BRIDGING THE GAP BETWEEN HETEROGENEOUS COMPUTING AND NEXT GENERATION MEMORY ARCHITECTURE USING HIGH LEVEL SYNTHESIS

by

Abhilash D. Rajagopala

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2019

Approved by:

Dr. Ronald R. Sass

Dr. James M. Conrad

Dr. Thomas P. Weldon

Dr. Andrew G. Schmidt

Dr. Erik J. Saule

ABSTRACT

ABHILASH D. RAJAGOPALA. Bridging the gap between heterogeneous computing and next generation memory architecture using high level synthesis. (Under the direction of DR. RONALD R. SASS)

The impact of transistor scaling on FPGAs is changing the role of FPGA from accelerators to a major role as processors. With this rapid development and the ability to implement complex systems on FPGAs, the conventional hardware language design flow is making way for software-like language using High-Level Synthesis (HLS). While academic and commercial HLS tools have made huge strides, nearly all these tools focus exclusively on the computation and the data path. Rarely do they directly address the memory subsystem and its impact on the overall performance. At best, the programmers can assist the tools with optimization which indirectly impact the memory subsystem performance. This has (unintentionally) exacerbated the already existing memory issues. The performance of DDR memory which has been the main stream off-chip memory has been lagging behind the processor performance. This has resulted in a performance gap and emerging memories such as Hybrid Memory Cube (HMC), High Bandwidth Memory (HBM), and others are promising prospects in reducing this gap. However, integrating these new memory technologies with HLS design flow has not been trivial. To fully utilize the performance benefits, the programmer must understand the low-level details of the hardware.

In this work, we conduct a systematic analysis of different HLS generated circuits on different off-chip memories. Our analysis identifies the root cause of the problem which is how the low-level hardware interacts with the memory subsystem. To mitigate these issues we introduce a hardware middle layer which establishes compatibility and a software transformation to improve the performance. This design is compared with the baseline system to evaluate the performance improvement from the methodology on heterogeneous systems.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and heartfelt thank you for everyone who helped me with this dissertation. Your advice, encouragement, and support has been very valuable to which I am always indebted. This work was supported by Electrical & Computer Engineering Department (my department) and Information Science Institute (ISI) to which I am always grateful.

Many thanks to my committee who dedicated many hours and been highly supportive throught this dissertation. Your advice and encouragement means a lot to me and has helped me to keep on track.

I would like thank ISI (where this idea was born) for giving me an oppurtunity to work in your lab and help me formulate the initial concept.

A special thanks to Andy at ISI, who has been inspirational, and has helped me throughout this process. He has shown me, by his example, what a good scientist (and person) should be. It has been a real pleasure working with you.

This work would not be completed without my advisor. Ron, your relentless support, valuable advices, insightful suggestions ... (running out of adjectives here) even with everything that was going on, you have been the best. It was a honour to be your student and I could not have asked for more.

A supportive family is a blessing. Thanks to my parents Shyla and Rajagopala for supporting me this entire time. I couldn't have done this without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: Introduction

Processor performance has steadily, and significantly improved since the introduction of integrated circuit in 1960s. The improvement is the result of advancement in the transistor scaling (as famously predicted Moore's law [1]), improvement in processor architecture, and compiler technology. While the relative contribution of each of these improvements can be argued[2], the exponential growth in single-core processor performance has been undeniable. However, Dennard scaling [3] which closely tracked Moore's law by reducing the threshold voltage as the transistors get smaller, ended in 2004 when power and thermal efficiency limitations with smaller technology nodes. This led to the emergence of multi-core and many-core architecture based on the accelerator technologies.

Multi-core architectures (also known as homogeneous parallel architectures) are simpler to implement since all computing cores are identical. However, many-core architectures (also known as heterogeneous parallel architectures) which is a mix of processors [4] and/or a mix of accelerators such as GPUs, FPGAs, *etc.* are better in terms of energy efficiency and performance [5]. Not surprisingly, as of 2019 more than one-fourth of the world's fastest machines are heterogeneous architectures [6]. Moreover, most of these machines are also in the top 100 of the GREEN500 list [7] which sorts the top 500 fastest machine by energy efficiency.

Unlike the scaling of processor performance which has dominated computer architecture, the impact of memory - specifically *main memory* technology is insignificant. Dynamic Random Access Memory (DRAM) [8] was invented in the 1960s has greatly benefited from transistor scaling in terms of capacity. However, in terms of latencies the technology is flat. While the bandwidth has seen the most gain it is the result

of improved interfaces. Specifically, the introduction of SDRAM (1992) and then the DDR interface (1998) [9] has historically had the most impact on bandwidth . The DDR family of DRAM (DDR1 through DDR5) has been the dominant interface and memory technology for the last two decades. However, researchers and industry are looking to "next-generation" memories for the future. This includes examples such as the Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM). Both use DRAM at the most basic storage level but dramatically change the interface. These changes results in increase bandwidth and capacity but with modest improvements to memory latency.

The confluence of this sixty year history is that we have to deal with heterogeneous architectures, a haphazard set of standard interfaces, and nascent programming tools to use many-core architectures, and a lack of computation performance models ("rule-of-thumb guidelines for practitioners). This thesis identifies the current state of FPGA based accelerators, identifies the root problems, offers incremental improvements, and suggests a future line of inquiry to realize the potential of the next generation systems.

1.1 Accelerator Technology

While heterogeneous architectures provide performance and energy efficiency, they pose serious challenges for programmers. Specifically, programmers should posses knowledge of new interconnection models, design tools, scheduling parallel tasks and new user/API interfaces [10]. Most of the programming models such as CUDA [11], Open-MP [12], and others addresses these design challenges by providing platform specific solutions. Unfortunately, these solutions are not generic and computational scientist has to make numerous technological decisions before even writing the first line of code.

Among different heterogeneous elements, FPGA devices has a great potential in generating computational performance and design flexibility for multi-core architectures. Though, early FPGA devices were small with limited logic blocks, the current

state-of-the-art devices are large and have a rich set of computing elements. Indeed, entire System-On-a-Chip (SoC) design that incorporates processors and FPGA accelerators are a reality. But, the FPGA devices are typically designed using Hardware Description Language(s) (HDL). These languages are reactive and concurrent as opposed their imperative software counterpart. Due to this HDLs are perceived as more difficult and to ease the burden, numerous High Level Synthesis (HLS) systems have been proposed [13, 14, 15, 16] HLS offers the flexibility of implementing FPGA design using software-like, high-level languages (HLLs). More recently, many academic and commercial tools can even generate an entire SoC design using HLLs such as C or C++. These technologies can significantly reduce the programmers burden and improve their productivity.

In theory, HLS is a transformation of imperative programs with conditional statements converted into multiplexers (MUX) or Look-Up-Table (LUT), loops into state machines, and computation into custom computing circuits or DSP slices. Unfortunately, many of these transformation tools have failed due to lack of well defined or universally accepted models for high level capture, poor quality of synthesis results, and lack of verification tools [17]. Modern HLS tools tries to address these issues but (as we will show) they still have specific design flows, vendor-specific programming models, and manual optimization techniques.

1.2 Memory Technology

While processor performance has steadily increased, the performance of memory and I/O has not caught up to the performance of the processors. The I/O performance has improved in fits and spurts and the main memory has steadily improved but at a very lower rate than processors. This slow progress has created a performance gap between the memory and single core processor performance. The well known Figure 1.1 which is cited by many articles and text books [18] illustrates the growing performance gap (Note the log scale on y-axis) of 50% per year between the memory

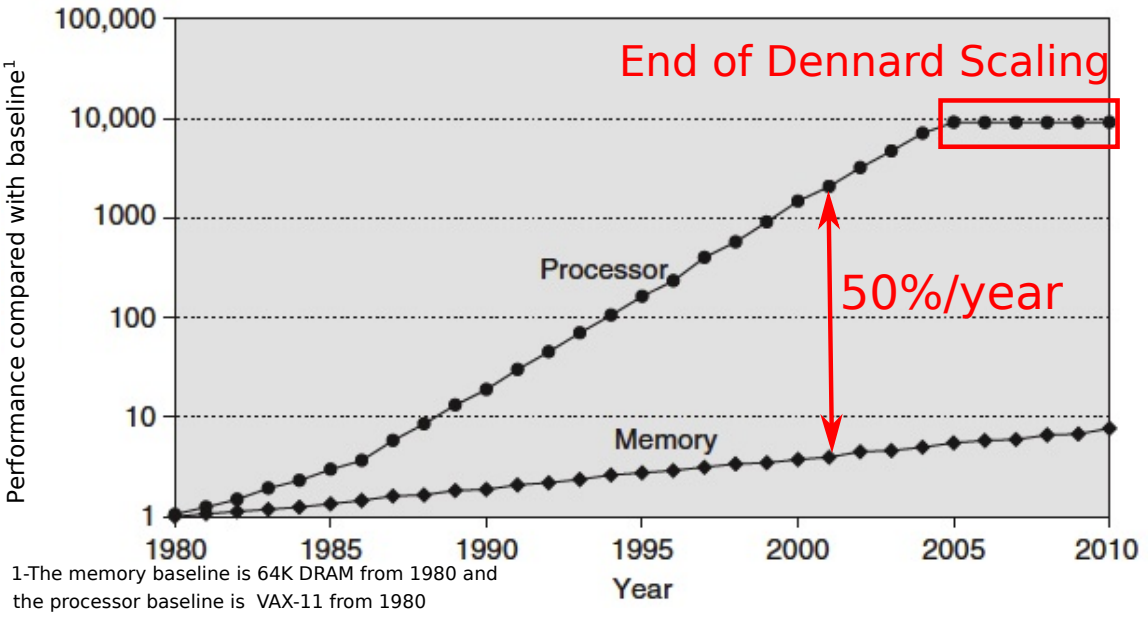

Figure 1.1: Performance Gap between Computing and Memory

and single core processors (After year 2010 new multi-core metrics has emerged and direct comparisons are not valid). This phenomenon of memory being the bottleneck to the overall performance is known as *Memory Wall problem* [19].

Conventional DRAM technology, has been ubiquitous since its introduction in 1960. The popular off-chip memory has improved in bandwidth, the latency has remained stagnant for decades. At the current rate of improvement the Memory wall problem cannot be resolved. In fact, future DDR improvements are unlikely to cater the needs and industry has been exploring novel memory technology to replace DDR. These new technologies, however, have very different performance characteristics and the programmer has to be aware of low-level details to fully utilize these devices. A memory agnostic approach to programming would solve this issue, but no such framework has been developed or explored. We propose to investigate these two issues — high level heterogeneous programming and novel memory— jointly in a single framework.

1.3 Thesis Statement

As heterogeneous architectures with different types of memories are prevailing, the design complexity has increased. HLS is a solution that has worked well with embedded applications. But, can the success of HLS be translated to HPC applications with larger data sizes? Additionally, the HLS generated accelerators are highly dependent on user-defined optimization. Most of these optimizations directly manipulate the internal memory. So, is there a directive or a set of directives that provide the performance for HPC applications on off-chip memory? Finally, with changing memory subsystem the high level programming requires low-level details of memory and this hinders the productivity. Can we describe a memory agnostic framework to restore the design productivity with high level programming?

Thus, the ultimate question we are trying to answer is *Can Computational scientists' benefit from High Level Synthesis to develop High Performance Computing applications?* In this work, we propose to answer all of these questions by designing a framework called *Volcan*. This framework provides a memory agnostic approach and conducts a systematic platform analysis for applications developed in HLS with different memory architecture. The analysis includes use of software profiling for different optimization, run-time measurements with different architecture and a software methodology to improve the performance with emerging memory architecture while preserving the designer productivity.

The remainder of this dissertation is as follows. The necessary background information the reader should be familiar with to understand the work is presented in Chapter 2. The expanded motivation for this research is in Chapter 3. The details and specifics of the design infrastructure, *Volcan* architecture and methodology is discussed in Chapter 4. The evaluation is presented in Chapter 5. Lastly, in Chapter 6, the summary of this work is discussed.

CHAPTER 2: Background

2.1 Nomenclature

Before we begin, it is valuable to highlight the acronyms and keywords used in this document. Some of these words get used and reused, but the specific meaning of each acronym is important to the rest of the discussion. First, a High-Level Language (**HLL**) is a *software* in the traditional sense (i.e., C, C++, Java, and others). Here, "higher" refers to more abstract to the low-level details of machine instructions but still uses the imperative (fetch-execute cycle) style of directing machine execution. A Hardware Description Language (**HDL**) is also a human-readable code in VHDL or Verilog that specifies the behavior of the hardware circuits. A subset of HDL is *synthesized* into a form that ultimately is *implemented* in some technology (such as FPGA). While modern HDLs are also very high-level languages, they are different from HLLs because they describe (hardware) circuits that are inherently parallel and foreign to many software programmers. The "holy grail" is to create a system that translates **HLL** source into an **HDL** code that is synthesized and implemented. The literature has adopted the name High-Level Synthesis (**HLS**) for this translation process (which is unfortunate because the goal of the translation is implementation, not synthesis). The HLS source code in this document is referred as 'HLS application'. The hardware circuits that HLS generates is referred as **RTL** (Register Transfer Level). A RTL is a design abstraction that models digital circuits in terms of the operations and data. These RTL circuits are referred as **HLS core**. The HLS can build an optimized RTL with user defined optimizations. These optimizations are referred as **directives** in certain tools (when passed as command line argument) or as **pragmas** (when added to the code directly).

2.2 Memory

The oldest known artifact of recorded information is a Lembo bone [20] with 29 distinct notches on a baboon's fibula which dates back to 35,000 B.C. However, the roots of the computer memory dates back to the *Punch cards* [21] demonstrated in 1801 with a loom. These punched cards were the storage device for the early computers since Charles Babbage concept of Analytical Engine [22] in 1837 until they were replaced by magnetic storage units in 1980s. The magnetic recording is based on the principles of electromagnetic fields. The laws of electromagnetic fields was discovered by Michael Faraday [23] in 1831. The aspect of electromagnetism was expressed as a differential equation by James Clerk Maxwell which are known as *Faraday's Law*. This law underlies the principles of electromagnetic induction and rotation which are key for magnetic recording. These principles were the fundamentals of magnetic storage device which dates back to the *Drum Memory* of 1932 to the modern day Hard disk drive (HDD). These memories could retain the information without any energy source and hence known as *Non-Volatile Memory*. Converse to this are *Volatile Memory* which requires constant power to store and hold the information.

2.2.1 Non-Volatile Memory

Non-volatile memory has been used as primary storage in early computer, to store firmware and BIOS as well as secondary storage device in current generation computers. The *Drum Memory* [24] introduced in 1932 by Gustav Tauschek is regraded the earliest conception of non-volatile memory. A drum memory contains a large metallic cylinder of ferro-magnets in the form of drum. Structurally these are similar to current generation hard disk drives (HDD) which are made of flat disks of magnetic materials instead of drums. These Drum memories were replaced by Magnetic Core memory [25] (or just 'core memory'). The core memory was introduced in 1951 and

used a grid of magnetic core materials wounded by toroid magnetic materials to store multiple bits of data. The early implementation of core memory used in Whirlwind I [26] computer could store about 32 words. But, these were improved in later implementations to a density of 32 kilo-bits per cubic foot. This memory was upgraded with *Plated wire memory* [27] by using a grid of iron-nickle coated wires to thread multiple cores of memory. There were other innovative core memories such as *Core rope memory* used in Apollo Guidance computer [28], *Thin film memory* [29] used in UNIVAC 1107, *Disk pack* in IBM 1311 computers [30], *Twistor memory* [31], and *Bubble memory* [32]. Some concepts of the core memory are still relevant in modern day memories but the core memory by itself is obsolete since the introduction of transistor based memories such as SRAMs and DRAMs.

The most dominant non-volatile memory (NVM) is the the hard disk drive (HDD). The HDD is an electro-mechanical data storage device that uses magnetic materials for log term storage of data. The IBM3340 [33] introduced in 1973 was the first HDD technology which is known as 'Winchester head'. Over the years, these systems improved in their speed and capacities and are found in most computers today. The second most used NVM technology is the flash based storage devices. The flash drives provides a reliable, low power, high performance storage by eliminating the moving parts of the HDD. These devices were introduced in 1980s, but due to their lower capacity they were not the main stream NVM until 2000. Improvement in the MOS technology has led to higher density flash drives and are now the most popular NVMs referred as 'Solid State Drives (SSD)'.

There are many emerging NVM technologies to replace the existing flash based technology. Technologies such as Magnetoresistive RAMs (MRAM, STT-RAM) [34], Ferro-electric RAMs (FeRAM, FeFET RAM) [35, 36, 37], Resistive RAM (ReRAM) [38] also known as 'Memristor', and Phase Change Memory (PCM) [39] are implemented in smaller densities. Other technologies such as Racetrack memory [40], Millipede

Table 2.1: Summary of Emerging NVM technologies [44, 45, 46]

NVM	Cell Factor (F^2)	Read (ns)	Write (ns)	Power (pJ)	Endurance (/10yr)
Flash	4-5	25,000	200,000	10,000	10^4
MRAM	16-40	3-20	3-20	50	10^{15}
STT-RAM	37	<10	12.5	0.02	10^{15}
FeRAM	4	60	75	2	10^{15}
ReRAM	>5	<10	10	2	10^{15}
PCM	6-12	20-60	50-120	1000	10^{12}
CBRAM	6	50	50	2	$10^6/Month$
NRAM	5	10	10	10	10^{16}

memory [41], CBRAM [42], NRAM (based on carbon nano tubes) and FJG RAM [43] memories are being evaluated. Details of these technologies are scarce as they are on-going efforts and is out of scope of this document since here we are concentrating on volatile memories. However, the Table 2.1, summarizes these emerging memories and compares with the existing NAND Flash based on their density (in terms of standard feature size (F^2), access time (in nano seconds), power (in pico Joules), and endurance.

2.2.2 Volatile Memory

Until 1947, every computer memory was a non-volatile memory. The introduction of *Williams Tube* [47], based on Cathode ray tubes [48] in 1946-47 is the first known volatile memory. The Williams tube worked on the principle of secondary emission. These tubes could typically store about 1024 to 2560 bits of data. Based on similar principles there were other vacuum tubes based memories such as *Mellon optical memory* [49] and *Selectron Tubes*. These systems were rendered obsolete by introduction of more reliable memory technologies.

Contrary to CRT based memory, in 1947 Presper Eckert invented the *delay line memory* [50] based on the principles of analog delay lines. These memories were refresh-able as the modern day DRAMs but were accessed sequentially. The delay line memories were constructed by modifying an analog delay line with an amplifier

and a pulse re-shaper. These two device re-circulated the signal (refresh) from the output back to the input. The early delay line memory used mercury(Hg) as storage medium, quartz crystals as signal generators and sound waves for propagation. The choice of mercury was due to the similar acoustic impedance with piezo electric quartz crystals. These memory served as the main memory for early computers such as EDSAC (Electronic Delay Storage Automatic Calculator) built in 1949 and UNIVAC1 (Universal Automatic Computer) built in 1951.

The EDSAC [51], the first full-size stored program computer was developed at University of Cambridge. This machine used 32 mercury based delay lines of 576 bits each to store 512 of 35-bit words. The UNIVAC I, the first general purpose computer produced in United States, also had a 1.5 KB of delay line memory to store 1000 words of 12 characters. This was implemented using 126 mercury channels to a 18 mercury filled tubes. The later versions of delay lines used metallic wires for storage with nickel transducers. This setup created a mechanical magnetostrictive delay lines to generate torsional waves. These delay lines were more convenient and reliable for low capacity memory storage. By the early 1970s the semiconductor IC technology gained momentum and delay line memories were replaced by SRAM (Static Random Access Memory) and DRAM (Dynamic Random Access Memory).

2.2.2.1 Static Random Access Memory

With the invention of transistors, many inventors in 1960s had an idea to make clusters of transistors on a single silicon wafer or later known as 'Integrated Circuits (IC)'. This idea was was independently invented by Jack Kilby (1958) of Texas Instruments and Robert Noyce (1959) of Fairchild semiconductor [52]. Based on this integration, the first integrated bipolar Static Random-Access Memory (SRAM) was invented by Robert H. Norman at Fairchild semiconductor in 1963. The first commercial use of SRAM dates back to the IBM System 360 Model 95 computer that was introduced in 1965. Initially these memories were offered as a separate chip. But,

with improved fabrication process these memories became a part of the microprocessor core. The early SRAMs were constructed using full-CMOS with poly-silicon load. But, these were unimplementable due to the high leakage current. With better scaling technique these were replaced by thin-film-transistor (TFT) PMOS [53]. The PMOS configurations was similar to CMOS with six transistor structures but they differ in their silicon technology. As an alternative to PMOS configuration, a load-less four transistor (LL4T) cells [54] was created in 2001. Although this resulted in smaller area it increased the complexity of the cell design. With scaling fabrication technology, this configuration produced unreliable memories resulting in 6T SRAM cell as the default SRAM configuration for implementing registers and caches in the modern microprocessors.

The SRAM memory consists of array of SRAM cells with with additional control logic. The Figure 2.1 shows a 4×4 array of SRAM cells with each cell consisting of 6 transistors, a decoder logic to select the address lines, four input and output data lines, and a read/write line. The 6 transistor (6T) structure logically forms SR-latch to store the data with a control logic. While powered on, the SRAM remains in one of three states: Standby state, read state or write state. The standby state holds the state of the SR latch when no word line is asserted. For a read state, the address for the row is decoded and the output line (D_{out}) contains the output data from the cell. For a write state, the R/W logic is asserted, the input line (D_{in}) contains the input data word and the address decoder activates the row in which the data is stored.

2.2.2.2 DRAM

Dynamic Random Access Memory (DRAM) is a non-volatile random access memory that stores each bit of data in a capacitor within an integrated circuit. Since capacitors leak their charge, it needs to be refreshed periodically. Due to this refresh mechanism it called as *dynamic* memory.

The first use of capacitors as memory device was developed in a crypt-analytic

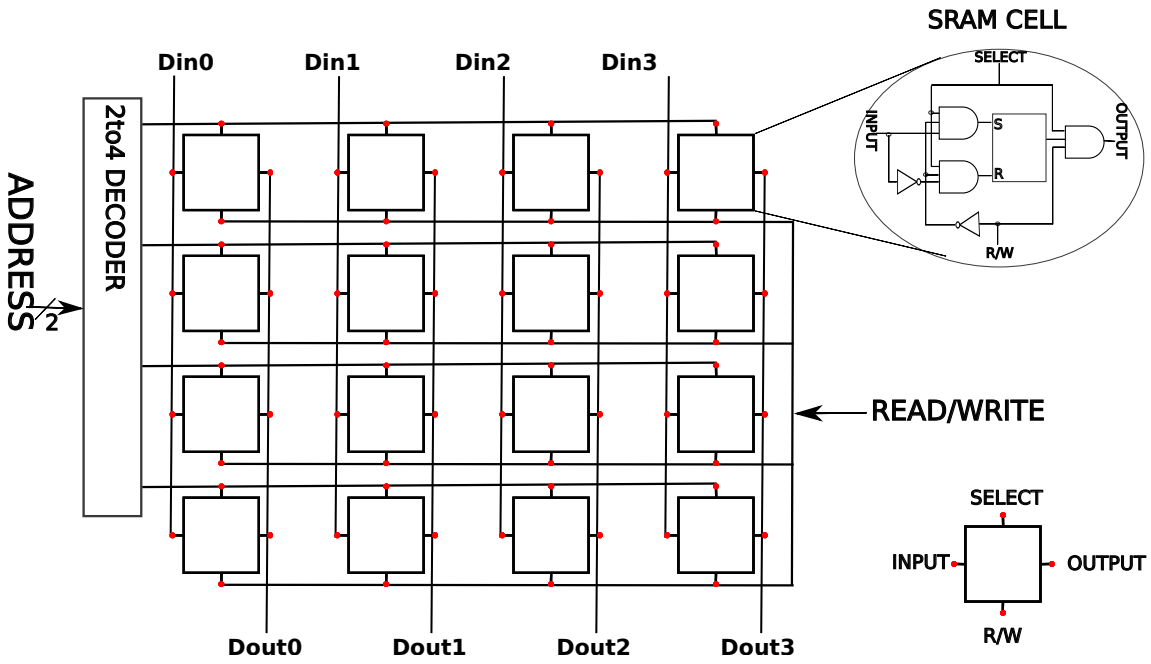

Figure 2.1: Structure of SRAM Cell Array

machine code named *Aquarius* used at Bletchley Park during World War II [55]. In 1964, Arnold Farber and Eugene Schlig working at IBM created a memory cell using a transistor gate and tunnel diode, which is known as Farber-Schlig cell [56]. In 1966, Dr. Robert H. Dennard of IBM Thomas J. Watson Research Center created a single transistor DRAM [8]. In 1970, William Regitz of Honeywell and Joel Karp of Intel designed a 1024 bit MOS memory named Intel 1102 [57]. This was a three transistor and single capacitor design but was commercially unsuccessful due to a narrow operating temperature. Based on this technology, Intel invented the first successful 1024-bit DRAM termed as Intel 1103. Though this memory was slow and difficult to manufacture, it established a viable low cost semiconductor memories in computers. The first DRAM with multiplexed row and column address lines was introduced in 1973 by Robert Proebsting with Mostek MK4096 4 Kbit DRAM. This was an important advancement, effectively halving the number of address lines required, which enabled the memory to fit into packages with fewer pins. In 1974, the 4Kbit DRAM was introduced with a single transistor cell using NMOS process.The

transition from 3 transistor (3T) to one transistor (1T) memory cell was the first major DRAM transition. From this time the DRAM increased in the capacity in relation with the transistor scaling. Until 1992, the DRAMs used asynchronous interface. The first commercial Synchronous DRAM chip was introduced in 1992 with the Samsung KM48SL2000, which had a capacity of 16 MB. The next milestone was the double data rate (DDR) which was also released by Samsung in 1998. From this time DDR-SDRAM has been the standard and has improved its performance and capacity from early DDR-1 to the latest DDR-4 memory.

The DRAM memories are typically arranged as a rectangle array of memory cell as shown in the Figure 2.2. Each memory cell consists of a transistor (T) and a capacitor (C) to hold a single bit of data. The array of memory cells are organized in columns and rows with a memory cell at each intersection. This structure allows an independent access to every bits with same latency also known as 'random access'. Apart from the memory cell, the DRAM memory consists of address decoders, sense amplifiers and data buffers. The address decoders are split into row and column decoders which selects the cell or a group of cells for read/write operation. Once the cell(s) are selected the sense amplifier drives the data from the data buffers into the selected cell(s) or read data from the cell to the data buffers.

Apart from the above the mentioned components there are four more components that makes the DRAM: the memory controller, memory banks, channels, and memory ranks. The memory controller manages the flow of data going to and from the DRAM. They have the necessary read and write logic and refresh rate control depending on the type of DRAM. Earlier, the memory controllers were a separate chip but in modern DRAM they are on the memory chip. The memory banks are concurrent portions of the memory which can perform independent memory operations (internal concurrency). In addition to banks, devices are grouped together into memory ranks. With these ranks the devices with relatively narrow interfaces can be used for wide

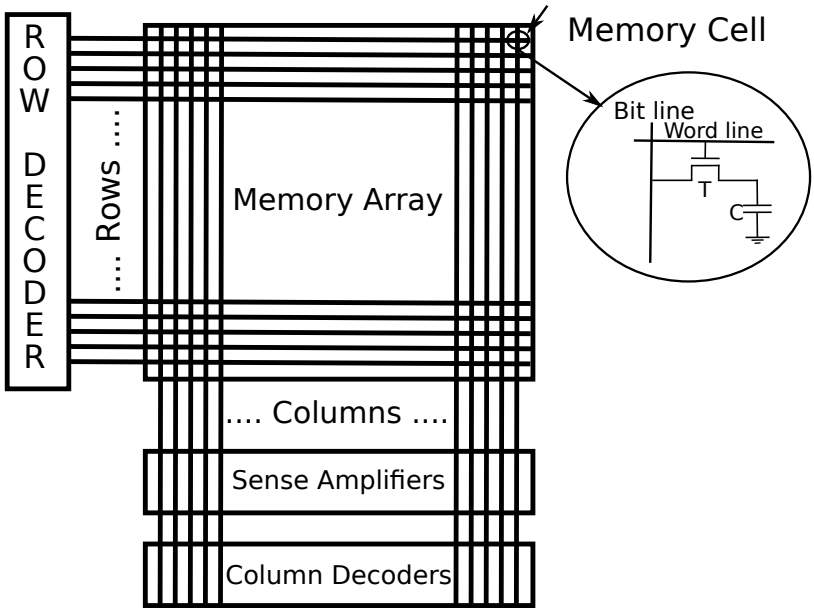

Figure 2.2: Structure of DRAM Memory

channels. For example, a rank of ×8 (8 bit wide) DRAMs would consist of eight physical chips and a rank of ×4 would consists of 16 physical chips all co-existing on a single or multiple DIMM (Dual In-line Memory Module) slot.

The read and write operations of a DRAM is done in multiple phases. In the first phase of data access, a row activation strobe (RAS) causes all the memory in a particular bank to activate a word for an entire row. The sense amplifier detects the value and stores it in the read buffer. After the data is stored, the column access strobe (CAS) drives data in or out depending on read or write operation. Before a different row is activated, a pre-charge command (PRE) is issued to ready the sense amplifiers. The memory protocol allows a row to be implicitly pre-charged in order to reduce the contention for the bus.

Apart from regular DRAM technology there are few specialized DRAM that currently exists. The DRAMs such as low power DRAMs (LPDDR), reduced latency DRAM (RLDRAM), and Synchronous graphics RAM (SGRAM) are introduced for specific purposes. The growth of mobile systems demands more RAM memory but with power constraints. To satisfy this low power DRAMs (LPDDR) are introduced.

Table 2.2: Performance of different DRAM technology

Memory	**Bandwidth (GB/s)**	**Latency(ns)**			
		t_{RCD}	t_{RAS}	t_{RP}	t_{CL}
DDR-400	5.6	20	40	20	15
DDR2-800	8	12.5	40	12.5	12.5
DDR3-1600	14.9	11.25	33.75	11.25	11.25
DDR4	25.6	14	33	14	14
GDDR5	48	14	28	12	16
HBM	256	14	34	14	14
HMC	320	14	27	14	14

t_{RCD} - Read Access Strobe to Column Access Strobe delay
t_{RAS} - Read Access Strobe active time
t_{RP} - Read Access Strobe pre-charge
t_{CL} - Column Access Strobe latency

The LPDDR supports 16 bit or 32 bit wide bus (as apposed to 64 bit on regular DDR) and can operated at 1.8V of supply voltage (instead of 2.5V). To improve the power efficiency they support temperature compensated refresh and ability to turn off device [58]. The LPDDR has improved over the years (LPDDR-1 to LPDDR-4) with increase in bandwidth and power efficiency. Another alternative to low power DRAM are reduced latency DRAM (RLDRAM). These device require more power and have less density but can offer 5× the speed of regular DDR [59]. They were designed for applications in networking, high-end commercial graphics, and level 3 caches which required low latency and capacities greater than SRAM. The Synchronous graphics RAM (SGRAM) are used for rendering the graphics such as texture memory [60] and frame buffers in video cards [61]. The earliest known SGRAM memory was introduced in 1994 by Hitachi [62]. This HM5283206FP chip was a 8Mb operating at 125 MHz. The modern data SGRAM known as Graphics Double Data Rate (GDDR) was introduced in 2000. The GDDR has improved from GDDR-1 to GDDR-6 which are constructed using the same base technology as DDR but optimized for power efficiency and throughput.

2.2.3 Next-generation Memory

The 3 dimensional ICs has been studied since 1980s [63]. But, feasible implementation were restricted due to the technical and economical challenges. These ICs offer a multiple advantage over the conventional 2D-IC. These 3D integration alleviates communication bottlenecks, integrates heterogeneous materials and enables novel architectures [64, 65]. There are multiple approaches to realize these 3D ICs. The most used methods are chip stacking [66, 67, 68], wafer stacking [69, 70, 71], and full monolithic integration [72, 73]. These technologies mainly differ in the size of their inter-layer via. The size can range from tens of micron meter in case of chip stacking to tens of nano meter in full monolithic integration. The key enabling elements for all these 3D technology is through-silicon-via (TSV). A good deal of research is performed to define the fabrication and process of the TSV [74, 75]. The chip stacking techniques with TSVs can significantly increase the density of the memory which can be seen in memories such as Hybrid Memory Cube(HMC) and High Bandwidth Memory (HBM).

2.2.3.1 High Bandwidth Memory (HBM)

High Bandwidth Memory (HBM) is a 3D stacked DRAM memory connected via Through-Silicon-Via (TSV) [76]. This memory technology adopted by JEDEC as an industry standard, can achieve higher bandwidth while using less power. Typically there are eight DRAM dies that are stacked up together to form a 3D circuit on top of a base die which has the memory controllers. The latest HBM standard [77] can support multiple DRAM stack of 2, 4, 8 or 12-Hi. A four DRAM dies (4-Hi) has two 128 bit channels per die and a total width of 1024 bits. The current HBM specification can support up to 24 GB per device at speeds up to 307 GB/s.

2.2.3.2 Hybrid Memory Cube

The Hybrid Memory Cube (HMC) [78] is structurally similar to HBM but they are different in memory characteristics. Like HBM, this memory is a stack of multiple DRAM memory and a logic die connected via Through-Silicon-Via. While both HMC and HBM are 3D memories, the HMC memory exists as a standalone system which can be connected via high speed serial links where as HBM co-exists with a processor (FPGA, GPU, or CPU) connected via 2.5D interposers [79]. Due standalone and non-JEDEC compliance of HMC, understanding the structure and the memory addressing is important for system integration.

The HMC memory structure as shown in Figure 2.3, is a stack of multiple DRAM memory dies and a logic die connected via Through-Silicon-Via (TSV). The stack of byte addressable DRAM memory blocks are known as *memory bank*. A group of eight memory banks is known as a *vault*. The HMC vaults are further grouped into *quadrants*. There are 16 vaults in a 4GB HMC and four of these vaults form one quadrant. Each quadrant is directly connected to a high speed serial link and connected to each other by a full crossbar switch. Access from a serial link to a local quadrant may have a lower latency than an access to a remote quadrant. Since, the remote quadrant is accessed through a crossbar switch and this requires multiple levels of arbitration.

Each vault has a separate memory controller in the logic area known as *vault controller*. This controller independently controls the timing and refresh rate requirements of the vault. Thus, the vaults are asynchronous to each other but the memory banks within a vault are synchronous. The vault controller buffers the memory request using a queue. The execution of the request is based on transactional efficiency which sets aside requests with bank conflict. This type of execution makes the HMC read access out-of-order. However, the request from a link to the same vault/bank address is executed in order.

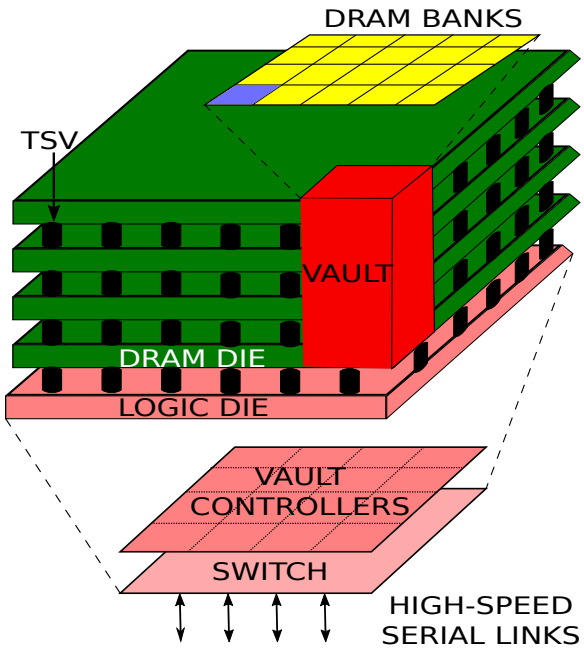

Figure 2.3: Structure of Hybrid Memory Cube

The HMC memory as shown in Figure 2.4 has a 34-bit addresses of which currently 31-bits are used. These addresses specify the quadrants, vaults, banks, and DRAM address bit. A 'vault interleaved' mapping algorithm is followed where the vault address are in Lower significant bit followed by the bank addresses. This mapping forces the sequential addressing to spread across multiple vaults. he address mapping in HMC is dependent on the memory size and the maximum block size. In this design, the maximum block size is set to 128B. The 34 bit HMC address field shown in Figure 2.4 can address up to 16 GB of memory. However, the current memory is 4 GB and hence the first three bits of MSB are ignored. The address mapping follows a *low-order-interleaving* policy for vaults and banks. Since, the minimum data granularity is 128 bit or *FLIT* the 4 LSB are ignored. The next three bits represents the total block size of 128 B. The next four bits from bit 7 to 11 are used to identify the 16 vaults followed by 4 bits to represent 16 banks of memory. The rest of the bits represents the DRAM address breaks into row and column address to access 1M bits blocks (Bits 4 to 7 and 15 to 31, total of 20 bits $2^{20} = 1M$ of 16 bytes

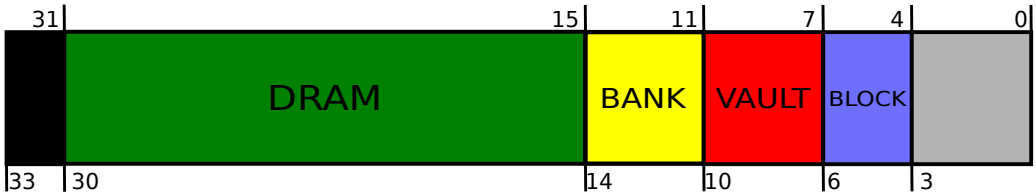

Figure 2.4: HMC address for 128B Block size and 4GB Memory

(Bits 0 to 3).

2.3 Interconnect

Interconnects are physical or logical links between devices. There are various interconnect and accompanying protocol standards. In this document, we use AXI interconnect to connect different logic blocks, Dual In-line Memory Module (DIMM) to connect DDR, PCIe to connect HMC backplane to system, and High Speed Serial Interface (HSSI) to connect FPGA and HMC. The specific DIMM, PCIe, and HSSI interface used in the design are discussed in Chapter 4.

Advance eXtensible Interface (AXI) [80], is an open standard specification and a part of Advanced Microcontroller Bus Architecture (AMBA) [81] from ARM. The AXI protocol is a parallel high performance specification, supporting multiple slave and masters with a synchronous interface. The AXI-4 specification which is the upgrade to the AXI-3 specification targets high-bandwidth and high clock frequency system designs. The AXI4 Interface is implemented as two major interface AXI stream and AXI bus. The AXI4-Stream protocol is used as a standard interface to connect a single master, that generates data, to a single slave, that receives data. The protocol can multiple data streams as well as multiple masters and slaves. The connection uses the same set of wires, allowing a generic interconnect to perform upsizing, downsizing and routing operations. The bus interface which is referred as 'AXI4', is capable of memory mapping up to 32 masters and slaves with a burst transactions up to 256 data transfer cycles per address phase. The AXI write and read transactions consists of 5 channels: Write address, data, and response channel and Read address and data

channel.

The AXI protocol supports burst data transfer to transfer multiple data transfer as a single request. There are three types of burst transfer that can be selected for read and write:

- Fixed: the address remains the same for every transfer
- Incr: the address increments for each transfer
- Wrap: the address increments up to the wrap boundary. The wrap boundary is determined by the bytes transferred and the burst length

2.4 High Level Synthesis (HLS)

High level languages (HLLs) has been tried for hardware circuit design since the 1970s [82]. Though these early research were unsuccessful, we finally have productive High-Level Synthesis (HLS) tools since the last decade. The HLS tools can be classified as commercial and open-source tools depending on the developer or as domain specific or generic language tools depending on the source language. The main distinction on each of these tools are the tool specification, design flow, and the optimization.

2.4.1 HLS tools

There are many commercial and open-source HLS tools that are available today. Commercial tools such as Catapult-C [83] introduced in 2004 initially oriented towards application-specific integrated circuit (ASIC) but can now convert C, C++, System C codes into VHDL/Verilog design for FPGAs. Bluespec [84] is another commercial tool that uses a high-level functional hardware language based on Verilog and inspired by Haskell for hardware design. There are many open source tools that are developed and Chisel [85] is one example of a hardware construction language based on Scala programming language. Although these domain specific tools differ from the generic

language HLS tool used in this research, they all have a common goal of improving designer productivity for hardware design using HLLs.

High Level Synthesis tools that support generic languages can be broadly classified into procedural languages and object oriented languages tools. There are few tools that supports object oriented languages such as Cynthesizer [86], JHDL (SeaCucumber) [87] and Max Compiler [88] which have limited success. On the other hand, procedural language tools such as Bambu [89], LegUp [90], Intel HLS [91], and Xilinx Vivado HLS [92] have all gained their fair share in HLS domain. Bambu, is an academic GNU compiler based HLS tool from Politecnico di Milano. This tool supports compiler based optimization and can produce target designs for both Xilinx and Intel FPGAs. LegUp, is another open source LLVM-based HLS tools from University of Toronto, that can synthesize C code into Verilog without building an infrastructure from scratch. The academic version of this tools targets Intel FPGAs but commercial version supports both Intel and Xilinx FPGAs.

Vivado HLS is a commercial tool by Xilinx which is based on an earlier HLS tool called AutoESL [93] (acquired by Xilinx in 2011). This tool uses LLVM compiler backend to generate hardware RTL for Xilinx devices using C, C++, and SystemC languages. This tool includes a complete design environment and ability to fine-tune the hardware RTL by using many design hints (pragmas). For System-On-a-Chip (SoC) devices, the tool allows a complete software solution for designing accelerators. Additionally, the tools adds appropriate *data movers* for off-chip memory access based on the accelerator's data pattern. For FPGA devices (non SoCs), the tool supports accelerators design in HLL and converts it into a RTL. This RTL can be integrated to a FPGA design and implemented using traditional FPGA design flow.

The accelerators are designed in C/C++ using Vivado High-level Synthesis (HLS) flow. The Vivado HLS compiler converts the High level language (HLL) design into a Register Transfer Level (RTL) that can be synthesized and implemented on FPGA.

The HLS transformation from C to RTL consists of three main process:

- Scheduling: determines the operations to be executed on each clock cycle. This is dependent on frequency of FPGA device, execution time for the operation to complete and optimization directives.
- Binding: determines the hardware resources to be implemented for the scheduled operations. This is dependent on the resource available in the FPGA devices.
- Control logic extraction: creates a finite state machine (FSM) in hardware language (Verilog in our design) to implement the sequence of operations in the RTL design.

The C/C++ code for HLS design has a top-level function with function arguments followed by number of sub-functions. The body of the function and sub-function contains variables, arrays, and loops. The HLS transforms the top level function and sub-functions into blocks in RTL. The function arguments are converted into inputs and outputs of this RTL block. The variables in the function are mapped into registers and the array variables declared inside the C function is mapped as block RAM (BRAM) or UltraRAM in the final FPGA design. A single iteration of the loop is transformed into a logic block by HLS and the RTL executes this logic block for the entire loop sequence. By default, the loops are *rolled* but this behavior can be modified by optimization directives.

2.4.2 HLS Optimization

HLS tools provides user with various optimization techniques to improve the performance of the application. As summarized in this survey paper by Nane *et. al*, the optimizations can be performed at various levels of the HLS process. The optimization can be achieved by chaining the operation resulting for lower latency, custom bit

width on the interface, memory space allocation, loop optimizations, *etc.* [94] Some of these optimizations are managed by the compiler and while for others user is required to add different pragmas or knobs to optimize the code. In this document, we classify the pragmas into two categories, the code pragma and interface pragmas. The code pragmas are the optimizations introduced to the body of the code (usually the loop) and the interface pragmas are added to the interface.

1. **Code pragma** The loop in a code performs a repetitive operation multiple times sequentially. By using the right pragma these loops can be transformed in HLS for better performance in hardware. There are many optimization techniques in this category which are provided by the tool. Since, there are various pragmas in this category we are concentrating on four most popular pragmas which are used in our design. The code examples shows are for Xilinx HLS tools but most other HLS tools have a similar or variation of the described pragma.

    - Pipeline: The `PIPELINE` pragma, transforms a sequentially executing loops into a pipelined loop. As shown in the Figure 2.5, the code on the left contains a `for` loop which does read, compute, and write operations. If we consider each operation to take one clock cycle, then an iteration of the loop is completed in 3 clock cycles (represented by the timing diagram on the left). When a PIPELINE pragma is added to this loop, the loop is pipelined such that each individual operation can be executed on every clock cycle. For example, in the below code without PIPELINE pragma the read operation is executed every 3 clock cycles but with PIPELINE this operation is executed on every clock cycle. Overall for two iterations of the loop, the execution time is reduced from 6 clock cycles to 4 clock cycles with pipelining. The example given represents an ideal case of one clock per operation but in reality this may vary (example: multiplication operation taking more than 3 clock cycle). The slowest executed operation affects the

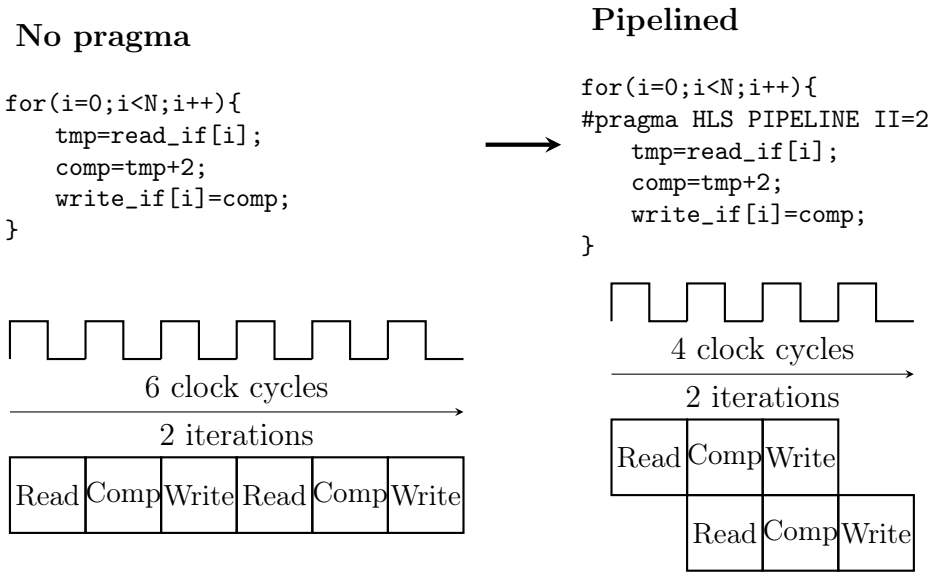

Figure 2.5: Comparison between no pragma and pipeline pragma

overall performance of the pipeline. Therefore, knowing this execution time becomes a critical parameter to determine the delay required to execute the rest of the operations. This time interval which is the time taken to execute the consecutive iterations of the pipelined loop is known as *Initiation Interval (II)*. In Xilinx HLS, this value by default is equal 1 and this value changes based on the user code. The current HLS compiler cannot predict the II value and it is the designers responsibility to provide one for pipelining.

- Unroll: The loop unrolling exploits parallelism by creating multiple copies of the loop body. The Figure 2.6 shows the HLS code with loop unrolling. The number of copies of the loop are determined by the numerical value known as *factor*. By default, the HLS compiler tries to unroll the entire loop, known as *full loop unroll*. If the factor used is less than the total loop iterations then it is known as *partial unroll*. Unrolling requires a large amount of resources, depending on the availability of the resource and the total iteration of the loop the user can decide either to fully or partially

```
for(i=0;i<N;i++){
#pragma HLS UNROLL factor=2
    tmp=read_if[i];
    }
```

Unrolled →

```
for(i=0;i<N/2;i++){
    tmp=read_if[i];
    tmp=read_if[i+1];
}
```

Figure 2.6: Loop unroll pragma in HLS

unroll the loop. In the example shown, the loop is partially unrolled with a factor of 2 on a loop of size N. This unrolls the loop twice and adjusts the iteration to N/2. Loop unrolling though advantageous is limited by the number of resources and the loop carried dependency (which is true for PIPELINE as well). For example, unrolling a loop with a single port memory may not result in performance gains since the memory access will be sequential. On the other hand, if a dependence exists within the operations of different iterations (know as loop-carried dependence) then this may stall the operation. Additionally, with nested loops the unrolled loop may result in inconsistent results.

- Array partition: Every memory optimization pragmas in HLS concentrate on the local memory which is implemented in the BRAM. The array partitioning is one such pragma which partitions a local memory into various memory segment. If each of the segments are connected to a parallel BRAM then this results in performance gain. The Listing 2.1, shows a declaration a local memory of M rows and N columns. This memory or array is partitioned using the `ARRAY_PARTITION` pragma which divides the first dimension of the array `mem_arr` into blocks of 8. If the *block* parameter is not mentioned then by default the array is divided completely into individual elements. The *block* creates smaller arrays from consecutive blocks of the original array, The other alternative is to use *cyclic* partition which creates smaller arrays by interleaving elements from the original array.

```
1    int mem_arr[M][N];
2    #pragma HLS ARRAY_PARTITION variable=mem_arr \
3    block factor=8 dim=1
```

Listing 2.1: Array partition pragma

- Loop trip count: The loop trip count pragma is used when the user need to manually supply the trip count information for the compiler. This trip count information is essential to calculate the latency of the module. In most cases, the compiler can calculate the trip count information. But, for the cases in which the variables used to determine the tripcount are either a input arguments or variables calculated by dynamic operation the HLS compiler cannot calculate this information and results in an unknown latency. At this point the user can provide the minimum and the maximum iterations of the loop using `LOOP_TRIPCOUNT` pragma.

2. **Interface pragma** The HLS framework supports different AXI interfaces such as AXI-Stream, AXI-lite, and AXI-full. To connect a HLS code to an external devices, user should choose one of the AXI port. These ports can be optimized with different optimization pragma. This allows to group different ports, specific the depth (in case of full AXI), and specify the outstanding and/or burst read and write factors.

   The HLS code Listing 2.2 has a C function with a name, argument, and a return type. In this example, the user defined function name is `ex_rt` which will be the top-level for the generated hardware. Every HLS generated hardware require a control interface. This interface is used for initialization, start-stop control, and for the status. By default the compiler adds a native interface called `ap_ctrl`. This can be modified by user to `axi_lite` as shown in line 2 of the Listing 2.2.

HLS allows use of C data types such as int, float, *etc* for the function arguments. Additional custom data width data type are available from textttap_int library. In the example shown, an integer pointed of 512 bit width is created. This argument is interfaced to an external memory. This can also be declared as an array if implemented as BRAM.

The interface details to the function can be defined using `INTERFACE` pragma. Line 3 defines the argument to be AXI master interface with the keyword *m_ axi*. The *depth* parameter in line 4 is required parameter for *m_ axi*. This parameter specifies the maximum number of test bench samples required for RTL co-simulation. The *offset* parameter controls the address offset of the interface. By default, the offset is 0 (*none*) and this can be modified to *direct*, in which case offset is entered into the HLS tool or as *slave*, where the offset is set at runtime using the control interface defined in line 2 (`axi_lite`).

```
1 void ex_rtl (ap_int<512> *inter1){
2 #pragma HLS INTERFACE s_axilite port=return
3 #pragma HLS INTERFACE m_axi depth=1024 port=inter1 offset=slave
```

Listing 2.2: Sample HLS code

2.5 Related Work

This dissertation covers two different domains: High-Level Synthesis (HLS) and Hybrid Memory Cube (HMC). Here we present some of the prior research that are conducted on these two domains individually. We also present some of the system level research and their distinction to this dissertation.

Most of the HLS research is focused on Design Space Exploration (DSE) [95, 96, 97]. These research have analyzed the HLS tools at compiler level and have provided various solutions to improve the performance of the HLS converted hardware. The

COMBA framework [98] explores a comprehensive set of HLS pragmas related to the functions, loops, and arrays. Further it finds the best configuration using analytical models and metric-guided DSE algorithm. These research are focused on computational performance and the local memory (with exception of data-center clusters) which is implemented using the BRAM. In our research, we conduct a similar DSE study focusing on the loop structure of the algorithm. But our analysis and observations are related to the performance and integration to the external main memory. Additionally, our research focus on HMC as main memory which is different from the DDR as well as the hybrid counterpart HBM.

Many HMC research efforts have focused on evaluating the characteristics of the hybrid memory [99, 100, 101]. The evaluations in these research explores the bandwidth, latency, thermal efficiency, and other unique memory characteristics of HMC (*FLIT*, addressing modes *etc*). Most of these works evaluates the memory with test applications that are interfaced using the HMC native interface. These evaluation has merits, since a sweeping study on memory parameters can be efficiently analyzed using native interfaces. We follow some of these evaluation in order to understand the baseline memory performance of HMC. We conduct our baseline experiments on HMC using the AXI interface by varying different memory parameters.

The Smart Memory Cube [102] design introduces a modular extension to the standard HMC using an AXI-4 interconnect. The design implements an interconnection among the Processing-In-Memory (PIM) [103] module, the link controller and the memory. This interconnect design sits between the link control of the standard HMC and the *vault* controller. This implementation includes an address re-mapper and scrambler to support near memory computation. Our design has an address manipulator to connect the HLS hardware and memory. But, this does not modify the existing memory interface or target any specific memory characteristics through addressing. Instead, it introduces a hardware middle layer for compatibility. Additional

performance improvement is achieved by modifying the HLS application.

Due to the presence of logic layer and the demand to reduce the memory latency many studies have focused on PIM with HMC [104, 105, 106, 107]. Some of these studies are simulation and some have actual implementation in the hardware. Our study does not involve any memory simulation. Instead, we design our HLS application as memory agnostic. Although our focus is on the time spent by the application on external memory all the computations are done inside the application. Thus, in terms of operation we focus on the basic read and write to the memory and do not exploit any `Read-Modify-Write` atomic instructions as done in PIM.

The HMC memory offers multiple parallel channels and performs better for large memory access. This can be exploited by mapping multi-core architectures which can generate large request. he GoblinCore-64 architecture [108] and RISC-V based architectures [109, 110] are some of the research in with multiple cores are mapped into HMC to generate large data request (some use memory coalescing) to exploit the performance. The research by Zhang *et. al.* [111, 112] optimizes the HMC for a particular application. In their experiment, they demonstrate it on graph applications such as Breadth First Search and graph traversal by minimizing the memory request to the external memory. Our research focuses on monolithic generic applications that are implemented as hardware. The software methodology tries to generate the large memory access for HMC by optimizing the HLS application to use a larger data width interface.

One of the motivating factor for our work is the fact that the existing HLS compiler do not support next generation memory addressing. However, there is a compiler level study that is done with CAIRO [113] compiler. This study targets the instruction-level offloading for Processing In-Memory using compiler assisted techniques. This research is related to identifying and supporting the in-memory execution and is not a generic compiler to use in HLS. Additionally, our work does not build any compiler

techniques to support HMC instruction. We design the applications such that the existing HLS compiler can support HMC.

There are few system level research that optimizes applications (not in HLS) for HMC. The research by Zhang *et. al.* [111, 112] optimizes the graph applications such as Breadth First Search and graph traversal by minimizing the memory request to the external memory. This is one of the approach in which individual algorithms (rather than generic) is acutely optimized for better performance in HMC. Since, HMC as well as HBM provides multiple channels some research have leveraged parallelism in an application (or architecture) to map the data to HMC. The GoblinCore-64 architecture [108] and RISC-V based architectures [109, 110] are some of the research in with multiple cores are mapped into HMC to generate large data request (some use memory coalescing) to exploit the performance. While application parallelism can benefit from HMC, HBM, as well Multi-Channel DDR (MCDDR), not all applications can be parallel and designing parallel applications affects designer productivity. Additionally running application on a soft-core processor has a lower performance than implementing them as a hardware. All the applications presented here are monolithic, non-parallel and are implemented as hardware circuits. However, some loop optimizations on HLS can produce parallelism in hardware but this does not change the application design.

CHAPTER 3: Motivation and Preliminary Results

In this chapter we discuss three preliminary research results that motivated the Vol can design. These preliminary results involves programmers productivity, HLS memory pattern, and bandwidth performance of HMC memory.

3.1 Programmers Productivity

More and more industries are adapting to the heterogeneous architecture for their performance and energy benefits. There are lot of potential examples and industry changes: Intel's HARP [114], Amazon EC2 F1 [115], Microsoft FPGA Bing search, Xilinx Deep Learning... But even with the adaptability there is a major challenge that has been overlooked: programmer's challenge. With all of these systems, bulk of the complexity has been shifted towards the programmer.

The programs that were simpler to design in the 70's has grown in complexity, even though it performs the same function. For example, consider the matrix multiplication application in a FORTRAN77, CUDA [116], and SDSoC with basic optimization as shown in the Listing 3.1 3.2 3.3. The CONVEX compiler [117] used for FORTRAN can convert a sequential code into a vectorized parallel optimized code by using a specific optimization flag during compile time. On the other hand, the CUDA programmer has to create and manage the blocks, threads, and memory. Finally, with SDSoC the programmer has to understand different interfaces and use the appropriate pragmas to implement it on FPGAs. This increase in complexity hinders the programmer productivity. This is one of the motivation for designing the Volcan framework that can abstract the low-level details to the programmer.

FORTRAN Code for Matrix Multiplication

Main

```
1 DO I = 1, N
2         DO J = 1, N
3                 c (I, J) = 0. 0
4                 DO K = 1, N
5                         C(I,J) = C(I,J) + A(I,K) * B(K,J)
6                 END DO
7         END DO
8 END DO
```

Command Line

```
1     %fc -03 -c mmult.f
```

Parallel Vectorized Code by Compiler

```
1    M = MVSL(N)
2 PARALLEL DO J = 1, N
3         DO IOUTER = 1, N, M
4                 C(IOUTER:MIN(N, IOUTER+ M - 1), J)= 0.0
5         END DO
6 END DO
7 PARALLEL DO J = 1, N
8         DO IOUTER = 1, N, M
9                 VO= C(IOUTER:MIN(N, IOUTER+ M - 1), J)
10                DO K = 1, N
11                        V1 = A(IOUTER:MIN(N, IOUTER+ M - 1), K)
12                        VO =VO + VI * B(K, J)
13                END DO
14                C(IOUTER:MIN(N, IOUTER+ M - 1), J) =VO
15        END DO
16 END DO
```

Listing 3.1: Matrix Multiplication in FORTRAN77/Convex (ca.1990)

CUDA Code for Matrix Multiplication

matrixMul.cu

```
1  /*
2   * Copyright 1993-2009 NVIDIA Corporation. All rights reserved.
3   * NVIDIA Corporation and its licensors retain all intellectual property and
4   * proprietary rights in and to this software and related documentation and
5   * any modifications thereto. Any use, reproduction, disclosure, or distribution
6   * of this software and related documentation without an express license
7   * agreement from NVIDIA Corporation is strictly prohibited.
8   */
9  #ifndef _MATRIXMUL_KERNEL_H_
10 #define _MATRIXMUL_KERNEL_H_
11 #include <stdio.h>
12 #include "matrixMul.h"
13 #define CHECK_BANK_CONFLICTS 0
14 #define AS(i, j) As[i][j]
15 #define BS(i, j) Bs[i][j]
16 #endif
17 __global__ void
18 matrixMul( float* C, float* A, float* B, int wA, int wB){
19     int bx = blockIdx.x;
20     int by = blockIdx.y;
21     int tx = threadIdx.x;
22     int ty = threadIdx.y;
23     int aBegin = wA * BLOCK_SIZE * by;
24     int aEnd  = aBegin + wA - 1;
25     int aStep = BLOCK_SIZE;
26     int bBegin = BLOCK_SIZE * bx;
27     int bStep = BLOCK_SIZE * wB;
28     float Csub = 0;
29     for (int a = aBegin, b = bBegin;a <= aEnd;a += aStep, b += bStep){
30         __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
```

```
31          __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
32          AS(ty, tx) = A[a + wA * ty + tx];
33          BS(ty, tx) = B[b + wB * ty + tx];
34          __syncthreads();
35          for (int k = 0; k < BLOCK_SIZE; ++k)
36              Csub += AS(ty, k) * BS(k, tx);
37          __syncthreads();}
38      int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
39      C[c + wB * ty + tx] = Csub;}
40  #endif // #ifndef _MATRIXMUL_KERNEL_H_
```

Command Line

```
1       nvcc matrixMul.cu -o matrixMul -gencode arch=compute_35,code=sm_35
```

Listing 3.2: Matrix Multiplication in CUDA

CUDA requires header file which is not shown here

SDSoC Code for Matrix Multiplication

mmult.cpp

```
1   #define SZ 32
2   //Pragma below instructs the HLS compiler to generate a PL
3   //design which has Direct Memory Interface with DDR and PL.
4   #pragma SDS data zero_copy(in1[0: SZ * SZ], in2[0: SZ * SZ],out[0: SZ * SZ])
5   void mmult_accel(int *seqOne, int *seqTwo, int *out){
6       //Local memory is implemented as BRAM memory blocks
7       int A[SZ][SZ],B[SZ][SZ],C[SZ][SZ];
8       int i,j,k,itr;
9       //Burst read data from DDR memory to BRAM
10      for (itr = 0, i = 0, j = 0; itr < SZ * SZ; itr++, j++){
11      #pragma HLS PIPELINE
12          if (j == SZ){ j = 0; i++;}
13          A[i][j] = in1[itr];
14          B[i][j] = in2[itr];}
```

```
15    //Performs matrix multiplication out = in1 x in2
16    for (i = 0; i < SZ; i++){
17        for (j = 0; j < SZ; j++){
18            int result = 0;
19            for (k = 0; k < SZ; k++){
20            #pragma HLS PIPELINE
21                result += A[i][k] * B[k][j];}
22            C[i][j] = result;}}
23    //Burst write the results from output matrix C to DDR memory
24    for (itr = 0, i = 0, j = 0; itr < SZ * SZ; itr++, j++){
25    #pragma HLS PIPELINE
26        if (j == SZ) { j = 0; i++;}
27        out[itr] = C[i][j];}}
```

main.cpp

```
1  #include <iostream>
2  #include <stdlib.h>
3  int main(int argc, char **argv) {
4      const int dim = SZ;
5      //Allocate memory:
6      int *in1 = (int *) sds_alloc(SZof(int) * dim * dim);
7      int *in2= (int *) sds_alloc(SZof(int) * dim * dim);
8      int *hw_results_1 = (int *) sds_alloc(SZof(int) * dim * dim);
9      //Create test data
10     for (int i = 0; i < dim * dim; i++) {
11         in1[i] = rand() % dim;
12         in2[i] = rand() % dim;
13         hw_results_1[i] = 0;
14      }
15     #pragma SDS resource (1)
16     mmult_accel (in1, in2, hw_results_1);
17     #pragma SDS resource (2)
18     mmult_accel (in1, in2, hw_results_2);
```

```
19     //Release Memory
20     sds_free(in1);
21     sds_free(in2);
22     sds_free(hw_results_1);
23     sds_free(hw_results_2);
24 }
```

Listing 3.3: Matrix Multiplication in SDSoC/HLS[b] [b] The accelerator code must be selected as hardware in the design flow

3.2 HLS memory pattern

The memory access pattern in an algorithm can be sequential or random. If memories are accessed with incremental address then these address can be used as a burst access. For a large data transfer these burst access improve the memory performance by generating single read or write request for a large data. One way to identify these pattern from the code is by analyzing the loop structure. For example, if a *for* loop in C code These burst pattern can be identified based on the algorithm. As shown in the Listing 3.4, a *for* loop with an incremental access will generate a burst access. But with High-level synthesis (HLS) these code are transformed based on the optimization of the loop. This can impact the performance of algorithm based on the loop optimization.

```
1 void addr_analy(int* inp){
2         int i=0;
3         for(i=0;i<size;i++)
4                 //Write data
5         for(i=0;i<size;i++)
6                 //Read data
```

Listing 3.4: *for* loop to create burst pattern

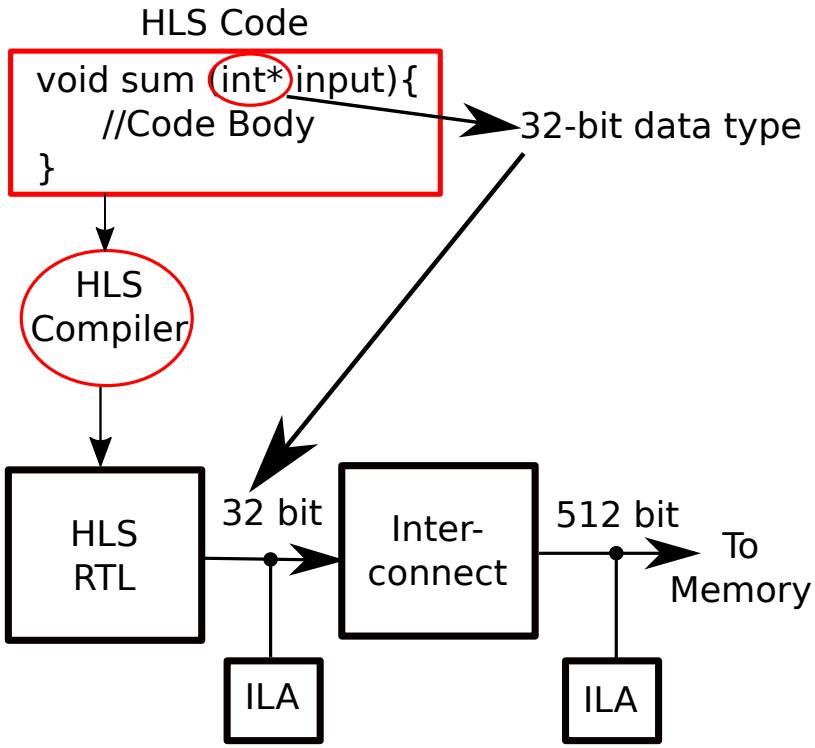

Figure 3.1: Address analysis setup

To understand the HLS behavior, we created the a hardware setup as shown in the Figure 3.1. This setup consists of HLS RTL of the code Listing 3.4 connected to an interconnect and memory. The integer pointer used in the top-level HLS code is transformed into a 32 bit data interface which is connected to the AXI Interconnect. The interconnect optimizes the burst transfer by packing multiple data into the 512 bit interface and generating a single address for the burst length. To capture the behavior of the HLS with different optimization and the interconnect, Integrated Logic Analyzers (ILA) [118] are connected to the interface.

The address capture for directives such as no optimization, pipeline, and loop split exhibit the same behavior. As shown in the Figure 3.2, the write signals at the 32 bit HLS interface generates a burst of 32 bit data. Each burst contains 16 data of 32 bit with total burst size of 512 bit. These burst are sent as a single address memory transaction to the memory by the interconnect. For subsequent address, the memory is incremented by 40h. In case of read both the HLS and interconnect, the

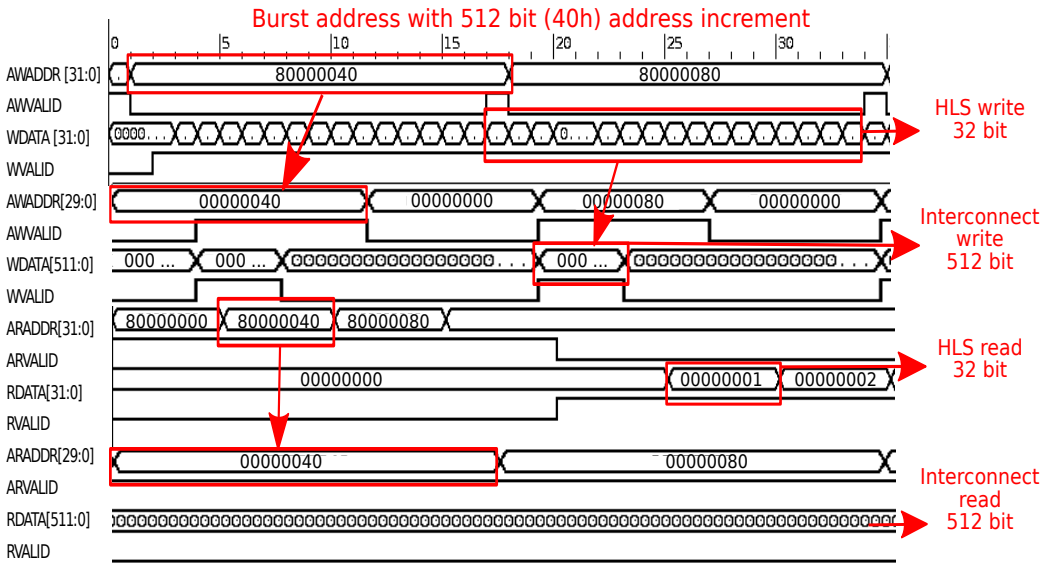

Figure 3.2: Address captured for NoPragma, Pipeline, and Loop split

interfaces generates a single read address by dropping two MSBs (used for memory map) to fetch the entire 512 bit of data from the memory. This 512 bit data is received by the interconnect and is sent to HLS as 32 bit data at a time.

The Figure 3.3, shows the address behavior for unroll and array partitioning optimizations. We can notice that the HLS RTL exhibit has no burst access. In this case, a 32 bit data is transferred one at a time with an address increment of 32 bit (shown by the signals `AWADDR[31:0]` and `ARADDR[31:0]` with 80030000h to 80030004h increment). The same address behavior is reflected at the 512 bit interconnect interface and for read access.

The above analysis shows that the burst behavior in HLS is not just dependent on algorithm but on the optimization in HLS. The Volcan framework has to consider this behavior and be adaptable based on the optimization used in HLS.

3.3 HMC Bandwidth

One of the key aspects of the next-generation memory is they are structurally different from the DDR memory. The performance of this 3D structured memory is controlled by the memory access pattern. As suggested by the previous studies in

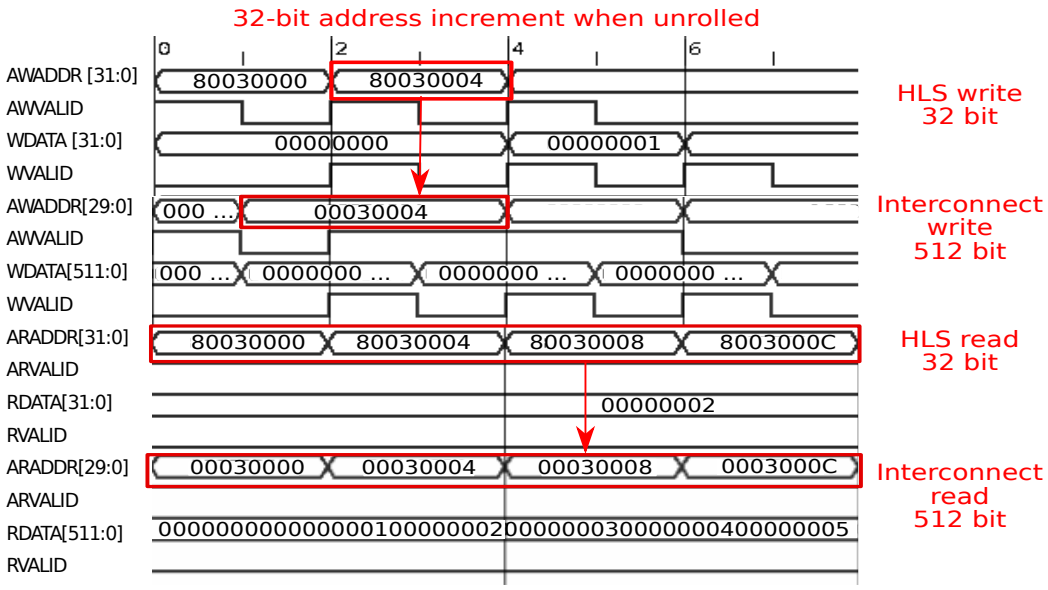

Figure 3.3: Address captured for Unroll and Array partitioning

Section 2.5, a large access size is key to tap the performance in HMC. To affirm this we conducted a preliminary study on Micron AC510 Hybrid Memory Cube (HMC) [119]. The bandwidth measurement experiments on HMC is conducted by using *GUPS* and *AXI HMC test* applications that are provided by Micron. These application differ in their memory access pattern. and are implemented as a Verilog module which is connected to the HMC memory controller. The applications perform read and write operations to the HMC with varying data sizes (FLIT), channels and burst sizes. The bandwidth is calculated as a product of operations per second and the transfer sizes.

The HMC memory controller provides two different interfaces. One is a parallel interface of 9 channels with each channel of 128 bit data width (referred as 'native') and the other is a AXI-4 interface of two channels of 512 bit data width. Irrespective of the interfaces these are physically connected to the memory using a 8 high speed serial links. Each native channel has a data width of 128 bit which is referred to as 'FLIT'. A single memory transaction can vary from a single channel and a single FLIT (referred as 'Default HLS access') to using all 9 channels in

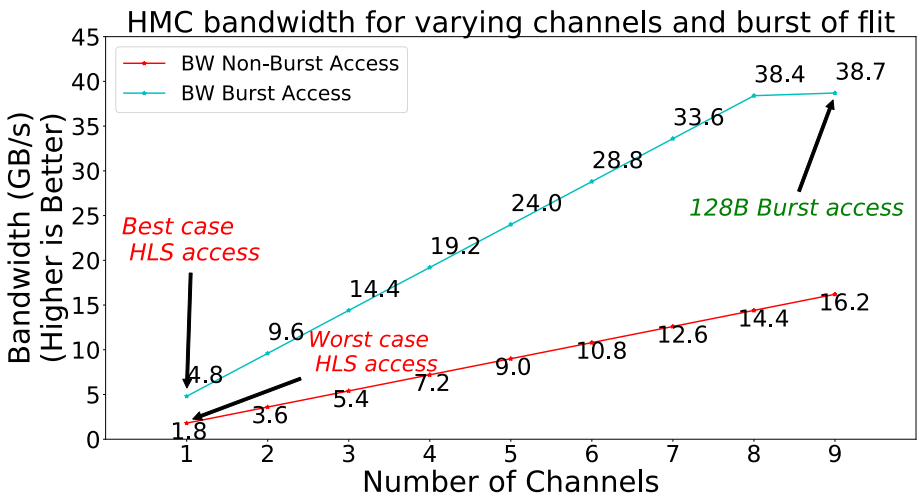

Figure 3.4: HMC Bandwidth for varying channels and burst sizes

parallel with maximum burst size of 128 bytes (burst length of 8 with each data of 128 bits). The Figure 3.4 shows that the bandwidth for these access can vary from 1.8-38.7 GB/s. As suggested earlier, a large data size of 128 bytes with multiple channel can achieve the peak bandwidth. But to reach this bandwidth the applications should be designed in parallel. While the degree of parallelism is limited to the algorithm and the HLS infrastructure has limited support for designing parallel HLS kernels. This results in the best case achievable bandwidth with current HLS infrastructure to 4.8 GB/s and the worst case to 1.8 GB/s.
The 512 bit AXI-4 channel is similar to the current DDR interfaces as well as the HBM. In this experiment we measure the bandwidth capabilities of HMC for different burst length (access size). The access sizes are varied by modifying the burst length from 0 to 255. For a single channel, the firmware is interfaced to one 512 bit AXI-4 interface of the HMC memory controller. For the dual channel, the same firmware module is instantiated to two 512 bit AXI-4 interface. Each instance of this firmware module independently executes a write transaction and places the write address and the data in a FIFO. After receiving the write response the write address is used from the FIFO for read transaction. After receiving the read data it

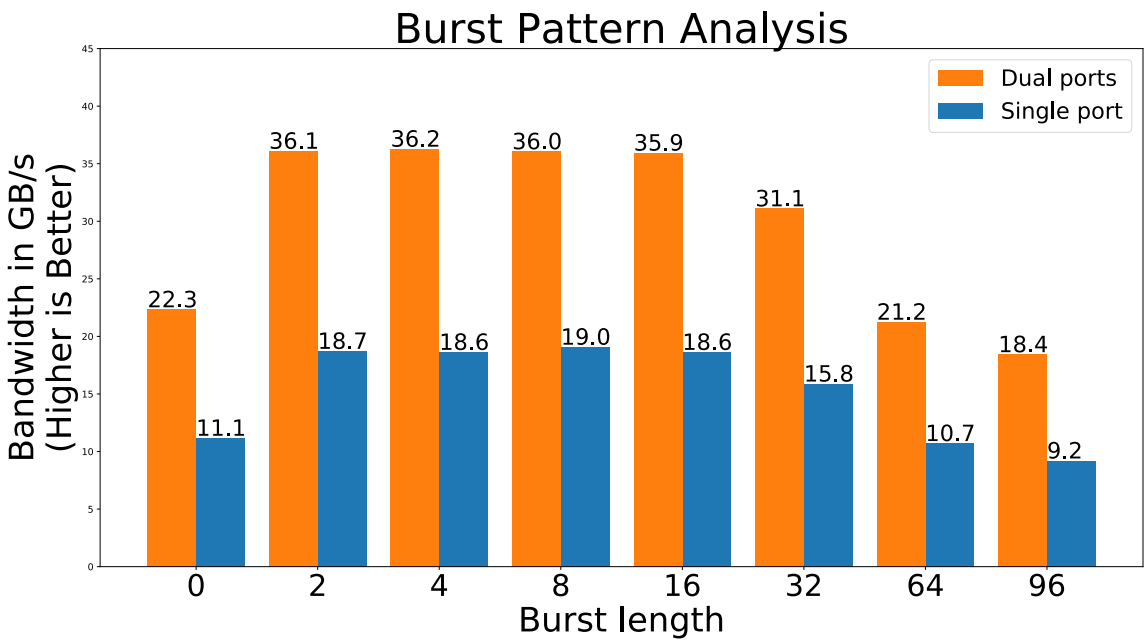

Figure 3.5: HMC bandwidth analysis for varying burst length on a single and dual channel AXI interface

is verified against the write data in the FIFO.

Figure 3.5 shows the bandwidth measured in GB/s for both single and dual AXI ports. Each AXI port has 512 bit data width and can be regarded as using 4 parallel native channels. This interface detail is hidden from the user and the conversion of AXI to native channel occurs inside the vendors memory controller. This AXI interface supports a burst length up to 255. But, due to firmware errors (at vendors memory controller) the bandwidth could only be measured for burst length up to 96. A burst length of 0 is a non-burst case which is similar to sending a single FLIT on a channel The bandwidth measured for a non-burst case (burst length=0) for single port AXI is 11.1 GB/s and for the dual port is 22.3 GB/s respectively. The peak bandwidth for a single port is 19 GB/s and is attained at burst length of 8. Similarly, the peak bandwidth for a dual port is 36.2 GB/s and is attained at burst length of 4. In both these peak bandwidth cases, the total burst size is $8 \times 512 = 4 \times 2 \times 512 = 4$ Kbit. These numbers are interesting because even though HMC can perform better for large access size, according to this experiment a burst

access to a memory location greater than 4 Kbit (maximum block boundary for a DRAM column) can yield lower performance. It is also important to note that the AXI based memory controller re-orders the read response which may add additional delays. These bandwidth numbers provide us the optimum access size to attain the system level performance in HMC. The peak bandwidth of 19 GB/s at a burst length of 8 is dependent on utilizing the 512 bit data width. But this requires an overhaul in application design which hinders the programmer productivity. This motivates us to design a framework/methodology to improve the performance of HLS cores on off-chip memories that can use wider memory interfaces in next-generation memories.

CHAPTER 4: Design

The design consists of three major sections, the infrastructure details all the hardware platforms and the hardware-software design of each platform, the High-Level Synthesis (HLS) design which details the applications and the HLS optimization, and finally the Volcan methodology.

4.1 SoC platforms

The Volcan design is implemented on four different hardware platforms. Each platform differ in the FPGA, the memory, and the high level design flow. Of the four hardware platforms, two of the platforms are System-On-a-Chip (SoC) and two are FPGA platform. The SoC platforms contains a hard core ARM processor(s) with a FPGA. In case of FPGA platform we implement a soft core processor that run on FPGA. The off-chip memories in these platforms are DDR3, DDR4, and HMC. Two platforms have DDR3 memory but they differ in their configuration. In terms of design flow, the SoC platforms uses SDSoC design flow and the FPGA platforms are designed with the combination of HLS and traditional FPGA design flow. As described in the Chapter 2, the SDSoC is a complete software design flow to program the SoC system where as the FPGA systems requires both software and hardware design flow. The flow of this section illustrate the increasing complexity in the design flow and the required infrastructure that is necessary to build the system.

4.1.1 Infrastructure

The two SoC platforms in our design are the Xilinx Zynq ZC706 SoC [120] and Xilinx UltraZynq ZCU102 MPSoC [121]. These platforms differ in the hard core processor, the FPGA, and the memory as shown in the Table 4.1. The ZC706 has a

Table 4.1: SoC platform specification

Resource	**Component**	**Zynq 706**	**ZCU102**
Hardware	Part number	XC7Z045 FFG900-2	XCZU9EG-2FFVB1156
	Processor	Cortex A9	Cortex A53
	Proc. Speed	800 MHz	1200MHz
FPGA	Logic Cells(K)	350	600
	Block RAM(Mb)	19.1	32.1
	DSP Slices	900	2,520
	I/O Pins	362	328
	Transceiver	16	16
	Frequency(MHz)	142.85	150
Memory	Memory Type	DDR3	DDR4
	Frequency (MHz)	533	1066
	Capacity(GB)	1	4
	Interconnect	SODIMM	SODIMM
	Bandwidth(GB/s)	5.3	17
	DMA width(bit)	32	64
	AXI BW(GB/s)	2	5.3

single core ARM A9 processor and the ZCU102 has a dual core ARM A53 processor. Additionally, the ZCU102 has a real time ARM processor R5 which is not used in our design. In terms of FPGA resources, the ZCU102 has a larger FPGA with more logic cell, BRAM memory, and high speed transceivers. Although, the ZCU102 uses a better silicon technology to implement these resources, both the Zynq platform have a comparable operating frequencies (1412.85 and 150 MHz). The off-chip memories are the most distinguishing factors in these platforms. The ZC706 uses the previous generation DDR3 at a frequency of 533 MHz and the ZCU102 uses DDR4 operating at 1066 MHz. The DDR4 has 4× the capacity of the DDR3 while both are interfaced using a SODIMM interface. In terms of logical interface speed, the ZCU102 uses a 64 bit DMA for data transfer and the single AXI bandwidth is 2× better than the ZC706. Clearly, in terms of memory and interconnect the ZCU102 is $2-3\times$ better than the ZC706 platform.

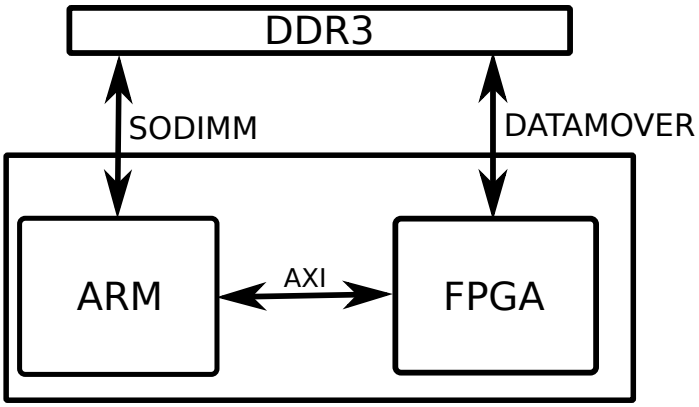

Figure 4.1: Zynq Architecture for SDSoC

4.1.2 Design Flow

The SoC architecture as shown in the Figure 4.1 consists of Processing System (PS), Programmable Logic (PL), and DDR memory. The PS system is physically connected to the DDR memory through SODIMM slot and is logically connected to PL through a AXI bus interface. Software Defined SoC (SDSoC) flow provides a variety of components to move the data between the PS and PL system. Depending on the data pattern the SDSoC tool assists these components which are known as *data movers*. The Table 4.2 lists all these components and their properties. As seen here, the *data movers* implements a SDSoC side and a Vivado side component. The SDSoC side components are called using software APIs and the equivalent Vivado side component is implemented in HDL (verilog or VHDL) by SDSoC tool. Since, our design targets large data-set, we use *zero_ copy data mover* in our design. The syntax of this data mover is given in Listing 4.1 below. In this example, the HLS core access `inp1` and `res` data of size 512×512 directly from the memory.

```
1 #define SZ (512*512)
2 #pragma SDS data zero_copy(inp1[0:SZ],res[0:SZ])
```

Listing 4.1: *zero_ copy* data mover

Table 4.2: Different Data Movers in SDSoC

SDSoC	Vivado	Accelerated IP	Property
axi_lite	processing_system7	register, axilite	Non-Contiguous
axi_dma_simple	axi_dma	bram, fifo, axis	Transfer size <32MB
axi_dma_sg	axi_dma	bram, fifo, axis	
axi_fifo	axi_fifo_mm_s	bram, fifo, axis	Transfer size < 300 B
zero_copy	accelerator IP	aximm master	Contiguous

The design for ZC706 is developed in Xilinx SDSoC 2017.2 tool [122] and for ZCU102 using Xilinx SDSoC 2018.3. The SDSoC design flow as shown in Figure 4.2 uses high level language such as C to build a system on a chip design. The SDSoC provides a pre-built hardware base platforms for both the SoC platforms. In this design flow, we define the HLS core design in C, the main application running on ARM, and the data movers to connect the HLS core with ARM. The design steps to define these are as follows:

1. Defining the main program: A software program in C is designed which has the `main` function. This function is the the primary entry for execution when SoC boots up. In the main function the HLS core is executed using a function call.

2. Defining HLS core: The accelerator application which is implemented on FPGA is designed using C.

3. Defining the data mover: Based on the data requirement of the accelerator function, appropriate data mover is selected in a C header file. This header file also includes the prototype of the HLS core function.

4. Functionality Test: This verifies if the hardware and software are working in synchronous i.e. main function can call the HLS core function.

5. Baseline for HLS: The HLS tools are used to analyze the function and identify the cycle budget and the performance of the HLS core. This time provides an

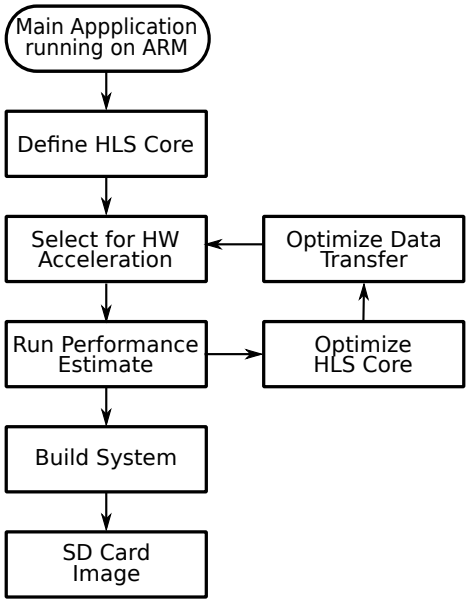

Figure 4.2: SDSoC Design Flow

estimate of HLS core execution time with an ideal memory (zero latency and infinite bandwidth).

6. HLS core optimization: Using the data gathered from the baseline, appropriate optimization pragma is introduced and the performance is measure.

7. Resource Utilization: With the optimized HLS core the resource utilization is measured. This provides an estimate of the HLS core's FPGA resource utilization.

8. Optimize Interface: Based on the memory pattern of the HLS core, the interface that connect the FPGA and memory is optimized. The Data Motion network is use to analyze the performance of the interface.

9. Add Monitors: The performance monitors are added to the design.

The above steps leads to the creation of SD card image which includes the Linux kernel and the necessary device drivers. This SD card image is used to boot the device and the appropriate application is executed from the shell.

Table 4.3: FPGA platform specification

Resource	**Component**	**Kintex 705**	**AC510**
Hardware	Part number	XC7K325T-2FFG900C	XCKU060-FFVA1156
FPGA	Logic Cells(K)	326	726
	Block RAM(Mb)	16.0	38.0
	DSP Slices	840	2,760
	I/O Pins	500	520
	Transceiver	16	32
	Frequency(MHz)	150	187.5
Memory	Memory Type	DDR3	HMC
	Frequency (MHz)	800	187.5
	Capacity(GB)	1	4
	Interconnect	SODIMM	HSSI
	Bandwidth(GB/s)	12.8	60
	AXI BW(GB/s)	12	12

4.2 FPGA platforms

The two FPGA platforms that are considered are Xilinx Kintex 705 development board [123] and Microns AC510 HMC Module [119]. For infrastructure perspective, these FPGA platforms do not have a hard core processors unlike the Zynq boards that are discussed earlier. The Table 4.3 summarizes the specification of the FPGA platforms. As it is noticeable, the FPGA resources are similar to the Zynq platforms with similar AXI interconnects. However, the AC510 uses a HMC memory which is connected by High Speed Serial Links (HSSI) to the FPGA. The other difference with the AC510 module is that it is not a standalone board. The AC510 module which consists of a Xilinx Kintex Ultrascale FPGA [124] and HMC memory is hosted on a EX700 [125] backplane. This backplane is connected to a Pico SC-6 mini [126] Linux system with a 8 GB/s PCIe $\times$16 Gen3 bus. Due to this infrastructure differences the AC510 module requires additional support in the hardware design. The FPGA platforms are not supported by SDSoC design flow and we require both hardware and HLS core design. The HLS core is designed using Vivado HLS 2018.3 design suite [92] and the hardware is designed using the

traditional FPGA design flow. We initially build the hardware design for the KC705 board. As we discussed earlier, the AC510 board requires additional hardware which we build upon the the KC705 design.

4.2.1 Kintex FPGA

The hardware design for Kintex FPGA is as shown in the Figure 4.3 which integrates the HLS core with the memory. The design includes three components for the integration, a) HPro module which orchestrates the entire process b) Smart Interconnect IP from Xilinx c) Memory controller which are connected through a AXI Interconnect.

The primary role of HPro module is to initiate the HLS core module, check the correctness of the HLS core, and gather the result. In case of KC705 FPGA, this module takes the input from JTAG debugger to start the process. Internally, the HPro module consists of Microblaze processor and application profilers. The Microblaze processor runs the software version of the HLS core and compares the results to verify the HLS core correctness. The application profilers consists of hardware timers that measures the runtime of the HLS core. More details of HPro module is provided when we discuss this design for AC510 module.

The Smart Interconnect is a replacement of AXI Interconnect IP from Xilinx. This IP can connect multiple memory mapped AXI master devices with multiple memory mapped AXI slave device. The main advantage of using this IP is that it can pack multi-beat burst from a smaller data-width interface to fill a larger data-width interface. For example, if the incoming interface is 32 bit data-width and the outgoing interface is 512 bit data width, then a multi-burst on 32 bit interface would be filled into a single transaction in a 512 bit interface. This reduces the number of memory transaction and minimizes the latency.

The memory controller for DDR IP is configured using Memory Interface Generator (MIG) from Xilinx. This involves a series of steps that are listed below:

1. Create design by selecting the number of controller as 1
2. Pin options: Select the part number as XC7K325I-FFG900
3. Memory selection: Select DDR3-SDRAM as the memory
4. Controller options
    (a) Clock period: 800 MHz
    (b) Memory type: SODIMM
    (c) Memory part: MT8JTF12864HZ-1G6
    (d) Number of bank: 4
    (e) Data ordering: Normal
5. AXI parameter options
    (a) Data width: 512
    (b) Arbitration scheme: Read Priority
    (c) Narrow bus support: Disable
    (d) Address width: 30 bits
6. Memory options
    (a) Input clock: 150 MHz
    (b) Read burst type: Sequential
    (c) Output impedance control: RZQ/7
    (d) Memory address map: Bank-Row-Column
7. Pin selection: Fixed pin out
8. Generate the memory controller

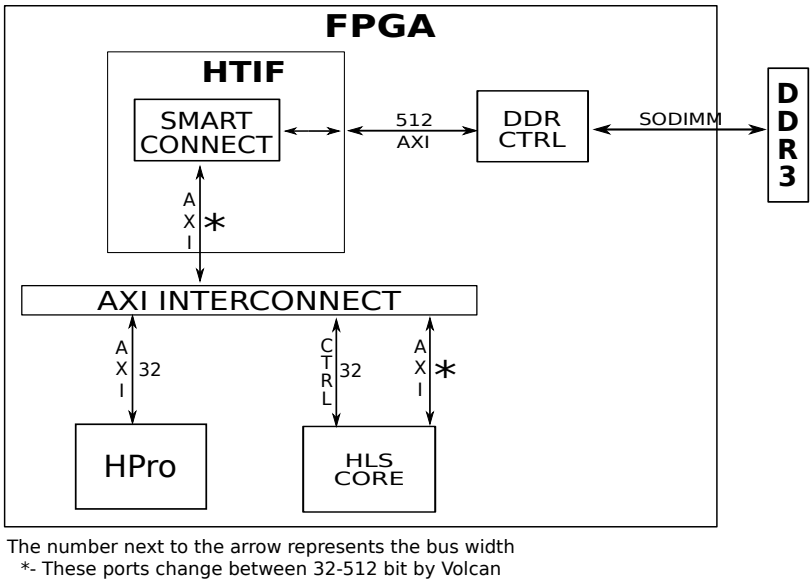

Figure 4.3: Volcan design on Kintex

4.2.2 AC510 HMC

The Volcan design for AC510 consists of a hardware-software system design. The hardware design as shown in the Figure 4.4 consists of three major modules: Hardware Translation Interface (HTIF), HPro core, and the HMC memory controller. These modules are interconnected using AXI4 bus interconnect. This hardware design interfaces to the memory using Micron HMC memory controller. The hardware design (including the accelerators) runs at a frequency of 187.5 MHz. This frequency is based on the HMC controller clock rate which is calculated as $Lanes \times Link\ Speed/Data\ per\ clock\ cycle$[1]. $= 8 * 15\ Gbps/640 = 187.5\ MHz$
The HMC memory controller offers two interface variants a) 10 channels of 128 bit native interconnect (these are different from the 8 physical links) b) 2 channels of 512 bit AXI interconnect (and additional 2 channels of native interface). Since, we are interested in monolithic applications that are generated using HLS (non-parallel) and their performance improvement, the Volcan design is connected to one of two channels of 512 bit AXI interconnect. The FPGA and the external

[1]640 bits of data is sent by HMC controller to a lane on every clock cycle

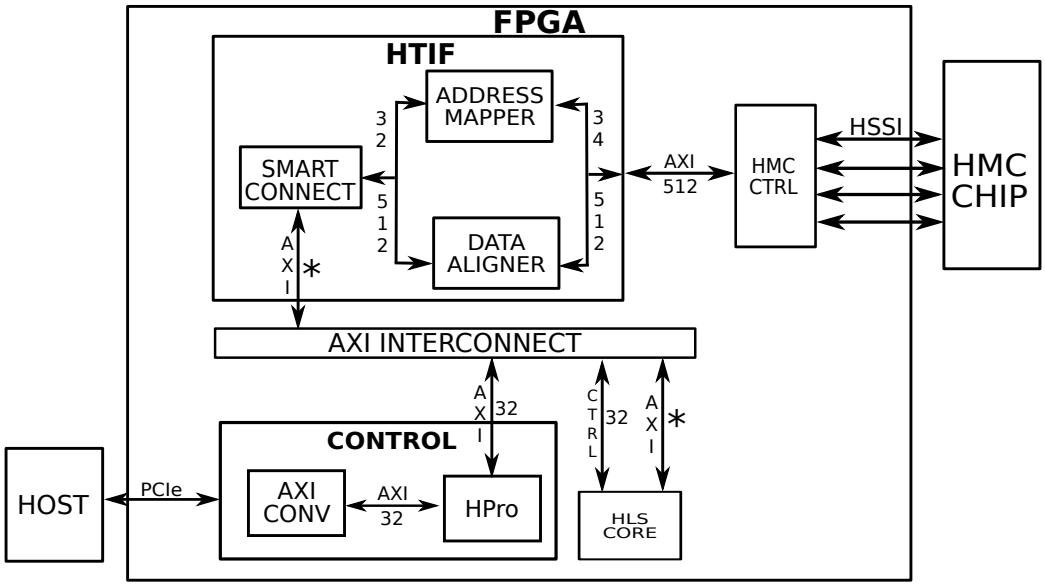

Figure 4.4: Volcan Design on AC510

HMC memory is physically connected using 8 lanes of half-width 15G (187.5 MHz) High Speed Serial Interface (HSSI) links. The entire hardware of FPGA and HMC memory (AC510 board) [119] is connected to the x86 Linux system (referred as 'host') via PCIe. A vendor framework, *Picoframework* provides the software driver which runs on host machine and a hardware interface (*PicoBus* and *Picostream*) which is implemented on FPGA. The Volcan design connects to the host machine using this *Picoframework* hardware interface (both *PicoBus* and *Picostream*). Apart from the hardware design, the Volcan also includes a software methodology. This methodology is applied on the HLS core for better performance.

4.2.2.1 Hardware Translation Interface (HTIF)

The structure of HMC described in Chapter 2 is addressed using a 34 bit memory address. We described earlier that HMC mandates the minimum data granularity to FLIT (16B). This is implicated in the address by ignoring the lower 4 bits of the 34 bit address. But, from a HLS compiler perspective the data is accessed in a byte addressable manner with a 32 bit address. This mismatch between the HMC requirement and compiler assumption leads to *compatibility issue*. With HTIF

design we introduce *address mapper* to convert the incompatible memory address from the HLS core to a compatible HMC memory address. The address mapper generates a 34 bit address with zeros in the last 4 LSB (complying to FLIT) and 3 MSB bits (which are also ignored by HMC). Then, ignoring the 2 MSB bits of 32 bit HLS core address (these are used for memory mapped I/O for FPGA design) it concatenates the 27 LSB bits into the 34 bit HMC address. This entire operation is done using combinational logic to avoid any additional delays and signal synchronization errors. The overall operation is given below:

$HMCAddress = concat\{3'h0, addr[26:0], 4'h0\}$

In Chapter 3, we conducted a study on HMC interfaces to understand a) what percentage of bandwidth in HMC is utilized with different access pattern b) how does the burst length impact the bandwidth. In these studies, we found that if HMC is accessed with a single FLIT and single channel then only 5% (10× less than max. DDR BW) of maximum bandwidth is attainable. On the other hand, the maximum bandwidth (18GB/s for a single 512 AXI channel) is achieved for a burst length of 8. So, in order to improve the burst transaction we added SmartConnect Core and designed a *data aligner* module in HTIF. This core can detected the burst access and pack 16 of 32 bit data into a single 512 bit data. This 512 bit is greater than a single FLIT and complies with HMC address. But, for a non-burst access this module cannot correctly handle the data. The reason is for a non-burst transfer, the `WSTRB` signal (for a write) is asserted to specify the location of the data in a 512 bit. But, this signal is not supported by HMC memory controller interface. To solve this the *data aligner* is introduced. This module re-aligns the data into a 512 bit channel (without need for `WSTRB`) and synchronizes with the address generator. This re-alignment is not required for non-burst read since the data written is already aligned.

4.2.2.2 HPro Core

The HMC memory requires initialization sequence before accessing it. This is done via C/C++ application that runs on the host machine. Similar way, the HLS core also requires initiation and this is done using a different C application that runs on the Microblaze soft core processor on FPGA. In an integrated system the initialization of HMC should occur prior to HLS initiation (else the system does not run). After the HLS core is initiated, we need to measure the performance and collect the statistics. In an isolated system, the HLS initialization and collection of statistics is done using the JTAG-UART port. But, in a HLS-FPGA-HMC integrated system it requires complex Microblaze Debug Module (MDM) configuration. This is due to the HMC memory controller which uses MDM in a separate design and results in Boundary SCAN placement conflict (Refer MDM Spec [127] Chapter 4). So, the only way to avoid conflicts is to initialize HMC, initiate HLS, and collect results using PicoFramework (PCIe). Further added to the complexity in this integrated system are the three different clock domains. There is the HMC memory clock running at 187.5 MHz used for the HMC memory, Volcan design, and the HLS core, Picostream clock running at 250 MHz and PicoBus clock which is a derived clock of Picostream divided by 62 which is 4 MHz (the frequency is set to minimum by vendor to avoid timing closure issues). Thus, in Volcan we design *HPro Core* which initializes HMC memory using Picoframework, initiates the HLS core using a custom handshake protocol, measures the performance of HLS core using hardware timers and synchronizes different clock domain using FIFO. The HMC initialization application is designed in C/C++ using Picoframework driver (*PicoDrv*). This application runs on the Linux host machine and uses *RunBitFile* command to initialize and flash the FPGA firmware. The details of the initialization is out of scope here and for further information refer to HMC specification (Fig. 8) [78] . Once the initialization is successful the application

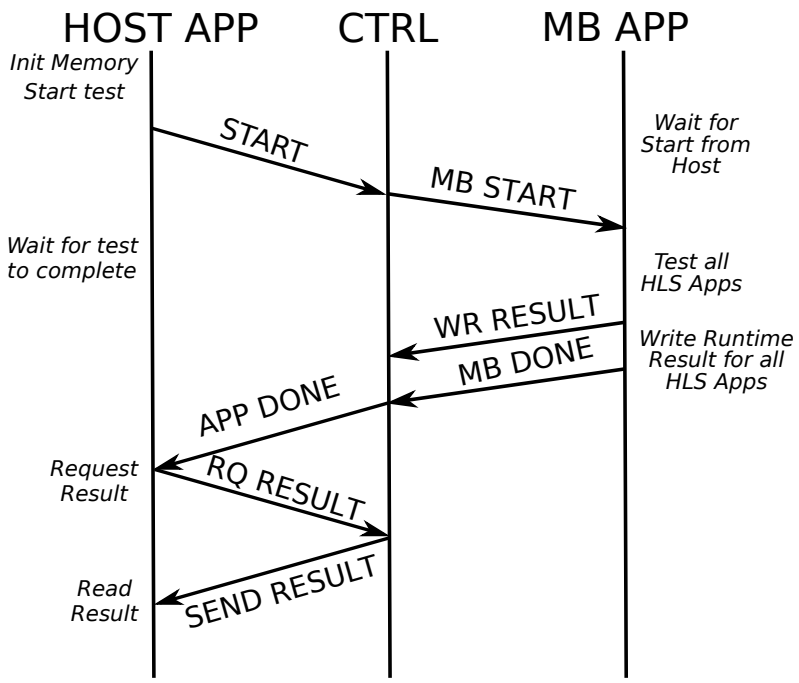

Figure 4.5: Handshake protocol between Host and FPGA

initiates the custom Volcan protocol. As shown in Figure 4.5, the host application sends a `START` signal to the Volcan controller via *PicoBus*. The Volcan controller has two modules: AXI Converter which synchronizes the PicoBus with the AXI interconnect using a cross-clock domain FIFO and Handshake protocol (HPro) module that orchestrates the system. The HPro module consists of a custom verilog module and a Microblaze core. The verilog module sends `MB_START` signal to the Microblaze. A C-application running on Microblaze initiates the HLS core(s) as well as the hardware timers to measure the performance. After completion of HLS core execution, the Microblaze informs the verilog module with a `MB_DONE` signal. The HPro verilog module collects all the statistics and conveys the `DONE` signal to the host application. The host application reads the runtime statistics through *Picostream* interface.

The last piece of the Volcan design is to ensure correctness. The HLS is a reliable methodology in producing accurate results. However, when an application with

variable loop is optimized with loop unrolling techniques it produces a sub optimal Quality of Result (QoR). Thus, it is critical to add a component to check the program correctness in runtime. For this purpose, a software function is developed which executes the same algorithm as the HLS core on Microblaze. This function further ensures the correctness by comparing the results of HLS core and the software application. The statistics of this test and the results are sent back to the host machine along with runtime statistics.

4.3 HLS Design

The application design follows a *memory agnostic* philosophy in which the minutiae of details of underlying memory architecture is hidden from the programmer. The application code written in C/C++ remains exactly the same for different memory architectures. The applications are designed using the Xilinx Vivado suite. The Vivado HLS tool converts these applications into a hardware block referred here as *HLS hardware* (HLS-HW). This HLS-HW is imported to a Vivado hardware project for further synthesis and implementation. These applications are monolithic since the HLS tools currently does not support any parallel cores.

The applications that are consider differ in their loop structures which implies memory pattern. The applications have single perfect loop, nested perfect loop, and nested imperfect loop. Since we are interested in the impact on external memory, the applications are designed to use the external memory as their primary memory storage.

4.3.1 HLS Applications

To evaluate the off-chip memory performance we design seven HLS cores in C/C++ and generate RTL using HLS compiler. These seven HLS cores differ in their loop structures, data dependencies, and data-type. These cores are designed such that there is a clear demarcation of read-compute and writes in the code structure. This

way, it is easier to introduce directives for optimization for read and write loops. These read and write loops directly access the data from the off-chip memories. In certain loops, where RAW (read after write) dependencies exists, BRAM buffers are used as temporary storage. The data is stored in BRAM till the dependencies are resolved (typically end of the inner loop) and is transferred to the off-chip memory before the next iteration.

The description of all the implemented HLS cores, their data sizes, loop structures, and read-compute-write sequences is given below.

1. Summation generates $512 \times 512 = 262144$ write request as input and computes the sum by reading $512 \times 512 = 262144$ integers. Both the write and read loops are single perfect loops.

```
for i -> 0 to N
  //Write input

for i -> 0 to N
  //Read input
  //Compute
```

2. Matrix Multiply performs multiplication of 2-dimensional matrices of integers X and Y of dimension 512×512. The Read-Compute is done on the inner loop and the Write is done on the outer loop. This algorithm has a perfect triply nested loops.

```
for i -> 0 to r
  for j -> 0 to c
    for k -> 0 to c
      //Read Input1
      //Read Input2
      //Compute
    //Write Result
```

3. Longest Common Subsequence computes the longest subsequence among two unsigned integer sequences of length 512. This algorithm is implemented using dynamic programming with separate loops for Read-Compute and Write.

```
for i -> 0 to N
  //Read Input1
  for j -> 0 to N
    //Read Input2
    //Compute
  for j -> 0 to N
    //Write Result
```

4. Knapsack problem for a given set of $N = 512$ items with weight and value (unsigned integers), determines the number of each items to include in a collection such that the total weight is less than or equal to the total weight $M = 512$.

```
for i -> 0 to N+1
  //Read Input1
  //Read Input2
  for j-> 0 to M+1
    //Compute
  for j-> 0 to M+1
    //Write Result
```

5. Dijkstra's algorithm (ASSP) finds the shortest path (unsigned integer) between sources/nodes of the graph. This algorithm is modified with an outer loop to find all source ($N = 512$) shortest path (ASSP).

```
for i -> 0 to N
  for j -> 0 to N-1
    for k -> 0 to N
      //Read Input
      //Compute
```

```
6   for j -> 0 to N
7     //Write Result
```

6. LU Decomposition: decomposes a floating-point matrix of size 512×512 into lower and upper triangular matrix. This algorithm has an imperfect nested loops where the iteration index of inner loops depend on the variable i, the iteration of the outer loop.

```
1 for i -> 0 to N
2   for j -> i to M
3     for k -> i to M
4       //Read Input1
5       //Compute
6       //Write Result1
7       //Write Result2
```

7. Cholesky Decomposition is a decomposition of a Hermitian, positive-definite matrix of size $N = 512 \times 512$ into the product of a lower triangular matrix and its conjugate transpose. This algorithm also has an imperfect inner loop and a square root computation for floating-point numbers.

```
1 for i -> 0 to N
2   for j -> 0 to i+1
3     for k -> 0 to j
4       //Read Input
5       //Compute
6       //Write Result
```

4.3.2 HLS Optimizations

The High-Level-Synthesis tool allows optimization using *directives*. As described in Section 2.5, there are tools that can predict the right directives but they all are related to ideal (zero latency) or BRAM memory (known latency). But, there is no

Table 4.4: Directive Combination (DC) and description

DC	Description
N_wN_r	No optimization
P_wP_r	Pipeline write and read
U_wU_r	Loop unroll write and read
P_wU_r	Pipeline write and unroll read
S_wP_r	Pipeline loop split write and pipeline read
S_wU_R	Pipeline loop split write and unroll read
B_wU_r	Pipeline, loop split, block partition write and unroll read

guarantee that every directive can lead to a better performance with off-chip memories. Hence, understanding the effect of these optimization is important. Many different optimizations on different applications can lead to a large set for Design Space Exploration (DSE). To simplify this we have created a custom list of directive combination. We use these combinations of directives (DC) to compare the performance between the ideal memory (no latency) and actual off-chip memory. The Table 4.4 shows the list of DC and their description. These directives follows the convention $writeloopoptimization_w readloopoptimization_r$. The exploration starts with no optimization which is denoted byN_wN_r. This acts as a base case with which different directives (DC) are compared. Results that are worse than the base case indicates the negative impact of optimization. The different directives that follow N_wN_r are the combination of four optimization techniques: Pipeline (P), Unroll (U), Loop split with Pipeline (S), and Block array partitioning with Pipeline(B). The set of directive listed in Table 4.4 can be further expanded for unrolling cases on write loops and pipeline cases for read loops. These cases are excluded because the unrolling the write is counter-productive to burst access and pipelining the read does not improve the performance for wider data interface. The design and optimizations follows certain assumptions and guidelines which are listed below:

1. For every DC the latency of the HLS core module should always meet the

timing. This timing guarantees that the every operation in the HLS core can be executed in the mentioned clock cycle. from the hardware implementation timing closure which depends

2. For loop pipeline, the *Initiation Interval (II)* value cannot be predicted by the HLS tool. We determine the optimal II value by successive approximation. The code is compiled with initial value is calculated based on the operations in the loop and the process is repeated until we find the optimal value.

3. Loop unrolling requires a lot of resource and a complete unroll for a large loop can lead to compilation error. So to avoid this, we do a partial unroll with factor of 16.

4. To maintain uniformity, the factors for loop split and array partition are set to 16.

5. For imperfect loops, the compiler cannot determine the iteration. As a coding guideline we add `LOOP_TRIP_COUNT` pragma for every loop in our design.

4.4 Volcan Methodology

The HTIF module of Volcan solves the address compatibility issue and improves the burst access. But these does not reflect in performance improvement. To improve the performance we introduce *Volcan Software methodology*. This software methodology transforms the source code of HLS core by memory blocking technique. The transformation follows a generic step which is independent of the source code and optimizations. However, we eliminate loop splitting and array partitioning optimization techniques with this methodology. The reason is the these techniques does not help in blocking the data and creates undesirable layers of memory hierarchy with the HLS core. The steps involved is listed below and the transformation is shown in the Figure 4.6.

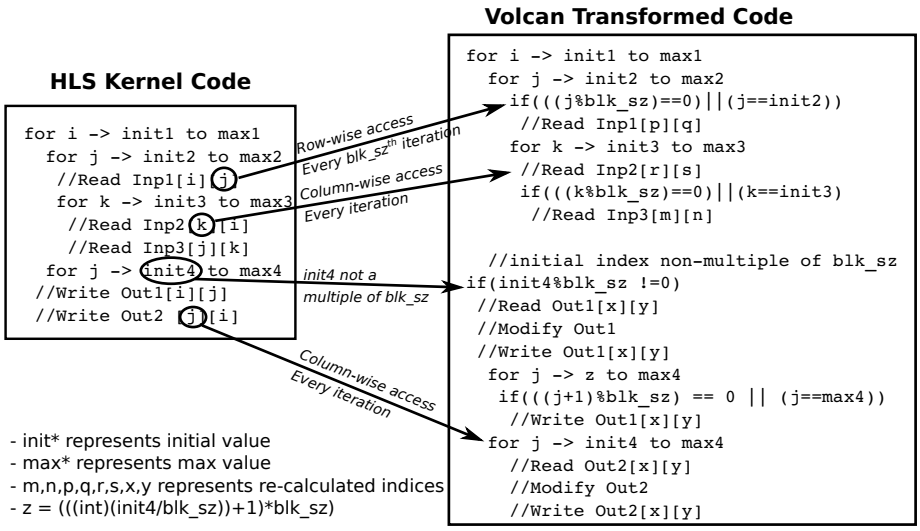

Figure 4.6: Code Transformation using Volcan

1. Interface Transformation: The inputs and outputs of the HLS core is declared as a pointer argument to the top level function. This pointer is a address of a 32 bit data. This is transformed into a 512 bit data using a custom Volcan data structure. The custom data structure consists of an array of HLS core interface data-type (*int, float, double ...*) of block size (`blk_sz` determined by: $512/sizeof(data\_type)$ (512 is the maximum data width of a single AXI interface in HMC controller).

2. Read Transformation: The methodology identifies the input read and optimizes it for a block access. If the input read is accessed row-wise then the access is modified to read 512 bit of data on every $\texttt{blk\_sz}^{\texttt{th}}$ iteration. The loop indices are re-calculated for the block access as: `floor((start and stop)_index/blk_sz)`. For a column-wise read access (which cannot be blocked), the 512 bit is read on every iteration.

3. Compute: The methodology preserves the computation part of the original HLS core. Since, the read data is 512 bit, the transformation extracts the expected 32/64 bit data required for the computation.

4. Write Transformation: The transformation creates two different types of block write access based on the HLS core write access and the iteration index. If the start of iteration is a multiple of `blk_sz` and write access is row-wise then the memory is blocked for every `blk_sz`$^{\texttt{th}}$ iteration and written as a 512 bit data. For a column-wise access or non-multiple of `blk_sz`, the entire 512 bit of memory is read, modified in the interested location, and written back to the memory. In case of non-multiple of `blk_sz`, this read-modify-write is done until the iteration reaches the multiple of `blk_sz` but for column-wise this operation continues till the end of the loop.

CHAPTER 5: Evaluation and Results

We evaluate the design on both SDSoC and FPGA platforms, with and without Volcan methodology. Each platform is tested with seven different HLS core and different set of directives. Each HLS core and directive is evaluated for ideal memory performance (zero latency and infinite bandwidth) which is estimated by the tool and the actual memory performance which is measured by running these HLS cores on FPGAs.

The evaluation starts with the experimental setup which discuss provides the testing parameters. Then we evaluate the HMC memory to understand the baseline bandwidth capabilities. Once we know the raw bandwidth of memories, we evaluate all the HLS core on all four platforms. This evaluation provides us the baseline off-chip performance of the existing FPGA and SoC platforms as well as the behavior of different directives. Using these performance data we evaluate the Volcan methodology in terms of computation and memory impact on both the FPGA platforms. We analyze these improvements of Volcan methodology and compare with the non Volcan performance. Finally, we conclude with a table of best optimization and the best suited platform for each HLS core and the relative performance with respect to running them as software applications.

5.1 Experiment Setup

The experiments are conducted on four different hardware platforms as described in Chapter 4. The SoC platforms are Xilinx ZC706 SoC [120] with an ARM Cortex A9 processor running at 800 MHz and Xilinx Ultrascale+ MPSoC ZCU102 [121] with an ARM A53 processor running at 1200 MHz. These platforms are connected to

Linux host system through a JTAG/UART connection. To conduct the test, the boot images are loaded into a SD card and the SoC platforms are configured for SD card boot. Once the system boots it is interacted through a terminal application on host system.

The FPGA platforms for the experiments are Xilinx Kintex KC705 FPGA [123] and Micron AC510 HMC [119]. Since, the FPGAs do not have a hard core processors like SoCs, the KC705 is designed with a Microblaze soft-processor which runs at 150 MHz. Similar to SoC platforms, the KC705 board is connected to a Linux host system using JTAG/UART. But, instead of SD card boot, it is configured for JTAG boot. The firmware, which includes the hardware design for the FPGA and the Microblaze application is downloaded using JTAG. The Microblaze Debugger Module (MDM) [127] is used as the debugger to interact with the board.

The AC510 platform differs from the rest of the hardware platforms used in this experiment. This module is connected to EX700 [125] and which in turn connect to Pico SC-6 mini [126] Linux system through a PCIe interconnect. As discussed in Chapter 4 Section 4.2.2.2, this FPGA platform cannot accommodate a MDM debugger module. For debugging this platform, we have created HPro Core module in Verilog which can be accessed using the PCIe interconnect. To interact with HPro Core we create a software application written in C and is compiled with a *gcc 5.4* compiler. This application runs on SC-6 mini Linux system and uses PicoAPI [128] functions to interact with AC510. With this application we initialize the HMC, initiate the firmware running on FPGA and gather the results from the FPGA.

5.2 Baseline Analysis

In baseline analysis, we test every HLS core discussed in Chapter 4 with the combination of directives defined in Table 4.4. These results act as a baseline with which can compare the Volcan methodology. The baseline result are presented as runtime measurement for ideal memory and the actual memory. The ideal memory

refers to a memory model where the memory has zero access latency and infinite memory bandwidth.This measurement is captured using SDSoC/HLS tools at synthesis stage of the design. This measurement is an estimation from the tool to measure the performance of generated HLS core with an ideal memory. The actual memory time is the runtime measurement of HLS core when implemented on FPGA. This runtime is measured using hardware timers that are inserted in the FPGA/SoC design. In case of SoC platforms (ZC706 and ZCU102) the hardware timers run at the processor speed (800 MHz and 1200 MHz respectively) and is scaled down to match the speed of the FPGA. For FPGA platforms an AXI based hardware timer is included which runs at FPGA design frequency of 150 MHz.

5.2.1 SoC platform

This experiment evaluates the behavior of HLS cores on off-chip memory with varying memory access pattern and optimization. As previously mentioned, the memory access pattern is not just dependent on the algorithm but also on optimization. The Table 5.1, represents the ideal memory and actual memory runtime for different HLS cores with varying optimizations. For every HLS core, the best runtime for actual and ideal memory is highlighted.

From the Table 5.1 we can notice that there are many different directive than provides the best runtime for an ideal memory. But these directives do not provide the best result for actual memory. The only directive that is consistent for all HLS core is the the pipeline directives ($P_W P_R$). This result is intuitive, since the pipeline optimizes the instructions executed on FPGA and also preserves the burst behavior in an algorithm.

When we analyze each HLS core, for summation it is clear that pipeline ($P_W P_R$) is the only directive which shows performance gain for both the ideal and actual memory. The rest of the directives show positive gain only for the ideal memory case and not for actual memory. This decrease in performance is caused by two

Table 5.1: Execution time (in kilo clock cycle) for Ideal and Actual Memory on SoC platforms (lower the better)

HLS Core	ZC706		ZCU102		DC
	Ideal	Actual	Ideal	Actual	
Sum	786	1,333	786	838	N_wN_r
	524	**1,242**	**524**	**576**	P_wP_r
	786	1,293	786	838	U_wU_r
	704	2,365	671	2,692	P_wU_r
	524	2,859	**524**	2,094	S_wP_r
	704	3,312	671	2,692	S_wU_r
	704	3,638	671	2,693	B_wU_r
M Mul.	1,746,928	9,004,899	1,344,275	6,776,866	N_wN_r
	134,217	2,773,357	**134,217**	**1,884742**	P_wP_r
	572,589	2,543,358	471,925	1,968,112	U_wU_r
	$\star$	$\star$	$\star$	$\star$	P_wU_r
	134,217	2,227,642	**134,217**	1,932,412	S_wP_r
	134,217	2,038,159	**134,217**	1,936,590	S_wU_r
	134,217	**1,911,922**	**134,217**	1,935,510	B_wU_r
LCS	1,314	33,693	1,314	27,168	N_wN_r
	800	**19,241**	800	**13,934**	P_wP_r
	719	25,133	657	27,985	U_wU_r
	562	27,021	**545**	26,373	P_wU_r
	1,047	23,021	800	15,231	S_wP_r
	608	23,283	546	27,701	S_wU_r
	608	25,982	546	27,408	B_wU_r
Knapsack	1,319	20,133	1,319	20,230	N_wN_r
	799	**4,597**	799	**3,709**	P_wP_r
	1,008	23,422	778	21,878	U_wU_r
	565	19,376	**564**	19,646	P_wU_r
	1,061	7,942	922	5,540	S_wP_r
	827	22,766	688	21,474	S_wU_r
	1,086	30,628	1,059	29,352	B_wU_r
Dijkstra	691,356	729,724	673,454	684,562	N_wN_r
	607,662	**658,382**	**673,445**	**684,058**	P_wP_r
	1,509,294	2,125,159	1,424,572	2,136,876	U_wU_r
	1,508,181	2,856,512	1,424,459	2,835,095	P_wU_r
	608,648	684,141	673,447	685,386	S_wP_r
	1,509,167	2,846,480	1,424,461	2,836,365	S_wU_r
	1,517,670	3,169,902	1,424,592	2,836,534	B_wU_r
LU Decom.	7,250,641	6,506,760	7,116,424	6,260,265	N_wN_r
	1,758,462	**1,500,838**	**1,088,422**	**1,975,787**	P_wP_r
	7,250,614	6,478,928	7,074,219	6,217,021	U_wU_r
	2,428,240	3,095,369	1,660,158	2,828,506	P_wU_r
Cholesky	2,987,450	1,467,215	2,696,152	1,397,094	N_wN_r
	1,104,732	**1,144,619**	**820,773**	**756,924**	P_wP_r
	3,180,527	1,463,981	2,775,057	1,403,355	U_wU_r
	2,987,450	1,463,628	2,634,023	1,394,266	P_wU_r

factors. The first factor is the directives such as loop unroll expect parallel access to the memory for unrolled loops. This is can be achieved in an on-chip memories like BRAM with multiple ports. But, the off-chip memories such as DDR have single port. This forces the serialization of memory access resulting in increase of memory access time. This flexibility in on-chip memory is one of the reason for HLS core to perform better with on-chip memory and also map poorly to off-chip memory. The second factor that affects the performance is the result of poor transformation of split loops ($S_W P_R$) by HLS tool. The split loops even after pipelining requires a longer latency to process the data within the loop. In terms of control steps, the split loops requires 83 steps to process 16 integer data. Where as a single loop with pipeline ($P_W P_R$) requires only 48 steps.

Matrix multiplication restricts the optimization to the inner loop for a large data size (512×512). Any optimization to the external loop requires more FPGA resources than available. Thus, the optimization such as $P_W U_R$ cannot be applied to this algorithm. This restriction also results in write access being not efficiently optimize. The inner loop performs the read compute and the write is performed on the exit of the inner loop. This is reflected in similar runtime performance for most of the optimization. Every directive combination can perform better than no optimization ($N_W N_R$). Although the behavior on both platform are similar, the amount of speed-up shown for ideal memory is not even close to the actual memory performance. Additionally, we observed HLS tools picking different datamover for ZC706 compared to ZCU102 for the same directives. For one of the unrolling case ($B_W U_R$) in ZC706, the read of input data used a cache coherent ACP port where as for all the other cases High performance (HP) port was selected. This choice of data-mover was picked by the SDSoC tool since all the cases uses the *zero_copy* data mover in the application source header. This change contributed to some performance gain for loop unroll cases when compared with other platforms.

In case of Longest Common Subsequence (LCS) the best performing directive for ideal memory is $P_W U_R$ but this is one of the worst performing for an actual memory. These cases defeats the purpose of estimation of HLS core performance before implementation. Every loop unroll case shows the memory bottleneck issue due to single port access. Like with the other HLS core the pipeline directive works better with the off-chip memory. Another observation for this algorithm is that the ZCU102 performs at least 40% better than the ZC706 platform. The cache coherence ACP port in ZC706 is a bad choice for this memory pattern and the High performance (HP) port is a better interface choice. This provides us with the information on the instance when ACP port can perform better than HP port and *vice versa*.

The Knapsack problem has a similar loop structure to that of LCS. This is mirrored in the performance behavior. As with other HLS cores the unrolling cases are poorly mapped to off-chip memory and have relatively lower performance. In this case both ZC706 and ZCU102 use the HP port which is reflected in similar performance behavior across both the platforms.

The Dijkstra's ASSP performance shows that most of the directives perform lower than the no optimization. This is true for both the ideal memory as well as the actual memory. This is because this design has input dependencies (6 of them) which needs to be resolved using local memory and then transferring it to off-chip memory in batch. The pipeline directive reduces the this latency to some extent and hence shows some performance gain. This performance shows how the HLS cannot work for some algorithms (with many input dependence) even with optimization.

The LU and the Cholesky decomposition HLS cores have imperfect loops. This restricts from applying the loop splitting technique to these HLS cores. These imperfect loops also results in an estimation error for the ideal memory by the HLS tools. This can be observed from the estimated ideal time being greater than the

actual memory. The ideal memory estimation is performed using the loop iteration data. But, this is calculated using the minimum and maximum iteration index ignoring the variable loop bounds. So, this estimate results in a time as if it was a perfect loop. Additionally, loop unrolling on these variable loop bounds generates a sub-optimal (incorrect) computation results. This is due to static loop unrolling by HLS for a variable bound loops. In terms of actual memory performance which is measured using the hardware timer, the pipeline directives shows performance gains for both ideal and actual memory. This trend is similar to matrix multiplication, LCS, and Knapsack. Similarly, Cholesky decomposition works well only for pipeline ($P_W P_R$) optimization.

In summary, for most applications pipeline directive results in the best optimized HLS core. Algorithms such as matrix multiplication restricts the optimization due to exhaustion of FPGA resources. The Dijkstra ASSP is an anomaly and no optimization works (except pipeline to some extend) due to input dependencies. Finally, variable loop bounds can result in error in ideal memory estimation and also sub-optimal results for unrolled loops. Even though the platform have different memories and one faster than the other, the performances are closely matched.

5.2.2 FPGA platforms

The Table 5.2, shows the performance of FPGA platforms with ideal and actual memory. The ideal memory time for both KC705 and AC510 are similar even though they use different version of HLS tools. As seen with the SoC platform, the pipeline directive ($P_W P_R$) is the best performing directive. The performance measurement of KC705 is similar to the SoC platforms that we saw earlier. Analysis on each HLS core done earlier holds good for the KC705 FPGA platform. However, the AC510 is slower than the SoC and the KC705 platform.

The relative performance gain/loss of HMC over DDR is shown in the Figure 5.1. The positive gains are shown in green and the negative performance is shown in red.

Table 5.2: Execution time (in kilo clock cycle) for Ideal and Actual Memory on SoC platforms (lower the better)

HLS core	KC705		AC510		DC
	Ideal	Actual	Ideal	Actual	
Sum	786	803	786	803	N_wN_r
	524	**540**	**524**	**563**	P_wP_r
	786	1,032	802	9,154	U_wU_r
	671	3,007	688	7,375	P_wU_r
	524	2,311	**524**	5,518	S_wP_r
	671	3,007	688	7,352	S_wU_r
	671	3,007	688	7,359	B_wU_r
M Mul.	1,476,920	8,050,408	1,745,358	19,738,235	N_wN_r
	134,217	2,487,371	**134,217**	**5,359,613**	P_wP_r
	352,846	2,845,237	352,846	9,560,861	U_wU_r
	$\star$	$\star$	$\star$	$\star$	P_wU_r
	134,217	2,481,048	**134,217**	5,359,842	S_wP_r
	134,217	**2,480,904**	**134,217**	5,359,842	S_wU_r
	134,217	2,480,960	**134,217**	5,359,840	B_wU_r
LCS	1,318	30,435	1,314	75,345	N_wN_r
	1,066	**15,618**	1,066	**15,173**	P_wP_r
	661	32,312	650	76,370	U_wU_r
	549	29,718	**549**	74,586	P_wU_r
	1,067	16,618	1,067	16,551	S_wP_r
	550	29,718	550	75,965	S_wU_r
	550	29,718	550	75,964	B_wU_r
Knapsack	1,319	28,706	1,319	57,412	N_wN_r
	799	**5,160**	799	**10,320**	P_wP_r
	778	29,472	778	58,944	U_wU_r
	565	28,314	**565**	56,629	P_wU_r
	923	6,086	923	12,172	S_wP_r
	688	29,240	688	58,481	S_wU_r
	1,059	33,136	1,059	66,272	B_wU_r
Dijkstra	673,447	709,108	673,447	709,108	N_wN_r
	539,492	**576,044**	**539,492**	**576,044**	P_wP_r
	1,432,945	2,055,577	1,432,945	19,303,361	U_wU_r
	1,432,834	2,053,779	1,432,834	19,301,694	P_wU_r
	673,447	710,158	673,447	710,169	S_wP_r
	1,432,834	2,055,187	1,432,834	19,302,995	S_wU_r
	1,432,965	2,055,309	1,432,965	19,303,144	B_wU_r
LU Decom.	7,384,859	6,633,027	7,222,090	14,771,532	N_wN_r
	1,088,947	**1,991,349**	**1,084,204**	**4,235,552**	P_wP_r
	8,185,709	6,523,045	7,198,020	14,678,756	U_wU_r
	1,590,397	2,650,576	1,791,656	5,178,869	P_wU_r
Cholesky	3,233,547	2,427,950	3,099,067	3,621,890	N_wN_r
	1,490,813	**2,235,974**	**1,490,813**	**3,354,731**	P_wP_r
	3,187,672	2,375,200	3,194,225	3,622,561	U_wU_r
	2,987,894	2,357,121	2,918,974	3,356,182	P_wU_r

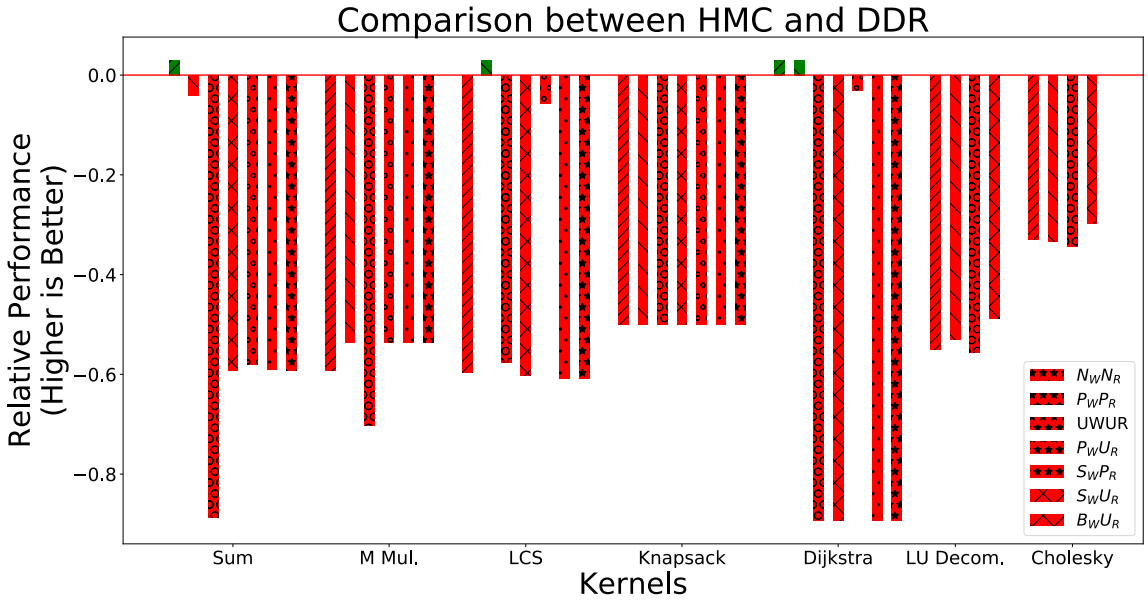

Figure 5.1: Performance comparison between HMC and DDR3 memory

As seen, for most HLS core and directives the performance with HMC is lower by 50% compared to DDR. In the Chapter 3, we saw that the HMC bandwidth is $4 - 10\times$ lower than the peak bandwidth for an non burst access. We also saw that the HLS directives such as loop unrolling modifies the burst memory pattern into non-burst pattern. The combine affect of this causes the HMC to perform $2 - 3\times$ slower than the DDR. This indicates that the design under-utilizes the HMC memory bandwidth. This behavior was prognosticated in the design, since HTIF can completely utilize the 512 bit interface for burst access only. For rest of the memory access it uses $(32/512) = 1/16^{th}$ of the maximum memory bandwidth.

5.3 Volcan Evaluation

In this evaluation, the HLS cores are transformed using the Volcan methodology and is tested on FPGA platforms KC705 and AC510. The performance measurements are compared with the baseline performance (referred as 'HLS Generated') that is presented in Table 5.1 5.2. The Volcan measurement includes the total runtime which is presented as the computation time and the memory overhead. The

computation time is measured exactly same as the ideal memory measurement that was conducted earlier. This represents the amount of time required by the HLS core for computation assuming an ideal memory. The memory overhead time is measured using the hardware timers and is difference between the actual memory time and the ideal memory time that was presented in previous section. These measurement can clearly indicate the impact of Volcan methodology on computing and memory.

5.3.1 DDR Memory

The Table 5.3, represents the computation and memory time for HLS generated and Volcan HLS cores. The HLS generated runtime is same as ideal memory runtime presented in Table 5.2. The difference is that in baseline analysis the focus is on the total runtime performance and compared with different directives. Where as here we are measuring it as computation and memory overhead time with and without Volcan methodology.

The Table 5.3 clearly indicates that the Volcan methodology improves the performance on 24 out of 27 tested cases. However, these improvements are dependent on the algorithm and their memory pattern. The methodology employs memory blocking techniques to improve the memory performance. In most cases, the time spent on memory is significantly reduced by $2 - 14\times$ which results in overall improvement. The baseline experiments concluded with pipeline as the best optimization and loop unrolling fails due to memory bottleneck. One of the goal of Volcan is to be adaptable for different directive requirements. So, in case of loop unrolling it provides a parallel access to the unrolled loops (up to factor 16) and then map it to the wide data-width of the memory interface. This can be regraded as a multi-port access from HLS core perspective to a single port wide off-chip memory interface.This technique has eliminated the bottleneck and improve the memory performance as it can be seen from the Table 5.3. For example, in case of Knapsack HLS core the loop unrolling performance is improved by $11\times$ and the

Table 5.3: Computation and memory overhead (lower the better) and relative gain (higher the better) by Volcan on KC705 (DDR)

HLS Core	HLS Generated		Volcan		Gain	DC
	Comp.	Mem.	Comp.	Mem.		
Sum	786	17	1,338	38	−1.7×	N_wN_r
	524	16	32	5	14.6×	P_wP_r
	786	246	67	200	3.8×	U_wU_r
	671	2,336	45	303	8.6×	P_wU_r
M Mul.	1,476,920	6,573,488	2,148,533	1,440,721	2.2×	N_wN_r
	134,217	2,353,154	273,154	1,384,906	1.5×	P_wP_r
	352,846	2,492,391	269,480	1,063,780	2.1×	U_wU_r
	⋆	⋆	⋆	⋆	⋆	P_wU_r
LCS	1,318	29,117	3,596	10,810	2.1×	N_wN_r
	1,066	14,552	768	3,118	1.5×	P_wP_r
	661	31,651	3,955	10,700	2.2×	U_wU_r
	549	29,169	3,171	10,537	2.2×	P_wU_r
Knapsack	1,319	27,387	2,108	2,507	6.2×	N_wN_r
	799	4,361	803	1,067	2.8×	P_wP_r
	778	28,694	729	1,906	11.2×	U_wU_r
	565	27,750	705	1,909	10.8×	P_wU_r
Dijkstra	673,447	35,661	1,207,700	387,333	−2.2×	N_wN_r
	539,492	36,552	674,233	320,468	−1.7×	P_wP_r
	1,432,945	622,632	746,212	320,366	1.9×	U_wU_r
	1,432,834	620,945	746,312	320,356	1.9×	P_wU_r
LU Decom.	↑	↑	↑	↑	6.3×	N_wN_r
	↑	↑	↑	↑	2.9×	P_wP_r
	↑	↑	↑	↑	6.8×	U_wU_r
	↑	↑	↑	↑	3.2×	P_wU_r
Cholesky	↑	↑	↑	↑	4.5×	N_wN_r
	↑	↑	↑	↑	4.1×	P_wP_r
	↑	↑	↑	↑	4.6×	U_wU_r
	↑	↑	↑	↑	4.5×	P_wU_r

⋆ - P_wU_r requires more FPGA resources than available
↑ - HLS measurement for imperfect loops are incorrect

memory overhead is one-eleventh of the HLS generated circuit.

The improvement by Volcan does not come without any cost. As seen from the Table 5.3, the methodology introduces additional overhead for computation. This overhead is due to a) extraction of 32 bit data from the 512 bit memory data for computation b) introduction of *if* statements. The data extraction overhead can be mitigated using optimization which can be seen from the table. Where as the *if* statement overhead is due to the poor mapping of decision statements by HLS tools. The only anomaly in Volcan performance is the Dijkstra's algorithm. As seen in our earlier baseline experiment, this HLS core has input dependencies which serializes the read operation. This serialization does not help memory blocking. For Volcan methodology, which relies on blocking of memory, this dependency transforms into a 512 bit data read on every iteration of the loop. Further, it adds some latency to extract the 32 bit required for the computation due to the *if-else* block introduced by Volcan. When measured, this latency is increased to 15 clock cycles with Volcan compared to 4 clock cycles for the native case. Although, these latency impact the $N_W N_R$ and $P_W P_R$ cases, the Volcan methodology improves the performance by 2× for the loop unrolling cases.

The relative gain measured between the Volcan and HLS generated is represented in Figure 5.2. The Volcan methodology improves the performance in 24 out of 27 cases. This is the result of memory blocking technique, that reduces the total memory access latency with fewer transaction and exploits the bandwidth. However, it introduces a computational overhead which can be minimized by optimization. As it can be noted from the Figure 5.2, two of three cases which does not improve are no optimization cases. In this case, the overhead introduced in computation exceeds the improvement in the memory. Overall, the Volcan methodology proves that the performance can be improved and is adaptable to the DDR memory.

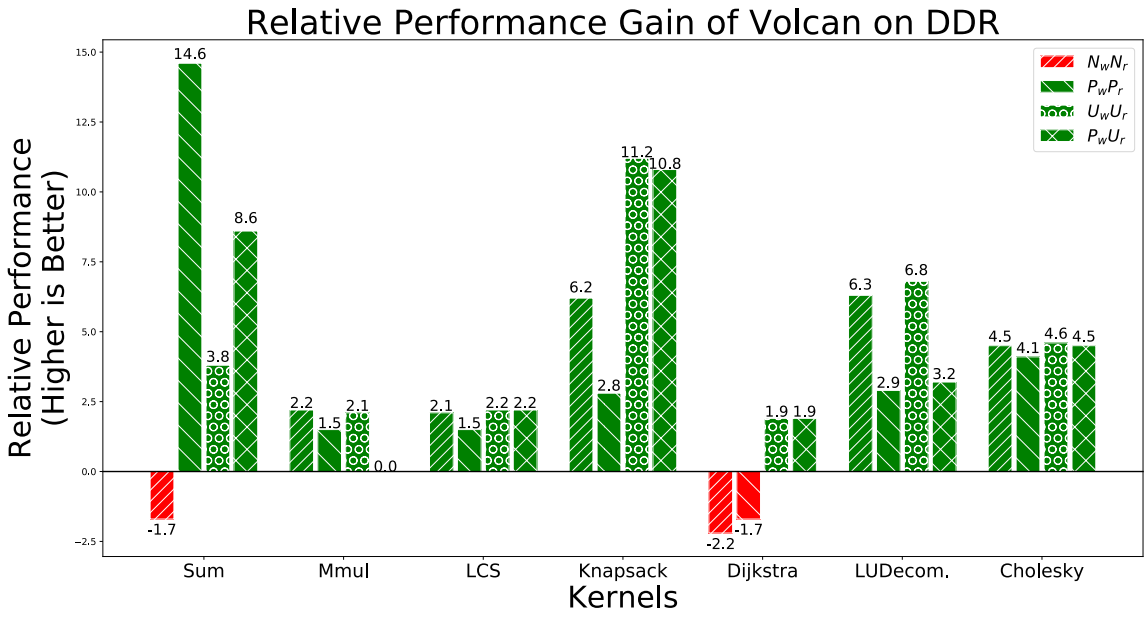

Figure 5.2: Relative gain by applying Volcan methodology on DDR

5.3.2 HMC Memory

The Volcan methodology evaluation similar to DDR is repeated on HMC . As mentioned in Chapter 4, the loop split and array partitioning directives are not applied to this methodology. The Table 5.4, represents the computation and memory overhead of Volcan and non-Volcan methodology over AC510 HMC platform. The performance gain of Volcan compared with non Volcan is given in Figure 5.3.

The performance improvement with HMC is similar to the DDR where 24 out of 27 cases shows improvement of $1.3\times - 16\times$ over same HLS core without the methodology. As seen earlier in the baseline analysis (Figure 5.1), HMC does not perform as expected and the performance is lower than DDR. This is due to the under-utilization of the memory bandwidth and poor performance for non-burst access. The loop unrolling directive which is one of the cause for non-burst access (other being the nature of algorithm) is resolved in Volcan with directive adaptability. The Volcan can recognize these patterns and map the unrolled loops

Table 5.4: Computation and memory overhead in terms of percentage of overall time (lower the better) and relative gain (higher the better) by Volcan on AC510 (HMC)

HLS Core	HLS Generated		Volcan		Gain	DC
	Comp.	Mem.	Comp.	Mem.		
Sum	786	17	851	1	−1.1×	N_wN_r
	524	39	32	3	16×	P_wP_r
	802	8352	65	613	13.5×	U_wU_r
	688	6687	43	418	16×	P_wU_r
M Mul.	1,745,358	17,992,877	1,879,835	2,489,596	4.5×	N_wN_r
	134,217	5,225,396	272,629	1,223,092	3.6×	P_wP_r
	352,846	9,208,015	252,445	3,380,350	2.6×	U_wU_r
	⋆	⋆	⋆	⋆	⋆	P_wU_r
LCS	1,314	74,031	3,334	38,292	1.8×	N_wN_r
	1,066	14,107	769	11,228	1.3×	P_wP_r
	650	75,720	3,431	38,804	1.8×	U_wU_r
	549	74,037	2,909	38,292	1.8×	P_wU_r
Knapsack	1,319	56,093	1,844	4,995	8.4×	N_wN_r
	799	9,521	801	3,500	2.4×	P_wP_r
	778	58,166	727	4,387	11.5×	U_wU_r
	565	56,064	703	4,379	11.1×	P_wU_r
Dijkstra	673,447	35,661	805,571	1,185,068	−2.8×	N_wN_r
	539,492	36,552	673,447	1,116,324	−3.1×	P_wP_r
	1,432,945	17,870,416	562,023	1,117,848	11.5×	U_wU_r
	1,432,834	17,868,860	562,122	1,117,846	11.5×	P_wU_r
LU Decom.	↑	↑	↑	↑	13.6×	N_wN_r
	↑	↑	↑	↑	4.7×	P_wP_r
	↑	↑	↑	↑	14.7×	U_wU_r
	↑	↑	↑	↑	5.1×	P_wU_r
Cholesky	↑	↑	↑	↑	3.6×	N_wN_r
	↑	↑	↑	↑	3.3×	P_wP_r
	↑	↑	↑	↑	3.8×	U_wU_r
	↑	↑	↑	↑	3.5×	P_wU_r

⋆ - P_wU_r requires more FPGA resources than available
↑ - HLS measurement for imperfect loops are incorrect

to access memory in parallel. These parallel access and mapped to the memory interface effectively converting the non-burst transactions. The rest of the performance are similar to the DDR memory as seen earlier.

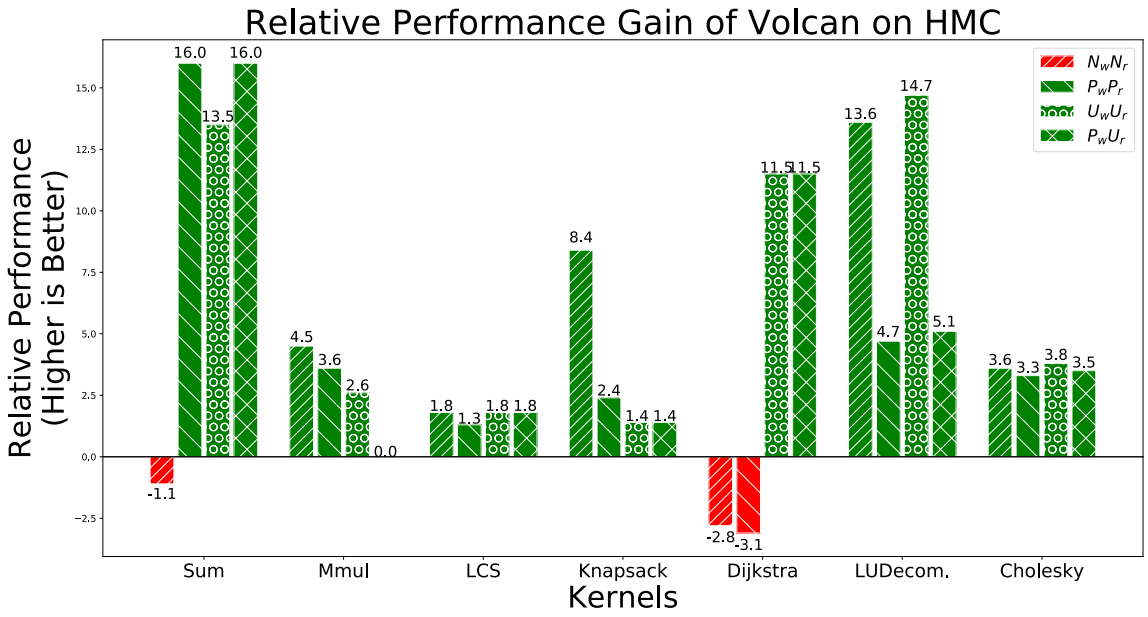

Figure 5.3: Relative gain by applying Volcan methodology on HMC

5.4 Final Evaluation

The goal of this dissertation is to evaluate the HLS technology for HPC applications. This can be addressed by answering the key question *Can Computational scientists' benefit from High Level Synthesis to develop High Performance Computing applications?* The Chapter 4 describes the Volcan design and its memory agnostic approach. The Volcan design integrates HLS core with different platform without changing the HLS core design for every platform. This improves the productivity for using high level languages by abstracting the low level details of architecture and memory. The result from the previous sections identifies the the right combination of directives, platform, and memory for every HLS core. In summary, the pipeline directive and Volcan methodology are the best case scenarios for the existing scenarios. Comparing HMC with DDR, HLS cores such as summation and matrix multiplication perform better with HMC and the rest of the HLS core are better with DDR. These results identifies the right optimization and platform for HPC applications designed using HLS.

Table 5.5: Comparison of execution time (lower the better) between HLS core and software

Application	Execution time (μ s)		Gain
	HLS Core	Software	
Summation	0.23	1.22	5.2×
M. Multiplication	9,971	2,812	−3.5×
LCS	25.90	2.85	−9.0×
Knapsack	12.46	2.16	−5.7×
Dijkstra	6,631	1,925	−3.4×
LU Decom.	4,634	453	−10.2×
Cholesky	3,611	107	−33.7×

To identify the impact of Volcan design, we compare the performance against an ARM A53 processor running at 1200 MHz. The execution time of best performing HLS core and software counterpart (which uses the same source code) is given in Table 5.5. The source code for the application remains exactly the same. In case of HLS, the source code is passed into the HLS compiler and implemented on FPGA. The best case memory is considered for each HLS core. As discussed above, for summation and matrix multiplication the best case memory is HMC and for the rest of the HLS cores the memory is DDR3. For software execution, the source is compiled with ARM compiler and executed on ARM processor. The off-chip memory for software execution is DDR4. As can be seen from the Table 5.5, except for summation the hardware execution time is higher than the software. The data pattern in case of summation is similar to streaming which has a high temporal locality. The rest of the application have random access pattern. Depending on the pattern the hardware is slower by $3.5 - 33\times$ than that of software.

CHAPTER 6: Conclusion

High performance computing (HPC) is moving from homogeneous architectures to heterogeneous architectures. The current generation heterogeneous architectures have a mix of computing elements such as CPUs, GPUs, FPGAs *etc.* and a mix of memory elements such as SRAM (cache), BRAM, DDR, HBM, HMC *etc.*

Though these architectures are efficient for performance and energy, they are more challenging to design for the computational scientist. Among different computing elements FPGAs play a key role in heterogeneous architectures as accelerators and main stream computers. Traditionally, FPGAs are programmed using hardware languages that are concurrent and reactive. High level synthesis (HLS) design allows FPGA implementation in High-level languages (HLL) such as C/C++. HLS supports optimizations which can further improve the performance of the generated hardware. This HLS technology is highly beneficial for embedded applications with small data size. The HPC applications process a large amount of data and hence it has to access data from the off-chip memory. This requires efficient mapping of HLS core for off-chip memory access by abstracting the underlying hardware. This dissertation introduced Volcan, a novel memory agnostic framework that integrates HLS generated HPC applications to different heterogeneous architectures.

To understand the behavior of HLS on off-chip memory, we tested seven HPC applications with seven optimizations on four different FPGA and SoC platforms, which had three different types of memory (DDR3, DDR4, HMC) using two different HLS flow. The test results suggests that by default, HLS poorly maps applications for off-chip memory access. This can be seen from the difference in

performance between ideal and actual memory as shown in Table 5.1 and 5.2. In terms of optimizations, pipeline directives works the best. But, the impact of loop unroll optimization is detrimental and is slower than no optimization. The reason is HLS maps the unrolled loop as a non-burst access to off-chip memory. Unlike BRAM the off-chip memories have a single channel interface. But, HLS assumption of multi-channel interface results in memory bottleneck which impacts the performance.

The Volcan design tries to improve the existing HLS infrastructure with hardware and software design. By analyzing low level hardware details of each architecture the necessary infrastructure is created. This establishes a a hardware abstraction and memory agnostic interface. Specifically, the compatibility issues such as incompatible address scheme in HMC is resolved by designing a Hardware Translation Interface (HTIF) module. Similarly the control issues are resolved by designing HPro module which implements a handshake protocol to establish connectivity. By building these infrastructure, the HLS design can be integrated to different FPGAs and different memory architectures. However, these low-level infrastructure does not improve the performance. To improve the performance of HLS cores with off-chip memories, a software transformation is introduced.

The Volcan software transformation known as 'Volcan Methodology' improves the performance by memory blocking technique. This involves analyzing the data dependencies of the HLS core and the optimization directives. The address analysis experiments showed that the HLS optimization can change the access pattern in an algorithm. From the experiment we noticed that directives such as pipeline and loop split exhibits the burst behavior. But with unroll and array partition directives the burst behavior is changed to non-burst access. These non-burst access can heavily impact the performance of HLS core on off-chip memories. As we saw from the burst analysis study, the non-burst bandwidth is $4 - 10\times$ lower than the peak

bandwidth. The software methodology address the performance issues by transforming the HLS core to access wider memory interface. This wider interface not only utilizes the memory bandwidth efficiently but transforms a single channel memory into a multiple parallel ports to benefit loop unrolling. The result shows that Volcan improves the performance in both DDR and HMC for 24 out of 27 test cases. The gain is dependent on the algorithm memory pattern and can range from $1.3 - 16\times$ compared to the default HLS infrastructure.

While Volcan improves the performance for the existing infrastructure, it does not outperform the software application as shown in the Table 5.5. The HLS tools can provide performance benefits for embedded like and streaming applications (as seen in summation application). But they poorly map HPC applications which have random access pattern to the off-chip memory. The problem lies in the underlying implementation of the tools which initially compiles the HLS source code as a software application (these are based on GCC and LLVM). This compilation technique does not reflect the hardware behavior which leads to poor implementation. The other important issue is that the hardware devices such as FPGAs are massively parallel. But, the current HLS tools does not support parallel application. This compromises the user to generate monolithic applications and use wider interfaces for performance. But, the utilization of wider interface is limited by the memory pattern of the application. Depending on the application the performance can deteriorate from $3\times - 33\times$ as shown in Table 5.5. In conclusion, the current HLS cannot be used for HPC applications. Changing the fundamentals of compilation technique would be key for future architectures and applications.

REFERENCES

[1] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, July 2008.

[2] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, pp. 1–12.

[3] M. H. Kryder and C. S. Kim, "After hard drives—what comes next?" *IEEE Transactions on Magnetics*, vol. 45, no. 10, pp. 3406–3413, Oct 2009.

[4] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sep. 2006.

[5] J. M. Shalf and R. Leland, "Computing beyond moore's law," *Computer*, vol. 48, no. 12, pp. 14–23, Dec 2015.

[6] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[7] J. Kahle, "The cell processor architecture," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 3–. [Online]. Available: https://doi.org/10.1109/MICRO.2005.33

[8] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[9] T. .-T. list, "June 2019," 2019. [Online]. Available: https://www.top500.org/lists/2019/06/

[10] T. G. 500, "June 2019," 2019. [Online]. Available: https://www.top500.org/green500/lists/2019/06/

[11] R. H. Dennard, "Evolution of the mosfet dynamic ram-a personal view," *IEEE Transactions on Electron Devices*, vol. 31, no. 11, pp. 1549–1555, Nov 1984.

[12] H. . Park, S. . Yang, M. . Jung, T. . Kang, S. . Kim, K. . Sohn, D. . Bae, S. . Kim, K. . Kim, B. . Sohn, H. . Kim, H. . Byun, Y. . Shin, and H. . Lim, "A 833 mb/s 2.5 v 4 mb double data rate sram," in *1998 IEEE International Solid-State Circuits Conference. Digest of Technical Papers, ISSCC. First Edition (Cat. No.98CH36156)*, Feb 1998, pp. 356–357.

[13] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. . Wang, "Heterogeneous computing: challenges and opportunities," *Computer*, vol. 26, no. 6, pp. 18–27, June 1993.

[14] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, July 2008.

[15] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[16] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 11, no. 4, pp. 44–54, Winter 1994.

[17] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, March 2007.

[18] P. A. Jackson, B. L. Hutchings, and J. L. Tripp, "Simulation and synthesis of csp-based interprocess communication," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, April 2003, pp. 218–227.

[19] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers, "Chimps: A c-level compilation flow for hybrid cpu-fpga architectures," in *2008 International Conference on Field Programmable Logic and Applications*, Sept 2008, pp. 173–178.

[20] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[21] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[22] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[23] E. P. Jr., "Lebombo bone," 2019. [Online]. Available: http://mathworld.wolfram.com/LebomboBone.html

[24] D. Anderson and J. Delve, "Biographies [f.c. williams; j. vaucanson; j.m. jacquard]," *IEEE Annals of the History of Computing*, vol. 29, no. 4, pp. 90–102, Oct 2007.

[25] A. G. Bromley, "The evolution of babbage's calculating engines," *IEEE Ann. Hist. Comput.*, vol. 9, no. 2, pp. 113–136, Jun. 1987. [Online]. Available: http://dx.doi.org/10.1109/MAHC.1987.10013

[26] F. Michael, *Experimental researches in electricity... reprinted from the Philosophical Transactions of 1831-1852 (with other electrical papers).* R. & J.E. Taylor, 1839.

[27] E. D. Daniel, C. D. Mee, and M. H. Clark, *About the Editors.* IEEE, 1999. [Online]. Available: https://ieeexplore.ieee.org/document/5265664

[28] J. W. Forrester, "Digital information storage in three dimensions using magnetic cores," *Journal of Applied Physics*, vol. 22, no. 1, pp. 44–48, 1951. [Online]. Available: https://doi.org/10.1063/1.1699817

[29] R. R. Everett, "The whirlwind i computer," *Electrical Engineering*, vol. 71, no. 8, pp. 681–686, Aug 1952.

[30] J. A. Rajchman, "Ferrite apertured plate for random access memory," *Proceedings of the IRE*, vol. 45, no. 3, pp. 325–334, March 1957.

[31] C. D. Brady, "Apollo guidance and navigation electronics," *IEEE Transactions on Aerospace*, vol. AS-3, no. 2, pp. 354–362, June 1965.

[32] W. Proebster, "The design of a high-speed thin-magnetic-film memory," in *1962 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, vol. V, Feb 1962, pp. 38–39.

[33] I. H. Yetter, "High-speed fault simulation for univac 1107 computer system," in *Proceedings of the 1968 23rd ACM National Conference*, ser. ACM '68. New York, NY, USA: ACM, 1968, pp. 265–277. [Online]. Available: http://doi.acm.org/10.1145/800186.810587

[34] W. A. Baker, "The piggyback twistor - an electrically alterable nondestructive-readout twistor memory," *IEEE Transactions on Communication and Electronics*, vol. 83, no. 75, pp. 829–833, Nov 1964.

[35] A. Bobeck, I. Danylchuk, F. Rossol, and W. Strauss, "Evolution of bubble circuits processed by a single mask level," *IEEE Transactions on Magnetics*, vol. 9, no. 3, pp. 474–480, Sep. 1973.

[36] R. B. Mulvany, "Engineering design of a disk storage facility with data modules," *IBM Journal of Research and Development*, vol. 18, no. 6, pp. 489–505, Nov 1974.

[37] G. Hu, J. H. Lee, J. J. Nowak, J. Z. Sun, J. Harms, A. Annunziata, S. Brown, W. Chen, Y. H. Kim, G. Lauer, L. Liu, N. Marchack, S. Murthy, E. J. O'Sullivan, J. H. Park, M. Reuter, R. P. Robertazzi, P. L. Trouilloud, Y. Zhu, and D. C. Worledge, "Stt-mram with double magnetic tunnel junctions," in *2015 IEEE International Electron Devices Meeting (IEDM)*, Dec 2015, pp. 26.3.1–26.3.4.

[38] J. T. Evans and R. I. Suizu, "Static fram: an emerging nonvolatile memory technology," in *Seventh Biennial IEEE International Nonvolatile Memory Technology Conference. Proceedings (Cat. No.98EX141)*, June 1998, pp. 26–.

[39] D. Takashima, "Overview of ferams: Trends and perspectives," in *2011 11th Annual Non-Volatile Memory Technology Symposium Proceeding*, Nov 2011, pp. 1–6.

[40] X. Yin, X. Chen, M. Niemier, and X. S. Hu, "Ferroelectric fets-based nonvolatile logic-in-memory circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 1, pp. 159–172, Jan 2019.

[41] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[42] H. . P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec 2010.

[43] A. J. Annunziata, M. C. Gaidis, L. Thomas, C. W. Chien, C. C. Hung, P. Chevalier, E. J. O'Sullivan, J. P. Hummel, E. A. Joseph, Y. Zhu, T. Topuria, E. Delenia, P. M. Rice, S. S. P. Parkin, and W. J. Gallagher, "Racetrack memory cell array with integrated magnetic tunnel junction readout," in *2011 International Electron Devices Meeting*, Dec 2011, pp. 24.3.1–24.3.4.

[44] E. Eleftheriou, T. Antonakopoulos, G. K. Binnig, G. Cherubini, M. Despont, A. Dholakia, U. Durig, M. A. Lantz, H. Pozidis, H. E. Rothuizen, and P. Vettiger, "Millipede - a mems-based scanning-probe data-storage system," *IEEE Transactions on Magnetics*, vol. 39, no. 2, pp. 938–945, March 2003.

[45] J. R. Jameson, P. Blanchard, C. Cheng, J. Dinh, A. Gallo, V. Gopalakrishnan, C. Gopalan, B. Guichet, S. Hsu, D. Kamalanathan, D. Kim, F. Koushan, M. Kwan, K. Law, D. Lewis, Y. Ma, V. McCaffrey, S. Park, S. Puthenthermadam, E. Runnion, J. Sanchez, J. Shields, K. Tsai, A. Tysdal, D. Wang, R. Williams, M. N. Kozicki, J. Wang, V. Gopinath, S. Hollmer, and

M. Van Buskirk, "Conductive-bridge memory (cbram) with excellent high-temperature retention," in *2013 IEEE International Electron Devices Meeting*, Dec 2013, pp. 30.1.1–30.1.4.

[46] M. Joodaki, *Selected Advances in Nanoelectronic Devices: Logic, Memory and RF*, ser. Lecture Notes in Electrical Engineering. Springer Berlin Heidelberg, 2012. [Online]. Available: https://books.google.com/books?id=rF3yIlMzcxUC

[47] F. C. Williams, T. Kilburn, and G. C. Tootill, "Universal high-speed digital computers: a small-scale experimental machine," *Journal of the Institution of Electrical Engineers*, vol. 1951, no. 3, pp. 99–, March 1951.

[48] J. M. Stinchfield, "Cathode ray tubes and their application," *Transactions of the American Institute of Electrical Engineers*, vol. 53, no. 12, pp. 1608–1615, Dec 1934.

[49] J. P. Eckert, Jr., "A survey of digital computer memory systems," *IEEE Ann. Hist. Comput.*, vol. 20, no. 4, pp. 15–28, Oct. 1998. [Online]. Available: https://doi.org/10.1109/85.728227

[50] J. P. Eckert, "A survey of digital computer memory systems," *Proceedings of the IRE*, vol. 41, no. 10, pp. 1393–1406, Oct 1953.

[51] M. V. Wilkes, "The edsac computer," in *1951 International Workshop on Managing Requirements Knowledge*, Dec 1951, pp. 79–79.

[52] E. Braun and S. MacDonald, *Revolution in Miniature: The History and Impact of Semiconductor Electronics*. Cambridge University Press, 1982. [Online]. Available: https://books.google.com/books?id=03c4wldf-k4C

[53] O. Minato, T. Sasaki, S. Honjo, K. Ishibashi, Y. Sasaki, N. Moriwaki, K. Nishimura, Y. Sakai, S. Meguro, M. Tsunematsu, and T. Masuhara, "A 42ns 1mb cmos sram," in *1987 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, vol. XXX, Feb 1987, pp. 260–261.

[54] K. Noda, K. Matsui, K. Takeda, and N. Nakamura, "A loadless cmos four-transistor sram cell in a 0.18-/spl mu/m logic technology," *IEEE Transactions on Electron Devices*, vol. 48, no. 12, pp. 2851–2855, Dec 2001.

[55] B. Copeland, *Colossus: The secrets of Bletchley Park's code-breaking computers*. OUP Oxford, 2006. [Online]. Available: https://books.google.com/books?id=e6ocfloTkJ4C

[56] W. Y. Stevens, "The structure of system/360: Part ii - system implementations," *IBM Syst. J.*, vol. 3, no. 2, pp. 136–143, Jun. 1964. [Online]. Available: http://dx.doi.org/10.1147/sj.32.0136

[57] W. Regitz and J. Karp, "A three transistor-cell, 1024-bit, 500 ns mos ram," in *1970 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, vol. XIII, Feb 1970, pp. 42–43.

[58] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving dram performance by parallelizing refreshes with accesses," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 356–367.

[59] C. Toal, D. Burns, K. McLaughlin, S. Sezer, and S. O'Kane, "An rldram ii implementation of a 10gbps shared packet buffer for network processing," in *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, Aug 2007, pp. 613–618.

[60] M. Cox, N. Bhandari, and M. Shantz, "Multi-level texture caching for 3d graphics hardware," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, July 1998, pp. 86–97.

[61] H. Lee, W. Yun, S. Kang, H. Moon, S. Kwack, D. Lee, K. Kwean, K. Kim, Y. Choi, J. Ahn, and J. Kih, "A low power high performance register-controlled digital dll for 2gbps x32 gddr sdram," in *2005 IEEE Asian Solid-State Circuits Conference*, Nov 2005, pp. 401–404.

[62] H. Ltd., "Hm5283206fp," 2019. [Online]. Available: https://www.hitachi.com/New/cnews/E/1997/970317B.html

[63] M. Yasumoto, H. Hayama, and T. Enomoto, "Promissing new fabrication process developed for stacked lsi's," in *1984 International Electron Devices Meeting*, Dec 1984, pp. 816–819.

[64] S. S. Wong and A. E. Gamal, "The prospect of 3d-ic," in *2009 IEEE Custom Integrated Circuits Conference*, Sep. 2009, pp. 445–448.

[65] K. Bernstein, P. Andry, J. Cann, P. Emma, D. Greenberg, W. Haensch, M. Ignatowski, S. Koester, J. Magerlein, R. Puri, and A. Young, "Interconnects in the third dimension: Design challenges for 3d ics," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007, pp. 562–567.

[66] S. Lu, J. Juang, H. Cheng, Y. Tsai, T. Chen, and W. Chen, "Effects of bonding parameters on the reliability of fine-pitch cu/ni/snag micro-bump chip-to-chip interconnection for three-dimensional chip stacking," *IEEE Transactions on Device and Materials Reliability*, vol. 12, no. 2, pp. 296–305, June 2012.

[67] B. Dang, M. Shapiro, P. Andry, C. Tsang, E. Sprogis, S. Wright, M. Interrante, J. Griffith, V. Truong, L. Guerin, R. Liptak, D. Berger, and J. Knickerbocker, "Three-dimensional chip stack with integrated decoupling capacitors and thru-si via interconnects," *IEEE Electron Device Letters*, vol. 31, no. 12, pp. 1461–1463, Dec 2010.

[68] H. Saito, M. Nakajima, T. Okamoto, Y. Yamada, A. Ohuchi, N. Iguchi, T. Sakamoto, K. Yamaguchi, and M. Mizuno, "A chip-stacked memory for on-chip sram-rich socs and processors," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 1, pp. 15–22, Jan 2010.

[69] P. R. Morrow, C. . Park, S. Ramanathan, M. J. Kobrinsky, and M. Harmes, "Three-dimensional wafer stacking via cu-cu bonding integrated with 65-nm strained-si/low-k cmos technology," *IEEE Electron Device Letters*, vol. 27, no. 5, pp. 335–337, May 2006.

[70] C. S. Premachandran, J. Lau, Ling Xie, Ahmad Khairyanto, K. Chen, Myo Ei Pa Pa, M. Chew, and Won Kyoung Choi, "A novel, wafer-level stacking method for low-chip yield and non-uniform, chip-size wafers for mems and 3d sip applications," in *2008 58th Electronic Components and Technology Conference*, May 2008, pp. 314–318.

[71] N. Maeda, H. Kitada, K. Fujimoto, K. Suzuki, T. Nakamura, A. Kawai, K. Arai, and T. Ohba, "Wafer-on-wafer (wow) stacking with damascene-contact tsv for 3d integration," in *Proceedings of 2010 International Symposium on VLSI Technology, System and Application*, April 2010, pp. 158–159.

[72] I. Radu, B. Nguyen, G. Gaudin, and C. Mazure, "3d monolithic integration: Stacking technology and applications," in *2015 International Conference on IC Design Technology (ICICDT)*, June 2015, pp. 1–3.

[73] P. Batude, M. Vinet, A. Pouydebasque, C. Le Royer, B. Previtali, C. Tabone, J. . Hartmann, L. Sanchez, L. Baud, V. Carron, A. Toffoli, F. Allain, V. Mazzocchi, D. Lafond, S. Deleonibus, and O. Faynot, "3d monolithic integration," in *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, May 2011, pp. 2233–2236.

[74] K. Takahashi, Y. Taguchi, M. Tomisaka, H. Yonemura, M. Hoshino, M. Ueno, Y. Egawa, Y. Nemoto, Y. Yamaji, H. Terao, M. Umemoto, K. Kameyama, A. Suzuki, Y. Okayama, T. Yonezawa, and K. Kondo, "Process integration of 3d chip stack with vertical interconnection," in *2004 Proceedings. 54th Electronic Components and Technology Conference (IEEE Cat. No.04CH37546)*, vol. 1, June 2004, pp. 601–609 Vol.1.

[75] C. A. Bower, D. Malta, D. Temple, J. E. Robinson, P. R. Coffinan, M. R. Skokan, and T. B. Welch, "High density vertical interconnects for 3-d integration of silicon integrated circuits," in *56th Electronic Components and Technology Conference 2006*, May 2006, pp. 5 pp.–.

[76] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 a 1.2v 8gb 8-channel 128gb/s

high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 432–433.

[77] JEDEC, "High bandwidth memory (hbm) dram,jesd235b," 2019. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

[78] Micron, "Hybrid memory cube — HMC gen2," 2019. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf

[79] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, May 2017, pp. 1–4.

[80] A. Developer, "Amba 4 overview," 2019. [Online]. Available: https://developer.arm.com/architectures/system-architectures/amba/amba-4

[81] ——, "Amba overview," 2019. [Online]. Available: https://developer.arm.com/architectures/system-architectures/amba

[82] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, July 2009.

[83] M. Graphics, "Catapult hls," 2019. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis

[84] Arvind and R. Nikhil, "Hands-on introduction to bluespec system verilog (bsv)," in *2008 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, June 2008, pp. 205–206.

[85] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[86] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, "Systemcodesigner: Automatic design space exploration and rapid prototyping from behavioral models," in *2008 45th ACM/IEEE Design Automation Conference*, June 2008, pp. 580–585.

[87] B. Hutchings and B. Nelson, "Developing and debugging fpga applications in hardware with jhdl," in *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers (Cat. No.CH37020)*, vol. 1, Oct 1999, pp. 554–558 vol.1.

[88] A. Studnitzer and O. Mencer, "Going to the wire: The next generation financial risk management platform," in *2013 IEEE Hot Chips 25 Symposium (HCS)*, Aug 2013, pp. 1–26.

[89] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field Programmable Logic and Applications*, Sep. 2013, pp. 1–4.

[90] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[91] Intel, "High-level synthesis compiler," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[92] Xilinx, "Vivado design suite," 2019. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[93] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[94] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

[95] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin, "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 157–162. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755788

[96] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on FPGAs using high level synthesis," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct. 2014, pp. 456–463.

[97] J. Cong and Y. Zou, "A comparative study on the architecture templates for dynamic nested loops," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 251–254.

[98] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 430–437.

[99] M. Gokhale, S. Lloyd, and C. Macaraeg, "Hybrid memory cube performance characterization on data-centric workloads," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '15. New York, NY, USA: ACM, 2015, pp. 7:1–7:8. [Online]. Available: http://doi.acm.org/10.1145/2833179.2833184

[100] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube," *CoRR*, vol. abs/1706.02725, 2017. [Online]. Available: http://arxiv.org/abs/1706.02725

[101] J. Schmidt, H. Fröning, and U. Brüning, "Exploring time and energy for complex accesses to a hybrid memory cube," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 142–150. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989099

[102] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *CoRR*, vol. abs/1701.06420, 2017. [Online]. Available: http://arxiv.org/abs/1701.06420

[103] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, April 1995.

[104] J. D. Leidel and Y. Chen, "Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 621–630.

[105] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600213

[106] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2016, pp. 204–216.

[107] L. Nai and H. Kim, "Instruction offloading with hmc 2.0 standard: A case study for graph traversals," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15. New York, NY, USA: ACM, 2015, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2818950.2818982

[108] X. Wang, J. D. Leidel, and Y. Chen, "Concurrent dynamic memory coalescing on goblincore-64 architecture," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 177–187. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989128

[109] ——, "Memory coalescing for hybrid memory cube," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 62:1–62:10. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225062

[110] J. Zhang, Y. Liu, G. Jain, Y. Zha, J. Ta, and J. Li, "Meg: A riscv-based system simulation infrastructure for exploring memory optimization using fpgas and hybrid memory cube," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 145–153.

[111] J. Zhang, S. Khoram, and J. Li, "Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021737

[112] J. Zhang and J. Li, "Degree-aware hybrid graph traversal on fpga-hmc platform," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 229–238. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174245

[113] R. Hadidi, L. Nai, H. Kim, and H. Kim, "Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 48:1–48:25, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3155287

[114] Intel, "Hardware accelerator research program," 2019. [Online]. Available: https://software.intel.com/en-us/hardware-accelerator-research-program

[115] Amazon, "Amazon ec2 f1 instances," 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[116] R. Hochberg, "Matrix multiplication with cuda | a basic introduction to the cuda programming model," August 2012.

[117] F. Baetke, B. Metzger, and P. Smith, "The convex application compiler - a major step into the direction of automatic parallelization," in *Supercomputer '92*, H.-W. Meuer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 158–172.

[118] Xilinx, "Integrated logic analyzer (ILA)," 2019. [Online]. Available: https://www.xilinx.com/products/intellectual-property/ila.html

[119] Micron, "AC510 specification," 2019. [Online]. Available: https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/ac-series-hpc-modules/ac-510

[120] Xilinx, "Xilinx Zynq-7000 SoC ZC706 evaluation kit," 2019. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html

[121] ——, "Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation kit," 2019. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

[122] ——, "SDSoC development environment," 2019. [Online]. Available: https://www.xilinx.com/cgi-bin/docs/rdoc?v=latest;d=ug1027-sdsoc-user-guide.pdf

[123] ——, "Xilinx kintex-7 FPGA KC705 evaluation kit," 2019. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html

[124] ——, "Kintex ultrascale," 2019. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale.html

[125] Micron, "Ex-700," 2019. [Online]. Available: https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/hpc-backplanes/ex-700

[126] ——, "Sc6-mini," 2019. [Online]. Available: https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/hpc-desktop-systems/sc6-mini

[127] Xilinx, "Microblaze debug module (mdm) v3.2," 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/mdm/v3_2/pg115-mdm.pdf

[128] Micron, "Picoframework," 2019. [Online]. Available: https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/picoframework