8-9-2014

MEMORY INTERFACE SYNTHESIS FOR FPGA-BASED COMPUTING

Zheming Jin
University of South Carolina - Columbia

Memory Interface Synthesis for FPGA-Based Computing

by

Zheming Jin

Bachelor of Science
Huaihai Institute of Technology, 2005

Master of Science
Shanghai University, 2008

Submitted in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2014

ACKNOWLEDGEMENTS

I express my gratitude to my advisor Dr. Bakos for his study and research guidance and financial support. He goes to town so we knock on wood. I also remember my advisor at Shanghai University. Many years ago, a little friend said that an advisor should be your mentor, in the past, now, and in the future. In addition, I acknowledge the committee members' reviews and comments, which better the contents of the dissertation.

It is pleasant to know about and learn from my lab mates, Tiffany, Krishna, Yan, Ibriham, Fan, Yang and Ivan. Some of them have graduated while others are going to complete the study. Impressed by their various interests in food, gadgets, technology, politics and religion, I am also thankful for their help and advice.

Finally, I express my gratitude to my parents!

ABSTRACT

This dissertation describes a methodology for the generation of a custom memory interface and associated direct memory access (DMA) controller for FPGA-based kernels that have a regular access pattern. The interface provides explicit support for the following features: (1) memory latency hiding, (2) static access scheduling, and (3) data reuse. The target platform is a multi-FPGA platform, the Convey HC-1, which has an advanced memory system that presents the user logic with three critical design challenges: the memory system itself does not perform caching or prefetching, memory operations are arbitrarily reordered, and the memory performance depends on the access order provided by the user logic. The objective of the interface is to reconcile the three problems described above and maximize overall interface performance. This dissertation proposes three memory access orders, explores buffering and blocking techniques, and exploits data reuse for the synthesis of custom memory interfaces for specific types of kernels. We evaluate our techniques with two types of benchmark kernels: matrix-vector multiplication and 6- and 27-point stencil operations. Experimental results show the proposed memory interface designs that combine memory latency hiding, access scheduling and data reuse achieve an overall performance speedup of 1.6 for matrix-vector multiplication, 2.2 for a 6-point stencil, and 9.5 for a 27-point stencil as compared to using a naïve memory interface.

Table of Contents

Acknowledgements iii

Abstract iv

List of Tables viii

List of Figures ix

Chapter 1 Introduction 1

1.1 Memory Performance 2

1.2 Overview of Dissertation 4

Chapter 2 Background 7

2.1 High-performance Reconfigurable Computing 7

2.2 Dynamic Random Access Memory 9

2.3 Convey HC-1 Memory System 13

2.4 Memory Access Ordering on the Convey HC-1 15

2.5 Memory Latency Hiding on the Convey HC-1 16

2.6 Data Reuse on the Convey HC-1 21

2.7 Buffering for Memory Latency Hiding and Data Reuse 22

2.8 Performance Measurement 22

Chapter 3 Related Work 25

3.1 the Convey HC-1 System 25

3.2 Memory Access Scheduling 26

3.3 Memory Latency Hiding 28

3.4 DATA REUSE 29

3.5 3D STENCIL 31

CHAPTER 4 PRELIMINARY RESULTS 33

4.1 IMPACT OF DRAM SCHEDULING 33

4.2 IMPACT OF DOUBLE BUFFERING ON MEMORY LATENCY HIDING 36

4.3 IMPACT OF THE LOOP AND ARRAY ACCESS ORDERS 37

4.4 DATA REUSE ANALYSIS OF 3D STENCIL 40

4.5 BANDWIDTH CHARACTERISTICS OF 3D STENCIL 42

4.6 CONCLUSION 43

CHAPTER 5 MEMORY INTERFACE DESIGNS ON THE CONVEY HC-1 45

5.1 STENCIL 45

5.2 MATRIX VECTOR MULTIPLICATION 59

5.3 CONCLUSION 64

CHAPTER 6 EXPERIMENTAL RESULTS 66

6.1 PERFORMANCE IMPACT OF THE LOOP AND ARRAY ORDERS 66

6.2 PERFORMANCE IMPACT OF PROPOSED 3D STENCIL KERNEL DESIGNS 74

6.3 PERFORMANCE IMPACT OF MVM DESIGNS WITHOUT DATA REUSE 80

6.4 PERFORMANCE IMPACT OF PROPOSED MVM DESIGNS 85

6.5 RESOURCE UTILIZATION 87

6.6 DISCUSSION OF EXPERIMENTAL RESULTS 88

CHAPTER 7 CONCLUSION 91

REFERENCES 94

LIST OF TABLES

Table 2.1 Divisions of memory address fields..14

Table 2.2 Memory operations of read reorder buffer 20

Table 4.1 DRAM controller efficiency test with 120 access orders 34

Table 4.2 Memory interface stall rate using double buffering...38

Table 4.3 DRAM controller efficiency of 6-point 3D stencil..39

Table 4.4 Data reuse rate of 6-point 3D stencil 41

Table 4.5 Data reuse rate of 27-point 3D stencil 41

Table 5.1 Reuse rates of using the loop, array and numerical orders 52

Table 6.1 Results of proposed 6-point stencil design (K-dimension partition) 75

Table 6.2 Results of proposed 6-point stencil design (J-dimension partition) 76

Table 6.3 Results of proposed 6-point stencil design (I-dimension partition)....................77

Table 6.4 Comparison of the ideal and actual speedup in proposed 3D 6-point stencil....77

Table 6.5 Results of proposed 27-point stencil design (K-dimension partition) 79

Table 6.6 Comparison of the ideal and actual speedup in proposed 3D 27-point stencil..79

Table 6.7 Interface efficiency (no data reuse and row-major)..82

Table 6.8 Interface efficiency (no data reuse and column-major) 82

Table 6.9 DRAM controller efficiency (no data reuse and row-major) 83

Table 6.10 DRAM controller efficiency (no data reuse and column-major).....................83

Table 6.11 Runtime (no data reuse and row-major) 83

Table 6.12 Runtime (no data reuse and column-major) 83

Table 6.13 Interface efficiency (Data reuse)...85

Table 6.14 DRAM controller efficiency (Data reuse) 85

Table 6.15 Runtime (Data reuse)...86

Table 6.16 Runtime (data reuse) vs. runtime (without data reuse)...................................86

Table 6.17 The ideal speedup and the actual speedup in the proposed MVM designs 86

Table 6.18 Resource utilization of 6-point and 27-point 3D stencils 87

Table 6.19 Resource utilization of MVMs 88

LIST OF FIGURES

Figure 2.1 Simplified version of FPGA internal architecture 7

Figure 2.2 Convey's hybrid-core computer 9

Figure 2.3 Modern DRAM organization 10

Figure 2.4 Memory access scheduling of eight memory operations 12

Figure 2.5 Convey HC-1 memory architecture 12

Figure 2.6 Memory access orderings on Convey HC-1 16

Figure 2.7 Double-buffering control FSM 17

Figure 2.8 Double buffer behaviors and memory interface and pipeline utilization 18

Figure 2.9 State transitions of the reorder buffer 19

Figure 4.1 6-point 3D stencil loop kernel 37

Figure 4.2 Block diagram of the 6-point 3D stencil for each PE 38

Figure 5.1 6-point and 27-point 3D stencils 46

Figure 5.2 A $5 \times 5 \times 4$ grid with elements numbered numerically 48

Figure 5.3 Block diagram of the proposed 3D 6-point stencil for each PE 50

Figure 5.4 Stencil computation of six output elements in 27-point 3D stencil 56

Figure 5.5 Block diagram of one of the three accumulators which operate in parallel 56

Figure 5.6 Block diagram of 27-input 64-bit pipelined adder 58

Figure 5.7 The standard algorithm of MVM 59

Figure 5.8 Matrix blocks assignments among four AEs with 16 PEs per AE 62

Figure 5.9 Block diagram of the proposed MVM for each PE 63

Figure 6.1 Partitions along the J (left) or I (right) dimensions in 3D stencil space 67

Figure 6.2 Efficiency vs. the block sizes (K-dimension partition, 6-point stencil) 67

Figure 6.3 Efficiency vs. the block sizes (J-dimension partition, 6-point stencil)............68

Figure 6.4 Efficiency vs. the block sizes (I-dimension partition, 6-point stencil)............68

Figure 6.5 Runtime vs. the block sizes in 3D 6-point stencil 69

Figure 6.6 Efficiency vs. the block sizes (K-dimension partition, 27-point stencil) 70

Figure 6.7 Efficiency vs. the block sizes (J-dimension partition, 27-point stencil)...........71

Figure 6.8 Efficiency vs. the block sizes (I-dimension partition, 27-point stencil)...........71

Figure 6.9 Runtime vs. the block sizes in 3D 27-point stencil 72

Figure 6.10 Block partitions along the J and I dimensions in 3D stencil space 74

Figure 6.11 Comparison of 6-point 3D DRAM controller efficiency results...................78

Figure 6.12 Row-major and column-major stride access orderings for each PE..............81

Figure 6.13 Scaled runtime vs. matrix and block sizes (No data reuse)............................84

CHAPTER 1

INTRODUCTION

High-performance reconfigurable computing (HPRC) is a computer architecture that combines the potential performance of reconfigurable computing accelerators like FPGAs with the flexibility of conventional general-purpose microprocessors. HPRC is suitable for numerous applications that spend substantial time on a few software kernel loops. Executing these kernels on HPRC systems reduces the execution time as compared to software implementations using general-purpose processors.

In HPRC systems, kernels are implemented with hardware description languages such as Verilog and System Verilog. Some HPRC systems also allow the use of high-level synthesis tools [1,2,3,4] to aid the design implementation and verification of kernels. Current high-density FPGAs enable the implementation of a kernel in a modular fashion to improve design productivity.

An FPGA is an integrated circuit designed to be configured by a customer after manufacturing. An FPGA contains programmable logic components and a hierarchy of reconfigurable interconnects that allow the components to be wired together in many different configurations for any arbitrary digital logic circuit. FPGAs often handle the kernel computation faster than general-purpose processors because they can be fully customized, allowing almost all hardware resources to be dedicated to the computation. On the other hand, CPUs generally utilize hardware resources less efficiently due to its general-purpose fixed microarchitecture.

The major challenge with designing FPGA-based kernels is that a designer must explicitly define the on-chip portions of memory hierarchy and cannot take advantage of the complex and refined structures on microprocessors that are taken for granted. On many HPRC systems, on-chip components do not exist between memory system and FPGAs, so the designer has to specify the memory access mechanism (e.g., generate memory addresses and employ buffering) to hide memory access latency and reuse commonly occurring items to improve performance. The design and optimization of on-chip components are challenges for the designer.

1.1 MEMORY PERFORMANCE

In HPRC systems, the main applications run on the microprocessors while the FPGAs handle kernels that have long execution time. Such kernels typically present parallel computations that can be efficiently implemented on the FPGAs. To support highly parallel processing on the FPGAs, the memory system in an HPRC system offers very high memory bandwidth and capacity. However, high memory bandwidth and capacity do not lead directly to high increase in the memory system performance. For memory-bound kernels, memory system performance is very important because the overall kernel performance depends on memory system performance. Memory system performance is often characterized by memory latency hiding, memory access scheduling and data reuse.

Memory latency hiding overlaps kernel computations with memory accesses. A high-performance microprocessor hides the memory latency through out-of-order instruction execution, nonblocking cache and prefetching. In out-of-order instruction execution, arithmetic instructions that follow the memory access instruction may bypass the pending memory instruction if there is no data dependency between them. Nonblocking cache also effectively reduces the cache miss penalty by overlapping execution with memory access. It allows the data cache to continue to supply cache hits during a cache miss. Another approach is to prefetch data before they are requested by the processor. The prefetched data can be put into the caches or into an external buffer that can be accessed more quickly than the main memory. In FPGA-based designs, support for memory latency hiding must be explicitly defined by the user based on the characteristics of the memory system on an HPRC platform.

The HPRC memory system generally utilizes a DRAM-based memory system. DRAM is organized in a way in which its effective bandwidth depends on the order in which accesses are made. Some CPUs perform *memory access scheduling* to dynamically schedule memory loads and stores to improve memory performance, but they are limited by the size of the memory load/store queue and/or prefetching accuracy. Other CPUs access the memory in the order statically scheduled by the compiler. An FPGA design of a kernel with a regular access pattern and a priori knowledge of all memory accesses presents a huge potential for scheduling memory accesses insofar as on-chip memory capacity permits. However, the user has to explicitly define memory access orderings based on DRAM architecture and memory access patterns of a kernel to achieve high memory bandwidth.

Data reuse is another memory performance optimization that improves memory performance. It takes advantage of temporal locality of memory accesses to avoid unnecessary off-chip memory accesses. CPUs reuse data in the caches by exploiting temporal locality of memory accesses. However, conflict misses in a cache still cause some unnecessary memory accesses. In an FPGA design, whenever there is regular access pattern, unnecessary off-chip memory accesses can be completely avoided insofar as the on-chip memory capacity permits. The support for data reuse must be integrated into the design as well.

1.2 OVERVIEW OF DISSERTATION

The Convey HC-1 is a modern HPRC system combining FPGAs with microprocessors. The goal of this dissertation is to develop methodologies that employ memory latency hiding, memory access scheduling and data reuse for two types of memory-bound kernels, matrix vector multiply (MVM) and 3D stencil, on the Convey HC-1 system.

We choose MVM and 3D stencil because both kernels are memory-bound. If the kernel is compute-bound, then its performance is not constrained by the memory performance. We select 6-point and 27-point 3D stencils as examples of simple and complex access patterns, respectively. The larger the number of stencil points, the more memory performance optimizations can be applied, but the fewer FPGA resources available for the implementation.

Our approach is to develop an optimized memory controller interface that integrates memory latency hiding, memory access scheduling and data reuse to achieve high overall performance. The goal of memory latency hiding is to maximize *interface efficiency*, the percentage of interface execution time in which memory controller interface performs a memory access. We achieve almost 100% interface efficiency for memory latency hiding.

For memory access scheduling, the goal is to maximize *DRAM controller efficiency*, the percentage of memory access time in which memory accesses are not stalled due to the conflicts. We improve DRAM controller efficiency from 39% to 88% for MVM without data reuse, and from 42% to 89% for MVM with data reuse, and from 68.7% to 74% for 6-point 3D stencil, and from 52% to 72% for 27-point 3D stencil.

With a limited size of on-chip buffers, the goal of data reuse is to minimize the total number of memory requests required to achieve high *data reuse rate*. Data reuse rate is the ratio of the number of reused memory requests to the total number of memory requests without data reuse. We are able to achieve the reuse rate of about 0.5 for MVM and the reuse rate of 0.6 and 0.9 for 6-point and 27-point 3D stencil respectively.

When combining all three optimizations in the memory controller interface, the goal is to improve (i.e., decrease) the execution time, so the overall performance of the memory controller interface is measured with speedup. Compared to the baseline memory interface designs, we are able to achieve averaged speedup of 1.6 for MVM, 2.2 for 6-point 3D stencil and 9.5 for 27-point 3D stencil, with data reuse contributing the most to the improvement of speedup. The experimental results show that our dissertation can provide design and optimization references to the end-user who needs to parallelize a specific application by implementing it on a multi-FPGA platform.

The rest of the dissertation is organized as follows. Chapter 2 describes the multi-FPGA Convey HC-1 platform. Then it describes the techniques of memory latency hiding, memory access ordering and data reuse in the memory interface. Chapter 3 discusses related work from the perspectives of the target platform, memory latency hiding, memory access scheduling and data reuse. Chapter 4 describes the preliminary results of our work on the Convey HC-1. Chapter 5 presents detailed memory interface designs of the kernels on the Convey HC-1. Chapter 6 shows the experimental results of memory interface designs on the Convey HC-1. Chapter 7 concludes the dissertation.

CHAPTER 2

BACKGROUND

2.1 HIGH-PERFORMANCE RECONFIGURABLE COMPUTING (HPRC)

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are built on a matrix of configurable logic blocks (CLBs) connected via programmable interconnections. Each CLB is basically composed of combinational logic (CL) and flip-flops (FFs). Figure 2.1 shows a simplified version of an FPGA internal architecture. As opposed to Application Specific Integrated Circuits (ASICs), where the devices are built specifically for an

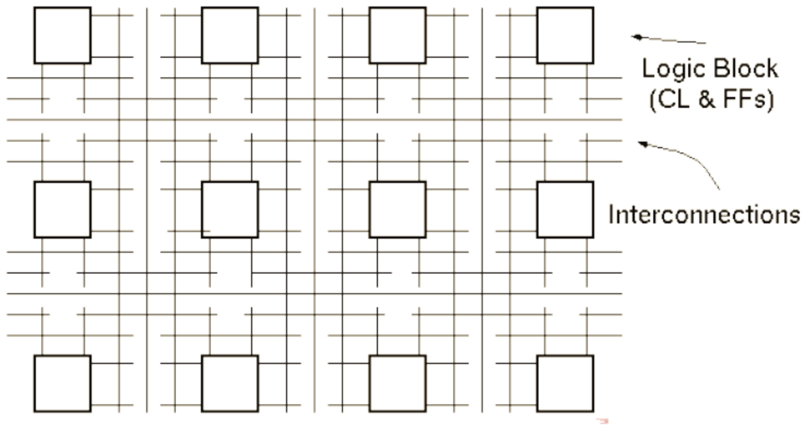


Figure 2.1 Simplified version of FPGA internal architecture

application, FPGAs can be programmed to the desired application or functionality requirements. Due to FPGAs' programmable characteristics, they are widely used in fields of vehicles, consumer electronics, high-performance computing, video and image processing. Two large companies, Altera and Xilinx, provide the broadest lineup of FPGAs with advanced features, low power, and high performance, for any FPGA designs [5,6].

What are the benefits of having FPGAs in an HPRC system? The potential speed benefits of using FPGAs are huge when taking advantage of massive parallelism in FPGAs. FPGAs also provide designers full control at the lowest hardware level, allowing fine-grained, manual optimizations for each application. In addition, FPGAs are low-power devices compared to traditional microprocessors. The power consumption of a typical design in FPGAs is very low. Finally, FPGAs are flexible in the reconfiguration of hardware accelerators for multiple applications.

The concept of reconfigurable computing has existed since the 1960s, when the concept of a computer made of a standard processor and an array of reconfigurable hardware was proposed [7,8,9]. The aim of the reconfigurable computing is to combine the flexibility of software with the speed and configurability of the hardware, with a main processor controlling the behavior of the hardware. Unfortunately the concept was not realizable until the advancements in silicon process technology allowed complex designs to be implemented on large and very large-scale integrated circuits.

Early HPRC systems were designed around one node of microprocessors and another of FPGAs. The two nodes were connected directly without a scalable interconnection. The modern architecture of scalable parallel systems is grouped into two classes based on the types of the nodes and how they are connected. In uniform node nonuniform systems, each node has either FPGAs or microprocessors and the nodes are connected via an interconnection network to globally shared memory. In nonuniform node uniform systems, there is only one type of node and each node contains FPGAs and microprocessors. FPGAs are connected directly to the microprocessors in the node. The nodes are connected the same way as in uniform node nonuniform systems.

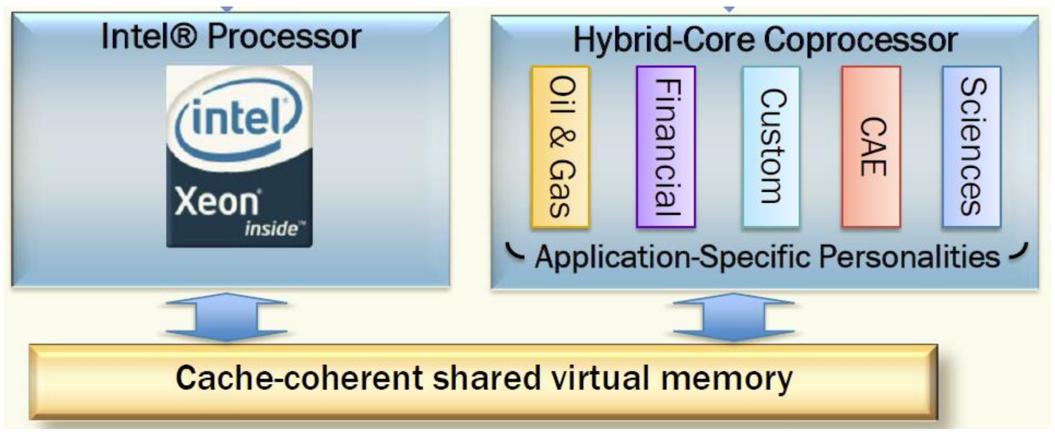


Figure 2.2 Convey's hybrid-core computer [10]

The Convey HC-1 platform, the first of a series of Convey's high-performance reconfigurable computing products, was unveiled in 2009. The reconfigurable computing system combines commodity Intel processors with a reconfigurable coprocessor based on FPGA technology. Figure 2.2 illustrates Convey's hybrid-core computer. The coprocessor can be dynamically reloaded with custom instructions that are optimized for different workloads. The instructions are a combination of x86 set of instructions and reloadable set of instructions. The reloadable set of instructions is called a personality. The platforms also feature globally shared memory that supports a standard load/store programming model. In the globally shared memory model, both the processor and the coprocessor have copies of the global memory space through a cache-coherent shared virtual memory.

2.2 DYNAMIC RANDOM ACCESS MEMORY

The memory system in an HPRC system provides very high memory bandwidth and capacity to support the highly parallel, high bandwidth processing of the coprocessors. In

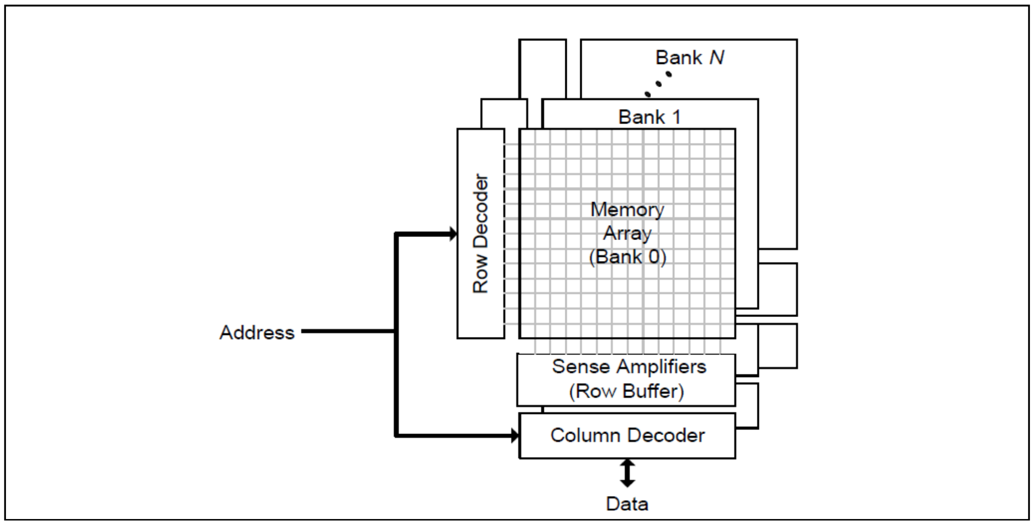


Figure 2.3 Modern DRAM organization

such memory system, multiple DRAM devices are interconnected together to form a memory system managed by one or more memory controllers. The functionality of a memory controller is the read or write requests to a given address in DRAM in accordance with the DRAM access timing requirements. DRAM is logically a 3D memory with the dimensions of bank, row and column. Figure 2.3 shows the logical organization of a modern DRAM. Each bank operates independently of the other banks and contains an array of memory cells that are accessed an entire row at a time. Modern DRAM devices contain multiple banks so that multiple, independent accesses to different DRAM arrays can happen in parallel. When a row of the memory array is accessed (row access) through the row decoder, the entire row (a.k.a. a DRAM page) is latched into the sense amplifiers (row buffer). The row buffer reduces the latency of subsequent accesses to that row. The time it takes to move data from the memory array to the row buffer is known as the Row Column Delay (RCD). When a row is held in the buffer, any number of reads or writes (column access) may be performed through the column decoder. Read data, for example, are moved from the sense amplifiers of a given bank to the memory controller. The time it takes to move the data from the sense amplifier to the memory controller after initiation of column access is known as the Column Access Strobe (CAS) latency. The CAS latency measures the time to transfer the first word of memory. After the reads or writes are complete, the row must be written back to the memory array through bank precharge to prepare the bank for a subsequent row access. A detailed introduction to the DRAM organizations, operations, timing, and the basic nomenclature can be found in [11].

Modern HPRC systems are becoming increasingly limited by DRAM based memory system performance. To improve memory performance, modern DRAM provides several independent memory banks, buffers the most recently accessed row of each bank and allows pipelined and burst memory accesses. Though these device-level optimizations increase the memory performance, they make memory performance dependent on the specific memory interface in the HPRC system. The functionality of memory interface is to efficiently manage the memory accesses to the memory system to achieve high system performance.

Memory access scheduling or ordering is an effective way to improve the memory performance. It is a mechanism of ordering the memory accesses to complete the set of pending memory references. Figure 2.4 shows the sequence of eight memory operations on DRAM. Each memory reference is represented by the address (bank, row and column) and we assume 3 cycles, 3 cycles and 1 cycle are required for bank precharge, row access and column access respectively. If the eight memory operations are performed in order, 56 cycles are needed to finish the eight references. If the memory references are reordered, they take only 19 cycles.

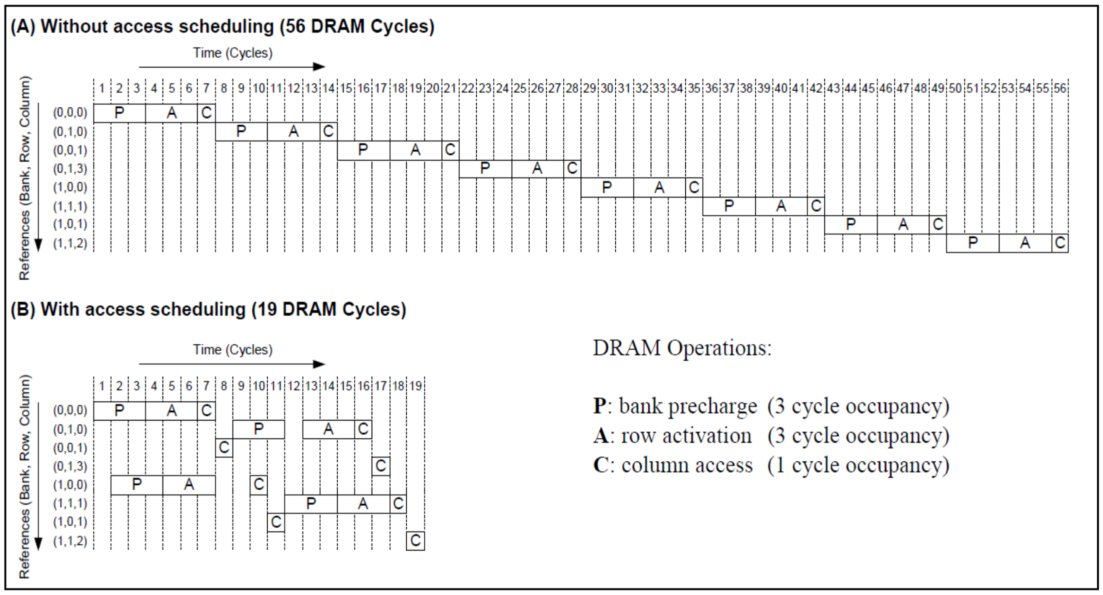

(A) Without access scheduling (56 DRAM Cycles)
Time (Cycles)
References (Bank, Row, Column)
(0,0,0)
(0,1,0)
(0,0,1)
(0,1,3)
(1,0,0)
(1,1,1)
(1,0,1)
(1,1,2)
P
A
C
(B) With access scheduling (19 DRAM Cycles)
DRAM Operations:
P: bank precharge (3 cycle occupancy)
A: row activation (3 cycle occupancy)
C: column access (1 cycle occupancy)

Figure 2.4 Memory access scheduling of eight memory operations.

The figure is from [12]

Another way to improve the overall performance of an HPRC system is the use of on-chip buffers for hiding the latency of memory access. As compared to the general-purpose processors that have hardware-managed caches to hold frequently accessed data, the on-chip buffers in the FPGAs store frequently accessed data. The overall performance increases significantly as the latency of accessing on-chip data is much faster than off-chip

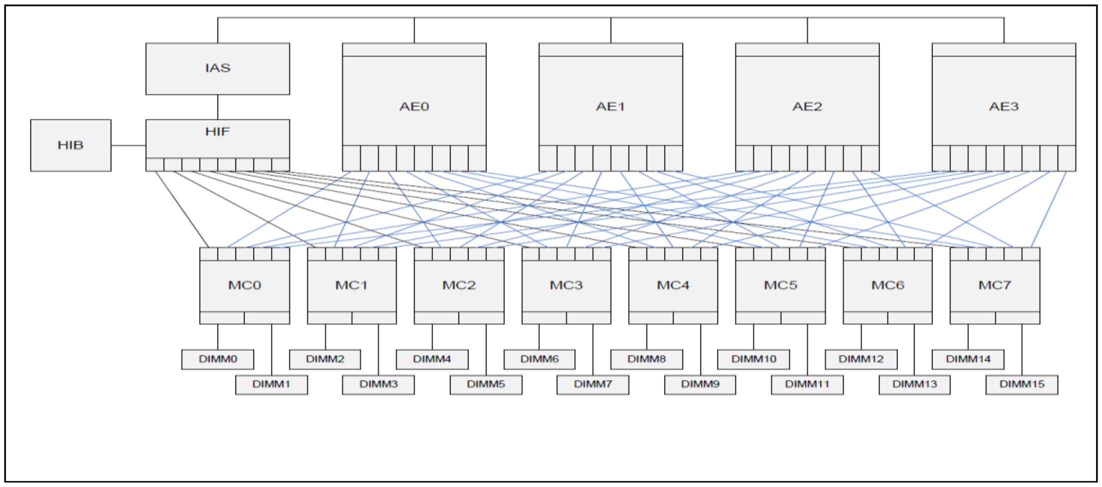


Figure 2.5 Convey HC-1 memory architecture

DRAM access. In addition, the on-chip buffers presents opportunities for data reuse, i.e., the access of data in the buffers more than once during application execution. By storing and accessing the reusable data in the on-chip buffers, the memory bandwidth contention is decreased, which in turn reduces memory access latency and improves the overall performance.

2.3 CONVEY HC-1 MEMORY SYSTEM

The Convey HC-1 is a high-performance reconfigurable computer containing an FPGA-based coprocessor attached to a host motherboard through a socket-based front-side bus interface. Unlike socket-based coprocessors from Nallatech [13], DRC [14], and XtremeData [15], which are confined to a footprint matching the size of the socket, Convey uses a mezzanine connector to bring the front side bus (FSB) interface to a large coprocessor board roughly the size of an ATX motherboard. This allows the coprocessor memory to be coherent with the host CPU's memory while providing sufficient real estate for the coprocessor board to be capable of hosting multiple large user FPGAs and a large and sophisticated memory system.

As shown in Figure 2.5, the coprocessor board contains four user-programmable Virtex-5 LX 330 FPGAs that Convey refers to as "application engines (AEs)". The coprocessor board also contains eight discrete memory controllers, each of which is implemented on its own Virtex-5 LX 110 FPGA. The memory space of the coprocessor board is physically partitioned into eight equal-size segments, and each segment is only accessible from one of the eight memory controllers. Specifically, each memory controller can access 64 contiguous bytes within each block of 512 contiguous bytes. Each AE is connected to all eight memory controllers through a crossbar switch that is instanced on each memory controller.

The interface between each AE and each memory controller allows up to two independent 64-bit memory transactions (read or write) per cycle on a 150 MHz clock, giving a peak theoretical bandwidth of 2.4 GB/s per memory controller, or 19.2 GB/s per AE, or 76.8 GB/s aggregate bandwidth for the coprocessor board [16]. In practice, each AE “sees” a memory interface consisting of 16 64-bit wide channels.

Memory addresses are virtual and mapped to 4 MB pages. Each memory controller contains a translation lookaside buffer (TLB) to cache the page table. Within each 4 MB page, the memory address is divided into the following physical fields as listed in Table 2.1. Each of the eight memory controllers are connected to two DIMMs, which are selected by bit 9. Both DIMMs contain eight DRAMs (i.e., eight 8-bit busses) and each DRAM has eight banks that can be accessed independently by the memory controller. The division

Table 2.1 Divisions of memory address fields

Bit Field	**Field Size**	**Field Name**
21 : 20	2	DRAM row
19 : 13	7	DRAM column
12 : 10	3	DRAM bank
9	1	DIMMs select
8 : 6	3	Memory controller
5 : 3	3	DRAM bus
2 : 0	3	Byte alignment

(mapping) of memory address fields implies that the requests are interleaved across the memory controllers as well as across the banks within the DIMMs attached to each memory controller. This ensures that sequential accesses will be spread across all 16 DIMMs, maximizing concurrency and bandwidth.

Each of the memory controllers also attempts to maximize bandwidth by scheduling incoming memory requests to each of the banks on its DIMMs, routing requests to non-busy banks and grouping reads and writes into bursts to minimize bus turns (state change between read and write). As a result, memory accesses are performed in a different order in which they were requested by the user. Since Convey will not reveal implementation details nor provide a cycle-accurate model of the memory controller, we performed a bandwidth test that accesses all the 65536 64-bit words accessible from a single memory controller within a single page. The effective bandwidth given by the Convey memory controller with various access patterns will be characterized in Chapter 4.

2.4 MEMORY ACCESS ORDERING ON THE CONVEY HC-1

Apart from the memory access scheduling in the memory controller, the memory controller interface introduces memory access orderings from the perspective of custom personality. In other words, a custom personality can request the memory through the memory control interface in different access orders. Figure 2.6 gives an overview of the memory access orderings A, B, and C that occur on the Convey HC-1 platform. Different memory access orderings may be performed among the personality, the memory controller interface, the memory controller and the memory. While the memory access ordering at B depends on Convey's proprietary memory controller, the ordering at A depends on the access orders of the custom personality. For example, a personality that implements a loop kernel of vector addition, may request an element of each array for the current loop iteration before proceed to request the elements of the next iteration. Alternatively, the personality may request two elements of one array for the two loop iterations before proceed to request two

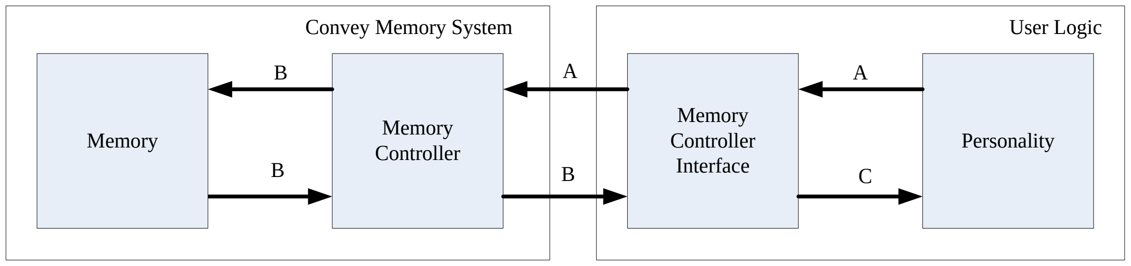


Figure 2.6 Memory access orderings on Convey HC-1

elements of another array. The ordering at C may be the same as the ordering at A when the receiving data return to the personality in the same order as the data requests from the personality. The ordering at C may also be different from the ordering at A when the data from the memory controller interface can be reused by the personality. In this case, the personality only needs to request the data once and the return data can be reused by the personality. In the dissertation, we explore the memory access ordering at A for the application-specific personalities to improve memory performance.

2.5 MEMORY LATENCY HIDING ON THE CONVEY HC-1

On the Convey HC-1 platform, the memory access latency is around 256 clock cycles. To hide the long memory latency, the memory controller interface needs to support pending memory requests to efficiently utilize the memory bandwidth. When requested data return from the memory, they are stored in the on-chip buffers from which the kernel read the input data for computation. To parallelize the kernel computation with the memory requests, the memory interface uses more than one buffer (multiple buffering) to hide the memory latency. Multiple buffering delivers a stream of inputs from the DRAM-based memory system to the kernel while minimizing the number and length of stream interruptions.

2.5.1 DOUBLE BUFFERING

The first attempt at memory interface design used a double-buffering (ping-pong) approach, where data returned from the memory controller fill one buffer while the contents of the other buffer are flushed into the pipelined kernel. To simplify the control of double-buffering, each of the two buffers has its FSM-based controller, shown in Figure 2.7. Each buffer proceeds through a sequence FILL, WAIT, and FLUSH with occasional transitions between FLUSH and STORE. In FILL, the buffer requests a sequence of load addresses from the memory channel. After all the addresses have been requested, the controller enters the WAIT state, where it waits for all the requested data to be delivered from memory. After all input values are delivered to the input buffer, the buffer enters the FLUSH state in which all the stored input values are sent into the pipelined kernel (assuming the other buffer has completed its own FLUSH procedure). Since both buffers share a single memory channel, only one buffer can be in the FILL state at a time. Likewise, since both buffers

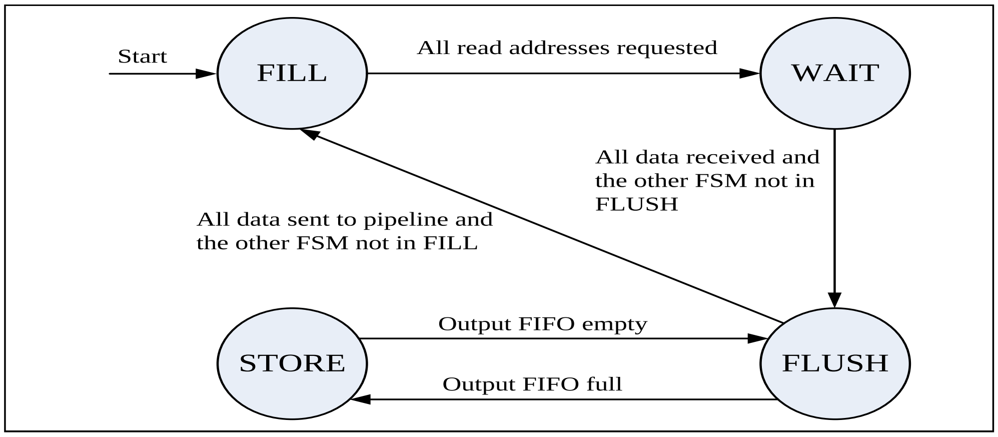

Start
FILL
All read addresses requested
WAIT
All data received and the other FSM not in FLUSH
All data sent to pipeline and the other FSM not in FILL
Output FIFO empty
STORE
FLUSH
Output FIFO full

Figure 2.7 Double-buffering control FSM

are sharing a single kernel pipeline, only one buffer can be in the FLUSH state at one time. This yields a combined behavior as shown in Figure 2.8. The top two horizontal lines represent the state transitions of both buffers. The bottom two horizontal lines represent when memory requests are being made (top) and when the kernel pipeline is enabled (bottom). As shown, buffer A is the first to enter the FILL state, while buffer B must wait. This is shown at point (a). The time required in the FILL state depends on the number of stall requests issued by the MC. After buffer A issues all of its memory read requests it transitions to the WAIT state to wait for the input data to be delivered from MC. The amount of time required in the WAIT state depends on the average memory latency. After all data are received, buffer A transitions to the FLUSH state to send its buffer contents into the pipeline. The number of cycles required in the FLUSH state is fixed. However, as shown at point (b), buffer B must wait until buffer A completes its FLUSH before it can also transition to FLUSH. During the time that elapses after buffer B completes its FILL and before buffer A completes its FLUSH, the memory channel is idle. Likewise, during

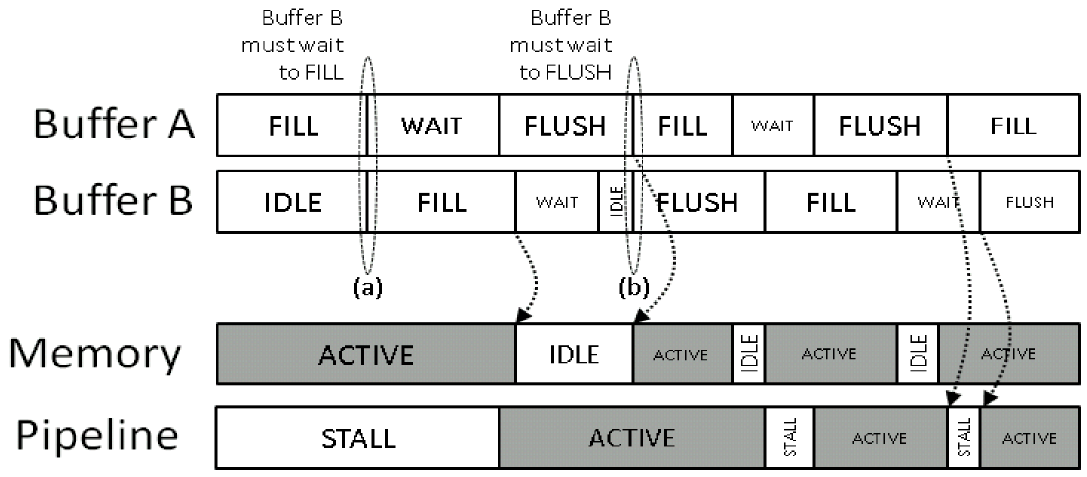


Figure 2.8 Double buffer behaviors and memory interface and pipeline utilization

the time that elapses between buffer A's subsequent FLUSH and buffer B's next FLUSH, the pipeline is idle.

The analysis of Figure 2.8 shows that the memory interface is idle when both buffer are in the WAIT state or one buffer is in the WAIT state while the other in the FLUSH state. In addition, the kernel pipeline is stalled when one buffer is in the FILL state and the other in the WAIT or IDLE state. These overlapping intervals introduce the inefficiency of the memory interface design with double buffering. Thus, the memory latency is not entirely hidden due to the contentions of the buffers for memory accesses and kernel pipeline.

2.5.2 CIRCULAR BUFFERING

In recent releases on Convey's Personality Design Kit, they included an optional memory reorder buffer that forms an interface between the user logic and the memory controller. The reorder buffer sends the requested words back to the user logic in the order they were requested from the user logic. As shown in Figure 2.9, when a request is received from the user logic, the reorder buffer's state transitions from *empty* to *pending*. When the

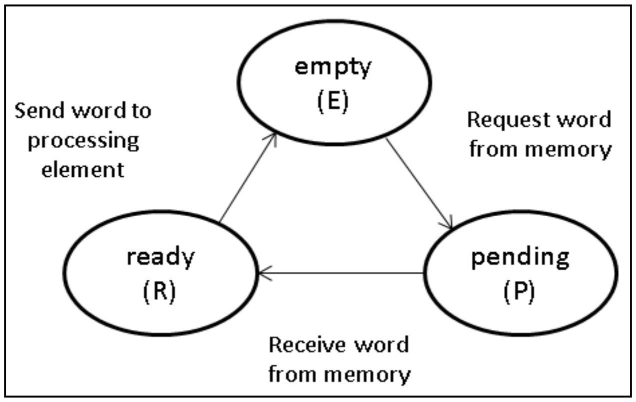
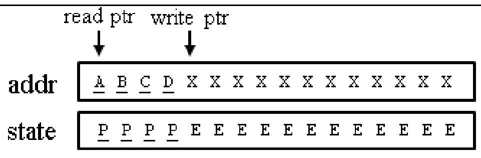
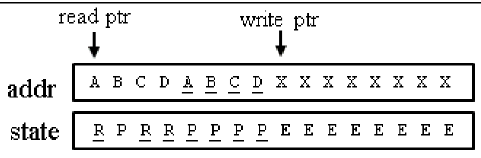
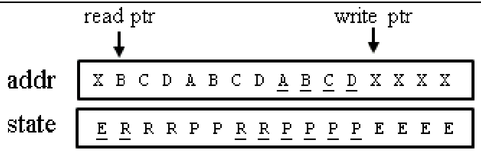
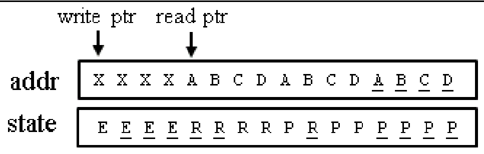


Figure 2.9 State transitions of the reorder buffer

Table 2.2 Memory operations of read reorder buffer

Cycles	**Address generator**	**Reorder buffer**	**Memory controller interface**	**Buffer state**
0-3	Load addresses of A[0], B[0], C[0], D[0]	Request A[0], B[0], C[0], D[0] from memory	Nothing	 read ptr write ptr addr A B C D X X X X X X X X X X X X state P P P P E E E E E E E E E E E E
4-7	Load addresses of A[1], B[1], C[1], D[1]	Request A[1], B[1], C[1], D[1] from memory	Deliver values of A[0], C[0], D[0] from memory to the buffer	 read ptr write ptr addr A B C D A B C D X X X X X X X X state R P R R P P P P E E E E E E E E
8-11	Load addresses of A[2], B[2], C[2], D[2]	Return A[0] to user Request A[2], B[2], C[2], D[2] from memory	Deliver values of B[0], C[1], D[1] from memory to buffer	 read ptr write ptr addr X B C D A B C D A B C D X X X X state E R R R P P R R P P P P E E E E
12-15	Load addresses of A[3], B[3], C[3], D[3]	Return B[0], C[0], D[0] to user Request A[3], B[3], C[3], D[3] from memory	Deliver values of A[1], B[1], B[2] from memory to buffer	 write ptr read ptr addr X X X X A B C D A B C D A B C D state E E E E R R R R P R P P P P P P

memory controller returns the requested word, the word is stored in the reorder buffer and marked as *ready*. When the read pointer points to a buffered word that is marked as ready, its value is sent back to the user logic.

Table 2.2 illustrates the operations of the reorder buffer. Assume the user logic implements a loop that computes an expression requiring one element from each of four arrays, A, B, C, and D. The kernel pipeline expects one entry from each array to be delivered in sequence, i.e., A[0], B[0], C[0], D[0], A[1], B[1], etc. The reorder buffer is based on a queue implemented as a circular buffer. As opposed to the double buffering where there are front and back buffers, the circular buffer has a single buffer that functions as both front and back buffers. When the user logic performs a load request, the state of the request is enqueued in the FIFO using a write pointer, it is requested from the memory controller, and its status is monitored.

Compared to the double buffering technique in which the memory interface could be idle without memory requests, our FPGA tests based on the kernel benchmarks show that the reorder buffer doesn’t introduce any idle or stall cycles.

2.6 DATA REUSE ON THE CONVEY HC-1

Data transfers between the DRAM-based memory system and the processing elements are often the performance bottleneck when using reconfigurable logics in FPGAs as a hardware accelerator. As a result, the use of on-chip buffers to store repeatedly accessed data is an effective way to reuse the data and increase the memory access bandwidth. The on-chip buffers are typically implemented using FPGA’s embedded RAM blocks. Given the architecture of the Convey HC-1 memory system in which each AE has 16 memory channels with each channel composed of request and response ports, the kernel computation is divided among 16 PEs per AE for maximum parallel execution and memory access bandwidth. Each PE accesses its on-chip buffers for reused data.

While each PE uses on-chip buffers for fast access to the reused data, the amount of reused data, however, is much larger than on-chip memory capacity. On-chip buffers have large bandwidth but limited size due to the number of available embedded RAM blocks on the target FPGAs. This means that some redundant memory accesses are still required as the on-chip memories cannot hold all reused data. So the buffers are updated at runtime based on the memory access patterns of the kernels. In section 2.9, we will define data reuse rate to measure the efficiency of data reuse.

2.7 BUFFERING FOR MEMORY LATENCY HIDING AND DATA REUSE

From the descriptions in sections 2.5 and 2.6, the on-chip buffers assume the roles of memory latency hiding and data reuse. Convey's circular buffering is better than double buffering in memory latency hiding because it doesn't introduce idle or stall cycles. In circular buffering, the reorder buffer sends the requested words back to the user logic in the order they were requested from the user logic. The problem with the reorder buffer is that once the requested words return to the user logic, they are not available in the reorder buffers for reuse. On the other hand, to reuse the requested words they must be kept (until they have to be updated due to the limited size of buffers) in buffers after they have been read. To achieve both memory latency hiding and efficient data reuse, the memory interface has two kinds of buffering for each PE: the reorder buffer requests memory and returns the requested words in order while custom circular buffer stores and accesses the requested data in a way suitable for reuse. This approach realizes the advantage of each buffer so that the reorder buffer returns the requested data without interface stall and the custom buffer is focused on maximizing data reuses.

2.8 PERFORMANCE MEASUREMENT

Chapter 1 briefly mentions overall performance metric and memory performance metrics for memory latency hiding, memory access scheduling and data reuse. For memory-bound kernels, the user logic attempts to access memory during every clock cycle of operation. Any cycle where the logic does not access memory is either due to inefficiency in the memory controller interface designs or due to a stall request from Convey's memory controller. To evaluate the efficiency of memory controller interface design, we compute *interface efficiency* as:

Interface efficiency = (rc + wc) / (execution time – sc),

where *rc* is the number of memory reads (loads) and *wc* the number of writes (stores) for each PE. Execution time is the time lapse from the time when all PEs in an AE start the kernel computation to the time when all PEs finish the kernel computation. Since not all PEs finish their kernel computation at the same time, the execution time is the worst runtime of all PEs. Similarly, *sc* is the maximum number of stall cycles among all PEs in an AE.

DRAM controller efficiency differs from interface efficiency in that it does not consider any inefficiency in the kernel design. To measure the performance of memory access scheduling, DRAM controller efficiency is computed as:

DRAM controller efficiency = (rc + wc) / (rc + wc + sc),

where *rc* is the number of memory reads (loads) and *wc* the number of writes (stores) that access a DRAM controller via its even and odd ports. As the crossbars distribute the memory requests of four AEs across eight DRAM controllers, we assume the number of memory accesses (*rc* + *wc*) to a DRAM controller is the number of memory accesses of all PEs of four AEs that are connected to the DRAM controller. *sc* is the sum of read and write stall cycles requested by the DRAM controller at its even and odd ports in a PE.

Assuming that memory requests are performed every execution cycle, the effective memory bandwidth of each memory channel can be computed as:

BW = 1.2 GB/s × *Interface efficiency* × *DRAM controller efficiency*

To measure the efficiency of data reuse, the *reuse rate* is computed as the ratio of the number of reused memory requests and the number of memory requests before (without) data reuse:

Reuse rate = (M1 – M2) / M1,

Where *M1 – M2*, the number of reused memory requests, is the difference of the number of memory requests before and after data reuse.

Since interface efficiency does not take into account any memory stalls, it is expected to be higher than DRAM controller efficiency. Just as the cache performance in a memory hierarchy is evaluated by the memory access time, we evaluate overall performance combining memory latency hiding, memory access scheduling and data reuse by the execution time. When combining all three optimizations in the memory controller interface, the goal is to improve (i.e., decrease) the execution time. We use speedup, the ratio of the execution time of two implementations with different optimizations, to compare the overall performance of the designs. In the Chapter on experimental results, we will define speedup in the context of specific implementations for each kernel.

CHAPTER 3

RELATED WORK

3.1 THE CONVEY HC-1 SYSTEM

The Convey HC-1 is a modern HPRC system. A few papers on the system were published to study the software/hardware interfaces, the programming methodologies, design and implementation optimizations, and system performance. However, the improvement of memory performance on the Convey HC-1 has not been studied well in the past.

Bakos examined the Convey HC-1 system in terms of system architecture, performance, ease of programming and flexibility [17]. The paper shows that the Convey system achieves better results than a CPU for memory-intensive applications such as Stride3 benchmark and Smith-Waterman algorithm. In [18] the authors analyzed the programming paradigm, the associated programming effort, and presented practical results on the Convey HC-1. Using a 3D 7-point stencil for solving the Laplace equation on grids of different sizes, they compared the peak floating-point performance on the Convey HC-1 and on a 2-way 2.53-GHz Intel Nehalem processor. They showed that the stencil performance on the Convey HC-1 is lower on smaller grids due to the lack of caching.

At the hardware design level, Ruthenberg [19] used the ROCCC compiler with Convey HC-1 support to generate the vector addition personality and evaluated the memory throughput using Convey's crossbar and read order buffer. The results show that the highest memory throughput is achieved when a read reorder buffer is instantiated alone as opposed to the combination of crossbar and read reorder buffer.

Cong et al. [20] reported their implementations of a memory-bound application on the Convey HC-1 and GPU platforms. Their FPGA design is around 30% faster than the GPU implementation and consumes less than half of the power consumption. However, the memory efficiency is around 30% on both platforms though the design utilizes all the memory access ports on the Convey HC-1.

Recently Nowak et al. [21] presented a highly parallel architecture that exploits the massive parallelism on the Convey HC-1 to accelerate prefiltering in HHblits. They achieved application speedup of 1.79 against the original, unmodified state-of-the-art SSE-based implementation. By considering data reuse, they transformed the memory-bound, SSE-based algorithm to a compute-bound FPGA implementation that has an average throughput of only 69 MB/s. The paper, however, does not mention memory efficiency of the memory system.

3.2 MEMORY ACCESS SCHEDULING

Memory access scheduling is an effective way to improve DRAM-based memory efficiency where the memory performance depends on how DRAMs are accessed by the DRAM controllers. Scott et al. [10] introduced the concept of ordering memory references to exploit locality within the 3-D memory structure. They described DRAM architectures and presented the memory controller architecture that supports memory access ordering. They showed that conservative and strong orderings can improve memory bandwidth by 40% and 93% respectively for a set of media benchmarks.

Zhang et al. [22] proposed a fine-grain priority scheduling method, which divides and maps memory accesses into different channels and returns critical data first, to fully utilize the available bandwidth and concurrency provided by DDR SDRAM and Direct Rambus DRAM systems in simultaneous multithreading systems. They showed DRAM optimizations make a larger difference than on single-threaded systems for workloads with intensive memory demands.

Fang et al. [23] proposed a core-aware memory access scheduling that classifies outstanding requests from the same processor core and issue them together. The idea is that the requests from the same core more likely exhibit data locality than the requests from different cores. A threshold of the maximum number memory requests is also set to prevent starvation of memory requests from other cores. Their simulation results on Solaris OS show that OS task scheduling schemes reduce the performance improvement of the benchmarks running with multiple threads.

Mutlu et al. [24] proposed a Stall-Time Fair Memory scheduler to provide quality of service to different threads using the DRAM memory system. Their scheduler achieves stall-time fairness by prioritizing requests from threads with very high memory slowdown. The memory slowdown is memory stall time of a thread running alongside other threads versus the stall time of the thread running alone. The scheduler is an extension to the first-come-first-serve scheduler with the additions of slowdown estimation and priority policy. The experimental results running on an 8-core chip multiprocessor show an average throughput improvement of 7.6%.

Yuan et al. [25] proposed a complexity-effective memory access scheduling for GPU architectures. They employed an interconnection network arbitration scheme that preserves the row access locality of each memory request streams to achieve DRAM efficiency. There is up to 91% performance improvement on the architecture with 8 memory channels, each controlled by its own DRAM controller and 1,792 in-flight memory requests. In [26], the authors proposed a CUDA-based API that allows programmer to improve the efficiency of memory accesses on the GPU platform. They added memory layout remapping and index transformation to NVIDIA's CUDA API and achieved about 3x speedup on GPU kernels and 20% overall performance improvement.

3.3 MEMORY LATENCY HIDING

Various software and hardware techniques of memory latency hiding have been proposed to improve the memory performance on the VLIW, HPRC and GPU architectures. The Decoupled Access/Execute architecture [27] in the early research decouples memory access and instruction execution. The access processor performs all memory access and address calculation; the execute processor does the computation. The communication between two asynchronously running processors is via the queues. The decoupled architecture hides memory latency by overlapping computation with memory transfer. Decoupled architectures are not very useful for general purpose computing as they don't handle controls and branches well, but they play an important role in scheduling in VLIW architecture. On HPRC platforms, decoupled architectures may be drawn as a reference to exploit parallelism between memory access and kernel computation to improve performance.

Software techniques were also proposed to hide the memory latency. Chen described a memory latency hiding algorithm that combines the loop pipelining with data prefetching [28]. The algorithm divides the iteration space into regular partitions, and then produces two parts of a schedule for ALU and memory operation in a nest loop. They studied the optimal partition shape and size for a well-balanced overall schedule. Xue et al. [29] proposed a software prefetching technique that can completely hide memory latencies for applications with multi-dimensional loops on architectures like CELL processor. They presented an algorithm, iterational retiming with partition, to achieve complete memory latency hiding for multi-core architectures.

For reconfigurable computers, Lange et al. [30] described a parameterized memory system suitable as target for high-level language to hardware compiler. The system combines support for control speculation, distributed memories, and clustered coherence mechanisms. Their experimental results using four benchmarks show that control-intensive codes gains much more from speculation than other types of codes. The speculation allows the hiding of memory latencies, but is limited to 10% improvement for some benchmarks that search for random key values.

Thielmann et al. [31] presented a compiler framework capable of generating application-specific microarchitectures supporting load values speculation on reconfigurable computers. They examined the re-execution control mechanisms, the associated speculation queues that resupply the appropriate data items, and the actual speculation predicators. All the techniques have been implemented in their hardware/software co-compiler. The experimental results show a 2.48x speedup over nonspeculative implementations.

3.4 DATA REUSE

Data reuse has been widely studied for bandwidth optimization during the past. In [32], the authors introduced a formal model for data reuse analysis by applying algebraic techniques specific to the data-flow analysis used in modern compilers. The analysis allows partitioning the index space of arrays in data-dominated application such that the heavily accessed array parts are used as redundant data in scratch-pad memories to reduce the dynamic energy consumption due to memory accesses.

In [33], the authors presented a multiprocessor data reuse analysis technique to enable the designer to explore solutions that meet power and area constraints while maintaining the necessary performance level. They described a shared buffer that can be accessed by multiple processors to reduce the number of accesses to main memory.

In [34] the authors introduced data block reuse to maximize the reuse of a data block before access the next data block. Their approach was based on dividing large data structure into logical blocks and clustering computations that access the same data block using a scheduler. They implemented a software-managed on-chip memory and gave no experimental results.

Wang et al. [35] summarized the research works in [36,37,38,39,40,41] and concluded that the reuse buffer is designed for sequential execution model and thus causes memory access conflict and low throughput in pipelined loops. They also studied the work in [42,43,44,45,46,47] and pointed out that these memory mapping and partitions approach fail to work when data reuse is performed in a non-affined, circular manner to save buffer size. Based on the previous work, they presented an integrated automatic approach for combining memory partitioning and merging with data reuse and pipelining to generate a memory optimization flow for FPGA behavioral synthesis. However, memory latency hiding is not part of their memory optimization flow.

3.5 3D STENCIL

Of all kernels selected for experiment in the dissertation, the 3D stencil is the most complex kernel. In [48] Brown evaluates efficient implementations of 2D finite-difference modeling. For evaluation Brown compared FPGA, quad-core and Cell machines. The best speedup was achieved by the FPGA implementation, but the work focused on 2D stencil computations.

The hardware implementations and optimizations for 2D and 3D stencils in the FDTD simulation and seismic imaging problems were proposed by He et al. [49,50,51]. The authors proposed to exploit data reuse of some grid values by sending grid values from the input cache to a set of cascaded FIFOs. The number of FIFOs depends on kernel's order of accuracy with respect to time and space. In 3D stencil each FIFO buffers 2D pages and each page generally contains hundreds to thousands of grid points along each spatial axis. So the onboard memory bandwidth has to be sacrificed to meet the buffering requirements. The authors didn't explore the memory access ordering to improve the performance.

Most work on stencil implementations has been done in the context of processor architectures. General microprocessor based solutions for solving scientific problems in three dimensions are cache-inefficient due to the high miss-rates for the large 3D arrays. In [52] the authors showed tiling is not needed for 2D stencil codes and developed techniques to apply tiling in 3D stencil codes by selection of non-conflicting tiles and/or padding array dimensions to eliminate cache access conflicts. Leopold [53] provides an analytical treatment to the problem to show the effects of tiling in Jacobi and Gauss-Seidel kernels on a 3D array. They proved a lower bound to the capacity miss of a cache of size C for an array of size N^3. They showed that rectangular tiles outperform square tiles for row-major storage order. Kamil et al. [54] investigated the impacts of trends in memory subsystems on cache optimizations for stencil computations. They considered the blocking strategies on prefetch-enabled microprocessors. They found that cache blocking is effective only for very large problem sizes and the software and hardware prefetching improved performance of long stride-1 accesses.

The research in [55] explored double-precision 3D stencil computations on the multicores by developing numerous optimization strategies and an auto-tuning environment. To hide memory latency, they employed hardware prefetching, software prefetching, DMA, and multithreading. They explored the technique of circular buffer for both read and write planes to minimize memory traffic. The advantage of the circular queue is the potential avoidance of cache's conflict misses.

CHAPTER 4

PRELIMINARY RESULTS

This chapter presents preliminary results of DRAM scheduling, memory latency hiding using double buffer, data reuse analysis and kernel bandwidth requirement. We performed all memory performance tests on the Convey HC-1 platform. The experiment in DRAM scheduling shows how the performance of Convey's DRAM controller depends on the memory access orders. The memory latency hiding test shows double buffering is not perfect due to the interface stall rate. We also calculate theoretical data reuse rate to show the benefit of data reuse for the kernels. In addition, we characterize the bandwidth requirement of 3D stencil kernel to show that 3D stencil is a memory-bound kernel. These results provide a basis for better understanding memory interface designs and implementations of memory-bound kernel.

4.1 IMPACT OF DRAM SCHEDULING

As described in Chapter 2, the Convey HC-1's proprietary DRAM controllers attempt to maximize memory bandwidth by dynamically scheduling incoming memory requests, which results in memory accesses performed in a different order in which they are requested by the user. In order to investigate the effective bandwidth given by the memory controller with various access patterns, we designed a memory test kernel that accesses all the 65536 64-bit words accessible from a single memory controller within a single page.

There are 120 (5!) possible access orders that define the relative frequency of when the address in each of the five fields changes. For example, for the following ordering: DRAM bus, DRAM bank, DIMMs select, DRAM column, DRAM row, an access pattern would access all eight words within a bus before changing the bank address, all eight banks before changing the DIMM address, both DIMMs before changing the column address, and all 128 columns before changing the row address.

Table 4.1 DRAM controller efficiency test with 120 access orders

Access Orders	Efficiency	Access Orders	Efficiency
bank,dimm,bus,col,row	0.93	bank,col,bus,row,dimm	0.19
bank,dimm,bus,row,col	0.93	bus,row,dimm,col,bank	0.19
dimm,bank,bus,row,col	0.92	dimm,col,bus,bank,row	0.19
dimm,bank,bus,col,row	0.92	bank,col,dimm,row,bus	0.18
bus,dimm,bank,row,col	0.92	bus,col,row,bank,dimm	0.17
bus,dimm,bank,col,row	0.92	bus,row,col,bank,dimm	0.17
dimm,bus,bank,col,row	0.92	col,dimm,bus,bank,row	0.16
dimm,bus,bank,row,col	0.92	bank,col,row,dimm,bus	0.15
bus,bank,dimm,col,row	0.92	dimm,col,bus,row,bank	0.15
bus,bank,dimm,row,col	0.92	col,dimm,bus,row,bank	0.15
bus,dimm,row,bank,col	0.91	bank,row,col,dimm,bus	0.15
dimm,bus,row,bank,col	0.91	col,bus,dimm,row,bank	0.15
bus,bank,row,dimm,col	0.90	col,bus,dimm,bank,row	0.14
bank,dimm,row,bus,col	0.88	bank,col,row,bus,dimm	0.13
dimm,bank,row,bus,col	0.87	col,bus,bank,dimm,row	0.13
bank,row,bus,dimm,col	0.84	bank,row,col,bus,dimm	0.12
bus,row,bank,dimm,col	0.80	col,bus,bank,row,dimm	0.12
dimm,row,bank,bus,col	0.66	bank,row,dimm,col,bus	0.12
row,bank,bus,dimm,col	0.64	col,bus,row,dimm,bank	0.12
row,dimm,bank,bus,col	0.63	col,bus,row,bank,dimm	0.10
row,bus,bank,dimm,col	0.58	dimm,col,bank,bus,row	0.10
dimm,row,bus,bank,col	0.50	row,bus,col,dimm,bank	0.10
row,dimm,bus,bank,col	0.50	dimm,row,bus,col,bank	0.09
bank,bus,dimm,col,row	0.50	row,dimm,bus,col,bank	0.09
bank,bus,dimm,row,col	0.50	row,bus,dimm,col,bank	0.09
bus,dimm,col,bank,row	0.46	row,bank,col,dimm,bus	0.09
bank,dimm,col,bus,row	0.46	dimm,row,bank,col,bus	0.09
dimm,bank,col,bus,row	0.45	row,dimm,bank,col,bus	0.09
dimm,bus,col,bank,row	0.43	row,bank,col,bus,dimm	0.08
bank,bus,row,dimm,col	0.36	col,dimm,bank,bus,row	0.08

bus,bank,col,dimm,row	0.35	row,bank,dimm,col,bus	0.08
bus,col,dimm,bank,row	0.35	row,bus,col,bank,dimm	0.08
bank,row,dimm,bus,col	0.33	col,bank,dimm,bus,row	0.07
bus,row,dimm,bank,col	0.33	dimm,col,bank,row,bus	0.07
row,bus,dimm,bank,col	0.33	col,bank,dimm,row,bus	0.07
row,bank,dimm,bus,col	0.33	col,bank,bus,row,dimm	0.07
bus,bank,row,col,dimm	0.32	col,bank,bus,dimm,row	0.07
bus,bank,col,row,dimm	0.32	col,dimm,bank,row,bus	0.06
bus,row,bank,col,dimm	0.32	dimm,col,row,bus,bank	0.06
bank,row,bus,col,dimm	0.32	row,dimm,col,bus,bank	0.05
row,bus,bank,col,dimm	0.32	col,dimm,row,bus,bank	0.05
row,bank,bus,col,dimm	0.32	dimm,row,col,bus,bank	0.05
bus,col,bank,dimm,row	0.28	col,bank,row,dimm,bus	0.05
bus,col,dimm,row,bank	0.28	col,bank,row,bus,dimm	0.05
bus,dimm,row,col,bank	0.26	dimm,row,col,bank,bus	0.04
bank,dimm,col,row,bus	0.25	col,dimm,row,bank,bus	0.04
bank,dimm,row,col,bus	0.25	row,dimm,col,bank,bus	0.04
dimm,bank,col,row,bus	0.24	row,col,dimm,bus,bank	0.04
dimm,bus,col,row,bank	0.24	col,row,dimm,bus,bank	0.04
dimm,bank,row,col,bus	0.24	dimm,col,row,bank,bus	0.04
dimm,bus,row,col,bank	0.24	col,row,dimm,bank,bus	0.04
bus,dimm,col,row,bank	0.23	row,col,dimm,bank,bus	0.03
bank,col,dimm,bus,row	0.23	row,col,bus,dimm,bank	0.03
bus,col,bank,row,dimm	0.22	row,col,bus,bank,dimm	0.03
bus,row,col,dimm,bank	0.21	col,row,bus,dimm,bank	0.03
bus,col,row,dimm,bank	0.21	col,row,bus,bank,dimm	0.03
bank,bus,col,dimm,row	0.20	col,row,bank,dimm,bus	0.03
bank,bus,row,col,dimm	0.20	row,col,bank,dimm,bus	0.02
bank,bus,col,row,dimm	0.20	row,col,bank,bus,dimm	0.02
bank,col,bus,dimm,row	0.19	col,row,bank,bus,dimm	0.02

The DRAM controller efficiency results listed in Table 4.1 range from 93% down to 2%, demonstrating that the access ordering makes a substantial difference in effective memory bandwidth. Only 21 of the 120 access patterns achieved greater than 50% efficiency and over half of the access patterns resulted in less than 20% efficiency.

Any access pattern where the DRAM column and row were the two least frequently changed address fields achieved 92-93% efficiency, except for the following two cases which resulted in only 50% efficiency:

1. bank,bus,dimm,col,row

2. bank,bus,dimm,row,col

In these cases, contention for a single DIMM reduces memory performance.

The worst efficiency was observed for access patterns where row and column change the most frequently. All twelve of these access patterns resulted in less than 4% efficiency. However, Convey's scheduler was able to achieve good efficiency in some cases when the DRAM row was changed every cycle, as shown in the following cases:

1. row,bank,bus,dimm,col (64%)
2. row,dimm,bank,bus,col (63%)
3. row,bus,bank,dimm,col (58%)
4. row,dimm,bus,bank,col (50%)
5. row,bus,dimm,bank,col (33%)
6. row,bank,dimm,bus,col (33%)

The memory bandwidth test shows that the access ordering makes a substantial difference in effective memory bandwidth. To achieve high memory bandwidth, the memory should be accessed consecutively as much as possible given the specific memory access pattern of a kernel.

4.2 IMPACT OF DOUBLE BUFFERING ON MEMORY LATENCY HIDING

As described in Chapter 2, double buffering is the first attempt to hide memory latency. However, double buffering introduces inefficiency in memory controller interface due to contentions of the buffers for memory accesses and kernel pipeline. To measure the inefficiency, a performance counter is used to count the number of cycles where there is no memory read or write without a corresponding memory stall request. This differentiates inefficiencies in DRAM controller with inefficiencies in the memory controller interface.

The number is then divided by the number of execution cycles to obtain the interface stall rate.

We used MrBayes kernel, a memory-bound kernel, to evaluate the inefficiency of double buffering in a memory interface design. The results in Table 4.2 shows the interface stall rate depends on the size of the each buffer. The stall rate is high for very small and very large buffers and drops when the buffer is around 1K-entry deep. The interface stall rate in double buffering leads to the pursuit of a better memory latency hiding technique. As compared to double buffering, FPGA test shows circular buffering using Convey's reorder buffer has zero interface stall rate. So all the experimental results that follow assume the memory controller interface has implemented circular buffering to eliminate the stall that arises from the inefficiency of the buffering for memory latency hiding.

4.3 IMPACT OF LOOP AND ARRAY ACCESS ORDERS

In order to evaluate the performance of the memory access orders, we implemented a 3D 6-point stencil loop on the Convey HC-1. Figure 4.1 shows the stencil loop kernel and Figure 4.2 shows the block diagram of the design for one PE.

The order of address generation affects the memory access pattern, which in turn affects memory performance. We define two memory access orders, *the loop order* and *the*

```
for i=1 to x-2 do
  for j=1 to y-2 do
    for k=1 to z-2 do
        S[i, j, k] =  A[i-1, j, k] + A[i+1, j, k] + A[i, j-1, k] +
                      A[i, j+1, k] + A[i, j, k-1] + A[i, j,k+1];
    endfor
  endfor
endfor
```

Figure 4.1 6-point 3D stencil loop kernel *array order*. In the loop order, all inputs for each loop iteration are requested before proceeding to request the inputs required by next iteration. The ordering leads to nonconsecutive accesses within each loop iteration. It is not necessary to request all inputs of one iteration before requesting all inputs of the next iteration. In the array order, all

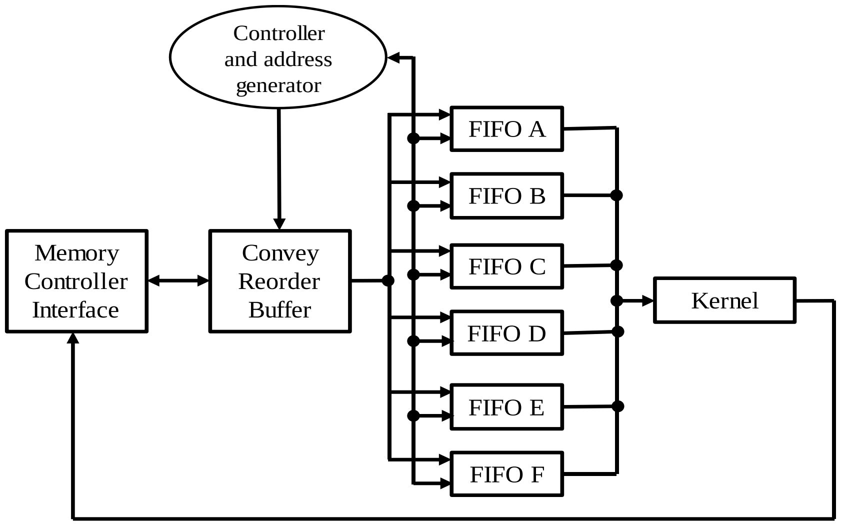


Figure 4.2 Block diagram of the 6-point 3D stencil for each PE

inputs belonging to each array are requested as needed by the iteration range covered by the buffer before moving to the next array. The ordering leads to non-consecutive access

Table 4.2 Memory interface stall rate using double buffering

Buffer size	Interface stall rate
256	20.7%
512	9.3%
1K	6.2%
2K	10.8%
3K	16.1%
4K	20.5%

for each array at the boundary of iteration range. The size of the iteration range is also referred to as *block size*.

The address generator produces the memory addresses in either the loop or array order. Convey's read reorder buffer receives data requests and returns the data to the kernel through the input FIFOs. Each of six input FIFOs corresponds to one of the six elements in the 6-point stencil. When any FIFO is almost full, the read reorder buffer will stop sending data to the FIFO. When all the FIFOs are not empty, the kernel accumulates six FIFO output data in parallel. The accumulation result is then written to the memory through the memory interface.

Table 4.3 DRAM controller efficiency of 6-point 3D stencil

Request Order	Loop	Array
Block size	1	85
Efficiency	67%	90%

The design is implemented using only two PEs accessing just one memory controller. Table 4.3 shows DRAM controller efficiency is 67% for the loop order and 90% for the array order. It is expected that the efficiency will be lower when 64 PEs of four AEs run in parallel. In the loop order, we request six elements in the innermost loop of the stencil code before proceeding to request the elements of the next loop iteration. In the array order, we request an array of elements of 85 iterations before proceeding to request next array of elements of 85 iterations. It is assumed that when block size is 1 the array order reduces to the loop order. Note this specific block size of 85 is only introduced as part of the preliminary 3D 6-point stencil result. The experimental result in Chapter 6 will take block size as a parameter for studying the impact of block size on the memory performance.

4.4 DATA REUSE ANALYSIS OF 3D STENCIL

In the stencil code shown in Figure 4.1, six references to the array A are located in the innermost loop of the nested loops. The references are R0 (A[i-1, j, k]), R1(A[i+1, j, k]), R2(A[i, j-1, k]), R3(A[i, j+1, k]), R4(A[i, j, k-1]) and R5(A[i, j, k+1]). An array element accessed by one reference in an iteration may be accessed again by other references in other iterations. For example, the array element accessed by reference R3 in iteration (i, j+1, k) can be reused by reference R2 in iteration (i, j+2, k).

As introduced in Chapter 2, data reuse is realized using on-chip buffers. If the size of on-chip buffers is large enough to store all elements of array A after requesting them from the memory, then all loop iterations can access the elements from the on-chip buffers, achieving maximum data reuse. If data reuse is not enabled, then each iteration has to access six elements from the memory to produce an output element. Since the size of on-chip buffers is limited, some redundant accesses are required to request the same elements that have been requested before.

We have built a FIFO-based model to analyze data reuse rate using different sizes of on-chip buffers and stencil space. The model counts the total number of reused memory references in the 3D stencil kernel loops and then divide it by the total number of memory references without data reuse to compute data reuse rate. When the FIFO buffer cannot hold a new memory reference due to the limited size, the oldest reference in the buffer is removed to provide space for storing the new reference.

Table 4.4 Data reuse rate of 6-point 3D stencil

	16	32	64	128	256	512	1024	4096	N^3
5	0.148	0.302	0.500	0.500	0.500	0.500	0.500	0.500	0.500
10	0.125	0.125	0.411	0.417	0.69	0.708	0.708	0.708	0.708
16	0.143	0.143	0.154	0.452	0.452	0.452	0.452	0.762	0.762
32	0.156	0.156	0.156	0.156	0.478	0.478	0.478	0.800	0.800
64	0.161	0.161	0.161	0.161	0.161	0.489	0.489	0.489	0.817
128	0.164	0.164	0.164	0.164	0.164	0.164	0.495	0.495	0.825
256	0.165	0.165	0.165	0.165	0.165	0.165	0.165	0.165	0.829
512	0.166	0.166	0.166	0.166	0.166	0.166	0.166	0.166	0.831

Table 4.5 Data reuse rate of 27-point 3D stencil

	16	32	64	128	256	512	1024	4096	N^3
5	0.198	0.442	0.689	0.829	0.829	0.829	0.829	0.829	0.829
10	0.259	0.579	0.583	0.826	0.826	0.928	0.928	0.928	0.928
16	0.275	0.616	0.619	0.619	0.855	0.855	0.945	0.945	0.945
32	0.286	0.643	0.644	0.644	0.644	0.874	0.874	0.955	0.955
64	0.292	0.655	0.655	0.655	0.655	0.655	0.882	0.882	0.959
128	0.294	0.661	0.661	0.661	0.661	0.661	0.661	0.885	0.961
256	0.295	0.663	0.664	0.664	0.664	0.664	0.664	0.664	0.962
512	0.296	0.665	0.665	0.665	0.665	0.665	0.665	0.665	0.963

Table 4.4 and Table 4.5 show data reuse rate depends on the sizes of buffer and stencil space. The first column of the table lists the sizes of the N-cube stencil space and the first row the sizes of the buffer. The sizes of the buffer range from 16 to 4096 while the sizes of the stencil space from 5^3 (N=5) to 512^3 (N=512). The last column of the table lists maximum data reuse rate for each N-cube stencil space when the buffer size is equal to N^3.

Table 4.4 lists the results of 6-point 3D stencil. When $N \geq 256$ data reuse rate stays at about 16% when the buffer size increases to 4096. This means the buffer is still too small to accommodate more references that can be reused for very large stencil space. When $N \leq 32$, data reuse rate is at maximum when $BS \leq 4096$. Notice a buffer as small as 16-entry deep can achieve the same data reuse rate as a much larger buffer for large 3D stencils. As shown in the last column, maximum data reuse rate increases monotonically from N = 16 to N = 512. The maximum rates are very close from N = 256 to N = 512. The highest data reuse rate is about 83% when N = 512.

Table 4.5 lists the results of 27-point 3D stencil. When N ≥ 256 data reuse rate stays at about 66% when the buffer size increases to 4096. Compared to 16% data reuse rate in 6-point 3D stencil, the rate is much higher, but the buffer is still too small to accommodate more references that can be reused. When N ≤ 32, data reuse rate is at maximum when BS ≤ 4096. Notice a 32-entry buffer increases more than twice data reuse rate compared to a 16-entry buffer. Maximum data reuse rate also increases monotonically. The maximum rates are very close from N = 128 to N = 512. The highest data reuse rate is about 96% when N = 512.

The results of using different sizes of buffer and stencil space provide an upper bound to the data reuse rate in a 3D stencil. In reality, there are many PEs on a multi-FPGA platform and each PE is assigned a block of stencil space to achieve high parallelism. The buffer size is also highly constrained by FPGA memory resources and timing requirement. As shown in Table 4.4 and 4.5, when the buffer size is 512, N = 10 has the highest data reuse rates.

4.5 BANDWIDTH CHARACTERISTICS OF THE 3D STENCIL

The 3D Stencil is a memory bound application. Without considering data reuse, the memory bandwidth required by the kernel to perform 3D stencil can be computed as:

$$readbytes = D \times PTS \times N^3 \qquad (4.1)$$

$$writebytes = D \times N^3 \qquad (4.2)$$

$$compute\ time = PTS / (kernelFLOPS \times F) \qquad (4.3)$$

$$Bandwidth = D \times N^3 \times (1 + 1 / PTS) \times (kernelFLOPS \times F) \qquad (4.4)$$

where *PTS* is the number of points in the stencil function, *D* the data type used in stencil, *N* the size of dimension of 3D stencil array with the assumption of a cubic array, *kernelFLOPS* floating-point operations in the accumulation stencil kernel and *F* the frequency at which the kernel executes. Assuming *kernelFLOPS* is 1 for the double-precision accumulation operation and the kernel operates at 150MHz, the required memory bandwidth for the stencil computation is approximately:

$$D \times N^3 \times 150e^6 \qquad (4.5)$$

When $N > 4$, the required memory bandwidth is larger than the 76.8 GB/s memory bandwidth on the Convey HC-1. Note the stencil kernel is still memory-bound when data reuse (i.e., $PTS = 1$) is enabled.

4.6 CONCLUSION

The preliminary results in this chapter show that the performance of Convey's DRAM controller strongly depends on the memory access orders even though it is optimized for scatter-gather operations. The access orders that achieve highest memory bandwidth imply that Convey's memory system should be accessed in sequential order for best performance. To hide memory latency, double buffering introduces high controller stall rate, which causes us to adopt circular buffering to avoid any controller stalls that lead to inefficiency of memory latency hiding. Based on the regular access pattern of a kernel, we define two memory access orders, the loop order and the array order. The array order is better than the loop order in DRAM controller efficiency for a specified block size. We also show data reuse rate depends on the sizes of the on-chip buffers and the stencil space. A 27-point 3D stencil has higher data reuse rate than a 6-point 3D stencil for a given buffer size. The next chapter will describe in detail the memory interface designs based partly on the preliminary results in this chapter.

CHAPTER 5

MEMORY INTERFACE DESIGNS ON THE CONVEY HC-1

In this chapter, we describe memory controller interface designs using two types of memory-bound kernels, 3D stencil and matrix vector multiplication. Taking advantage of the static access pattern of each kernel, we explore design techniques for improving memory latency hiding, memory access ordering and data reuse to maximize performance.

5.1 STENCILS

Stencil computations are widely used in scientific computation, such as in structured grids, partial differential equation solvers for simulation of thermo/fluid dynamics and electromagnetics. These applications are often implemented using iterative finite-difference techniques that sweep over a spatial grid performing nearest neighbor computations called stencils.

5.1.1 STENCIL COMPUTATION

As shown in Figure 5.1, a stencil calculation reads N input points to calculate one output point. In this dissertation, we chose the 6-point and 27-point stencil computation as representative kernels:

- 3D 6-point Stencil: Each point (element) in the output grid is updated by the six neighbors offset by 1 in each direction. Each point is a 64-bit number. The six

- neighbors are accumulated to generate the output point, so there are five adds for each point.
- 3D 27-point Stencil: Each point computation involves all points in a 3×3×3 cube surrounding the center output point. The 27 64-bit points are accumulated to generate the output point, so there are 26 adds for each point.

The stencil computation is based on the Jacobi (out-of-place) iterative method. We keep two separate memory arrays — one for input data and another for output data. This means that the stencil computation of each output point is independent of every other output point. As a result, they can be computed in any order. On the other hand, Gauss-Seidel (in-place) iterations require a single array for read and write. However, the dependencies in Gauss-Seidel stencil complicate the performance optimization and evaluation. This topic will be

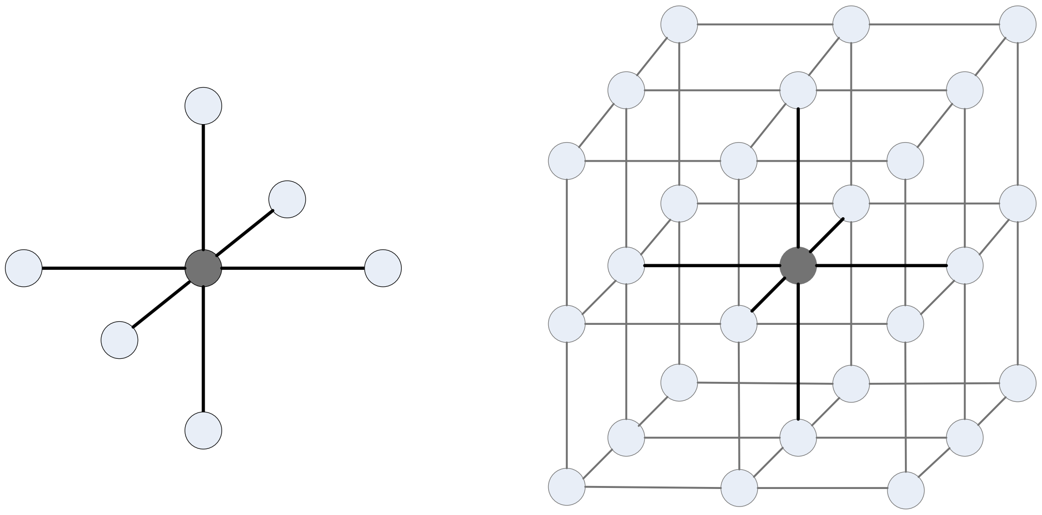


Figure 5.1 6-point and 27-point 3D stencils

left as future research.

While a 6-point stencil is simple and quite common, there are cases where larger stencils with more neighboring points are required. The 27-point stencil is a good approximation for other compute-intensive stencil kernels such as the NAS Parallel Multigrid benchmark [56]. This is why it is chosen for the dissertation.

Note all the points in our stencil computations have a scaled constant of 1, so there is no multiplication. Our focus is more on the impact of memory access patterns on the kernel performance than on the exact stencil computation of an N-point 3D stencil. The abstraction of any specific stencil computation with the accumulation provides us with the insight for implementing any stencil kernels.

5.1.2 27-POINT STENCIL KERNEL ON THE CONVEY HC-1

The preliminary result shows 6-point stencil implementation requires six 512-entry deep input FIFOs for each PE. When 16 PEs are implemented, the total FPGA memory resource used, including the memory resources for Convey's own hardware infrastructure, accounts for 77% of total memory resources on a Xilinx Virtex-5 LX330 FPGA.

For a 27-point stencil, 27 512-entry deep input FIFOs for each of 16 PEs are too large to fit on the FPGA. To implement 27 FIFOs without exceeding the FPGA memory limit, the depth of each FIFO can be reduced from 512 to 32 and both block RAMs (BRAMs) and distributed RAMs can be used. Using distributed RAM is necessary because the available BRAMs on the target FPGA are not sufficient for implementing 27 BRAM-based FIFOs per PE. Though the shallow FIFO limits the exploration space of block sizes on the Convey HC-1 platform, this will not be an issue when targeting the applications on the newer Convey HC-1ex platform. The Convey HC-1ex platform is equipped with four higher-capacity Virtex-6 LX760 FPGAs for the Application Engines.

5.1.3 PROPOSED 6-POINT STENCIL KERNEL DESIGN ON THE CONVEY HC-1

The proposed design is to combine memory latency hiding, memory access scheduling and data reuse to achieve high performance. We propose *the numerical order*, a memory access mechanism based on static access pattern, which facilitates the hardware implementations of memory latency hiding and data reuse. The idea can be explained using a small $5 \times 5 \times 4$ grid (block) in the example of 6-point 3D stencil.

Figure 5.2 illustrates a $5 \times 5 \times 4$ block composed of four planes P0, P1, P2 and P3. Each element in the planes is numbered numerically starting from 0 in P0 and ending at 99 in P3. A point in a plane is not considered as an output point if any of its N neighbors are outside the grid. As such, plane P1 has nine output points corresponding to the locations at 31, 32, 33, 36, 37, 38, 41, 42 and 43 and plane P2 has nine output points (56, 57, 58, 61, 62, 63, 66, 67 and 68). Since each output element is accumulated by N neighbors offset by 1 in each direction in the stencil computation, the 6-point stencil computation of an output point in P1 requires the neighboring elements in P0, P1 and P2 while the computation of

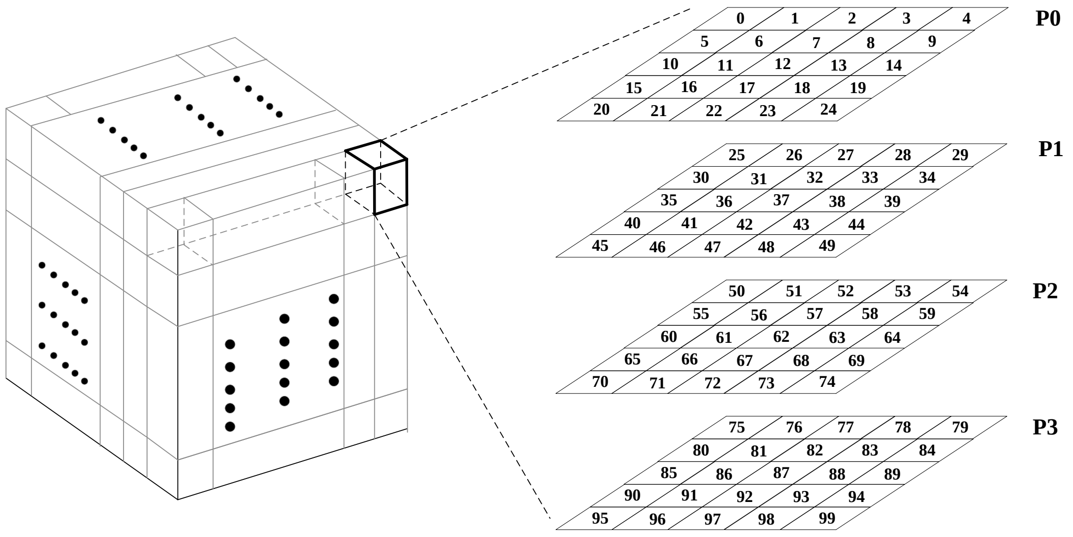


Figure 5.2 A $5 \times 5 \times 4$ grid with elements numbered numerically

an output point in P2 the neighboring elements in P1, P2 and P3. For simplicity we refer to these planes as top, middle and bottom. For example, output element corresponding to location 31 in P1 is a sum of elements at locations 6 (P0), 26 (P1), 30 (P1), 32 (P1), 36 (P1) and 56 (P2).

5.1.3.1 MEMORY LATENCY HIDING IN THE PROPOSED DESIGN

Recall the design of 6-point stencil using the loop or array order has six input FIFOs that receive six elements for computing an output element. Each FIFO receives the appropriate data with the FIFO select control signal. The control signal specifies which of the six FIFOs the requested element from the memory interface will be stored into. When the address generator requests the elements numerically, it is not feasible to have a large decoder to decode the location of each element to know which FIFO the element will be stored into. So all the requested elements have to be stored in the same on-chip buffer. On the other hand, if all the elements are stored in one buffer with a single read port, then it would take six cycles to read six elements for computing an output element, decreasing the computation throughput of the kernel and thus requiring a very large buffer to store the elements.

In order to effectively hide memory latency, we duplicate the buffer six times for the 6-point stencil, so each buffer receives the same data from the memory controller interface. Then the kernel can read six elements from the buffers in one cycle, increasing the buffer read throughput. Moreover, because the size of each buffer is limited, each buffer is a rotation buffer (RTB) that is read and written simultaneously to hide memory latency without introducing double buffering's stall rate. Figure 5.3 shows the block diagram of the proposed design.

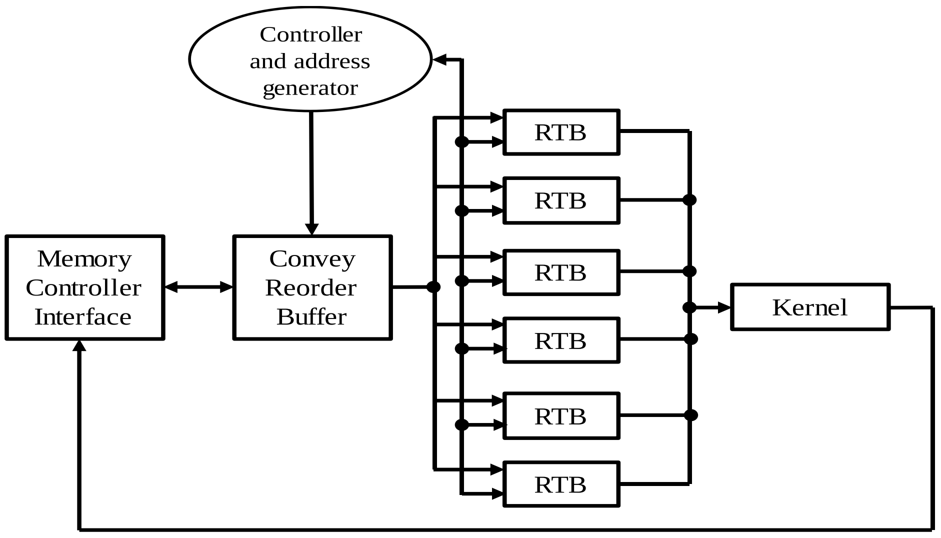


Figure 5.3 Block diagram of the proposed 3D 6-point stencil for each PE

As compared to the FIFO, each of the rotation buffers can be randomly addressed like static random access memory (SRAM), which facilitates simultaneous accesses of six elements at different memory addresses. In addition, unlike FIFO the data in RTB is still available after reading as long as it is not overwritten, which facilitates data reuse of the elements for computing the output elements. The RTBs always store the requested elements from the memory in the numerical order. To read six elements for computing an output element just as dequeueing six elements from six FIFOs, we must know where in the RTBs the kernel reads the data. The location (address) of reading each element in the buffer is specified by the numerical order of each element in a grid. For example, to compute the output element corresponding to location 31, six RTBs are accessed at the addresses 6, 26, 30, 32, 36 and 56 respectively. Thus each output element can be computed by reading the buffers at the numerically ordered addresses. However, it is not hardware efficient for the address generator to store in memory the read addresses for each output.

To optimize the address generation, we use relative addressing based on the static access patterns of 3D stencil.

Observe in Figure 5.2 that if address 31 of the output point is assumed as the reference address, then the differences between the reference address and the addresses of six neighbors are -25, -5, -1, 1, 5 and 25. It can also be observed that the address differences (offsets) are the same given the size of a plane when computing any output element.

For an N-point stencil, we take advantage of the static nature of the address differences to simplify the implementation of read addresses generation. As long as the location of an output element and the size of the plane are known, the addresses of the N neighbor can be calculated using the relative addresses without knowing their exact addresses. It should be noted, however, that the circular nature of the RTB requires any address be calculated in modular arithmetic.

5.1.3.2 MEMORY ACCESS ORDERING IN THE PROPOSED DESIGN

The numerical order does not imply the elements numbered numerically are stored in the memory consecutively or in the array order. In a large stencil space that is divided into subspaces for parallel computations among the PEs, the elements in each row of a plane are stored in the array order, but there is an address stride from the end of a row to the start of the next row in a plane. For example, in plane P0 as shown in Figure 5.2, elements at location 4 and location 5 are not stored in memory consecutively.

Compared to the loop or array order, the address generator of each PE requests the elements in a block in the numerical order. This means every row of a plane in a block is accessed in the array order.

Just as a block is composed of a set of planes, the entire stencil space is divided into many blocks as shown in Figure 5.2. On a multi-FPGA platform, these blocks are assigned to all PEs for highly parallel operations. To divide the entire workload among 64 PEs of four AEs on the Convey HC-1 platform, we can partition 3D stencil space along I, J and K dimensions. The experimental results in the next chapter will describe these partitions and their impact on the performance.

5.1.3.3 DATA REUSE IN THE PROPOSED DESIGN

In the memory interface designs with the loop or array order, the address generator requests six times the number of output elements so that each output element is computed using six elements. In this case, there is no data reuse and computing all output elements in the $5 \times 5 \times 4$ block requires 108 ($9 \times 2 \times 6$) requests, 6 requests for each of 18 output elements. The numerical order facilitates data reuse in that the address generator can request all the

Table 5.1 Reuse rates of using the loop, array and numerical orders

Num. of Planes	Numerical order	Loop or array order	Reuse rate
3	75	54	0.00
4	100	108	0.07
5	125	162	0.23
7	175	270	0.35
8	200	324	0.38
32	800	1620	0.51
53	1325	2754	0.52
87	2175	4590	0.53
104	2600	5508	0.53
172	4300	9180	0.53
257	6425	13770	0.53
512	12800	27540	0.54

elements in the $5 \times 5 \times 4$ block for computing 18 output elements in planes P1 and P2. In this case, the number of requested elements is 100 ($5 \times 5 \times 4$) elements.

Table 5.1 compares the number of elements that need to be requested in the loop, array and numerical orders for computing all the output points in a set of 5×5 planes. For a small $5 \times 5 \times 3$ grid, the numerical order requests more elements than the array or loop order does, so the reuse rate is 0. However, as the size of the grid increases the numerical order reduces the number of requests by around 50% compared to the loop or array order. Since the size of each buffer is limited, it cannot hold all the data and has to be rotationally updated with the new data. The update is based on the knowledge of regular access patterns of a grid in the stencil space. As the buffers always store data in the numerical order, the old data in a buffer can be updated with new data when all elements for computing all output elements in a plane have been read by the kernel. For example, after all elements in P0 for computing all output elements in P1 have been read from the buffers, all elements in P0 in the buffers can be replaced safely. In the course of reading RTBs from the kernel and updating RTBs from the memory interface, we must read and write the buffer in a way such that the stencil computation is performed correctly.

Assume a block assigned to each PE is composed of four 5×5 planes. After each RTB has stored three planes of elements, the kernel can start stencil computation in P1 by reading simultaneously six RTBs. Since there are nine output elements in P1, it takes nine cycles to read all input elements with each cycle reading six input elements for one output element. After nine cycles of reading all 54 (6×9) elements, the kernel can start reading six input elements for the stencil computation in P2 if a plane of elements for P3 has been stored in RTB. As the size of a plane is 5×5, it takes at least 25 cycles to store a plane of elements for P3 in RTB, so the kernel cannot start stencil computation in P2 when RTBs are still receiving elements of P3. In other words, the read control logic must ensure that stencil computation is not started until three planes of elements are ready in the buffers. So the size of the buffer is at least three times the number of elements in a plane. For this example, the size of RTB could be 25×3 when a block is composed of 5×5 planes. Note it is possible to start stencil computation a little early in P1 when the buffers are still receiving the last few elements of the 5th row of P2. However the read control logic is simplified when stencil computation is started after three planes of elements are ready because we do not need to know when the remaining elements will be delivered from memory to RTB. This does not decrease the performance as RTB read is faster than RTB write.

As describe in the last paragraph, when a RTB contains three planes for stencil computation in the current plane (e.g., P1), it is also receiving data elements of P3 that will be used for stencil computation in the next plane (e.g., P2). When the buffer size is three times the number of elements in a plane, the received data have to be written in the buffer region that contains top of three planes. To avoid the possible cases that elements received overwrite the elements that have not been read in the stencil kernel (e.g., 27-point stencil) when overlapping read and write in RTB, the size of the buffer is increased to four times the number of elements in a plane. As long as RTB read time for all output elements in each plane (e.g., P1) is faster than the time to update RTB with a plane (e.g., P3), it guarantees that data written in the RTB region that does not contain top of the three planes being read for stencil computation.

Unfortunately, the maximum size of each buffer is limited by the available memory resources on the target FPGA. Given the constraints of 16 PEs per AE, with six or more

RTBs and one output FIFO per PE, our experiment shows the maximum size of each buffer is 512 while the size of output FIFO is 2048 for 6-point stencil and 1024 for 27-point stencil. The size of the buffer provides us with a little more room for evaluating the performance impact of a relatively large plane.

5.1.6 PROPOSED 27-POINT STENCIL KERNEL DESIGN ON THE CONVEY HC-1

The 27-point stencil design is more challenging. While it is feasible to have 27 buffers, as implemented using 27 32-entry deep FIFOs in the loop and array order designs, the size of each buffer is too shallow to store three planes of elements. To reconcile limited memory resources with the buffer size requirement, we choose nine BRAM-based 512-entry deep buffers for each PE. While six elements can be read in one cycle using six buffers in 6-point stencil, it takes three cycles to read 27 elements using nine buffers.

For a relatively large $10\times10\times4$ grid, each plane (P1 or P2) has 64 output elements. It takes 100 (10×10) cycles for RTBs to be written with a plane of elements. On the other hand, as three cycles are required to read 27 input elements from nine RTBs, it would take 192 (64×3) cycles to read RTBs for all output elements in a plane (e.g., P1). Since RTB write is faster than RTB read, RTB overflow would eventually occur.

If RTB read time can be reduced below 100, then the overflow can be prevented from happening. Notice the buffer read time is not only constrained by the number of buffers, but also related with the computational capacity of the kernel. If we can increase kernel's computational capacity, we can reduce RTB read time by reusing the data without reading them again from RTBs.

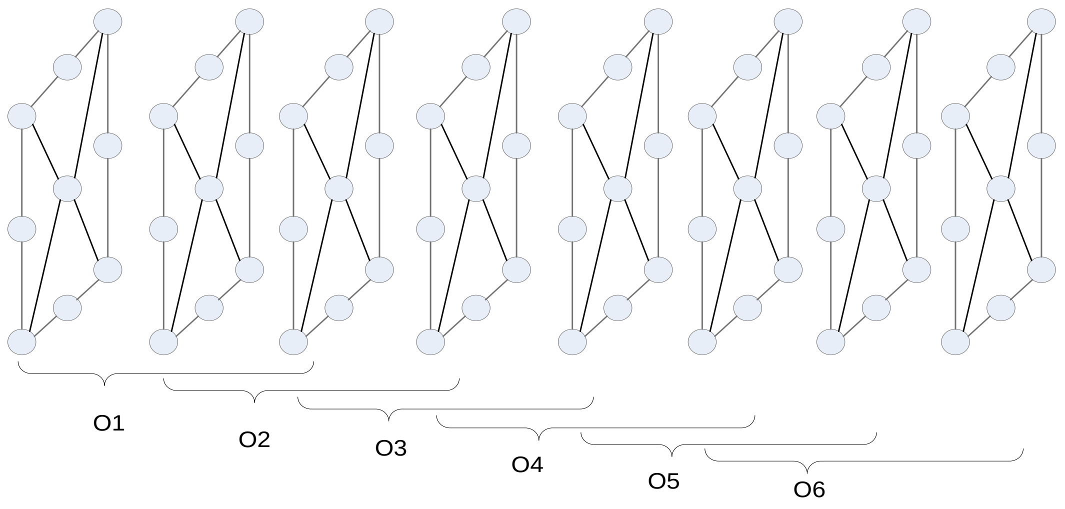


Figure 5.4 The stencil computation of six output elements in 27-point 3D stencil

Figure 5.4 shows a 3×8×3 block composed of eight planes parallel to each other vertically. The output element O1 is accumulated by reading nine input elements of the

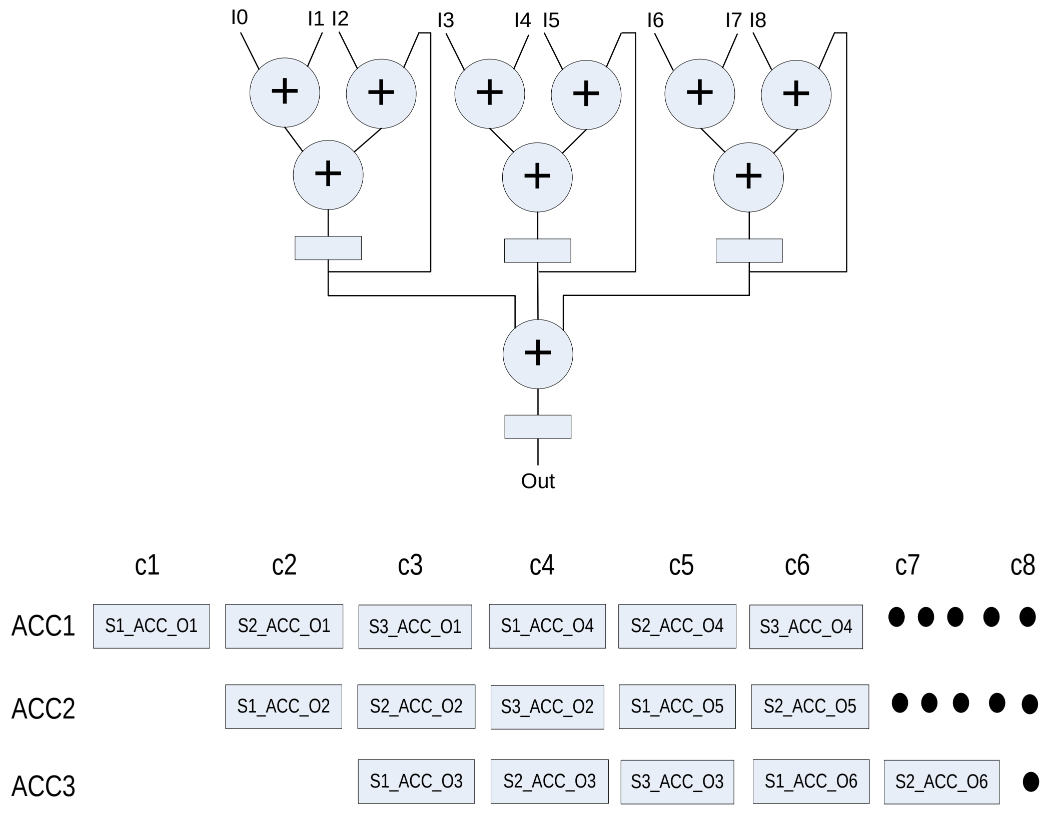


Figure 5.5 Block diagram of one of the three accumulators which operate in parallel

first plane followed by nine inputs in the second plane and then nine inputs in the third plane. When the second or third plane is read from RTBs and delivered to the kernel for computing the first output element O1, it is also used for computing the second output element O2. By increasing the computational capacity of the kernel to share RTB read data, it takes eight cycles to read all input elements for six output elements as compared to 18 cycles.

There are two logic designs for reading and accumulating 27 inputs for each output element as shown in Figure 5.4. The first design instantiates three 9-input accumulators. Figure 5.5 shows the block diagram of a single 9-input accumulator and the timing diagram of three 9-input accumulators operate in parallel. The functional unit receives nine 64-bit inputs (I0 to I8) from the buffers and three inputs from the intermediate registered results. To increase the kernel's computational capacity, we instantiate three accumulation units. The timing diagram shows how three functional units operate in parallel. The first unit (ACC1) starts to read nine inputs and accumulates them on cycle 1 (c1). This is the first-stage accumulation for output element 1 (i.e., S1_ACC1 _O1). On the next cycle (c2) the second unit (ACC2) starts while ACC1 is performing the second-stage accumulation. On cycle 3 the third unit (ACC3) starts while ACC1 and ACC2 are performing the third-stage and second-stage accumulations respectively. After three-cycle latency, ACC1 finishes reading all inputs for O1 and begins to read and accumulate the inputs for output element O4. Each accumulator is fully utilized during the stable phase. The latency of accumulating 27 64-bit inputs is four cycles, three cycles for reading 27 inputs plus one cycle to produce the final result. This is the minimum latency under the 150MHz timing requirement.

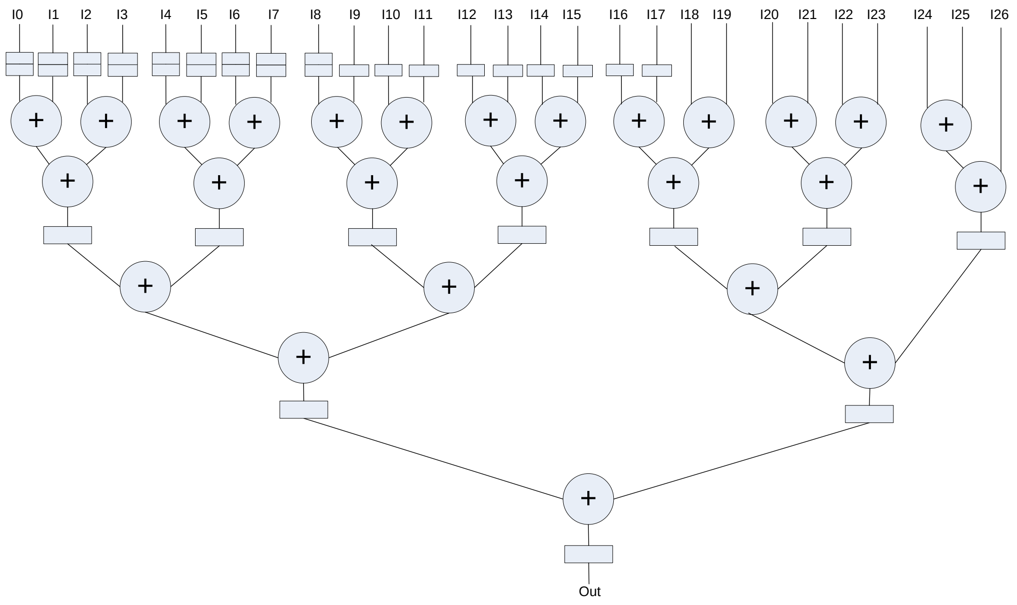


Figure 5.6 Block diagram of 27-input 64-bit pipelined adder tree

As shown in Figure 5.6, the second design has a 27-input pipelined adder tree with certain inputs buffered appropriately for reusing the read data. It accumulates every 27 input elements when they are ready. Comparing both designs in terms of the number of adders and registers (assuming each register or adder is 64-bit), the first one has 12 registers and 33 adders while the second one 37 registers and 26 adders. There is a tradeoff between the number of registers and the number of adders. However, the resource usage depends on how each design is implemented and optimized by the EDA tools on the FPGA.

By reusing the RTB read data and increasing the computational capacity of the kernel, RTB read time is reduced from 192 cycles to 66 cycles. Another approach to prevent buffer full is to stall the buffer write before buffer write full occurs. The drawbacks of this approach are the complexity of the buffer control logics, and the introduction of rotation buffers stall that degrades the performance of the design.

Though the accumulation-based stencil computation is simpler than other stencil computations that involve weighted multiplications and additions, the proposed 3D stencil designs provide insights for solving the issues that arise from the combination of memory latency hiding, memory access scheduling and data reuse.

5.2 MATRIX VECTOR MULTIPLICATION

Matrix vector multiplication (MVM) is a fundamental operation in linear algebra. Assuming C, A, and B are matrices of dimensions $M \times 1$, $M \times N$ and $N \times 1$. The standard algorithm that implements MVM is shown in Figure 5.7:

The algorithm shows that each C[i] is the dot product (inner product) of the *i*th row of matrix A with matrix B (i.e., vector B). Our MVM kernel ignores matrix C writes so that we can explore the impact of maximum data reuse of matrix B for each PE upon memory performance. The implementation of matrix C write will be left for future research.

```
for i=1 to M do
   for k=1 to N do
    $C[i]= C[i] + A[i, k] \times B[k]$;
   endfor
endfor
```

Figure 5.7 The standard algorithm of MVM

5.2.1 BANDWIDTH CHARACTERISTICS OF MVM

The memory bandwidth required by the kernel to perform MVM is:

$$readbytes = D \times N^2 + D \times N \tag{5.6}$$

$$writebytes = D \times N \tag{5.7}$$

$$compute\ time = (2 \times N - 1) \times N / (kernelFLOPS \times F) \tag{5.8}$$

$$bandwidth = (D \times N + 2 \times D) / (2 \times N - 1) \times (kernelFLOPS \times F) \tag{5.9}$$

where *D* is the data type used in MVM, *N* the column size of the matrix *A* assuming it is square (M = N), *kernelFLOPS* kernel’s floating-point operations per cycle and *F* the frequency of the kernel. To maintain a balanced system, the total memory bandwidth must scale with the computational throughout [57]. The Convey HC-1 has an aggregate memory bandwidth of 76.8 GB/s and operates at 150MHz. Then *kernelFLOPS* of the double-precision floating-point MVM for 64 PEs is approximately:

$$(128 \times N) / (N + 2) \tag{5.10}$$

or two floating-point operations per PE for very large N. The target Virtex-5 LX330 FPGA is large enough to provide the kernel with the resources for two double-precision floating-point operators per PE.

5.2.2 DATA REUSE IN MATRIX VECTOR MULTIPLICATION

For simplicity of description we assume M and N are 4 in the standard MVM algorithm as shown in Figure 5.7, so four references to the 4×4 matrix A and the 4×1 matrix B are located in the innermost loop of the nested loops. The references are A0(A[i,1]), A1(A[i,2]), A2(A[i,3]), A3(A[i,4]), B0(B[1]), B1(B[2]), B2(B[3]), and B3(B[4]). An element accessed by one reference in one iteration is accessed again by references in other iterations. For example, the element accessed by B0 in iteration i is accessed again in iteration i+1 for the dot product. Furthermore, each element in matrix B is reused by matrix vector multiplication for all iterations, so data reuse of matrix B is at maximum.

Without data reuse, each PE has to request the elements of matrix B every time the inner product is performed. When the workload of MVM is divided among multiple PEs, data reuse is implemented by each PE requesting part of the elements of matrix B once, and then reusing them after storing them in on-chip buffers. On the Convey HC1, the reuse rate is 0.5 – 8/M assuming there are 16 PEs in each AE and M is the row dimension of matrix

A. This shows there is no data reuse until the row size of matrix A is larger than 16. On the other hand, the reuse rate theoretically reaches the maximum rate of 0.5 if M is infinite.

5.2.3 PROPOSED MVM DESIGN ON THE CONVEY HC-1

The basic challenge in optimizing MVM is to reduce memory accesses of off-chip memory when computing very large matrices and vectors that do not fit in limited on-chip memory. The effective approach to reducing the off-chip memory accesses is to exploit reuse through blocking [58]. MVM is highly parallelizable since all the dot product calculations are independent. Blocking divides a large MVM into smaller MVMs that can fit within the limited on-chip memory constraint. The following C-style algorithm gives an example of blocking in MVM, where M and N are the dimensions of matrix A and the block size is $NB \times BS$ where NB is the number of rows and BS the number of columns in the block.

```
void block_mvm_kernel
(Data* A, Data* B, Data* C, int N, int NB, int BS) {
 int i, k;
 for (i = 0; i < NB; i++)
   for (k = 0; k < BS; k++)
    C[k] += A[i, k] * B[k];
}

void block_mvm
(Data* A, Data* B, Data *C, int M, int N, int NB, int BS){
 int i, k;
 for (i = 0; i < M; i += NB)
   for (k = 0; k < N; k += BS)
    block_mvm_kernel(&(A[i, k]), &(B[k]),&(C[i]), N, NB, BS);
}
```

To divide the workload of MVM on the Convey HC-1 platform, four AEs execute the block MVMs in parallel and access the entire memory through Convey's crossbars. Each AE is assigned a block of matrix A of size $NB \times BS$ and BS elements of matrix B. Since each AE can accommodate 16 PEs on the Convey HC-1, we fully utilize memory bandwidth from the AEs to coprocessor memory by assigning each PE in an AE a subrow of size BS in matrix A and a column of size BS in matrix B. So all PEs in an AE access a block of size 16 $\times$BS.

Figure 5.8 shows the matrices are divided among four AEs in shaded gray and each AE has 16 PEs and each PE accesses its local row elements in matrix A and its local column elements in matrix B. Without data reuse, the address generator in each PE produces the row addresses that request a subrow and then the column addresses that request a subcolumn. Since there are 16 PEs and 4 AEs on the Convey HC-1, the number of memory requests for each PE is $(N/16) \times (N/4) \times 2$.

Notice all subrows in a group assigned to a PE (e.g., PE0) in matrix A require the same column elements in matrix B for inner product operation. To achieve data reuse in matrix B, the address generator is optimized to skip the redundant column requests if the column

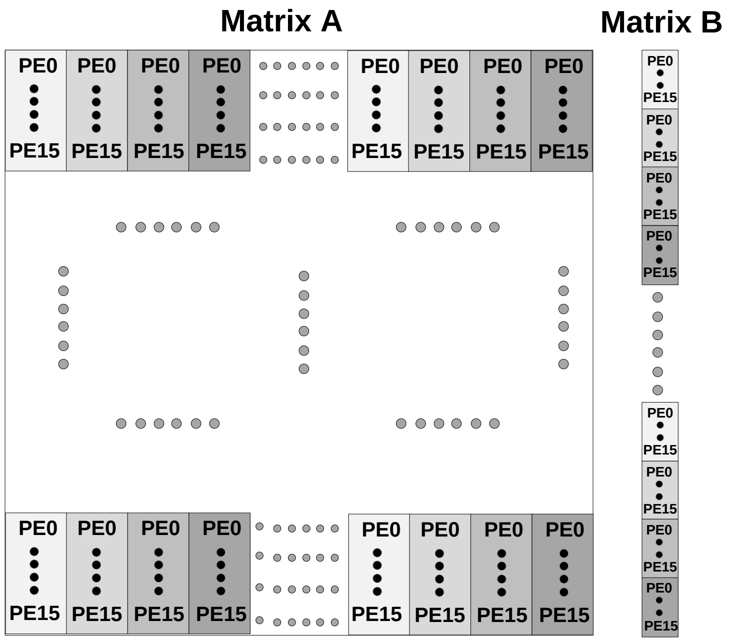


Figure 5.8 Matrix blocks assignments among four AEs with 16 PEs per AE

elements have already been requested before. The elements that correspond to the skipped requests can be obtained by reading the on-chip buffers that store the reused column data. Since column data are reused for every subrows assigned to a PE, the number of memory requests of each PE is $(N/16) \times (N/4) + N/4$.

To support column data reuse a column reorder buffer is integrated into the memory controller interface. The controller interface has implemented Convey's read reorder buffer as described in Chapter 2. The size of the column reorder buffer is the same as the size of the read reorder buffer, which is 512-deep and 72-bit wide. Unlike Convey's read reorder buffer which receives all the requested data from the memory, the modified read reorder buffer (MVM row ROB) only receives row data of Matrix A while the column reorder buffer (MVM column ROB) only receives column data of Matrix B. The distinction is made possible by the row/column select control sent by the address generator to the memory control interface when requesting the data. Compared to MVM column ROB that reuses column data, MVM row ROB does not reuse row data.

Figure 5.9 shows the block diagram of the proposed design for MVM. The multiplexer (MUX) selects the row or column data with the knowledge of row and column access patterns in the block MVM. For each PE, the MUX selects BS elements from the row

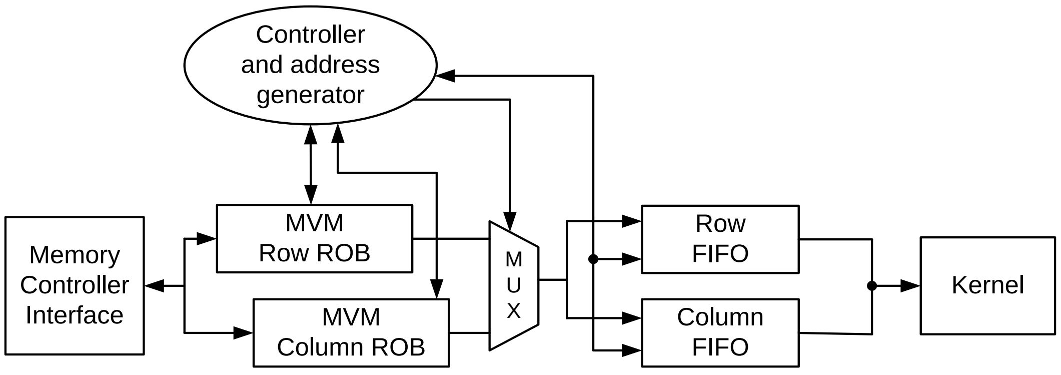


Figure 5.9 Block diagram of the proposed MVM for each PE

reorder buffer and then selects BS elements from the column reorder buffer. The MUX repeatedly selects between row and column data until the PE finishes all the inner products in the blocks assigned to the PE. The MUX output is delivered to the kernel through the row and column FIFOs. When both FIFOs are not empty, the kernel reads a row element and a column element, multiplies them to get the product and saves it for inner product operation.

As shown in Figure 5.8, all subrows in a group assigned to a PE (e.g., PE0) in matrix A require the same column elements in matrix B for inner product operation. After the requested column elements for the group are stored in the column reorder buffer, they are delivered to the column FIFO through the MUX every time the row reorder buffer has sent all row elements in a subrow to the row FIFO. After all the subrows in a group assigned to a PE are sent to the kernel, the column reorder buffer finishes reusing the column elements of the group. This process then repeats for each group of subrows and column elements assigned to the PE. Because the size of the column reorder buffer is limited, the buffer has to be updated in a circular manner. When the column elements have been reused by all subrows in the group, the spaces that are occupied by the column elements can be freed for receiving new column elements in the next group.

5.3 CONCLUSION

The chapter describes the proposed memory control interface designs for two types of memory-bound kernels, 3D stencil and MVM. Memory latency hiding is achieved by using custom reorder buffers and FIFOs. Three memory access orderings, the loop, array and numerical orderings, are defined in 3D stencil kernels. In stencil data reuse is achieved inside a 3D block composed of a set of planes for each PE. The reuse rate depends on the size of the 3D block and the size of on-chip buffer. The proposed numerical order facilitates the implementations of memory latency hiding and data reuse in the stencil kernel. In MVM each PE requests a subrow and a subcolumn in the array order. The sizes of the subrow and subcolumn depend on the number of AEs and the dimensions of the matrix. Each PE then performs inner product on the row and the column. The column can be reused for all iterations in a block assigned to a PE, which maximizes data reuse. The differences in blocking and data reuse for each kernel require the memory interface design tailored for each kernel to maximize performance. The next chapter will present the experimental results of the designs.

CHAPTER 6

EXPERIMENTAL RESULTS

In Chapter 4, we compared the performance of the loop and array orderings using two PEs and one AE. In Chapter 5, we described the proposed designs in details. In this chapter, we present and discuss the experimental results of the 3D stencil and MVM using 64 PEs on four AEs. Each PE runs at 150 MHz and the crossbar is enabled for each PE.

6.1 PERFORMANCE IMPACT OF THE LOOP AND ARRAY ORDERS

We evaluate the performance of 3D stencils with the loop and array orders by using a large 3D space of 512×512×512. Since there are 64 PEs, we divide the stencil space along one of the K, J and I dimensions for workload assignment. Figure 6.1 illustrates the partitions of stencil space along the J dimension (left) and I dimension (right). The partition along the K dimension is not shown, as it is very similar to the partition along the J dimension. Each PE is assigned a chunk of stencil space. For the J-dimension partition, each PE is assigned a chunk of size 512×8×512 (k=512, j=8, i=512) except PE63 which is assigned a chunk of size 512×6×512. For the I-dimension partition, each PE is assigned a chunk of size 512×512×8 except PE63 which is assigned a chunk of size 512×512×6. We assume the stencil computation for each PE is based on the algorithm shown in Figure 4.1 where the innermost loop index is *k* and the outmost loop index *i*.

Figure 6.2 shows interface efficiency (black line) and DRAM controller efficiency (gray line) of 6-point stencil when the stencil space is partitioned along the K dimension. As mentioned before, the loop order corresponds to a block size of 1 while the array order has

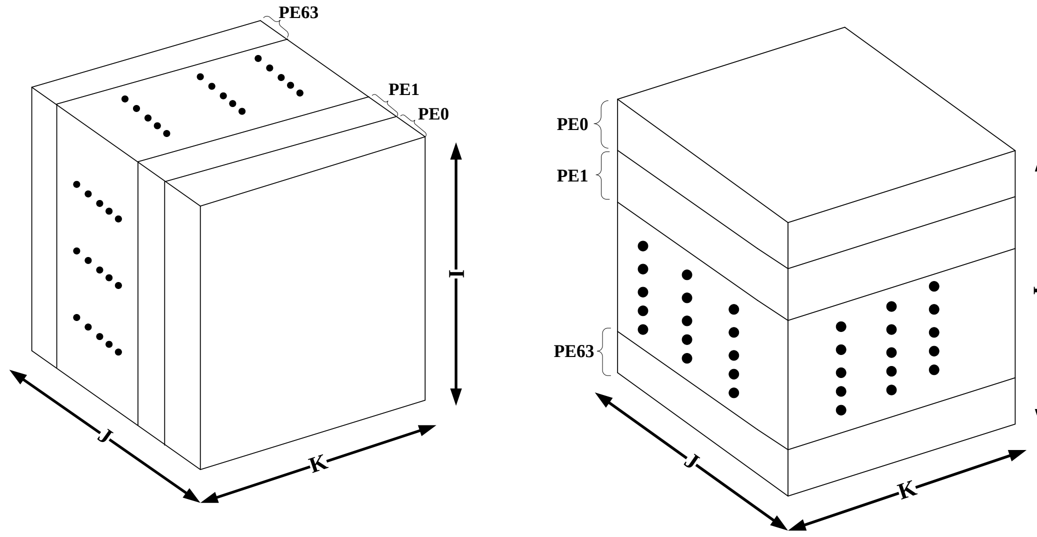


Figure 6.1 Partitions along the J (left) or I (right) dimensions in 3D stencil space

parameterized block size. The range of block sizes is from 1 to 500. For each block size, the figure plots the corresponding efficiency results. As shown in the figure, DRAM

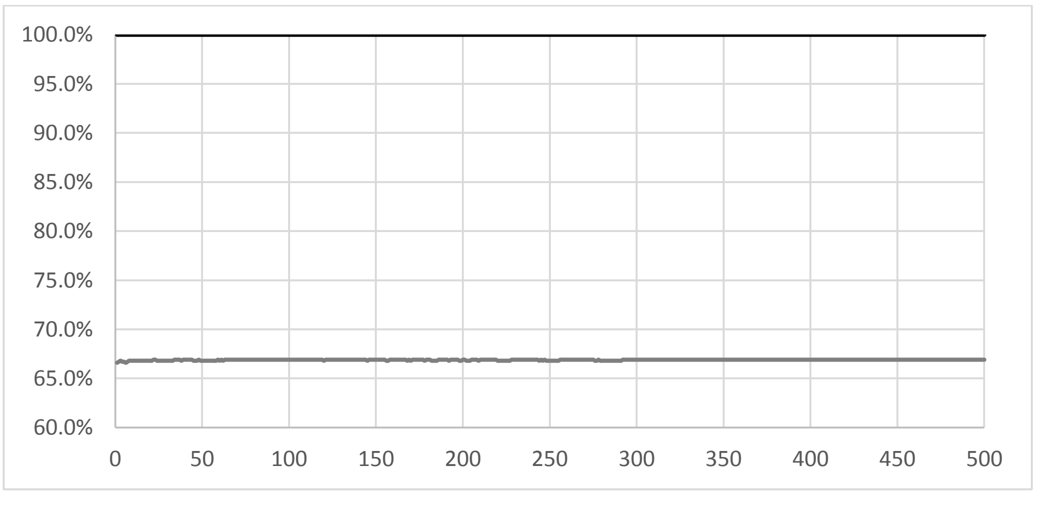


Figure 6.2 Efficiency vs. the block sizes (K-dimension partition, 6-point stencil)

controller efficiency stays close to 66.8% while interface efficiency is almost 100% for any block size.

Figure 6.3 shows the efficiency results of 6-point stencil when the stencil space is partitioned along the J dimension. As shown in the figure, DRAM controller efficiency is around 68.7% for the loop order and increases to about 74% as the block size reaches around 100. Interface efficiency is almost 100% except at block sizes of around 32 and in

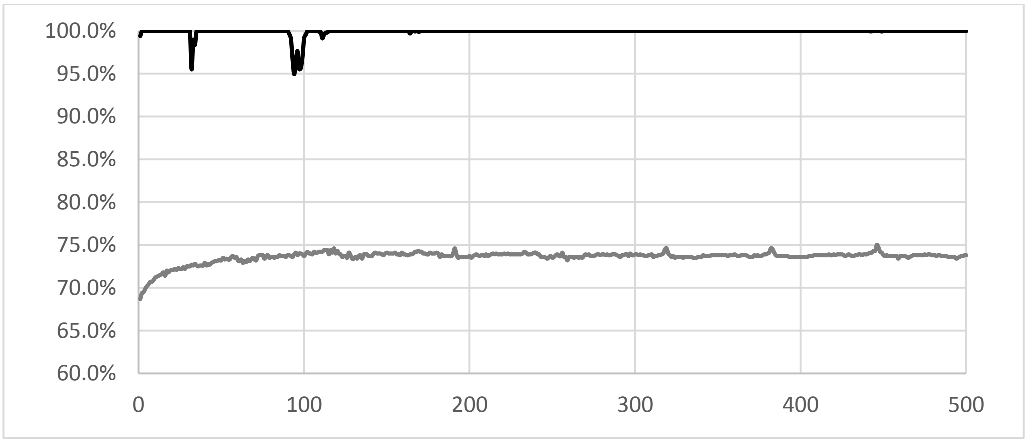


Figure 6.3 Efficiency vs. the block sizes (J-dimension partition, 6-point stencil)

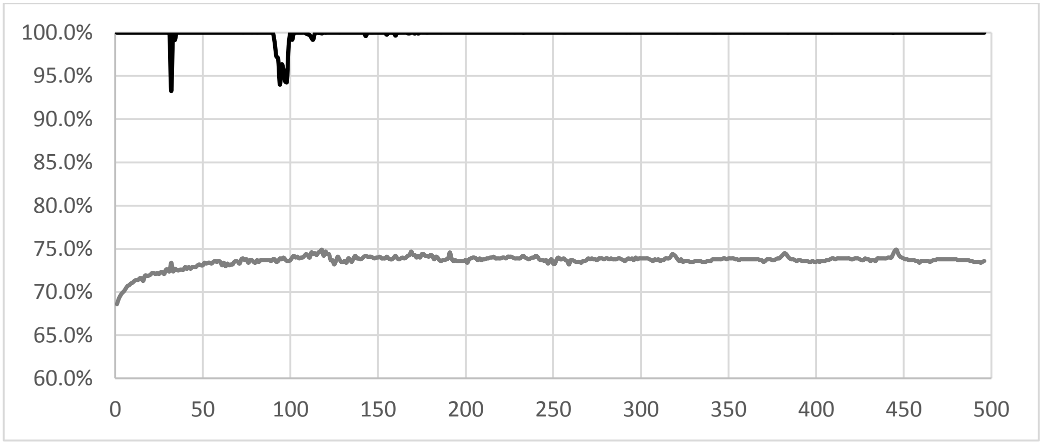


Figure 6.4 Efficiency vs. the block sizes (I-dimension partition, 6-point stencil)

between 90 and 100 where they drop by 3% to 5%. When the block sizes are larger than 100, DRAM controller efficiency results level off.

Figure 6.4 shows the efficiency results of 6-point stencil when the stencil space is partitioned along the I dimension. Notice the results are similar to those in Figure 6.3. DRAM controller efficiency is around 68.6% for the loop order and increases to about 74% as the block size reaches around 100. Interface efficiency is almost 100% except at sizes in between 90 and 100 where it drops by 3% to 6%.

Figure 6.5 compares the averaged runtime in cycles of all PEs in an AE for K-dimension (black line), J-dimension and I-dimension partitions in 6-point stencil. The runtime of K-dimension partition, above 29 million cycles, is about 16% higher than the runtime for J-dimension or I-dimension partition when the block size is larger than 100. The runtime of J-dimension partition is very close to the runtime of I-dimension partition over the block sizes, and they decrease from about 28 million cycles to about 25 million cycles when the block size increases from 1 to 100. Then the runtime slightly fluctuates at 25 million cycles when the block sizes are larger than 100. The minimum runtime is about 24.3 million cycles.

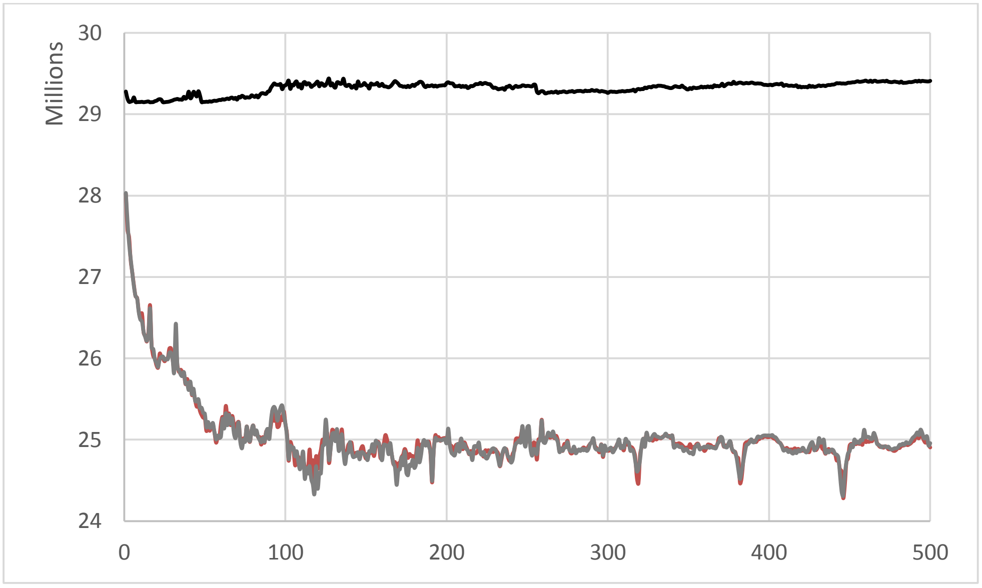


Figure 6.5 Runtime vs. the block sizes in 3D 6-point stencil

Comparing these figures shown above, we see that interface efficiency is almost 100% regardless of the way the stencil space is partitioned and the block sizes. DRAM controller efficiency increases from 68.6% to 74% for the partitions along the J or I dimension when the block sizes increase. However, DRAM controller efficiency is almost not affected by the block sizes and stays at around 66.8% for the partition along the K dimension. Though DRAM controller efficiency results are close for three partitions, the K-dimension partition does not achieve the highest DRAM controller efficiency given the elements are stored consecutively along the K dimension. For the 6-point stencil, none of the six memory accesses of six input points are consecutive for each output point, but there are more consecutive memory accesses as the number of stencil computations along the K dimension increase. The more consecutive memory accesses to the DRAM controllers, the fewer number of memory access stalls. Compared to the partitions along the J and I dimensions in which the size of the K dimension of the chunk is 512, the K-dimension partition reduces the number of consecutive memory accesses to the DRAM controllers, but the impact is quite small.

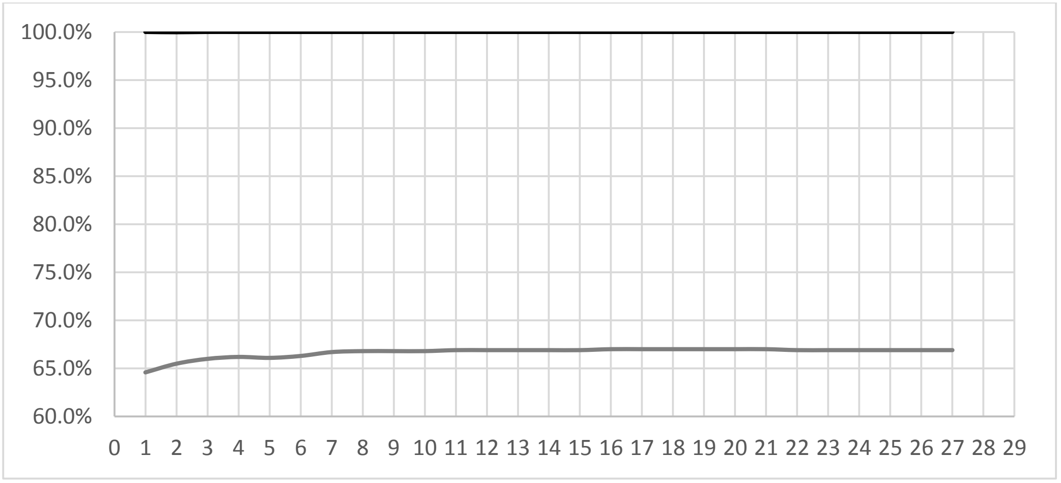


Figure 6.6 Efficiency vs. the block sizes (K-dimension partition, 27-point stencil)

We apply the same partitions to 27-point stencil space and the results are shown from Figure 6.6 to Figure 6.8. The range of block sizes is from 1 to 27 due to the memory constraint as described before. For each block size, the figure plots the corresponding efficiency results.

For the partition along the K dimension, Figure 6.6 shows DRAM controller efficiency increases very slightly from 64.6% to 67% and stays at around 67% for larger block sizes. Interface efficiency is almost 100% for any block sizes.

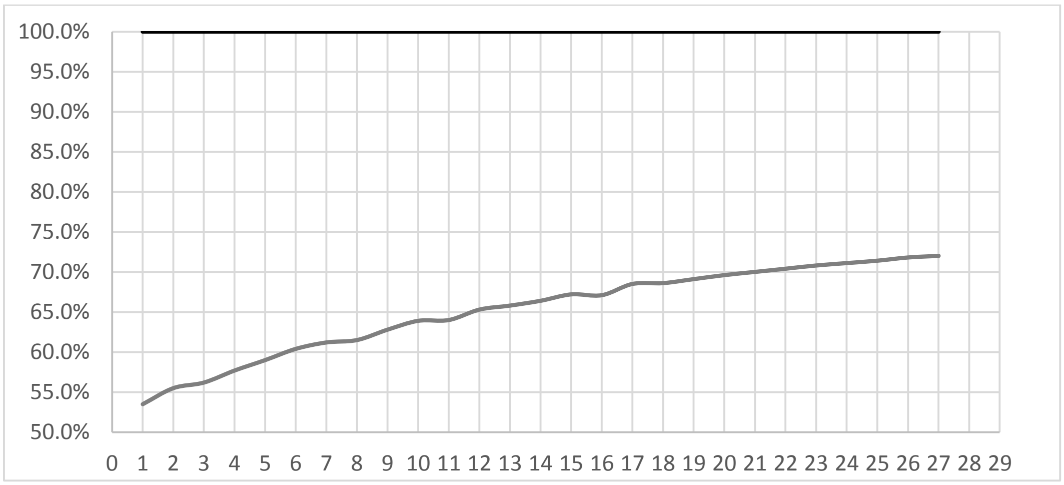


Figure 6.7 Efficiency vs. the block sizes (J-dimension partition, 27-point stencil)

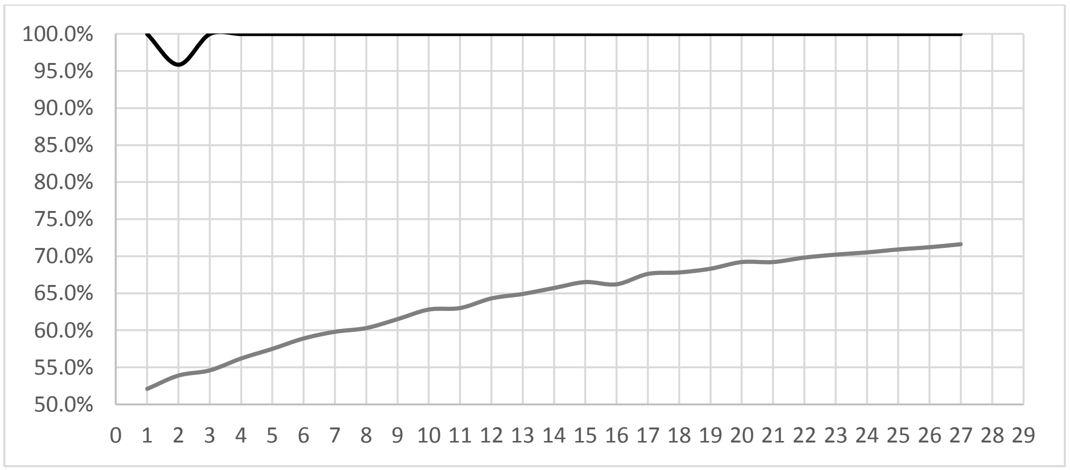


Figure 6.8 Efficiency vs. the block sizes (I-dimension partition, 27-point stencil)

For the partition along the J dimension, Figure 6.7 shows DRAM controller efficiency increases from 53.5% to 72% almost monotonically while interface efficiency is almost 100% for any block sizes.

For the partition along the I dimension, the results shown in Figure 6.8 are very similar to those in Figure 6.7. DRAM controller efficiency increases from 52.1% to 71.6% almost monotonically while interface efficiency is almost 100% for any block sizes except a 4% drop when the block size is 2.

Figure 6.9 compares the runtime in cycles of K-dimension (black line), J-dimension (gray) and I-dimension (light gray) partitions in 27-point stencil. The runtime of I-dimension partition is about 1% to 4% higher than the runtime of J-dimension partition. When the block size is smaller than 17, the runtime of K-dimension partition is lower than the runtime of I-dimension or J-dimension partition. The runtime of K-dimension partition then stays at around 116.5 million cycles when the block sizes are larger than 16. However, the runtime of J-dimension or I-dimension continues improving when the block sizes increase further. The minimum runtime is about 104.5 million cycles when the block size is 27.

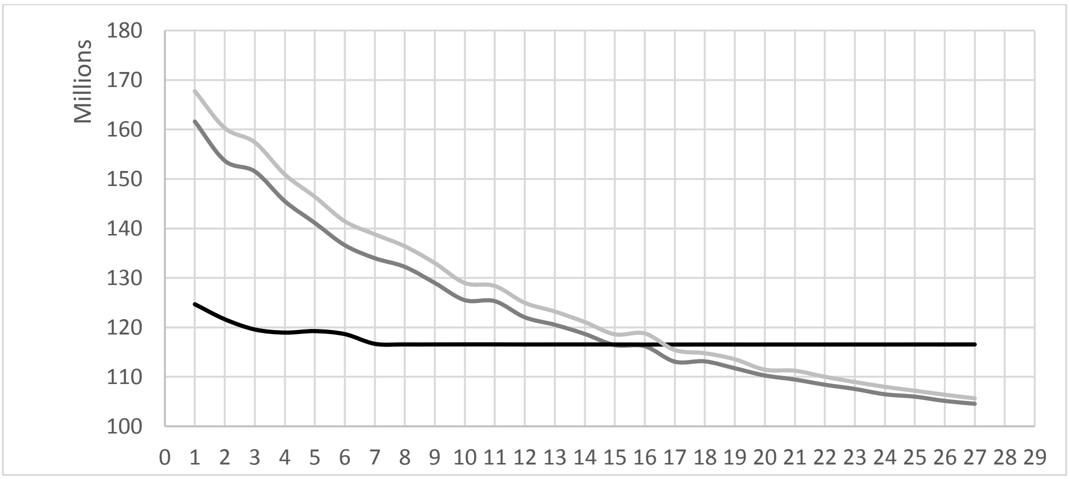


Figure 6.9 Runtime vs. the block sizes in 3D 27-point stencil

Comparing these figures we see that interface efficiency is almost 100% regardless of the way the stencil space is partitioned. However, DRAM controller efficiency increases from about 52% to 72% for the partitions along the J or I dimension when the block sizes increase. For the partition along the K dimension, DRAM controller efficiency is only slightly affected by the block sizes and stays at around 67%. Compared to the K-dimension partition in the 6-point stencil, the K-dimension partition allows 27 memory accesses for each stencil computation of each PE to be spread across all DRAM controllers for scheduling, reducing the bank conflicts that arise from the strided memory accesses in the J-dimension or I-dimension partition that interact with the interleaving in a way that memory accesses are to the same memory controller. As the block size increases, however, there are more consecutive memory accesses in each PE that can be exploited by the DRAM controllers in the J-dimension or I-dimension partition.

Based on the efficiency results of 6-point and 27-point 3D stencils, the blocking technique is an effective way to improve memory performance of stencil applications on the Convey HC-1 FPGA platform. The difference in DRAM efficiency is related with the access patterns of 6-point and 27-point stencils. When accessing input points in the loop order, 27-point stencil presents more irregular access pattern than that of 6-point stencil for each loop iteration, so the DRAM controller efficiency is 52% for 27-point stencil and 68.7% for 6-point stencil. On the other hand, when the block size increases, 27-point stencil sees a larger improvement of memory performance than 6-point stencil. And the efficiency will be expected to further increase if the size of input FIFO is not limited by the FPGA's memory resources. From the perspective of access pattern in a block, 27-point stencil presents more consecutive accesses than the 6-point stencil when the block size increases.

This helps DRAM controller schedule memory accesses more efficiently. However, for both stencil kernels memory efficiency will reach a point where further increase of block size has little effect on it.

6.2 PERFORMANCE IMPACT OF PROPOSED 3D STENCIL KERNEL DESIGNS

We evaluate the performance of proposed 3D stencil designs using the numerical ordering by a slightly larger 3D space of 514×514×512. We adjust the stencil space from 512×512×512 to 514×514×512 so that each of 64 PEs can request a grid (block) composed of a number of 10×10 planes of elements without introducing load imbalance in the *j* or *k* dimension of the block.

We also divide the stencil space along the K, J or I dimension for workload assignment to 64 PEs. Figure 6.10 illustrates the partitions of stencil space along the J dimension (left) and I dimension (right). The partition along the K dimension is not shown, as it is very similar to the partitions along the J dimension as shown in Figure 6.10. Since there are 64 PEs for four AEs, each PE is assigned a chunk of stencil space. The size of each chunk is

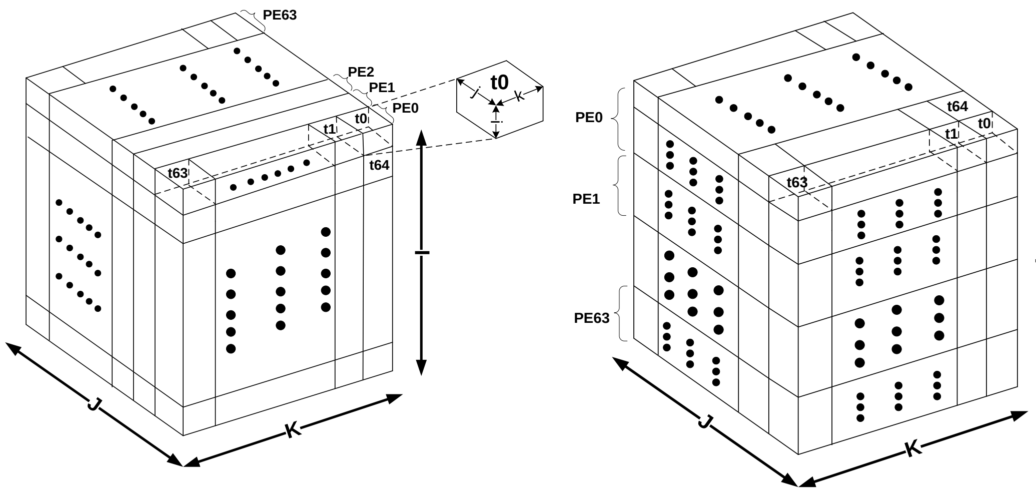


Figure 6.10 Block partitions along the J and I dimensions in 3D stencil space

514×10×512 in the J-dimension partitions. For the I-dimension partitions, each PE is assigned a chunk of size 514×514×10 except PE63 which is assigned a chunk of size 514×514×8. Each chunk is further divided into blocks. The size of each block (e.g., t0) is j×k×i where j=k=10. For each PE (e.g., PE0), it requests a block of elements in the order from block t0, through block t63, to block t64 until all blocks have been accessed in the chunk. The size of a block in *i* dimension is a parameter that correlates with the data reuse. The larger the value of the parameter is, the fewer number of blocks that are accessed in the chunk, increasing the data reuse rate. We choose the value of *i* such that the number of planes that contain the output points in the block is integer divisible by the number of planes that contain the output points in the chunk.

Table 6.1 lists the performance results of 6-point stencil for the partitions along the K dimension. The first column lists the size of the block in *i* dimension. The second and third columns list the number of load requests (ld_req_nu) and the number of store requests (st_req_nu) for each PE respectively. The fourth column lists the averaged runtime in cycles of all PEs of each AE. The fifth and sixth columns list the interface efficiency (I/F

Table 6.1 Results of proposed 6-point stencil design (K-dimension partition)

***i* dim**	**ld_req_nu**	**st_req_nu**	**runtime**	**I/F eff**	**DRAM eff**	**reuse rate**
4	6528000	2088960	17210316	99.988%	67.3%	0.408
5	5440000	2088960	15513067	99.986%	66.0%	0.483
7	4569600	2088960	14046638	99.985%	65.0%	0.543
8	4352000	2088960	13769022	99.984%	64.4%	0.558
32	3481600	2088960	11479047	99.982%	65.9%	0.618
53	3392000	2088960	11372618	99.982%	65.7%	0.624
87	3340800	2088960	11251270	99.982%	65.7%	0.627
104	3328000	2088960	11141877	99.981%	66.1%	0.628
172	3302400	2088960	11103086	99.982%	66.1%	0.630
257	3289600	2088960	10994738	99.982%	66.2%	0.631
512	3276800	2088960	10965916	99.982%	66.2%	0.632

eff) and DRAM controller efficiency (DRAM eff) respectively. Recall reuse rate is the ratio of the number of reused memory requests and the total number of memory requests without data reuse. For each PE, the total number of requests without data reuse is $510 \times 510 \times 8 \times 7$ where 7 is the number of load and store requests for the 6-point stencil computation of each point.

As shown in Table 6.1, when the *i* dimension of the block increases, the number of load requests decreases and the number of store requests is the same. Therefore the reuse rate increases when *i* value increases. The runtime also decreases as *i* value increases. From the smallest block with *i* = 4 to the largest block with *i* = 512, the runtime decreases by 38.9%. For all sizes of *i*, interface efficiency is close to 100% and the DRAM controller efficiency close to 65.8%. Compared to the lowest runtime in Section 6.1, the runtime decreases by 63% when *i* = 512.

Table 6.2 lists the performance results of 6-point stencil for the partitions along the J dimension. Compared to the results in Table 6.1, on average the runtime increases by 1% and memory efficiency is 1% lower. The reuse rates are the same as those in Table 6.1.

Table 6.2 Results of proposed 6-point stencil design (J-dimension partition)

***i* dim**	**ld_req_nu**	**st_req_nu**	**runtime**	**I/F eff**	**DRAM eff**	**reuse rate**
4	6528000	2088960	18098117	99.988%	65.5%	0.408
5	5440000	2088960	16032267	99.986%	64.8%	0.483
7	4569600	2088960	14341699	99.985%	64.3%	0.543
8	4352000	2088960	13862271	99.984%	64.3%	0.558
32	3481600	2088960	11409848	99.982%	66.5%	0.618
53	3392000	2088960	11297963	99.982%	66.0%	0.624
87	3340800	2088960	11245668	99.982%	65.9%	0.627
104	3328000	2088960	11278774	99.982%	65.6%	0.628
172	3302400	2088960	11232771	99.982%	65.5%	0.630
257	3289600	2088960	11129135	99.982%	65.8%	0.631
512	3276800	2088960	11105254	99.982%	65.8%	0.632

Table 6.3 lists the performance results of 6-point stencil for the partitions along the I dimension. As mentioned before, the I dimension of the chunk is 10 so the *i* dimension of the block is among 4, 6 and 10. Compared to the best runtime and DRAM controller efficiency in Table 6.1, the runtime increases by 19% when *i* = 10 and DRAM controller efficiency is 2% lower. The reuse rate is close to the reuse rate when *i* = 8 in Table 6.1.

Table 6.3 Results of proposed 6-point stencil design (I-dimension partition)

***i* dim**	**ld_req_nu**	**st_req_nu**	**runtime**	**I/F eff**	**DRAM eff**	**reuse rate**
4	6553600	2097152	18191555	99.988%	65.4%	0.406
6	4915200	2097152	15170615	99.985%	64.1%	0.519
10	4096000	2097152	13046077	99.983%	65.1%	0.575

To compare overall performance of the proposed 6-point 3D stencil design that combines memory latency hiding, memory access scheduling and data reuse with the design that has no data reuse, we define the *ideal speedup* as the ratio of the number of memory accesses without data reuse to the number of memory accesses with data reuse for each PE. As mentioned before, the total number of memory requests without data reuse is

Table 6.4 Comparison of the ideal and actual speedup in proposed 3D 6-point stencil

***i* dim**	**Ideal speedup**	**Actual speedup**	**Actual speedup vs. Ideal speedup**
4	1.69	1.41	83.48%
5	1.93	1.57	80.92%
7	2.19	1.73	79.03%
8	2.26	1.76	77.99%
32	2.61	2.12	80.91%
53	2.66	2.14	80.35%
87	2.68	2.16	80.46%
104	2.69	2.18	81.06%
172	2.70	2.19	80.96%
257	2.71	2.21	81.56%
512	2.71	2.21	81.58%

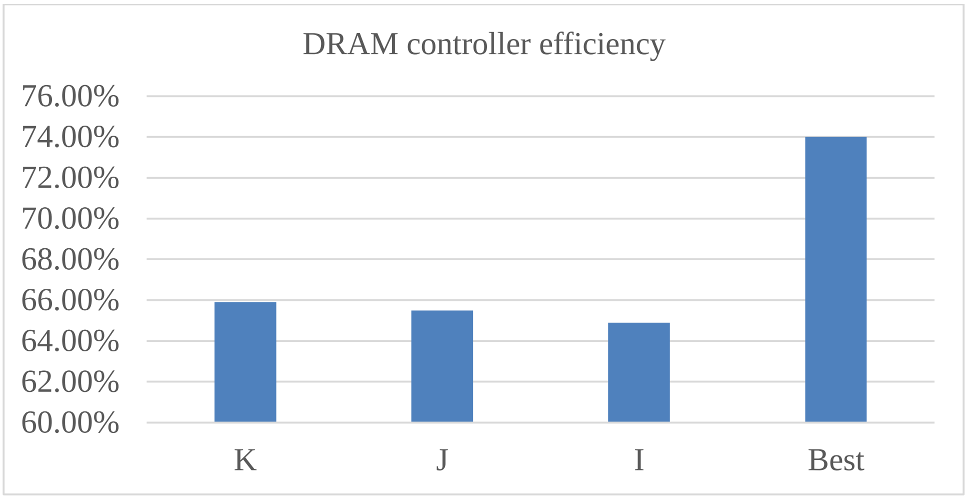


Figure 6.11 Comparison of 6-point 3D DRAM controller efficiency results

$510 \times 510 \times 8 \times 7$. For each *i* dimension of the block, the *actual speedup* is calculated by the ratio of the lowest runtime when the block size is 446 in the I-dimension partition to the corresponding runtime in Table 6.1. Table 6.4 compares the ideal and best actual speedup of 6-point stencil. With the increase of *i* value, the ideal speed up increases from 1.7 to 2.7 while the actual speedup from 1.4 to 2.2. The actual speedup achieves on average 80.76% of the ideal speedup.

Figure 6.11 compares DRAM controller efficiency results of the proposed 6-point 3D stencil design with the best DRAM controller efficiency. The best DRAM controller efficiency is 74% as described in Section 6.1. Since DRAM controller efficiency results across all *i* dimensions of the block are close, we average the efficiency results in Table 6.1, Table 6.2 and Table 6.3 respectively. As shown in Figure 6.11, the averaged DRAM controller efficiency of the K-dimension partition is about 11% lower than the best.

It is expected that the results of proposed 27-point stencil design are similar to those of proposed 6-point stencil design as their access patterns of the numerical order are the same. Because the results of the partitions along the K dimension are best in the proposed

6-point stencil design, we choose the same partition and compare the results with the best results in Section 6.1.

Table 6.5 shows the results of the partitions along the K dimension for 27-point stencil. Noticeably, the reuse rate reaches 0.9 as compared to the reuse rate of 0.63 in the 6-point stencil in Table 6.2. For all dimensions of *i*, interface efficiency is close to 100% and the DRAM controller efficiency close to 65.8%.

Table 6.5 Results of proposed 27-point stencil design (K-dimension partition)

***i* dim**	**ld_req_nu**	**st_req_nu**	**runtime**	**I/F eff**	**DRAM eff**	**reuse rate**
4	6528000	2088960	17427111	99.976%	66.9%	0.852
5	5440000	2088960	15499617	99.972%	65.9%	0.871
7	4569600	2088960	14057215	99.969%	65.0%	0.886
8	4352000	2088960	13668242	99.969%	64.8%	0.889
32	3481600	2088960	11453561	99.966%	66.0%	0.904
53	3392000	2088960	11383692	99.964%	65.7%	0.906
87	3340800	2088960	11303780	99.965%	65.7%	0.907
104	3328000	2088960	11226497	99.964%	65.7%	0.907
172	3302400	2088960	11175431	99.964%	65.7%	0.907
257	3289600	2088960	11057459	99.964%	66.0%	0.908
512	3276800	2088960	11034504	99.964%	66.1%	0.908

Table 6.6 Comparison of the ideal and actual speedup in proposed 3D 27-point stencil

***i* dim**	**Ideal speedup**	**Actual speedup**	**Actual speedup vs. Ideal speedup**
4	6.76	6.00	88.73%
5	7.74	6.75	87.17%
7	8.75	7.44	85.00%
8	9.05	7.65	84.57%
32	10.46	9.13	87.28%
53	10.63	9.18	86.40%
87	10.73	9.25	86.20%
104	10.76	9.31	86.59%
172	10.81	9.36	86.57%
257	10.83	9.46	87.29%
512	10.86	9.48	87.26%

Table 6.6 compares the ideal and actual speedup of the proposed 27-point stencil design across all *i* dimensions. As defined before, the *ideal speedup* is the ratio of the number of memory accesses without data reuse to the number of memory accesses with data reuse for each PE. The total number of memory requests without data reuse is $510 \times 510 \times 8 \times 28$. For each *i* dimension of the block, the *actual speedup* is calculated by the ratio of the lowest runtime when the block size is 27 in the I-dimension partition to the corresponding runtime in Table 6.5. With the increase of *i* value, the ideal speedup increases from 6.76 to 10.86 while the actual speedup from 6 to 9.48. The actual speedup achieves on average 86.64% of the ideal speedup.

As described in Section 6.1, the best DRAM controller efficiency for the 27-point stencil is 72% when the block size is 27. The averaged DRAM controller efficiency of the K-dimension partition in Table 6.5 is 65.8%, which is about 8.6% lower than the best.

6.3 PERFORMANCE OF MVM DESIGNS WITHOUT DATA REUSE

We evaluate the performance of MVM with different sizes of matrices and blocks. In our experiments on the Convey HC-1 platform, matrix A is a large $N \times N$ square matrix and N is 2048, 4096, 8192 and 16384. Matrices A and B are stored in the memory in the row-major order. The kernel is implemented with 64 PEs on four AEs with *BS* ranging from 16 to 512. The minimum size of the matrix is 2048 so that each PE in an AE can be assigned a subrow when BS is 512 at maximum. Without data reuse, the number of memory requests for MVM is $(N/16) \times (N/4) \times 2$ for each PE assuming there is no write operation.

As shown in Figure 5.8, matrices A and B are divided among 64 PEs of four AEs. Without data reuse, the address generator in each PE produces the row addresses that request a subrow and then the column addresses that request a subcolumn. Depending on the next subrow and subcolumn each PE in an AE is going to request, Figure 6.12 shows the *row-major stride access ordering* and the *column-major stride access ordering* for each PE (e.g., PE0) based on Figure 5.8.

Let us take PE0 in AE0 for the descriptions of the two orderings. In the row-major stride access ordering, PE0 first requests a subrow of size BS in the first row of matrix A, and then requests a subcolumn of size BS in matrix B. Then PE0 requests the next subrow which is pointed by the arc labeled with “row”. After all subrows in the first row of matrix A and all subcolumns in matrix B are requested, PE0 will request the subrows in the 16th row of matrix A. The process continues until PE0 requests all subrows and subcolumns assigned to it.

In the column-major stride access ordering, PE0 requests a subrow of size BS in the first row of matrix A, then requests a subcolumn of size BS in matrix B. Then PE0 requests

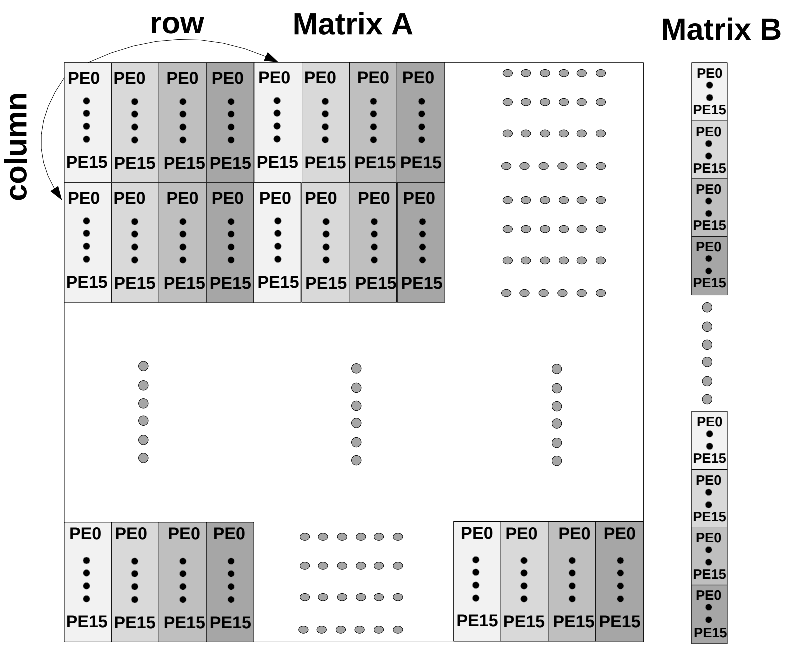


Figure 6.12 Row-major and column-major stride access orderings for each PE

the subrow which is pointed by the arc labeled with “column”. After all subrows in between the first column and the BSth column of matrix A and the subcolumns in matrix B are requested, PE0 will request the subrows in between the $(4 \times BS)^{th}$ column and the $(5 \times BS)^{th}$ column of matrix A. The process continues until PE0 requests all subrows and subcolumns assigned to it.

Table 6.7 Interface efficiency (no data reuse and row-major)

I/F eff	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	93.644%	96.478%	97.724%	98.277%	98.302%	99.254%
N=4096	94.000%	96.857%	98.196%	99.037%	99.337%	99.683%
N=8192	94.088%	96.937%	98.354%	99.181%	99.559%	99.766%
N=1638	94.110%	96.962%	98.455%	99.214%	99.591%	99.795%

Table 6.8 Interface efficiency (no data reuse and column-major)

I/F eff	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	93.398%	96.761%	98.306%	99.021%	98.344%	99.046%
N=4096	93.933%	96.937%	98.427%	99.149%	99.311%	99.647%
N=8192	94.070%	96.958%	98.452%	99.156%	99.531%	99.764%
N=1638	94.105%	96.966%	98.421%	99.171%	99.562%	99.790%

Table 6.7 and Table 6.8 show the interface efficiency results of row-major and column-major stride access orderings respectively. The interface efficiency depends on the sizes of BS for each matrix. When BS = 16, interface efficiency is around 94%. As BS increases from 16 to 512, interface efficiency rises from 94% to almost 100%. The increase of interface efficiency is due to the fact that the larger the block size, the fewer number of blocks resulting from blocking MVM, and the less overhead caused by the address generations of the blocks.

Table 6.9 DRAM controller efficiency (no data reuse and row-major)

DRAM eff	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	39.3%	57.1%	80.1%	85.3%	79.2%	86.3%
N=4096	39.8%	57.3%	80.2%	86.4%	83.1%	87.2%
N=8192	40.9%	55.3%	80.2%	88.2%	85.1%	86.0%
N=16384	41.2 %	57.2%	80.2%	88.3%	84.7%	88.0%

Table 6.10 DRAM controller efficiency (no data reuse and column-major)

DRAM eff	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	26.8%	54.2%	75.9%	85.8%	80.1%	85.4%
N=4096	26.8%	54.3%	76.1%	86.2%	82.6%	87.3%
N=8192	26.7%	55.0%	76.1%	85.3%	83.9%	87.5%
N=16384	26.6 %	54.6%	64.4%	84.9%	78.3%	87.2%

Table 6.9 and Table 6.10 show the DRAM controller efficiency results of row-major and column-major stride access orderings respectively. As shown in the tables, DRAM controller efficiency depends on the sizes of BS for a given matrix of size N. For the row-major stride access ordering, DRAM controller efficiency increases from 39% to 88% when BS increases from 16 to 512. For the column-major stride access ordering, DRAM controller efficiency increases from around 27% to 87% when BS increases from 16 to 512. For small sizes of BS, the row-major stride access ordering is much better than the column-

Table 6.11 Runtime (no data reuse and row-major)

Runtime	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	948626	532093	264434	225311	268430	214358
N=4096	3714137	2104296	1036304	868384	954309	849226
N=8192	14389481	8425879	4214742	3283330	3583135	3427312
N=16384	55871701	33781148	16822756	12936476	14438786	13014154

Table 6.12 Runtime (No data reuse and column-major)

Runtime	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	1568436	587004	329660	241471	263733	217579
N=4096	6284594	2317150	1223201	905288	976044	830685
N=8192	25235340	9100608	4827260	3671456	3625848	3296994
N=16384	101281117	35767664	19483892	14448023	15065649	12996661

major stride access ordering in improving DRAM controller efficiency. However, the improvement diminishes as the block size further increases. For both access orderings, DRAM controller efficiency drops when BS=256 and increases again from BS=256 to BS=512.

Table 6.11 and Table 6.12 show the runtime results of row-major and column-major stride access orderings respectively. For small sizes of BS, the row-major stride access ordering is better than the column-major stride access ordering in improving runtime. However, the improvement diminishes as the block size further increases.

Figure 6.13 plots the relationships between logarithmically scaled runtime time and N and BS using Table 6.12. The figure shows that runtime scales up with the matrix size N increasing from 2048 to 16384. For a matrix of size N, runtime decreases when BS increases from 16 to 256. From BS=16 to BS=128 there is 50% more reduction in runtime. However, due to the effect of DRAM controller efficiency, runtime increases from BS=128

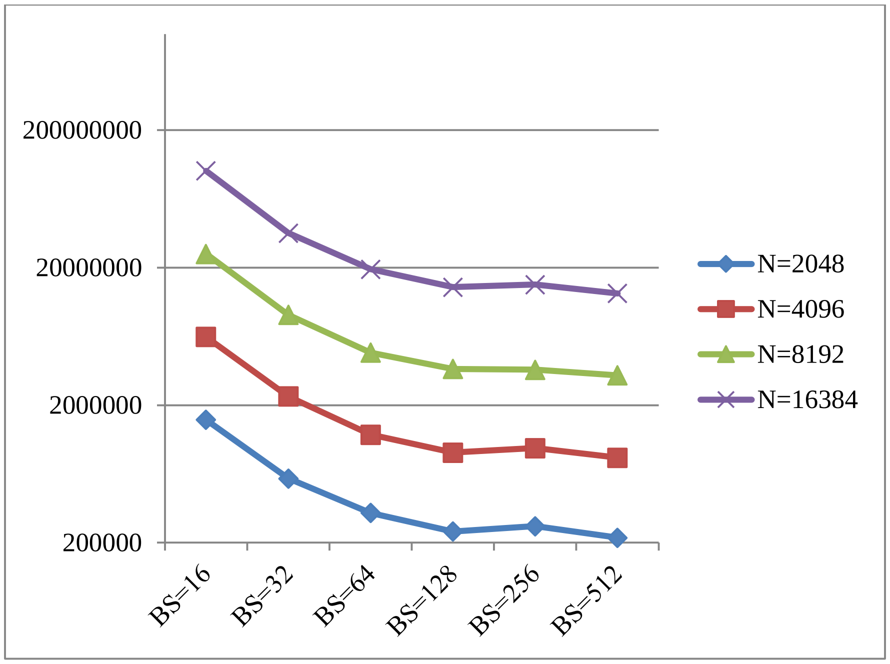


Figure 6.13 Scaled runtime vs. matrix and block sizes (No data reuse)

to BS=256, but decreases from BS=256 to BS=512. The experimental results suggest that a large BS generally achieves better performance on the Convey platform, but the performance improvement diminishes as BS increases.

6.4 PERFORMANCE OF PROPOSED MVM DESIGNS

We evaluate the performance of proposed MVM design with the same sizes of matrices and blocks as those in Section 6.3. The kernel is implemented with 64 PEs with *BS* ranging from 16 to 512. The minimum size of the matrix is 2048. For the proposed designs that maximize column vector data reuse, the number of memory requests for each PE is $(N/16) \times (N/4) + N/4$. So the reuse rate is 0.5-8/*N*, which reaches 0.5 when the size of matrix is infinite.

Table 6.13 lists the results of interface efficiency for the proposed design. Compared with the results in Table 6.7 and Table 6.8, the efficiency also increases from about 93% to almost 100% when BS increases from 16 to 512. Although the address generator skips the generation of addresses that can be reused, the overhead of address generation still depends on the number of blocks in the blocking MVM.

Table 6.13 Interface efficiency (data reuse)

I/F eff	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	92.091%	93.161%	95.749%	96.987%	97.721%	98.093%
N=4096	93.026%	94.083%	96.930%	98.090%	98.844%	99.226%
N=8192	93.244%	94.250%	97.186%	98.369%	99.130%	99.514%
N=16384	93.281%	94.492%	97.289%	98.439%	99.201%	99.586%

Table 6.14 DRAM controller efficiency (data reuse)

DRAM eff	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	42.7%	70.9%	85.9%	88.9%	89.0%	88.7%
N=4096	42.3%	71.9%	86.1%	89.1%	88.9%	88.9%
N=8192	42.3%	71.9%	86.9%	89.1%	88.9%	88.9%
N=16384	42.2 %	72.2%	85.9%	89.1%	88.9%	88.9%

As shown in Table 6.14, with data reuse the efficiency results are better than those without data reuse in Tables 6.9 and 6.10. When the block size is 32, DRAM controller efficiency even increases from 57% to 72%. However, the increase diminishes when the block size further increases. The results suggest that decreasing the number of memory requests using data reuse may reduce the number of DRAM controller stalls.

Table 6.15 lists the runtime with data reuse and Table 6.16 compares the runtime results with the *lowest* runtime results in Table 6.9 and Table 6.10. The proposed design reduces the runtime by a minimum of 10% and a maximum of 50% compared to the design without reusing the column elements.

Table 6.15 Runtime (data reuse)

Runtime	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	787796	298171	180709	164571	132672	134403
N=4096	3148240	1149857	683249	546120	525852	528046
N=8192	12590893	4576754	2532722	2146875	2098629	2100585
N=16384	50491697	17982722	10146566	8528111	8390234	8392746

Table 6.16 Runtime (data reuse) vs. runtime (without data reuse)

Runtime	BS=16	BS=32	BS=64	BS=128	BS=256	BS=512
N=2048	17.0%	44.0%	31.7%	27.0%	49.7%	37.3%
N=4096	15.2%	45.4%	34.1%	37.1%	44.9%	36.4%
N=8192	12.5%	45.7%	39.9%	34.6%	41.4%	38.7%
N=16384	9.6%	46.8%	39.7%	34.1%	41.9%	35.4%

Table 6.17 The ideal speedup and the actual speedup in the Proposed MVM designs

N	Ideal speedup	Actual speedup	Actual speedup vs. Ideal speedup
2048	1.984	1.57	78.98%
4096	1.992	1.58	79.51%
8192	1.996	1.59	79.46%
16384	1.998	1.57	78.66%

In MVM we define the ideal speedup as the ratio of the number of memory accesses without and with data reuse for each PE. So the ideal speedup is 2 / (1+16/*N*). The actual speedup is calculated by the ratio of the *lowest* runtime without data reuse to the runtime with data reuse for each PE. Table 6.17 lists the ideal speedup and the actual speedup for each matrix size *N*. The actual speedup for each N is the average of the actual speedup results of all values of BS. As shown in the Table 6.17, the ideal speedup is close to 2 while the actual speedup is around 1.6. The actual speedup achieves on average 79% of the ideal speedup.

6.5 RESOURCE UTILIZATION

All the designs are compiled using Xilinx ISE 14.4 with the same constraints and settings for synthesis, map, place and route. Table 6.18 summarizes the resource utilization of the 6-point and 27-point stencils with the loop, array and numerical orders when the stencil space is divided along the K dimension. The resource utilization results of dividing the space along the other two dimensions are not listed as they are almost the same as the results in the table. As shown in the table, the proposed design with the numerical order is more resource efficient in terms of the number of Slices, Slice registers, Slice LUTs and Block RAMs. Note DSP48E macros are not used as all the arithmetic operations are

Table 6.18 Resource utilization of 6-point and 27-point 3D stencils

	k-dim (6-point)			k-dim (27-point)		
Resource	loop	array	numerical	loop	array	numerical
Slice	83%	84%	81%	98%	98%	88%
Slice reg	57%	58%	53%	72%	73%	58%
Slice LUT	47%	48%	48%	78%	80%	62%
BRAM	80%	80%	80%	97%	97%	86%

implemented with Slice LUTs. The 27-point stencil design with the loop or array order utilizes 98% of total FPGA slices due to the large number of 64-bit adders and the FSM controller in the address generator. For the 6-point stencil, the BRAM utilization is 80% for the three orderings. For the 27-point stencil, the BRAM utilization is 97% for the loop and array orderings, and 86% for the numerical ordering.

Table 6.19 summarizes the resource utilization of the MVM designs. The MVM with data reuse utilizes a little more resources due to the additional control logics for column ROB and the address generator. The BRAM utilization is 25% for the MVM without data reuse and 30% for the MVM with data reuse.

Table 6.19 Resource utilization of MVMs

Resource	MVM (no reuse)	MVM (reuse)
Slice	64%	66%
Slice reg	44%	44%
Slice LUT	38%	39%
BRAM	25%	30%

6.6 DISCUSSION OF THE EXPERIMENTAL RESULTS

For the N-point 3D stencil kernels, we partition the stencil space along the J, K and I dimensions to evaluate the performance of each partition. For the designs that accesses memory in the loop and array order without reusing the input elements for stencil computation, the performance results of J-dimension and I-dimension partitions are very similar. Compared to the K-dimension partition, they achieve lower runtime and higher DRAM controller efficiency for large block sizes in both 6-point and 27-point stencils. However, the runtime and DRAM controller efficiency are not affected much by the block sizes in the K-dimension partition in both stencils.

For the proposed designs that access memory in the numerical order and reuse input elements for stencil computation, the efficiency and runtime results of J-dimension and K-dimension partitions are very close, but K-dimension partition is slightly better. Because the I dimension of the chuck assigned to each PE in the I-dimension partition is 10, the best efficiency and runtime of the I-dimension are lower than the best results in J-dimension and K-dimension partitions. The partition along the K dimension allows each PE to request part of the consecutive elements in 3D stencil array and the requests from all PEs are distributed in an interleaving way to the memory system through the crossbars.

By varying the size of *i* dimension of a $10 \times 10 \times i$ block assigned to each PE, the runtime is significantly reduced while interface efficiency and DRAM controller efficiency are almost the same as shown in Table 6.1, Table 6.2, Table 6.3 and Table 6.5. This means data reuse contributes the most to the overall performance improvement.

Though the runtime of the proposed designs that reuse data is much better than that of the design that accesses memory in the array order without data reuse, the memory performance is 8% to 11% lower than the best performance in the array order. When accessing a block in the numerical order, each row of a plane is accessed consecutively, but the memory accesses are strided between the rows of the plane(s). On the other hand, there are more consecutive memory accesses in a block in the array order when the block size increases. So the array order causes fewer load and store stalls from the DRAM controller per memory access than the numerical order, which in turn increases the memory performance. For the proposed designs, we believe increasing the block size in the k dimension will improve the memory performance. However, this will be left to the future research when the target FPGA platform offers a larger number of memory resources.

For MVM designs, increasing the block size can effectively increase the memory performance regardless of data reuse. Without data reuse, the row-major stride access ordering is much better than the column-major stride access ordering when the block size is small. However, the improvement in memory performance with the row-major stride access ordering diminishes as the block size increases. In addition, the drop in DRAM controller efficiency when the block size is 256 for both access orderings indicates that the memory controller cannot schedule the memory requests very effectively when the memory accesses are to the same memory controller and the rest are unused, causing more bank contentions.

As mentioned before, the experimental results show that data reuse is the most effective optimization in improving overall performance. For 6-point and 27-point stencils, the speedup of the proposed design over the design without data reuse is 2.2 and 9.5. In MVM, the speedup is 1.6 when column vector reuse is maximized.

CHAPTER 7

CONCLUSION

This dissertation presents generations of custom memory interface on the multi-FPGA platform for FPGA-based kernels that have a regular access pattern. The target platform, the Convey HC-1, is a multi-FPGA platform whose memory system performs dynamic access scheduling but presents the user logic with the critical challenges:

- Memory system itself does not perform caching or prefetching.
- Memory operations are arbitrarily reordered.
- Memory performance depends on the access order provided by the user logic.

To reconcile these problems and maximize overall performance of memory interface, the interface of each kernel should provide explicit support for the latency hiding, memory access scheduling and data reuse.

3D stencil and MVM are two memory-bound kernels. The stencil computation presents nonconsecutive memory accesses for N points in the innermost loop iterations. In MVM each AE accesses the matrices blocks in a stride pattern as each matrix is divided among four AEs. Both kernels are parallelizable for their hardware implementations on the HPC platforms such as the Convey HC-1. So they are benchmarks appropriate for evaluating performance of the memory interface designs and DRAM-based memory system.

To hide memory latency, we compare the performance of double buffering and circular buffering and show that double buffering introduces interface stall whereas circular buffering can eliminate the interface stall. For memory access scheduling, we propose three memory access orders, the loop order, the array order and the numerical order to evaluate the effect of memory access scheduling on the performance of memory interface. Based on the regular access pattern of the kernel, the numerical order facilitates the hardware implementations of memory latency hiding and data reuse in 3D stencil. In MVM data reuse is achieved by maximizing the reuse of column vector for each PE. In 3D stencil data reuse is realized by each PE reading 3D blocks composed of a set of planes and then performing stencil computation in the blocks. To combine memory latency hiding, memory access ordering and data reuse, we develop custom circular buffering techniques, custom address generators and kernel implementations in both kernels.

Experimental results show that when the block size increases, DRAM controller efficiency increases from 39% to 88% for MVM without data reuse using row-major stride access ordering, from 42% to 89% for MVM without data reuse, from 68.7% to 74% for 6-point 3D stencil, and from 52% to 72% for 27-point 3D stencil. With the proposed designs that enable data reuse, we are able to achieve the reuse rate of about 0.5 for MVM and the reuse rate of 0.6 and 0.9 for 6-point and 27-point 3D stencil respectively. We also achieve a speedup of 2.2 and 9.48 for 6-point and 27-point 3D stencil and a speedup of 1.6 for MVM with data reuse increasing the overall performance of the memory interface more than memory latency hiding and memory access ordering.

As mentioned before, the stencil computation in the dissertation has no multiply. When multiplication is considered, multiply will increase the runtime assuming multiply takes more time than add. However, the runtime increase is negligible. Our interface can hide memory latency almost 100% so that the stencil computations can be overlapped with memory accesses. When memory accesses are finished, the kernel will finish all the remaining stencil computations, increasing the runtime at the last loop iteration in the stencil algorithm.

As described in Chapter 4 and Chapter 5, the memory resource on the target Virtex-5 LX330 FPGA constrains the exploration space of our research. The future work beyond the dissertation includes (1) Explore memory efficiency results by increasing the FIFO size from 32 to 512 in the 27-point stencil implementation on the Convey HC-1ex platform. (2) In the implementations of proposed N-point stencil design that combines memory latency hiding, memory access ordering and data reuse, increase the stencil space and the plane size of a 3D block to further improve the memory efficiency. (3) Implement a version of the MVM kernel that contains memory write operations.

REFERENCES

[1] Impluse CoDeveloper C-to-FPGA Tools, Impulse Accelerated Technologies, 2012.

[2] Synphony C Compiler Reference Manual, Synopsys Inc, 2012.

[3] Cadence C-to-Silicon Compiler Datasheet, Cadence Inc, 2008.

[4] AutoESL User Guide, Xilinx Inc, April 2012.

[5] ———, http://www.xilinx.com/fpga/index.htm

[6] ———, http://www.altera.com/products/fpga.html

[7] ———, http://en.wikipedia.org/wiki/Reconfigurable_computing

[8] Estrin, G. 2002. *Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer*, IEEE Ann. Hist. Comput.24, 4 (Oct. 2002), 3–9.

[9] Estrin, G., *Organization of Computer Systems—The Fixed Plus Variable Structure Computer,* Proc. Western Joint Computer Conf., Western Joint Computer Conference, New York, 1960, pp. 33–40.

[10] ———, http://www.hotchips.org/wp-content/uploads/hc_archives/hc21/3_tues/

HC21.25.500.ComputingAccelerators-Epub/HC21.25.526.Brewer-Convey-HC1-Instruction-Set.pdf

[11] D. Wang, *Modern DRAM Memory Systems: Performance Analysis and a High Performance, Power-Constrained DRAM-Scheduling Algorithm*, PhD dissertation, 2005

[12] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, J.D. Owens, *Memory Access Scheduling*, Proc. ISCA00.

[13] Nallatech, *Intel Xeon FSB FPGA Accelerator Module*, http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html.

[14] DRC Computer, *DRC Reconfigurable Processor Units (RPU)*, http://www.drccomputer.com/drc/modules.html.

[15] XtremeData Inc., *XD2000i™ FPGA In-Socket Accelerator for Intel FSB,* http://www.xtremedata.com/products/accelerators/in-socket-accelerator/xd2000i.

[16] Convey Tech Note: *Convey HC-1 Memory Subsystem*, Convey Corporation.

[17] Jason D. Bakos, *High-Performance Heterogeneous Computing with the Convey HC-1*, Computing in Science and Engineering, Vol. 12, No. 6, November/December 2010.

[18] Werner Augustin, Jan-Philipp Weiss, and Vincent Heuveline, *Convey HC-1 Hybrid Core Computer − The Potential of FPGAs in Numerical Simulation,* Proceedings of the second International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'11), San Antonio, Texas, USA, February 2011.

[19] Gunnar Ruthenberg, *Optimizing ROCCC-generated stream processors for Convey HC-1,* project thesis, October, 2012.

[20] Cong J, Huang M, Zou Y: *3D Recursive Gaussian IIR on GPU and FPGAs, A Case Study for Accelerating Bandwidth-Bounded Applications,* Proceedings of the 9th IEEE Symposium on Application Specific Processors (SASP 2011). San Diego, CA: 201

[21] Fabian Nowak, Michael Bromberger, Martin Schindewolf, and Wolfgang Karl. 2013. Multi-parallel prefiltering on the convey HC-1 for supporting homology detection. In *Proceedings of the 20th European MPI Users' Group Meeting* (EuroMPI '13). ACM, New York, NY, USA, 169-174.

[22] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang, *A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality,* Proceedings of the 33rd Annual International Symposium on Microarchitecture, 2000. 1:70–73.

[23] Zhibin Fang, Xian-He Sun, Yong Chen, Byna, S., Core-aware memory access scheduling schemes, *IEEE International Symposium on Parallel & Distributed Processing,* pp.1-12, 23-29 May 2009

[24] Onur Mutlu and Thomas Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 40). IEEE Computer Society, Washington, DC, USA, 146-160.

[25] Yuan, G.L., Bakhoda, A., Aamodt, T.M., Complexity effective memory access scheduling for many-core accelerator architectures, *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on,* vol., no., pp.34,44, 12-16 Dec. 2009

[26] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. 2011. *Dymaxion: optimizing memory access patterns for heterogeneous systems*. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '11). ACM, New York, NY, USA

[27] Smith, J.E. *Decoupled access/execute computer architectures*, Computer Systems, ACM Transactions on; Volume 2, Issue 4, November 1984, Pages 289-308

[28] Fei Chen; Sha, E.H.-M., *Loop scheduling and partitions for hiding memory latencies, Proceedings of 12th International Symposium on System Synthesis,,* pp.64-70, Nov 1999.

[29] Hun Xue; Zili Shao; Meilin Liu; Meikang Qiu; Sha, E.H.-M., Loop scheduling with complete memory latency hiding on multi-core architecture, *12th International Conference on Parallel and Distributed Systems*, vol.1, *ICPADS 2006.*

[30] LANGE, H., WINK, T., AND KOCH, A. 2011. MARC II: *A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers*. In DATE.

[31] Thielmann, B., Huthmann, J., Koch, A. 2012. *Memory Latency Hiding by Load Value Speculation for Reconfigurable Computers*. ACM Trans. Reconfig. Technol. Syst. 5, 3, Article 13 (October 2012)

[32] Ilie I. Luican, Hongwei Zhu, and Florin Balasa. 2006. *Formal model of data reuse analysis for hierarchical memory organizations*. In Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD '06). ACM, New York, NY, USA, 595-600.

[33] Ilya Issenin, Erik Brockmeyer, Bart Durinck, and Nikil Dutt. 2006. *Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies*. In Proceedings of the 43rd annual Design Automation Conference (DAC '06). ACM, New York, NY, USA, 49-52

[34] M. Kandemir, G. Chen, and F. Li. 2006. *Maximizing data reuse for minimizing memory space requirements and execution cycles*. In Proceedings of the 2006 Asia and South Pacific Design Automation Conference (ASP-DAC '06). IEEE Press, Piscataway, NJ, USA, 808-813.

[35] Yuxin Wang, Peng Zhang, Xu Cheng, Jason Cong: *An integrated and automated memory optimization flow for FPGA behavioral synthesis*. ASP-DAC 2012: 257-262

[36] R. Banakar, S. Steinke, B. Lee, *Scratchpad memory design alternative for cache on-chip memory in embedded systems*, in Proc. of the 10th Int. Symp. on hardware/Software Codesign (CODES), 2002, pp. 73 - 78

[37] P.R. Panda, N.D. Dutt, A. Nicolau, *Efficient utilization of scratch-pad memory in embedded processor applications*, in IEEE Trans. European Design and Test Conference (ED&TC), 1997, pp. 7.

[38] M. Kandemir, J. Ramanujam, M.J. Irwin, et al, *A Compiler-Based Approach for Dynamically Managing Scratch-Pad Memories in Embedded Systems*, in IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2004, pp. 243 - 260.

[39] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. 2007. *DRDU: A data reuse analysis technique for efficient scratch-pad memory management*. ACM Trans. Des. Autom. Electron. Syst. 12, 2, Article 15 (April 2007).

[40] J. Cong, H. Huang, C. Liu, Y. Zou, *A Reuse-Aware Prefetching Scheme for Scratchpad Memory*, in Proc. of the 48th Annual Design Automation Conference (DAC), 2011, pp. 960-965.

[41] J. Cong, P. Zhang and Y. Zou, *Combined Loop Transformation and Hierarchy Allocation in Data Reuse Optimization*, in Proc. of the 2011 Int. Conf. on Computer-Aided Design (ICCAD), 2011, pp. 185-192

[42] Y. Tatsumi, H. Mattausch, *Fast quadratic increase of multiport-storge-cell area with port number,* in Electronics Letters, 1999.

[43] W.K.C. Ho, S.J.E. Wilton, *Logical-to-Physical Memory Mapping for FPGAs with Dual-Port Embedded Arrays*, in Field Programmable Logic and Applications, Lecture Notes in Computer Science, 2004, pp. 111-123.

[44] L. Benini, L. Macchiarulo, A. Macii, et al, *Layout-driven memory synthesis for embedded systems-on-chip*, in IEEE Trans. Very Large Scale Integration Systems (TVLSI), 2002, pp. 96 - 105.

[45] N. Baradaran, P.C. Diniz, *A compiler approach to managing storage and memory bandwidth in configurable architectures*, in ACM Trans. on Design Automation of Electronic Systems (TODAES), 2008, Vol. 13, No. 4, Article 61.

[46] J. Cong, W. Jiang, B. Liu, Y. Zou, *Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization*, in ACM Trans. on Design Automation of Electronic Systems (TODAES), 2011, Vol. 16 Issue 2, Article 15

[47] Y. Ben-Asher, N. Rotem, *Automatic memory partitioning: increasing memory parallelism via data structure partitioning*, in Proc. of the 8th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010, pp, 155 - 162.

[48] S. Brown, *Performance comparison of finite-difference modeling on Cell,, FPGA and multi-core computers*, in SEG/San Antonio 2007 Annual Meeting, 2007, pp. 2110–2114.

[49] C. He, W. Zhao, and M. Lu, *Time Domain Numerical Simulation for Transient Waves on Reconfigurable Coprocessor Platform*, in FCCM ’05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Washington, DC, USA: IEEE Computer Society, 2005, pp. 127–136.

[50] C. He, M. Lu, and C. Sun, *Accelerating Seismic Migration Using FPGA-Based Coprocessor Platform*, in FCCM ’04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Washington, DC, USA: IEEE Computer Society, 2004, pp. 207–216.

[51] C. He, G. Qin, M. Lu, and W. Zhao, *An Efficient Implementation of High-Accuracy Finite Difference Computing Engine on FPGAs*, in ASAP ’06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors. Washington, DC, USA: IEEE Computer Society, 2006, pp. 95–98.

[52] G. Rivera and C.-W. Tseng, *Tiling optimizations for 3D scientific computations*, in Supercomputing ’00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM). Washington, DC, USA:IEEE Computer Society, 2000, p.

[53] C. Leopold, *Tight Bounds on Capacity Misses for 3D Stencil Codes*, in Computational Science ICCS 2002. Springer Berlin / Heidelberg, 2002, pp. 843–852.

[54] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, *Impact of modern memory subsystems on cache optimizations for stencil computations*, in MSP ’05: Proceedings of the 2005 workshop on Memory system performance. New York, NY, USA: ACM, 2005, pp.36–43.

[55] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker,D. Patterson, J. Shalf, and K. Yelick, *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*, in SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing.Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[56] ———, https://www.nas.nasa.gov/publications/npb.html

[57] Eric S. Chung, *CoRAM: An In-Fabric Memory Architecture for FPGA-Based Computing*, PhD Thesis, August 2011.

[58] ———, http://en.wikipedia.org/wiki/Block_matrix