A Formal Approach to Memory Access Optimization: Data Layout, Reorganization, and Near-Data Processing

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

BERKIN AKIN

B.S., Electrical and Electronics Engineering, Middle East Technical University
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA, USA

July 2015

Abstract

The memory system is a major bottleneck in achieving high performance and energy efficiency for various processing platforms. This thesis aims to improve memory performance and energy efficiency of data intensive applications through a two-pronged approach which combines a formal representation framework and a hardware substrate that can efficiently reorganize data in memory.

The proposed formal framework enables representing and systematically manipulating data layout formats, address mapping schemes, and memory access patterns through permutations to exploit the locality and parallelism in memory. Driven by the implications from the formal framework, this thesis presents the HAMLeT architecture for highly-concurrent, energy-efficient and low-overhead data reorganization performed completely in memory. Although data reorganization simply relocates data in memory, it is costly on conventional systems mainly due to inefficient access patterns, limited data reuse, and roundtrip data traversal throughout the memory hierarchy. HAMLeT pursues a near-data processing approach exploiting the 3D-stacked DRAM technology. Integrated in the logic layer, interfaced directly to the local controllers, it takes advantage of the internal fine-grain parallelism, high bandwidth and locality which are inaccessible otherwise. Its parallel streaming architecture can extract high throughput from stringent power, area, and thermal budgets.

The thesis evaluates the efficient data reorganization capability provided by HAMLeT through several fundamental use cases. First, it demonstrates software-transparent data reorganization performed in memory to improve the memory access. A proposed hardware monitoring determines

inefficient memory usage and issues a data reorganization to adapt an optimized data layout and address mapping for the observed memory access patterns. This mechanism is performed transparently and does not require any changes to the user software—HAMLeT handles the remapping and its side effects completely in hardware. Second, HAMLeT provides an efficient substrate to explicitly reorganize data in memory. This gives an ability to offload and accelerate common data reorganization routines observed in high-performance computing libraries (e.g., matrix transpose, scatter/gather, permutation, pack/unpack, etc.). Third, explicitly performed data reorganization enables considering the data layout and address mapping as a part of the algorithm design space. Exposing these memory characteristics to the algorithm design space creates opportunities for algorithm/architecture co-design. Co-optimized computation flow, memory accesses, and data layout lead to new algorithms that are conventionally avoided.

Acknowledgement

I owe all my success to my great advisors Professor James Hoe and Professor Franz Franchetti. Their support, advice and expertise made this work possible, I am very thankful for their time and effort. James has tremendous experience and expertise in addition to his great personality and sense of humor. Although this thesis was "done in his head" long before, it took me a while to "make it perfect". Franz is one of the smartest people that I have ever met. He always had brilliant ideas and creative solutions in every subject. Our long and productive discussions, which are made enjoyable thanks to his unique wit, contributed to a huge portion of this thesis. To me both James and Franz are both excellent advisors and good friends.

I would like to thank my thesis committee members Professor Larry Pileggi and Professor Steve Keckler. Larry provided me great support and feedback throughout my PhD life including my thesis. I'm thankful to him for giving me the opportunity to access the state-of-the-art silicon technology and also teaching me how to use it. His collaboration and efforts were really valuable to me. Steve provided helpful feedback, constructive discussions and great architecture insights which made this thesis possible. I'm grateful for his time and effort.

This work was partially sponsored by the Defense Advanced Research Projects Agency (DARPA) PERFECT program under agreement HR0011-13-2-0007. I'm thankful for their generous supports to fund this research.

I'd like to also thank all the SPIRAL and CALCM team members in CMU. First, I'm thankful to Peter Milder for introducing me to his hardware infrastructure in SPIRAL, and for his help and col-

laboration during my first years in CMU. I'd like to also thank Michael Papamichael and Gabe Weisz for their company in the office and in various conferences, and also for their useful discussions on a lot of research or non-research topics. Special thanks to Michael for his suggestions on a lot of "how to fix a VW Golf" topics. I had the honor to work with great lab mates, researchers and post-docs in SPIRAL and CALCM including Nikos Alachiotis, Christos Angelopoulos, Kevin Chang, Eric Chung, Qi Guo, Yoongu Kim, Peter Klemperer, Bob Koutsoyannis, Tze Meng Low, Joe Melber, Marie Nguyen, Eriko Nurvitadhi, Thom Popovici, Fazle Sadi, Lavanya Subramanian, Richard Veras, Yu Wang, Guanglin Xu, Jiyuan Zhang, Zhipeng Zhao, Qiuling Zhu, and many more.

I would like to also thank excellent professors in CMU that I got a chance to meet and work together. Especially, I'd like to thank Professor Onur Mutlu for teaching me advanced computer architecture, Professor Jose Moura and Professor Markus Puschel for our collaborations in SPIRAL, special thanks to Markus for being a great host in Zurich.

I thank all of the brilliant ECE staff, especially Claire Bauerle and Samantha Goldstein who made administrative tasks run smoothly.

I feel lucky to be around great friends and fellow graduate students in Pittsburgh. There are too many to list but special thanks go to Ekin Sumbul and Cagla Cakir for their answers to my yet another circuits questions, and to Onur Albayrak, Melis Hazar, Meric Isgenc, Tugce Ozturk, Mert Terzi, Sercan Yildiz, and many more. Thank you all for making the life here so much fun.

Trying to push processing "near" memory was a bit difficult, when I was "far away" from my family. I cannot express with words my gratefulness to my parents Adil Akin and Leyla Akin, and my sister Gizem Akin. Although they will not understand much of it, this thesis would be impossible without their love and support.

Finally, I'd like to thank Sila Gulgec, my "home" here, for her continuous love and support. She was always with me both in the most stressful and the happiest times throughout this journey. This thesis would be meaningless without her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Transistors continue to shrink in size with every new generation following the Moore's law, which suggests doubling the number of transistors on chip every 18 months. However, transistor threshold and supply voltages have not been scaling proportionally in the post Dennard era [43]. Future processing platforms are heading towards an energy constrained era, where parts of the chip, known as the dark silicon, are under-clocked or completely turned off to meet the power consumption limitations.

In addition to the stringent on-chip energy constraints, scaling trends for the DRAM technology and off-chip pin count do not match the transistor scaling trends, pointing towards the memory wall problem. This problem is exacerbated in the dark silicon era by multiple cores and on-chip accelerators sharing the main memory and demanding for more bandwidth. Memory bandwidth becomes a limiting factor to supply data for increasing number of on-chip compute units. Furthermore, memory energy consumption is another limiter for future processing platforms. Currently a 64-byte cache line size DRAM memory access consumes around 20–35 nJ (70 pJ/bit for DDR3, 40 pJ/bit for LPDDR2) which is orders of magnitude more energy than an on-chip double-precision

fused multiply-add operation (50 pJ in 40nm) or a 64-byte on-chip SRAM access (110 pJ for 8 KB SRAM in 40nm) [56, 69, 77]. From a technology scaling standpoint, the DRAM memory system is a fundamental limiting factor both in terms of energy and bandwidth for future parallel processing platforms.

Moreover, with the emergence of the big data era, data intensive workloads require intensive memory usage. Limited data reuse, disorganized data placement, and inefficient access patterns in such workloads put a substantial pressure on the memory system. Due to large working sets and limited locality, these workloads require large roundtrip data movement between the processor and DRAM. It is reported that current systems spend significant energy and time on data movement. Specifically, for scientific, mobile and general-purpose workloads 28–40%, 35% and more than 50% of the total system energy is spent on data movement respectively [70, 91]. Although there are various disruptive memory technology developments such as 3D-stacked DRAM or wide I/O, promised energy and bandwidth potentials can only be achieved with the efficient use of the memory system.

Exploiting the memory level locality and parallelism is the key for efficiently utilizing the DRAM based main memories. There exists architectural mechanisms that aim to recover the lost parallelism and locality through reordering memory accesses by scheduling [101], distributing accesses by interleaving [118], or data prefetching to hide the memory latency [27]. These approaches are mainly limited with the size of reorder queues and none of them addresses the data placement problem. Compiler based code transformations, on the other hand, are limited with data dependencies lacking dynamic runtime information [42, 80, 112]. Data layout transformation via reorganizing data in memory aims the inefficient memory access pattern and the disorganized data placement issues at their origin. However, modern processing platforms lack the ability to perform efficient data layout transformations mainly due to roundtrip data movements and bookkeeping costs for remappings.

Moreover, the complex hierarchy of the memory is abstracted as a flat address space in the modern programming models. Low level memory system details including data layout, address mapping, and memory access behavior are not exposed to the users. Hence, programmers spend significant

manual effort to optimize the algorithmic behavior of the application for an improved memory access performance. Often times these algorithmic optimizations are specific to the platform's abstracted away memory hierarchy and architectural details, lacking portability.

Therefore, given the landscape for the future processing platforms, memory system should be a first class design consideration to achieve high performance and energy efficiency.

1.2 Thesis Approach and Contributions

To achieve the desired first class design consideration of the memory system as a part of the high-performance and energy-efficient processing platforms, this thesis combines a formal abstraction framework and hardware/software mechanisms for optimizing the memory access. It focuses particularly on the following goals:

- A formal framework to represent and manipulate critical memory access optimizations through data layout, access pattern or address mapping transformations.
- An efficient hardware substrate that allows implementing formally represented optimizations such as changing the data layout and address mapping schemes, which are either fixed or very difficult to change dynamically in conventional processing platforms.
- Architectural/software mechanisms that utilize the developed hardware substrate to perform the memory access optimizations either explicitly as a part of the application, or transparently to improve the memory system performance.
- An algorithm/architecture co-design capability which enables co-optimizing computation flow, memory access and data layout where the memory system is exposed to the algorithm design space through the formal framework.

To achieve these high-level goals, this thesis makes the following main contributions:

A Formal Approach to Memory Access Optimization. From a mathematical standpoint, this thesis makes the key observation that manipulating the memory access patterns, data organization as well as address mapping schemes can be abstracted as permutations. The thesis demonstrates a formal framework based on tensor (Kronecker product) algebra that allows structured manipulation of permutations represented as matrices. This formal framework serves as a mathematical language to express the memory access optimizations as set of rewrite rules. This allows integration of these rules into an existing formula rewrite system called Spiral which enables automation [98]. Hence various implementation alternatives can be derived automatically to exploit the locality and parallelism potentials in memory.

HAMLeT Architecture for Data Reorganization. This thesis presents the HAMLeT (Hardware Accelerated Memory Layout Transform) architecture with the goal of highly-concurrent, low-overhead and energy-efficient data reorganization performed in memory. HAMLeT architecture is driven by implications and requirements from the permutation based mathematical framework. Parallel architecture with multiple SRAM blocks connected via switch networks performs high throughput local permutations. A configurable address remapping unit implements the derived affine index transformations for data reorganizations, which allows handling the address remapping completely in hardware. Memory access optimizations derived by the formal framework are directly mapped onto the HAMLeT architecture.

HAMLeT architecture is specifically tuned for near-data processing implementation using 3D-stacked DRAM. Integrated within 3D-stacked DRAM, behind the conventional interface, it not only minimizes the roundtrip data transfer but also utilizes high-bandwidth through-silicon via (TSV) based access and abundant parallelism provided by multiple layers/vaults/banks. Parallel streaming architecture can extract high throughput via simple modifications to the logic layer, keeping the DRAM layers unchanged. Overall, HAMLeT provides an efficient substrate for highly-concurrent, low-overhead and energy-efficient data reorganization performed in memory.

Software-transparent Data Reorganization. Taking the HAMLeT architecture as a substrate that enables very efficient data reorganizations, we demonstrate a software-transparent data reorganization performed in runtime. In this operation, the memory controller determines a disorganized data placement by monitoring the physical address stream. It uses HAMLeT to change the data layout and the address mapping. Reorganized data layout leads to better utilization of memory locality and parallelism, which improves the host processor performance. This thesis demonstrates series of hardware/software mechanisms to perform these operations, and their side effects, efficiently in memory.

Explicit Data Layout Transformation. As another fundamental use case, this work focuses on offloading and accelerating common data reorganization routines observed in high-performance computing libraries (e.g., matrix transpose, scatter/gather, permutation, pack/unpack, etc.) using the HAMLeT architecture. This gives the ability to efficiently perform data layout and address mapping transformations from the software, exposing these critical memory optimizations to the programmers. Furthermore, the thesis demonstrates special software/hardware mechanisms to handle the offloading and coherence management efficiently for the targeted near-data processing architecture.

Block Data Layout FFTs: Co-optimization of Computation, Memory Access and Data Layout. For certain types of problems, the separation of the computation and memory access is very difficult due to the interdependent nature of the two. This thesis also analyzes such a problem, fast Fourier transform (FFT), where the high-performance and energy-efficient implementations require a co-optimization between computation, memory access and data layout. Using the tensor based formal framework as a unifying representation, compute data flows, memory access patterns and data layouts are co-optimized as a part of the algorithm design space, where the customized block data layouts and their address remapping are achieved by utilizing the HAMLeT architecture.

1.3 Thesis Organization

The rest of this dissertation is organized as follows.

First, Chapter 2 gives the background on the fundamental concepts that the thesis builds upon. It introduces the modern memory system organization, operation and various DRAM technologies including planar and stacked DRAMs. It further introduces the near-data processing (NDP) systems then elaborates on 3D-stacked DRAM based NDP alternatives. Furthermore, it gives a background on data reorganization operation and its capabilities.

Next, Chapter 3 presents the mathematical framework used to describe and manipulate critical memory system optimizations, most importantly data reorganization and address remapping. It demonstrates various test cases that manipulate permutations for transforming data layouts, access patterns and address mappings.

Then, Chapter 4 introduces the HAMLeT architecture for efficient data reorganization and address remapping in memory. It demonstrates the micro-architecture of the fundamental components in the HAMLeT, namely data reorganization unit (DRU) and address remapping unit (ARU).

Chapter 5 demonstrates two fundamental use cases for the highly-concurrent, energy-efficient and low-overhead data reorganization performed in memory. It describes the software-transparent and explicit operations. It also poses the critical architectural and software support required for these modes of operations.

Next, Chapter 6 presents the system architecture and integration. It first presents the near-data processing architecture exploiting the 3D-stacked DRAM technology. Then, focuses on host/HAMLeT integration issues such as parallel host and HAMLeT memory accesses, in-memory coherence management, and software offload mechanisms for explicit operation.

Chapter 7 provides an experimental evaluation for the fundamental use cases of the 3D-stacked DRAM based implementation of the HAMLeT architecture. It provides the modeling and simulation details for the 3D-stacked DRAM based architecture. Then it demonstrates detailed analysis for

improvements and overheads of the data reorganization for both explicit and transparent operation. The evaluations are concluded by hardware synthesis results to analyze power and area cost of the fundamental components of the HAMLeT architecture.

Then, Chapter 8 demonstrates a case where the critical details of the memory system optimizations are considered as a part of the algorithmic optimizations. Specifically, it presents a co-optimization of computation, memory access and data layout through block data layout FFT implementations which exploits the dynamic data layout operation.

Finally, following the related work in Chapter 9, Chapter 10 offers conclusions and future directions.

Chapter 2

Background

2.1 The Main Memory System

The main memory is becoming increasingly critical bottleneck in achieving high performance and energy efficiency for various computing platforms. This problem, also known as the memory wall, is further exacerbated in the dark silicon era by multiple cores and on-chip accelerators (such as GPU, FPGA, ASIC cores) sharing the main memory and demanding more memory bandwidth. A single chip heterogeneous multicore computing system is demonstrated in Figure 2.1. Even though, emerging technologies including 3D-stacked DRAM address the main memory bottleneck by providing more bandwidth while consuming less energy, in practice, the offered high performance and energy efficiency potentials is only achievable via the efficient use of the main memory. In this section, a detailed description about the fundamental operation of the DRAM based main memory systems is followed by the 3D-stacked DRAM technology.

2.1.1 DRAM Organization and Operation

DRAM is a dynamic random access memory which has been adopted as the main memory in almost all of the computing platforms ranging from mobile, desktop, graphics processors up to supercom-

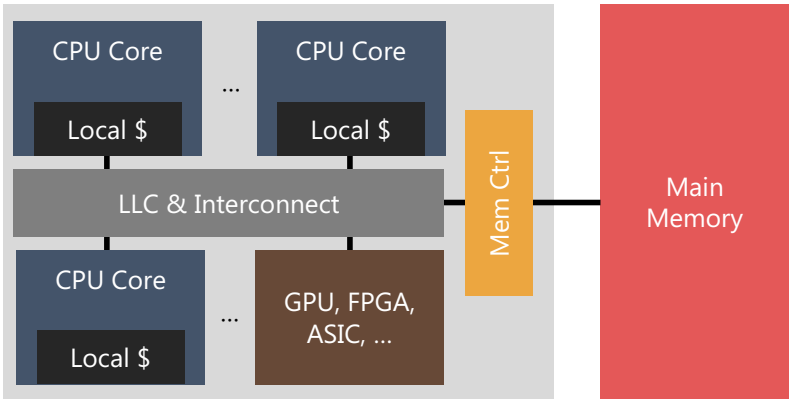


Figure 2.1: A single chip heterogeneous multicore computing system.

puters. DRAM memory cell has a very simple structure—a single transistor connected to a capacitor forms a single bit storage. This allows DRAM to achieve very high density. However, DRAM is a volatile memory so the contents of the memory cells leak over time. They have to be refreshed periodically to keep the stored values.

As shown in Figure 2.2, DRAM modules are divided hierarchically into (from top to bottom): ranks, chips, banks, rows, and columns. Each rank is constructed out of multiple DRAM chips. DRAM chips within a rank contributes to a portion of the DRAM word; they are accessed parallel in lock-step to form a whole memory word. Each bank within a DRAM chip has a *row buffer* which is a fast buffer holding the most recently accessed row (or page) in the bank. If the accessed bank and row pair are already active, i.e. the referenced row is already in the row buffer, then a *row buffer hit* occurs reducing the access latency considerably. In this case only the CAS (column address strobe) command is sent to access a column out of the active row. On the other hand, when a different row in the active bank is accessed, a *row buffer miss* occurs. In this case, the DRAM array is *precharged* and the newly referenced row is *activated* in the row buffer, increasing the access latency and energy consumption. Furthermore, when the accessed row buffer is not active then the DRAM array stays precharged which is called a *closed row*. An access to such bank still requires the activate command before the target row can be accessed. Therefore, to minimize the energy consumption and to achieve the maximum bandwidth from DRAM, one must minimize the row buffer misses. In other words, one must reference all the data from each opened row before switching to another row by

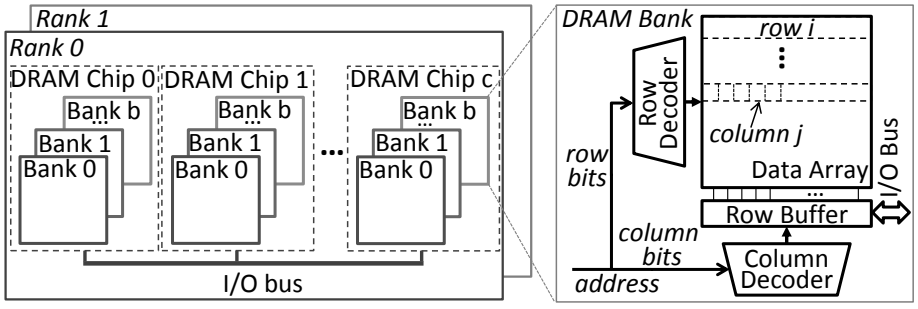


Figure 2.2: Overview of an off-chip planar DRAM module organization.

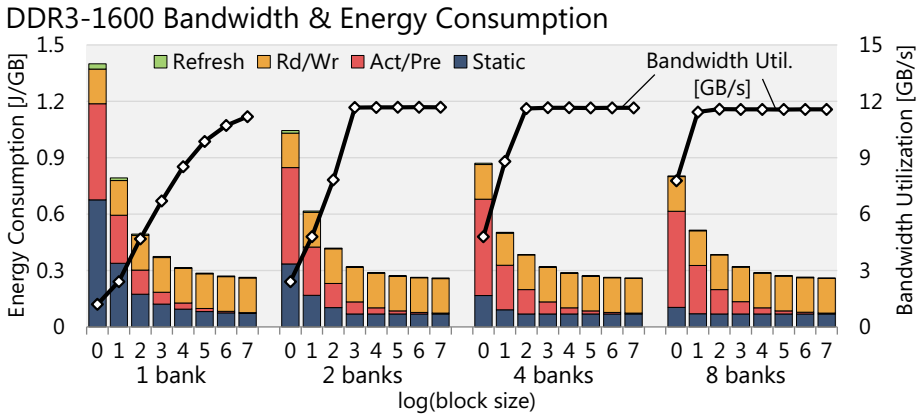


Figure 2.3: Locality/parallelism vs. bandwidth, power and energy tradeoffs for DDR3-1600 DRAM (1.5V, single rank, 8 bank, x8 width) [7].

exploiting the spatial locality in the row buffer.

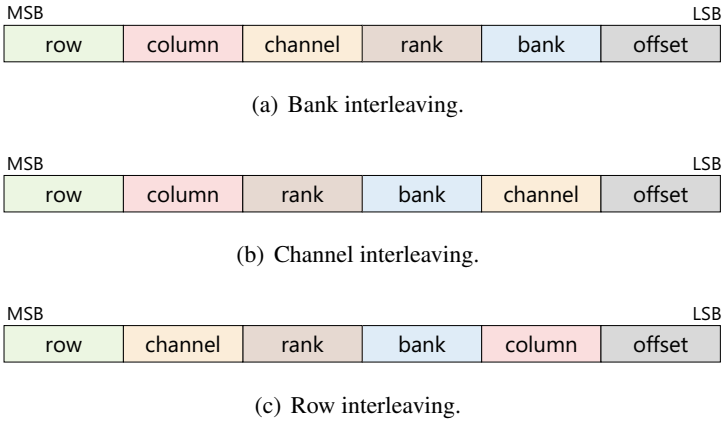
In addition to the row buffer locality (RBL), bank level parallelism (BLP) has a significant impact on the DRAM bandwidth and energy utilization. Given that different banks can operate independently, one can overlap the latencies of the row precharge and activate operations with the data transfer on different banks. BLP enables high bandwidth utilization even if the RBL is not fully utilized provided that the accesses are well distributed among banks. However, frequently precharging and activating rows in different banks increase the power and total energy consumed.

Figure 2.3 demonstrates the impact of RBL/BLP on DRAM bandwidth, power and energy consumption. In this experiment contiguous blocks of data are transferred from DRAM, where adjacent blocks are perfectly distributed among different banks. Therefore, the size of the data block corresponds to the number of elements referenced from an opened row. In Figure 2.3 we observe that the

achieved bandwidth increases with RBL (i.e. size of the data blocks) and/or BLP (i.e. number of banks to which the data blocks are distributed). If the BLP is limited (e.g. accesses are concentrated on a single bank), then RBL must be maximized to reach the maximum bandwidth. On the other hand, if the blocks are well distributed among banks, the maximum bandwidth can be reached with smaller block sizes, but with the cost of additional bank precharge/activate operations. Figure 2.3 also shows the energy consumption in Joules/GByte which corresponds to the total energy spent in transferring unit GB of data. Both BLP and RBL decrease total static energy by transferring the same amount of data faster, yet RBL is the key to reduce the total activate/precharge energy.

Access Scheduling. DRAM memory controllers implement access scheduling to maximize the system throughput by exploiting RBL and BLP. Considering the latency disparity of different commands in the DRAM and the capability of handling multiple requests concurrently, memory access scheduling can improve the overall performance by reordering the requests. Prioritizing the row-hit requests, i.e. first-ready first-come-first-served (FR-FCFS) [101], batch scheduling [86] or stall-time fair scheduling [87] are among widely adopted scheduling mechanisms. A limitation of the memory access scheduling to optimize for the system throughput is the limited size request queues of the memory controller. Reordering capability in the memory controller is limited with the total number of on-the-fly requests in the scheduling queues. Access scheduling does not address the inefficient memory access issue at its origin. It only helps to reclaim some of the lost parallelism/locality by reordering requests in a limited window.

Address Mapping. As described previously, DRAM memory modules consist of several hierarchical units such as channels, ranks, banks, rows and columns. Hence a particular location in the DRAM memory system is described by these *coordinates*. However, the main memory is abstracted with a flattened linear address space. The memory controller implements an address mapping scheme that maps this linear physical address space into the DRAM coordinates. In other words, the address mapping determines the DRAM coordinates, i.e. channel, rank, bank, row, column, of a memory accesses. Address mapping scheme determines how physically contiguous data,



(a) Bank interleaving.

(b) Channel interleaving.

(c) Row interleaving.

Figure 2.4: Various address mapping schemes including cache line interleaving (bank and channel) and row interleaving.

from the processors abstraction, are actually laid out in the DRAM. Hence, it is an important factor in determining the DRAM parallelism and locality utilization.

Most widely used address mapping policies include row (page) and cache line interleaving schemes. Row interleaving scheme interleave consecutive cache lines first into a row from a bank, then interleave the rows among banks and channels. Whereas, cache line interleaving scheme places consecutive cache lines into consecutive banks, ranks or channels. Example address mapping schemes bank interleaving, channel interleaving and row interleaving are shown in Figure 2.4(a), Figure 2.4(b) and Figure 2.4(c) respectively. Row interleaving is optimized for locality hoping that sequential accesses exhibit spatial locality where sequential locations in the address space are mapped to the same row. Hence an application that exhibits spatial locality can enjoy row buffer hits, improving the memory access performance. On the other hand, cache line interleaving is optimized for parallelism where consecutive locations in the address space are scattered to different banks aiming to increase the bank level parallelism. There are various hybrid approaches and sophisticated address mapping techniques, such as permutation based interleaving [118], that aim to exploit the memory parallelism and locality better. Moreover, application-specific efficient address mapping schemes can be determined via profiling.

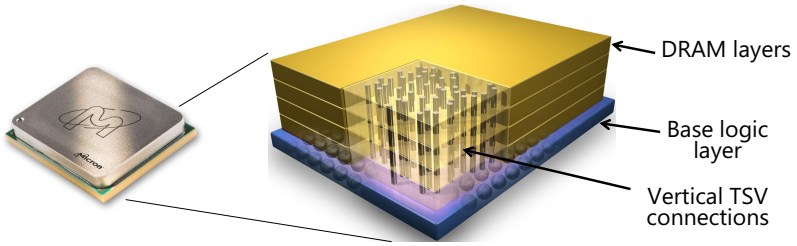


Figure 2.5: Micron's hybrid memory cube (HMC). Reprinted from [94].

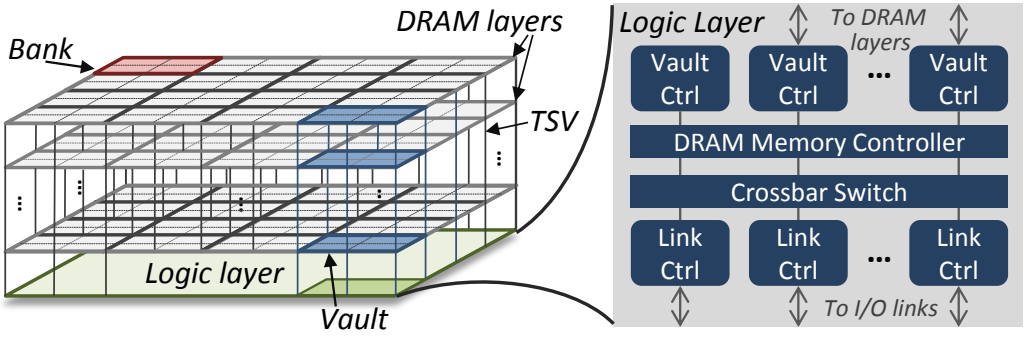


Figure 2.6: Overview of a HMC-like 3D-stacked DRAM [94].

2.1.2 3D-stacked DRAM Technology

3D-stacked IC is a broad term covering systems where multiple dies and/or wafers are stacked vertically. In the 3D-stacked DRAM technology, multiple DRAM dies and/or logic die(s) are stacked on top of each other and connected by vertical through silicon vias (TSV). TSVs allow connection through the wafer or die substrate where the active layer of the die can communicate vertically. Multiple such dies are connected using micro-bumps. By sidestepping the I/O pin count limitations, dense TSV connections allow high bandwidth and low latency communication within the stack. There are examples of 3D-stacked DRAM technology both from industry such as Micron Hybrid Memory Cube (HMC) [94], JEDEC standard High Bandwidth Memory (HBM) [16], Tezzaron Di-RAM [11], and from academia [49, 71].

Hybrid Memory Cube (HMC). Figure 2.5 shows the overview of the Hybrid Memory Cube (HMC). HMC features multiple DRAM dies and a base logic die which implements control units as well as high-speed serial communication interfaces as shown in Figure 2.6.

It consists of multiple layers of DRAM (currently 4 to 8 DRAM layers are planned) where each layer also has multiple banks. A vertical slice of stacked banks (or groups of two banks) form a structure called *vault*. HMC 2.0 features 32 vaults. Each vault has its own independent TSV bus and vault controller [65]. This enables each vault to operate in parallel similar to independent channel operation in conventional DRAM based memory systems. We will refer to this operation as *inter vault parallelism*.

Moreover, the TSV bus has very low latency that is much smaller than the typical tCCD (column to column delay) values [38, 111]. This allows time sharing the TSV bus among the layers via careful scheduling of the requests which enables parallel operation within the vault (e.g. [120]). We will refer to this operation as *intra vault parallelism*.

HMC is not a JEDEC standard. Instead of the conventional DDR standard, it has a serial communication interface. Serializer/deserializer (SerDes) units in the logic layer used to interface to off-stack, implementing a packet-based network communication. Despite its complexity, this interface allows direct short-reach links eliminating the need for silicon interposer. Currently 10, 12.5 and 15 Gb/s data rate lanes are planned where each link features 16 lanes and logic layer implements 4 to 8 links. 15 Gb/s lane configuration with 8 links can reach up to 480 GB/s aggregate off-stack bandwidth.

HMC can achieve very high energy-efficiency as well compared to conventional DRAM modules. It is reported that the overall energy efficiency is 10.48 pj/bit including DRAM access, TSV transfer and SerDes based I/O [65]. 6.78 pj/bit accounts for the logic layer which combines the SerDes and controller units. This is a significant improvement when compared to 70 pj/bit energy efficiency of the DDR3 and 18-22 pj/bit energy efficiency of the GDDR5 [65, 89].

Furthermore, the base logic layer implements memory/vault controllers that schedule the DRAM commands while obeying the timing/energy constraints. This allows less complex memory controller on the host side. Data switching between vaults and I/O links is achieved by a crossbar interconnect. Typically, these native control units do not fully occupy the logic layer and leave a real estate that could be taken up by custom logic blocks [65]. However, the thermal and power

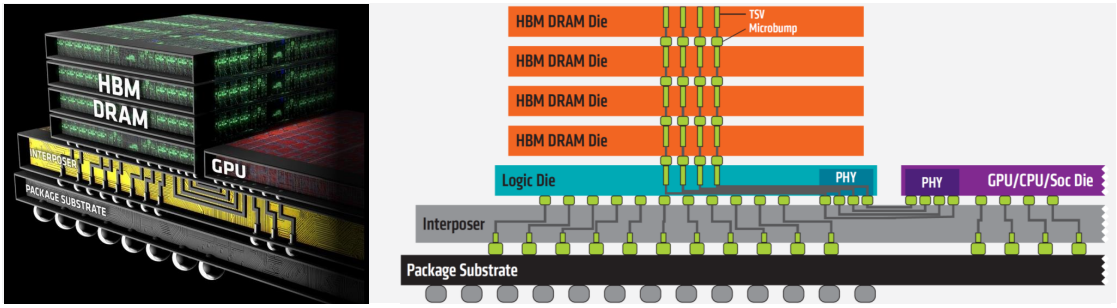


Figure 2.7: AMD's high bandwidth memory (HBM). Reprinted from [1].

constraints limit the complexity of the custom logic.

High Bandwidth Memory (HBM). As shown in Figure 2.7, High Bandwidth Memory (HBM) integrates multiple DRAM dies and an optional logic die by using TSVs and micro-bumps. Dense TSV connections at the base layer are directly connected to the host processor, as opposed to the HMC system. This connection is implemented through a silicon interposer substrate. HBM is a standardized interface adopted by JEDEC [16].

HBM features a wide interface of 128-bits per channel. In HBM, vertical stack of banks form a channel, equivalent to a vault in HMC terminology. Each channel can operate in parallel and not necessarily in lock-step fashion. This enables a high parallelism. Furthermore, these channels are directly exposed to the host processor. Currently, HBM supports 8 channels per stack where each stack supplies up to 32 GB/s of bandwidth which results in 256 GB/s per stack. Note that this bandwidth is directly available to the host processor, in contrast to HMC where part of the total bandwidth is spent on non-data packets over the serial network communication. Moreover, HBM can reach 6-7 pj/bit of energy efficiency. Due to its simplicity, HBM is more energy efficient than the HMC system which achieves around 10.5 pj/bit [65, 89].

Wide I/O. Wide I/O and Wide I/O 2 are also JEDEC compliant technologies, currently geared towards the mobile devices. They involve a DRAM layer directly stacked on top of a processor. This can be achieved through flip-chip, 2.5D interposer or 3D-stacked using TSV. Wide I/O uses large amount of TSVs at lower frequency to achieve high energy efficiency. This results in the

Table 2.1: Comparison of 3D-stacked DRAM systems Wide IO2, HBM, HMC.

Feature	Wide IO 2	HBM	HMC
Target system	Mobile	Graphics	Server/Enterprise
JEDEC compliant	Yes	Yes	No
Interface	Wide parallel	Wide parallel	Serial SerDes
Interface width	256-512 bits	8x128 bits	4-8 links, each 16 lanes
Interface voltage	1.2 V	1.2 V	1.2 V
Data rate per pin	1066 Mbps	2000 Mbps	10-12.5-15 Gbps
Max. Bandwidth	68 GB/s	256 GB/s	320 GB/s
System integration	Stacked on host	Interposer	High-speed links
Main shortcoming	Thermal/power impact	Depends on interposer	Non JEDEC standard

lowest I/O power for large bandwidth.

Table 2.1 summarizes the differences and salient features of the discussed 3D-stacked DRAM technologies (HMC, HBM and Wide IO2).

Silicon Interposer Based Connection. Host memory connection technology is mainly driven by the implemented memory interface. For example, HBM and Wide I/O technologies implement a parallel direct connection interface. This requires high number of TSV based IO connections. A silicon interposer based substrate used to implement the high number of IO connections between the processor and the memory. Silicon interposer can provide high number of connections but it requires a separate interposer layer on the main substrate. Despite its standardized JEDEC compliant interface, interposer based connection limits the drop-in replacement flexibility. Replacing and distributing memory modules for different host processors becomes more difficult as the connection is tied to an interposer. Yet, direct connection from the internal TSVs to the host enables a simple communication. An example system architecture using silicon interposer based connection is given in Figure 2.9.

Short-reach Link Based Connection. HMC features SerDes units integrated in the logic layer which allows serial network based interface. This technology requires high-speed link based connection. Point-to-point link connection eliminates the need for silicon interposer substrate. The

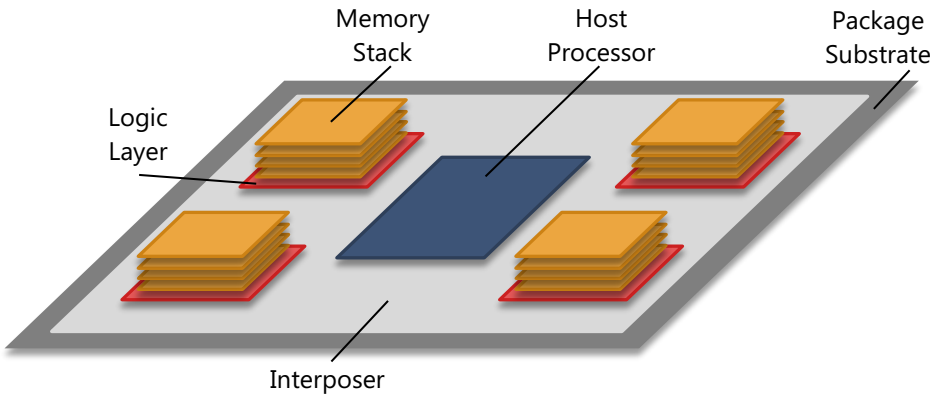


Figure 2.8: System architecture using silicon interposer based connection.

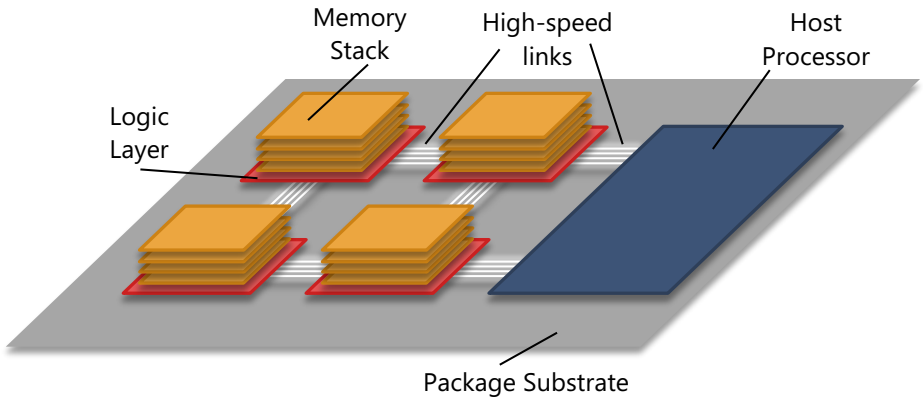


Figure 2.9: System architecture using point-to-point SerDes based link connection.

HMC goes into a conventional package and it is integrated to the PCB (printed circuit board) where the links directly connect it to the host. This limits its off-stack bandwidth however it enables flexible drop-in replacement capability. An example system architecture using point-to-point SerDes based link connection is shown in Figure 2.9.

2.2 Hitting the Memory and Power Walls

Following the Moore's law which suggests doubling the transistors on chip every 18 months, improvements in the transistor technology scaling enables putting more compute elements on chip in the form of processors and accelerators [84]. However, DRAM technology scaling has been lack-

ing behind in matching the same trend which causes the well-known memory wall problem [115]. With the rapid growth of the Internet and world wide web, the big data era emerged where the data intensive applications require intensive memory usage. Especially the common roundtrip data movement between the processor and DRAM for the data intensive applications puts more pressure on the memory system.

Moreover, after the Dennard scaling era, energy became the limiter of the high-performance systems. With the Dennard scaling, every processor generation implemented twice as many transistors which are clocked almost half times faster than the previous generation while consuming the same power. But after the Dennard scaling, doubled transistors will consume 40% more power when clocked at the same frequency [68]. This creates the dark silicon problem where the part of the chip is under-clocked or completely turned off to meet the power consumption limitations.

In addition to the on-chip power consumption, memory power consumption poses even more critical problem for high-performance systems. Accessing the off-chip data is 2 to 3 orders of magnitude more costly than on-chip access. Energy cost of a double-precision fused multiply-add operation and a 64-bit on-chip SRAM access are estimated as around 50 pJ and 14 pJ, whereas the same width off-chip access is more than 10 nj [56, 69, 77]. Moreover, with the transistors shrinking every generation, wire energy becomes more critical since wire capacitance per mm remains the same. When executing commodity data intensive workloads, modern processors spend substantial amount of time and energy on data movement. It is reported that current systems spend 28–40%, 35% and more than 50% of the total system energy on data movement for scientific, mobile and general-purpose workloads respectively [70, 91]. According to technology scaling trends this ratio will increase in the future systems [69].

The given landscape of computing systems and the technology scaling trends point towards an energy constrained computing era. In this era, the energy consumption will be dominated by the communication costs, both on-chip and off-chip. Emergence of data intensive applications which require intensive memory access, data movement and I/O, further aggravates the communication energy problem. Future high-performance computing systems have to find ways to minimize the

data movement to tackle the energy consumption limitation.

2.3 Near Data Processing

Pursuing the Von Neumann computation model, conventional processing platforms have been built following a computation-centric model. In the computation-centric model, data is moved throughout the memory hierarchy from disk to on-chip caches as necessary to perform arithmetic and logic operation on the centralized host processor. This model has been sufficient for decades where the intensive arithmetic and logic computations are much more costly than the data movement. Furthermore, deep memory hierarchies and large caches help alleviate the cost of data movements by exploiting the data locality. However, prevalent technology scaling trends point towards processing systems where the latency and energy costs are dominated by the data movement. Furthermore, data intensive applications that operate on large datasets require data movement across the entire memory hierarchy which further exacerbates the communication costs.

The shift from the computation dominated processing systems to the communication dominated systems requires a shift from the computation-centric processing model to a data-centric one. In contrast to the computation-centric model, the data-centric processing model distributes the computation to multiple levels of the memory hierarchy. Hence, instead of moving the data to a centralized processor, the processing is handled where the data reside using nearby processing elements. With the near-data processing (NDP) paradigm, instead moving data throughout the system, the compute is *moved* to the data. Hence, NDP paradigm can reduce the data movement costs significantly.

Near data processing is not a new concept. It has been explored in variety of technology contexts in the forms of custom compute units near on-chip/off-chip memory, I/O or disk [39, 46, 57, 60, 67, 90, 93]. Most of these NDP approaches focused on computation near main memory in the form of processing elements and DRAM or embedded DRAM (eDRAM) integrated in the same chip. Although, these efforts demonstrate significant improvements, NDP technology suffered in widespread adoption. One of the main factors in this limitation is the manufacturing difficulty and

costs. DRAM and logic process technologies are optimized differently. Proposed NDP systems either difficult and costly to manufacture, or the resulting implementations offered limited performance.

Nevertheless, with the emergence of big-data workloads and communication-dominated processing systems, reducing data movement becomes inevitable. Furthermore, with the technology advances such as 3D stacked integration, NDP regained a substantial research attention [65, 76, 114, 120]. Next we will focus on NDP systems exploiting 3D-stacked integration technology to reduce the off-chip data movements.

2.3.1 3D Stacking Based Near Data Processing

3D-stacked IC technology enables integration of multiple dies implemented in different transistor technologies. 3D-stacked DRAM is an example for the 3D integration technology where multiple DRAM layers and logic layer(s) are vertically integrated using the through silicon via (TSV) based connection [16, 94]. Dense TSV based vertical connection enables very high bandwidth and low latency internal communication, bypassing limited off-chip pins. This creates an efficient substrate to implement near-data processing systems. There exists several alternatives for integrating NDP accelerators near 3D-stacked DRAM that provide tradeoffs in cost, energy, complexity and performance. Figure 2.10 demonstrates three main alternatives for 3D-stacked DRAM based NDP.

First option is to integrate the accelerator off-stack (① in Figure 2.10). This is the simplest scheme where the accelerator is external to the stack. Hence, the memory stack design is not modified. Furthermore, the thermal impact of the accelerator on the DRAM layers is eliminated. However, the accelerator and the memory stack communicate through external links or interposer. This limits the bandwidth between the accelerator and the DRAM. Moreover, each access still suffer from high energy data transfer cost.

Integration of the accelerator as a part of the memory stack (i.e. ② and ③ in Figure 2.10) allows utilizing the bandwidth and energy-efficiency potential of the 3D-stacked DRAM more efficiently.

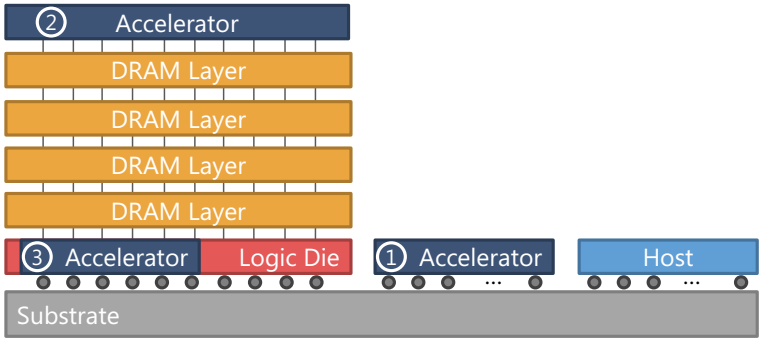


Figure 2.10: Accelerator integration to a 3D-stacked DRAM: (1) Off-stack, (2) stacked as a separate layer, (3) integrated in the logic layer.

Integration as a separate layer (②) reduces the memory access distance to TSV based accesses. However, the data transfer requires additional hops through the logic controller layer to access the DRAM layers. To avoid the access through logic layer, accelerator layer can implement its own DRAM controller interfaces. However, this further increases the complexity in the accelerator layer. It also requires arbitration between multiple controllers in the stack.

Furthermore, integrating the accelerator as a part of the logic layer (i.e. ③), behind the external interface, unlocks the access to internally available bandwidth and energy efficiency provided by TSV based vertical connection. In this scheme, the accelerator directly communicates with the logic layer memory controllers. Similar to ②, this scheme also eliminates the external data transfer. Moreover, it eliminates the detour on the logic layer for accesses between the DRAM and the accelerator. The main shortcoming of this approach, however, is the limited area, power consumption and temperature budget for a custom accelerator implementation in the logic layer. As discussed, the logic layer already includes native control units such as an interconnection fabric, memory/vault controllers, SerDes units, etc. It is reported that these units leave a real estate to be taken up by custom logic [94]. However, compared to a separate die, the area and power consumption headroom is more limited.

2.3.2 Hardware Specialization for NDP

Complete processing elements integrated in the logic layer are limited in utilizing the internal bandwidth while staying within an acceptable power envelope. For example, in [97], although simple low EPI (energy per instruction) cores are used in the logic layer, some of the SerDes units are deactivated to meet the power and area budgets, sacrificing the off-chip bandwidth. In [64], it is reported that more than 200 PIM (processing in memory) cores are required to sustain the available bandwidth which exceeds the power limit for the logic layer. Furthermore, NDP with simple logic layer configuration that meets the power and area constraints can even lead to application slowdowns [117].

Specialized hardware units are more efficient in providing higher throughput (hence memory intensity) per power consumption that is far beyond what general purpose processors can provide. Modern general purpose processors implement mechanisms such as instruction decoding, register renaming, out-of-order scheduling, branch prediction, prefetching, etc. that does not make actual computation but assist the processor to increase the execution throughput. These overheads create 2 to 3 orders of magnitude energy consumption difference between a specialized ASIC and a general purpose processor while processing the same application [62]. Hence, hardware accelerators specialized in simple compute mechanisms can be a good fit for area and power limited near-data processing systems.

2.4 Data Layout and Reorganization

For a simple compute mechanism candidate that can be a good fit for 3D-stacked DRAM based NDP, we focus on data reorganization. We refer to an operation that relocates data in the memory as *data reorganization*. Data reorganization is a fundamental and powerful technique to optimize the memory access performance.

An application's memory access performance is fundamentally determined by its algorithmic be-

havior that generates the logical sequence of memory accesses and the data layout in memory that determines the actually accessed physical locations. In the current modern programming model abstractions, actual data layout and memory access behavior are not exposed to the users. Hence, programmers spend significant manual effort to optimize the algorithmic behavior of the application for an improved memory performance. Often times these optimizations are specific to a platform's memory hierarchy and architectural details. Moreover, existing mechanisms such as memory access scheduling and compiler transformations provide limited improvements by reshaping the memory access patterns to partially recover the underutilized memory locality and parallelism.

The underlying data layout can be the limiting factor where solely optimizing the application's algorithmic behavior or the recovery mechanisms would not be sufficient for the best memory access. For example, if the two data elements sequentially required by an application are mapped to two different rows in a DRAM bank, neither algorithmic optimizations nor memory access reordering can solve the inevitable row buffer miss. Moreover, different applications or even different algorithms for the same application may favour different data layouts for a better memory access. The optimized layout for an application can also change with different architectures.

For an application which exhibits poor memory access performance (i.e. low bandwidth, low energy efficiency, high row misses, low memory parallelism) reorganizing its data layout can improve the memory access performance significantly. In Figure 2.11 we demonstrate a motivational example for memory access improvement via data reorganization. We run the memory trace for the PARSEC [32] benchmark *facesim* available from [14] on the USIMM DRAM simulator [35] modeling a single channel DDR3-1600 DRAM [7] with open page policy and FR-FCFS (first-ready, first-come first-serve) scheduling. Figure 2.11(a) demonstrates the normalized average address bit flip rate in the memory access stream. Address bit flip rate (BFR) is determined by recording the changes (or flips) for each DRAM address bit. The basic idea is that the highly flipping bits correspond to frequent changes in short time which are better suited to be mapped onto bank or rank address bits to exploit the parallelism. On the other hand, less frequently flipping bits are better suited to be mapped onto row address bits to reduce the misses in the row buffer. The data layout for this

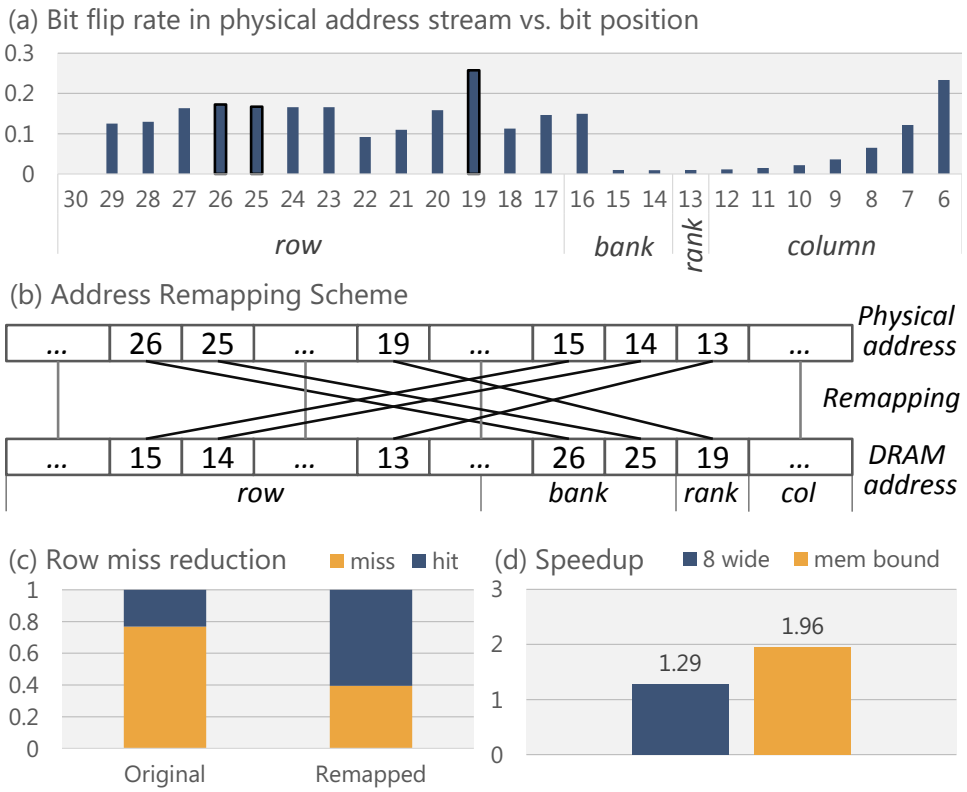


Figure 2.11: (a) Normalized bit flip rate in the physical address stream, (b) simple address remapping scheme and (c) row buffer miss rate reduction via the address remapping and (d) the resulting performance improvements for 8-wide and infinite compute power systems for *facesim* from PARSEC.

application can be reorganized such that the high BFR bits are mapped into bank/rank region of the DRAM address and the low BFR bits are mapped into the row region of the DRAM address. Figure 2.11(b) shows an example remapping that changes the address mapping such that highly flipping bits {26, 25, 19} are swapped with {15, 14, 13}. Figure 2.11(c) presents the reduction in the row buffer miss rate where data layout is reorganized as described. Finally, Figure 2.11(d) shows the achieved speed-up via the new mapping on an 8-wide 3.2 GHz processor. It also shows the upper bound for performance improvement where the overall runtime is purely memory bound such that the non-memory instructions are executed in a single cycle.

The results in Figure 2.11 show that the default data layout for an application can cause inefficient performance where reorganizing its dataset leads to significant performance improvement. Hence, global data layout and address mapping schemes can cause inefficient memory system utilization.

There exists efficient address mapping schemes such as [118] that improves the memory access performance for variety of applications. Moreover, application-specific address mapping schemes that will lead to higher parallelism and locality can be determined.

The previous example demonstrates an architectural solution; given the fixed application behavior, data reorganization improves the memory performance transparently. Data reorganization can also be a critical building block of an applications. For example, data reorganizations appear as an explicit operation in several scientific computing applications such as signal processing, molecular dynamics simulations and linear algebra computations (e.g. matrix transpose, pack/unpack, shuffle, data format change etc.) [12, 21, 30, 55, 58]. High performance libraries generally provide optimized implementations of these reorganization operations [5, 55]. There are several works demonstrating reorganization of the data layout into a more efficient format to improve performance [21, 36, 58, 92, 107, 108]. Hence, such applications can benefit from a substrate that allows efficient data reorganization exposed to the programmer.

In addition to the ability to accelerate critical data reorganization type operations, efficient data reorganization capability exposed to the programmer gives an ability to reconsider entire algorithms to exploit efficient dynamic data layout computing. There are intrinsically difficult problems where an efficient solution include the co-optimization of the algorithm behavior both in terms of compute dataflow and memory access pattern, as well as the underlying data layout [21]. Such problems can benefit from an ability to consider and more importantly manipulate the underlying data layout as a part of their algorithm design space.

However, reorganizing data in the memory comes with substantial performance overheads, and bookkeeping costs. Considering the high precision and large data sizes, significant fraction of the dataset reside in the main memory. Therefore, ideally, most of the data reorganization operations are memory to memory. However, on conventional systems data needs to traverse the memory hierarchy and the processor, incurring large energy and latency overheads. Roundtrip data transfer consumes the shared off-chip memory bus bandwidth which also degrades the performance of the other applications that run on the system. Furthermore, there exists a bookkeeping cost for track-

ing the mapping from original to reorganized locations. Conventionally this is either handled by implementing a hardware lookup table that store the mappings from each original location to the remapped location, or through costly OS page table updates and TLB invalidations.

NDP Approach for Data Reorganization. From a workload characteristic standpoint, data reorganization do not require intensive arithmetic and logic computations. Furthermore, these operations are memory to memory, in other words the input data reside in the main memory and the output data end up in the main memory in a different shape. These two fundamental characteristics make the data reorganization a good fit for NDP.

Chapter 3

A Formal Approach to Memory Access Optimization

This chapter presents a mathematical framework used to describe and manipulate memory access optimizations. Transforming data organizations, access patterns and address mappings are represented by permutations. A tensor (or Kronecker product) based mathematical framework is introduced to structurally manipulate the permutations represented as matrices. This framework enables extracting important practical implications related to the data and control flow, address mapping, memory access behavior, and data layout. After exemplifying the key practical implications, a formula rewrite system is introduced that enables automated restructuring of permutations.

3.1 Mathematical Framework

A permutation is defined as a rearrangement of all the members of a set to an order. It is a bijective function from a set to itself. From a memory access optimization perspective, data layouts, memory access patterns and address mapping schemes all correspond to a particular mapping function that represent an order of entities. For example, data layout corresponds to the mapping of a set of data to a physical memory space. Memory access pattern represents the order of locations that are accessed

sequentially. Furthermore, address mapping is a translation function which maps virtual addresses to physical addresses, or physical addresses to DRAM locations. Hence, data layout transformation, access pattern reordering or address remapping can be captured by a permutation.

3.1.1 Formula Representation of Permutations

Permutations can be expressed as matrix-vector multiplication such that $d_{out} = P_n.d_{in}$ where P_n is the n-by-n permutation matrix, d_{in} and d_{out} are n-element input and output vectors. A permutation matrix is a binary matrix where each row or column has a single non-zero element. It permutes the input vector based on the locations of the non-zero elements. For example, a permutation matrix P_8, which is a special permutation $L_{8,2}$ as we will see later (stride permutation), is given as follows:

$$P_8 = L_{8,2} = \begin{bmatrix} 1 & . & . & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & 1 & . \\ . & 1 & . & . & . & . & . & . \\ . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & 1 \end{bmatrix} \tag{3.1}$$

The Kronecker (or tensor) product based formalism developed in [109] captures the structures in the decomposition of the permutation matrices. This formalism is further extended in SPL (Signal Processing Language) to represent fast signal processing algorithms [116]. In this thesis, the Kronecker product formalism serves as the language for expressing and manipulating permutations and it is the basis of the developed mathematical framework.

The basic elements of the framework are special structured matrices and matrix operators. Before going forward, we define these special matrices and operators.

Special Matrices. Special structured matrices are building blocks of the formal framework. First, $n \times n$ identity matrix I_n and $n \times n$ reverse identity matrix (swap permutation) J_n are given.

$$\mathrm{I}_n = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}, \quad \mathrm{J}_n = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix}$$

Permutation matrices are another family of the important building blocks. For example the *stride permutation* operation, denoted as $\mathrm{L}_{nm,n}$ or L^{nm}_{n}, takes the elements from d_{in} at stride n in a modulus nm fashion and puts them into consecutive locations in the nm-element d_{out} vector:

$$d_{in}[in + j] \rightarrow d_{out}[jm + i], \quad \text{for } 0 \le i < m,\ 0 \le j < n.$$

An example stride permutation matrix $\mathrm{L}_{8,2}$ is previously given in (3.1). Another specific permutation matrix is the cyclic shift matrix $\mathrm{C}_{m,n}$ which applies a circular shift of n elements in the m element input vector:

$$\mathrm{C}_{m,n} = \left[\begin{array}{c|c} & \mathrm{I}_n \\ \hline \mathrm{I}_{m-n} & \end{array}\right].$$

For example, the cyclic shift matrix $\mathrm{C}_{5,2}$ is given as:

$$\mathrm{C}_{5,2} = \begin{bmatrix} . & . & . & 1 & . \\ . & . & . & . & 1 \\ 1 & . & . & . & . \\ . & 1 & . & . & . \\ . & . & 1 & . & . \end{bmatrix}$$

Finally, two basic vectors, n-element column vector of 0's μ_n and n-element column vector of 1's ν_n, are given as follows.

$$\mu_n = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \nu_n = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

Matrix Operators. Product is a straightforward matrix operator given as $A \cdot B = AB$. Then, we

introduce two matrix operators, direct sum ($\oplus$) and tensor (Kronecker) product ($\otimes$). These operators are useful in structured combinations of the matrices.

The direct sum operator ($\oplus$) composes two matrices into a block diagonal:

$$A \oplus B = \left[\begin{array}{c|c} A & \\ \hline & B \end{array}\right].$$

Tensor (Kronecker) product is a very critical matrix operator in this framework which is defined as:

$$A \otimes B = \left[a_{i,j}B\right], \text{ where } A = \left[a_{i,j}\right].$$

Two important special cases arise when either of the matrices of the tensor product is the identity matrix. If the left operand is the identity matrix ($\mathrm{I}_n \otimes A$), then it forms a simple block diagonal matrix:

$$\mathrm{I}_n \otimes A = \underbrace{A \oplus A \oplus \ldots \oplus A}_{\text{n times}} = \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix}$$

As we will see later, this operation corresponds to repetition of the small kernel or permutation over a large dataset. On the other hand, if the right hand operand is the identity matrix ($A \otimes \mathrm{I}_n$), then it also produces a interesting case where each element is replicated n times:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \mathrm{I}_2 = \begin{bmatrix} a & & b & \\ & a & & b \\ c & & d & \\ & c & & d \end{bmatrix}$$

Interpretations. The permutations created by the provided matrices and matrix operators can be used to express a data layout, memory access pattern or an address mapping.

For example, a streaming access to n elements corresponds to the identity matrix I_n. On the other

hand, $\mathrm{L}_{m,k}$ represents stride-k accesses to a group of m elements. These can be combined such that the overall access pattern is given as $\mathrm{L}_{m,k} \otimes \mathrm{I}_n$. This case represents a block-stride access pattern, where n element burst accesses are separated by $k * n$ element strides.

Similarly, a permutation can express a data layout. For example, given an $n \times n$ matrix and assuming a row-major increasing indices for the matrix elements, I_{n^2} maps the consecutive elements linearly into the memory address space which leads to row-major data layout. However, $L_{n^2,n}$ maps the consecutive elements into stride-n separated locations in the memory, which leads to column-major data layout.

As we will see next, these matrix expressions can be restructured through formula identities rules.

3.1.2 Formula Identities

Matrix expressions discussed so far which combine provided special matrices through matrix operators are called *formulas*. These formulas are the basic representations in the SPL [116], so formula and SPL expression terms are used interchangeably. Formulas essentially represent an algorithm (or a dataflow) of a matrix that transforms the input vector. The matrices can be any linear transform but this thesis mainly focuses on permutations.

Different formulas can correspond to the same matrix which means that they essentially perform the same computation but with a different algorithm. This is achieved through the *formula identities*. A formula identity is a mathematical identity that restructures a matrix which allows representing the same matrix, i.e. ultimately the same computation, with different formulas, i.e. different algorithms. A few example formula identities are given below.

$$\begin{aligned} A_n \otimes B_m &\rightarrow (A_n \otimes \mathrm{I}_m)(\mathrm{I}_n \otimes B_m) \\ A_n \otimes B_m &\rightarrow \mathrm{L}_{mn,n}(B_m \otimes A_n)\mathrm{L}_{mn,m} \end{aligned}$$

We will particularly develop formula identities to manipulate permutations expressed in SPL for various goals. These goals include, but not limited to, optimizing memory accesses for better par-

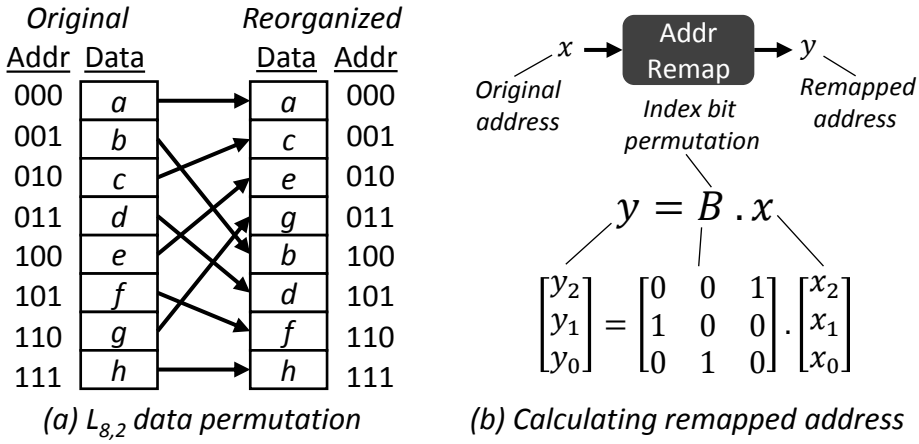


Figure 3.1: An example data reorganization ($L_{8,2}$) on 8 elements and the corresponding address remapping scheme.

allelism and locality, transforming the data layout for a particular application, changing the DRAM address mapping and restructuring a permutation to fit the local permutations into the local memory.

Next we particularly focus on the data reorganization where the data layout transformation, address remapping and memory access patterns are represented by permutations.

3.2 Permutation as a Data Reorganization Primitive

From a data reorganization perspective, a permutation takes an input dataset d_{in} and rearranges the elements to produce d_{out} such that $d_{out} = P_n.d_{in}$. Here, P_n is the n-by-n permutation matrix, d_{in} and d_{out} are n-element input and output datasets represented as vectors.

Let's take $L_{8,2}$ as an example data reorganization. $L_{8,2}$ is an 8×8 permutation matrix, hence it permutes 8-element input vectors. Figure 3.1(a) shows the corresponding reorganization of an 8-element dataset according to $L_{8,2}$. After the permutation reorganizes the dataset, the original addresses will hold stale data. For example an access to the location 001 will return the data *c*, which originally stored data *b*.

As presented in Figure 3.1(b) an address remapping mechanism that forwards the input addresses

(x) to their new locations (y) can solve this problem. Each permutation has a corresponding index transformation that represents the address remapping for a data reorganization. Figure 3.1(b) shows the index transformation of the permutation $L_{8,2}$, given as $y = Bx$. This unit will forward every access x to their new locations y via $y = Bx$. Following the previous example, an access to $x = 001$ will be forwarded to the location $y = 100$ via $y = Bx$ such that the returned data will be b as expected. This simple example demonstrates that the index transformation can be used to remap the addresses after a reorganization.

Permutations also capture the memory access pattern information of the data reorganization. Until now, the discussed permutations focused only on spatial locations and did not consider the time ordering. While reorganizing a dataset through a permutation, the sequence of the memory accesses are generated based on the performed permutation. When performing the permutation L^8_2, as shown in Figure 3.1(a), the corresponding memory accesses for reading and writing will be sequential and permuted order respectively. It is possible to rearrange the order of accesses again through permutations.

Overall, permutation is an effective means of representing and manipulating data reorganization operations. In this section our main goals are:

- Use the permutations as primitive operations to represent data reorganizations.
- Develop a systematic way of determining the index transformation for general permutation based reorganization operations to represent the address remappings.
- Develop a formula rewrite system and restructure the permutations to optimize for memory access patterns.

3.3 Index Transformation

Each permutation has an index transformation that transforms the indices of the input data and maps them to the corresponding locations in the output. In the context of data reorganizations, the index

transformation corresponds to the address remapping.

Our goal is to define a mapping function f_π that determines the corresponding index transformation for a given permutation. Specifically, it will determine the address remapping from the original location x to the remapped location y such that:

$$y = Bx + c \tag{3.2}$$

Here, assuming n bit addresses, x and y are n-element bit-vectors that represent the input and remapped addresses respectively. B is an $n \times n$ bit-matrix and c is an n-element bit-vector. In this form, the index transformation is an affine transformation on the address bits. This thesis only focuses on the class of permutations whose index transformations are affine functions and B is a permutation matrix, i.e. single non-zero in each row or column. This class covers many important practical permutations including stride permutation, bit reversal, swap permutation and their combinations through product, tensor product and direct sum. Hence, this class of permutations can express many important data reorganizations such as, matrix transpose, pack/unpack, swap, shuffle, copy/move, multidimensional data array rotation, recursively blocked data layouts, etc.

The bit representation for the index transform allows us to calculate all the remapped addresses. A crucial implication of this property is that one can reorganize the elements within the memory and then use the index transformation for address remapping such that the logical abstraction kept unchanged. As we will see later, this enables data layout transformation completely in hardware transparent to the software stack.

Stride permutation. First, we focus on the class of permutations that are constructed by combinations of the stride permutations (L) with identity matrices (I) via tensor product ($\otimes$) and matrix multiplication (.). This class of permutations can represent a variety of important data reorganization operations such as shuffle, pack/unpack, matrix transpose, multi-dimensional data array rotation, blocked data layout, etc. For this class of permutations, the fundamental properties of f_π

are given as follows:

$$f_\pi(\mathrm{L}^{2^{nm}}_{2^n}) \to B = \mathrm{C}_{nm,n}, \quad c = \mu_{nm} \tag{3.3}$$

$$f_\pi(\mathrm{I}_{2^n}) \to B = \mathrm{I}_n, \quad c = \mu_n \tag{3.4}$$

$$f_\pi(P \cdot Q) \to B = B_P \cdot B_Q, \quad c = c_P + c_Q \tag{3.5}$$

$$f_\pi(P \otimes Q) \to B = B_P \oplus B_Q, \quad c = \begin{bmatrix} c_P \\ c_Q \end{bmatrix} \tag{3.6}$$

For this class of permutations, the B matrix is constructed out of cyclic shifts, multiplication and composition (direct sum) operators. Here we emphasize that the combinations of these operators can represent "all" possible permutations on the address bits. Hence this class covers all of the data permutations whose index transformation is also a permutation on the address bits. We will later focus on the practical implications of these properties.

Swap permutation. We extend the reorganization operations to include different classes of permutations. First, we define the *swap permutation* J_n as follows:

$$d_{in}[i] \to d_{out}[n-i-1], \quad \text{for } 0 \le i < n.$$

J_n is simply the I_n matrix with the rows in the reversed order, as previously shown in (3.1.1). Swap permutations are especially useful to represent out-of-place transformations and swap type of operations. For example, $\mathrm{J}_2 \otimes \mathrm{I}_n$ can be interpreted as simply swapping the two consecutive n-element regions in the memory. Address remapping for the swap permutation is given as follows:

$$f_\pi(\mathrm{J}_{2^n}) \to B = \mathrm{I}_n, \quad c = \nu_n \tag{3.7}$$

Morton layout. Morton data layouts, or in general space filling curves, are based on recursive blocking [85]. Large data sets are divided into blocks recursively until the small leaf blocks reach to the desired size. There are various forms of Morton layouts depending on the order of blocking

while the most commonly used one is the Z-Morton ($Z_{2^n,2^m}$). Z-Morton layout is useful for various scientific, high-performance and database applications [33, 92]. Z-morton layout can be represented only by stride permutations, so the properties (3.3)-(3.6) are sufficient to represent it. However, we present this as a specific case since its address remapping corresponds to a stride permutation on the address bits:

$$f_\pi(Z_{2^n,2^m}) \to B = \mathrm{L}_{n,2} \otimes \mathrm{I}_m, \quad c = \mu_{nm} \tag{3.8}$$

Conditional permutations. Some applications require different permutations on separate regions of their dataset. Also, for some cases, data reorganization is applied only on a portion of the dataset keeping the rest unchanged. Direct sum operator ($\oplus$) is useful to express these cases. Direct sum operator calls for a conditional, address dependent remapping function:

$$f_\pi(P_k \oplus Q_l) = \begin{cases} f_\pi(P_k) \to B = B_P, \quad c = c_P \quad \text{for } 0 \le i < k \\ f_\pi(Q_l) \to B = B_Q, \quad c = c_Q \quad \text{for } k \le i < k+l \end{cases} \tag{3.9}$$

Here both of the B matrices, B_P and B_Q, are padded with the identity matrix (I) towards the most significant bit to match the height of x (and y), i.e. address bit-width. Similarly, the c vectors, c_P and c_Q, are padded with the zero vector (μ) again towards the most significant bit to match the address bit-width.

We also define a reverse direct sum operator, $\bar{\oplus}$ as follows:

$$P \bar{\oplus} Q = \begin{bmatrix} & P \\ Q & \end{bmatrix} \quad \text{and} \quad \underbrace{P \bar{\oplus} P \bar{\oplus} \ldots \bar{\oplus} P}_{n} = \mathrm{J}_n \otimes P$$

The index transformation for the reverse direct sum operator is given as:

$$f_\pi(P_k \bar{\oplus} Q_l) = \begin{cases} f_\pi(Q_l) \to B = B_Q, \quad c = c_Q \quad \text{for } 0 \le i < l \\ f_\pi(P_k) \to B = B_P, \quad c = c_P \quad \text{for } l \le i < k+l \end{cases} \tag{3.10}$$

Here, similar to the direct sum, both of the B matrices, B_P and B_Q, are padded with the identity

matrix (I) towards the most significant bit to match the height of x (and y), i.e. address bit-width. However, the c vectors, c_P and c_Q, are padded again towards the most significant bit but with the one vector (ν) in this case.

Example. Now let's take a simple example data layout transform $(L_{4,2} \otimes I_2) \oplus (I_2 \otimes J_4)$ on 16 element dataset. The dataflow representation is given in Figure 3.2 (note that the dataflow representation in Figure 3.2 is broken into two parts for demonstration purposes).

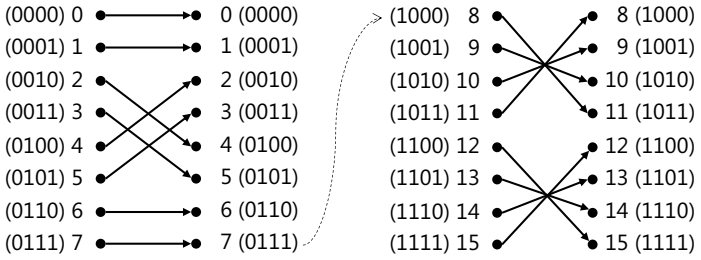


Figure 3.2: Dataflow representation of $(L_{4,2} \otimes I_2) \oplus (I_2 \otimes J_4)$.

By using the fundamental properties of the f_π we derive the bit matrix for address remapping as follows:

$$f_\pi((L_{4,2} \otimes I_2) \oplus (I_2 \otimes J_4)) \rightarrow \begin{cases} B = I_1 \oplus C_{2,1} \oplus I_1, \quad c = [\mu_1 | \mu_3], & \text{for } 0 \le i < 8 \\ B = I_1 \oplus I_1 \oplus I_2, \quad c = [\mu_1 | \mu_1 | \nu_1], & \text{for } 8 \le i < 16 \end{cases}$$

Here we padded the B matrices with identity matrix (I_1) to match their width with the height of the address vector. Similarly we padded the c vectors with zero vector (i.e. μ_1) again to match their height with the address vector.

$$f_\pi((L_{4,2} \otimes I_2) \oplus (I_2 \otimes J_4)) \rightarrow \begin{cases} B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad c = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, & \text{for } 0 \le i < 8 \\ B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad c = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, & \text{for } 8 \le i < 16 \end{cases}$$

Now let's test the address remapping via index transformation for the examples of accessing elements at address 3 and address 11. For an access to the address 3 (i.e. $x = 0011$) the index transfor-

mation is given as:

$$y = Bx + c = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = 5$$

Similarly, for the case where there is an access to the address 11 (i.e. $x = 1011$) the index transformation is given as:

$$y = Bx + c = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = 8$$

Hence, based on the index transformations, accesses to the locations 3 and 11 are remapped to 5 and 8 respectively. An analysis of Figure 3.2 demonstrates that these remappings are indeed correct.

Table 3.1: Summary of the address remapping rules.

$$f_\pi(\mathrm{L}_{2^m,2^n}) \to B = \mathrm{C}_{m,n}, \quad c = \mu_m \tag{3.11}$$

$$f_\pi(\mathrm{I}_{2^n}) \to B = \mathrm{I}_n, \quad c = \mu_n \tag{3.12}$$

$$f_\pi(\mathrm{J}_{2^n}) \to B = \mathrm{I}_n, \quad c = \nu_n \tag{3.13}$$

$$f_\pi(Z_{2^n,2^m}) \to B = \mathrm{L}_{n,2} \otimes \mathrm{I}_m, \quad c = \mu_{nm} \tag{3.14}$$

$$f_\pi(P \cdot Q) \to B = B_P \cdot B_Q, \quad c = c_P + c_Q \tag{3.15}$$

$$f_\pi(P \otimes Q) \to B = B_P \oplus B_Q, \quad c = \begin{bmatrix} c_P \\ c_Q \end{bmatrix} \tag{3.16}$$

$$f_\pi(P_k \oplus Q_l) \to \begin{cases} f_\pi(P_k) & \text{for } 0 \le i < k \\ f_\pi(Q_l) & \text{for } k \le i < k + l \end{cases} \tag{3.17}$$

$$f_\pi(P_k \bar{\oplus} Q_l) \to \begin{cases} f_\pi(Q_l) & \text{for } 0 \le i < l \\ f_\pi(P_k) & \text{for } l \le i < k + l \end{cases} \tag{3.18}$$

3.4 A Formula Rewrite System for Permutations

This section describes the key components of the SPL formula rewrite system for permutations.

3.4.1 Permutation Rewriting

Permutations described as formulas are restructured via formula identities, as defined in Section 3.1.2. Applied formula identities change the dataflow and create a restructured algorithm while keeping the ultimate core computation same. For example, by restructuring a permutation formula, memory access patterns can be optimized for locality/parallelism. The proposed formula rewrite system uses the formula identities as *rewrite rules*.

Table 3.2: Basic formula identities.

$$(AB)^T = B^T A^T \tag{3.19}$$
$$(A \otimes B)^T = A^T \otimes B^T \tag{3.20}$$
$$\mathrm{I}_{mn} = \mathrm{I}_m \otimes \mathrm{I}_n \tag{3.21}$$
$$A_n \otimes B_m = (A_n \otimes \mathrm{I}_m)(\mathrm{I}_n \otimes B_m) \tag{3.22}$$
$$A \otimes (BC) = (A \otimes B)(A \otimes C) \tag{3.23}$$
$$A_n \otimes B_m = \mathrm{L}^{mn}_{n}(B_m \otimes A_n)\mathrm{L}^{mn}_{m} \tag{3.24}$$
$$(\mathrm{L}^{mn}_{m})^{-1} = \mathrm{L}^{mn}_{n} \tag{3.25}$$

Table 3.3: Key permutation rewrite rules.

$$\mathrm{L}^{nm}_{n} = (\mathrm{L}^{nm/k}_{n} \otimes \mathrm{I}_k)(\mathrm{I}_{nm/k^2} \otimes \mathrm{L}^{k^2}_{k})(\mathrm{I}_{m/k} \otimes \mathrm{L}^{n}_{n/k} \otimes \mathrm{I}_k) \tag{3.26}$$
$$\mathrm{L}^{nm}_{m} = (\mathrm{I}_{m/k} \otimes \mathrm{L}^{n}_{k} \otimes \mathrm{I}_k)(\mathrm{I}_{nm/k^2} \otimes \mathrm{L}^{k^2}_{k})(\mathrm{L}^{nm/k}_{m/k} \otimes \mathrm{I}_k) \tag{3.27}$$
$$\mathrm{L}^{nk}_{n} = (\mathrm{I}_{n/k} \otimes \mathrm{L}^{k^2}_{k})(\mathrm{L}^{n}_{n/k} \otimes \mathrm{I}_k) \tag{3.28}$$
$$\mathrm{L}^{nk}_{k} = (\mathrm{L}^{n}_{k} \otimes \mathrm{I}_k)(\mathrm{I}_{n/k} \otimes \mathrm{L}^{k^2}_{k}) \tag{3.29}$$
$$\mathrm{L}^{kmn}_{km} = (\mathrm{I}_k \otimes \mathrm{L}^{mn}_{m})(\mathrm{L}^{kn}_{k} \otimes \mathrm{I}_m) \tag{3.30}$$
$$\mathrm{L}^{kmn}_{n} = (\mathrm{L}^{kn}_{n} \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathrm{L}^{mn}_{n}) \tag{3.31}$$

Set of the basic formula identities, or rewrite rules, are given in Table 3.2. In addition to the basic rewrite rules, Table 3.3 provides the key formula identities that restructure the permutations. Hence, given a permutation expressed as an SPL formula, the rewrite system uses the set of rules given in Table 3.2 and Table 3.3 to rewrite the permutation and produce restructured algorithms. Moreover,

it uses the set of rules given in Table 3.1 to derive the corresponding address remapping for the permutation.

3.4.2 Labelled Formula

The rewriting system can be used to restructure the permutations expressed in SPL for variety of goals in different contexts. After the rewrite operation, special labels are attached to formula constructs which represent the implied functionality in the implementation. In this thesis we will use the following labels: $\overrightarrow{(.)}$, $\overleftarrow{(.)}$, $\tilde{\otimes}$, $\underline{(.)}$ and $\overline{(.)}$.

The label $\overrightarrow{(.)}$ represents a permutation corresponding to data layout and $\overleftarrow{(.)}$ represents the permutation corresponding to the address mapping. These constructs are explicitly labelled for the cases where the data layout is changed transparently as a part of an optimized algorithm (e.g. block layout FFTs [21]).

Furthermore, $\tilde{\otimes}$ corresponds to iteration operator. It generally appears in the form of $\mathrm{I}_\ell \tilde{\otimes} \underline{(.)}$, where the right hand side operand is applied ℓ times. Here, $\underline{(.)}$ represents a kernel that will be performed locally, i.e. local permutation.

Finally, $\overline{(.)}$ represents a permutation that is performed through memory accesses. These constructs generally appear as the leftmost and rightmost operands, and determine the memory access patterns.

Summary. Permutations can represent memory access optimizations through transforming data layout, access pattern or address mapping. The formal framework demonstrates set of formula identities to manipulate the permutations expressed as formulas. These formula identities are used as rules in a rewrite system. This enables restructuring permutations with various goals including optimizing access patterns, transforming data layout or changing address mapping. As we will see, the formal approach helps to automate this process through integration into an existing rewrite system Spiral. Next, we will demonstrate the rewrite system in action through an example case

study.

3.4.3 Case Study

In this case study, we demonstrate permutation rewriting examples for a data layout transform and memory access pattern optimization. We assume a given application with fixed problematic memory access pattern on a given memory system. We assume that the memory system includes a custom hardware substrate that can perform high throughput local permutations. The goal is to demonstrate that without changing the application, just by reorganizing the data in memory using the custom hardware substrate, the performance and the energy efficiency of the application can be improved. We will also show that the additional data reorganization operation itself can be performed very efficiently by optimizing its memory access patterns.

Memory system. We assume a 3D-stacked DRAM system with the following configuration:

- 4 layers, 8 vaults, 256 TSV/vault, 1 KB DRAM pages, 8 SerDes links.
- Internal stream bandwidth: 516.9 GB/s @ 22.8 Watts (5.51 pj/bit)
- External stream bandwidth: 320.0 GB/s @ 30.7 Watts (12.0 pj/bit)

We assume that the host processor is connected to this memory using external SerDes based links which provide 320 GB/s of maximum bandwidth. We also assume that a specialized hardware accelerator for data reorganization is integrated in the memory which can perform local permutations of up to 1 million elements where each element is a byte. This unit can sustain the maximum internal bandwidth of 516.9 GB/s.

Application. We are given an application with fixed memory access pattern, say P. Let's also assume that $P = \mathrm{L}_{32M,4k}$ (where M=million, k=thousand elements). When we simulate the memory performance of P, we get:

$$P \rightarrow 27.2 \text{ GB/s @ } 3.98 \text{ Watts (18.33 pj/bit) (external)}$$

We observe that the application reaches very low bandwidth utilization and energy efficiency compared to the streaming case. This is mainly due to inefficient access patterns that do not utilize the locality and parallelism of the memory system. Since, the access patterns are assumed to be fixed, we will perform a data reorganization such that the given access patterns are a good match for the reorganized data layout.

After reorganization. Let's assume it is determined that R will be the reorganization permutation where $R = \mathrm{L}_{32M,8k}$. R will reorganize the data layout of the application such that, after the reorganization, the resulting access pattern will correspond to $P' = P \times R$ where $P \times R = \mathrm{I}_{32M}$. And the memory performance after the reorganization will be:

$$P' \rightarrow 320.0 \text{ GB/s @ } 30.73 \text{ Watts } (12.01 \text{ pj/bit}) \text{ (external)}$$

With the reorganized data layout, the application can reach the bandwidth and energy efficiency of the streaming case.

Reorganization operation. The data reorganization is performed separately to optimize the data layout. The memory performance of the reorganization, $R = \mathrm{L}_{32M,8k}$, is simulated as:

$$R \rightarrow 28.3 \text{ GB/s @ } 2.67 \text{ Watts } (11.83 \text{ pj/bits}) \text{ (internal)}$$

We observe that the reorganization operation itself has a very poor bandwidth and energy efficiency when implemented by definition. Although reorganized data layout provides significant improvements, overhead of the data reorganization overshadows the improvements in this form.

Optimized data reorganization. Instead of directly implementing, we restructure R by using the rewrite rule 3.26 (see Table 3.3) such that

$$R = \mathrm{L}_{32M,8k} = \overline{(\mathrm{L}_{32k,8k} \otimes \mathrm{I}_{1k})}(\mathrm{I}_{32} \tilde{\otimes} \underline{\mathrm{L}_{1M,1k}})\overline{(\mathrm{I}_4 \otimes \mathrm{L}_8^{8k} \otimes \mathrm{I}_{1k})}$$

In this restructured form, $\underline{(.)}$ corresponds to local permutation that needs to be performed in the specialized hardware accelerator for data reorganization in the logic layer. We observe that the hardware accelerator can support the required size of the local permutation $L_{1M,1k}$ (i.e. 1 million elements). Furthermore, $\overline{(.)}$ corresponds to memory access permutations that are performed by transferring data from/to DRAM layers. In this restructured form, memory access permutations reach the maximum internal streaming bandwidth utilization:

$$\overline{(L_{32k,8k} \otimes I_{1k})} \rightarrow 516.0 \text{ GB/s @ } 22.7 \text{ Watts } (5.51 \text{ pj/bit}) \text{ (internal)}$$

$$\overline{(I_4 \otimes L_{8k,8} \otimes I_{1k})} \rightarrow 516.0 \text{ GB/s @ } 22.7 \text{ Watts } (5.51 \text{ pj/bit}) \text{ (internal)}$$

Hence in this optimized form, memory access permutations can be performed at the maximum internal bandwidth rate where the accesses are generated by the hardware accelerator in the logic layer. We also assumed that the hardware accelerator can sustain the streaming data rate at the maximum internal streaming bandwidth while performing the local permutation. Therefore, the data reorganization can be performed very efficiently, i.e. at the throughput that matches the maximum internal streaming bandwidth, minimizing the high overhead of the original version.

Summary. We demonstrated that given an application with fixed inefficient memory access patterns on a given memory system, data layout reorganization in the memory can improve the overall performance and the energy efficiency. After the reorganization, previously inefficient memory access patterns are transformed into efficient accesses which yield high bandwidth and energy efficiency. Furthermore, the optimized reorganization operation can be performed efficiently which amortizes the overhead.

3.5 Practical Implications

This section discusses the important features, interpretations and the practical implications of the developed mathematical framework.

Unified Framework. The developed mathematical framework serves as a unified language to express algorithms, capture the target architecture machine model, and represent optimizations through formula rewrite rules. In the data reorganization context, it allows expressing address mapping, data layout, memory access pattern, control flow as well as memory system parameters and behavior (e.g. bank/rank/vault/layer parallelism, row-buffer locality, etc.). Given these representations and constraints, it serves as an optimization substrate through formula rewriting.

Efficient Address Remapping. The properties of the f_π function given in Table 3.1 demonstrate a structured way of deriving the index transformation, or address remapping, for a given permutation. The index transformation captures the address remapping information for the entire dataset in a closed form expression (i.e. $y = Bx + c$). This allows calculating the remapped addresses on the fly, instead of keeping a large lookup table or updating the corresponding page table entries via OS calls. Furthermore, the B matrix is a permutation matrix and the c vector is a binary vector. Hence, given an input address x, B shuffles the bits and c inverts them if necessary to produce y. Bit shuffle and inversion can be implemented in hardware at a very low cost.

Inverse Problem. There exists an inverse f_π function that takes the index transform and derives the corresponding data reorganization as a permutation. Hence, to achieve a particular address remapping, it can derive the corresponding data permutation which can be optimized and performed efficiently.

Efficient Memory Access. Set of rewrite rules given in Table 3.2 and Table 3.3 demonstrate a structured way of restructuring memory access patterns of the permutations. Permutations can be restructured by targeting various goals including memory parallelism and locality, which provides an effective means of optimizing the access patterns for bandwidth utilization, latency and energy efficiency.

Generalization. Permutations are basic building blocks of the data reorganization operations. Developed framework demonstrates a generalized way of handling the permutations. Furthermore, it includes set of rules which allows a systematic way of combining various building blocks for generalization. Although the developed techniques are limited to permutations only, this thesis demonstrates that they can cover a wide range of reorganization operations.

Automation. The mathematical formalism enables using the SPL domain specific language to express the data reorganizations. Furthermore, it enables inclusion of the developed rewrite rules into the existing rewriting system of Spiral [98]. This provides an automated approach to rewrite, optimize and compile SPL expressions, which will be discussed next.

3.6 Spiral Based Toolchain

Spiral automatically generates high-performance hardware/software implementations for linear transforms. The generated implementations are automatically tuned to a given target platform through formula rewriting and auto-tuning. Target platforms are abstracted via so called *paradigms*. These paradigms include multi-core parallelism [54], SIMD vectorization [79], streaming hardware generation [83], etc.

In this thesis, however, we view Spiral mainly as a SPL compiler and formula rewrite system. We integrate the developed mathematical framework into the rewrite system of the Spiral and extend it with a customized backend. Our main goals include: (i) optimize (rewrite) the reorganization operations to exploit memory parallelism and locality and (ii) derive the address remapping for a given reorganization.

A high level overview of the toolchain is given in Figure 3.3. Here, block ① parses the high level function calls to the native format (SPL). Given the native representation of the permutations in SPL, block ② derives the address remapping function (i.e. B and c). Finally, block ③ rewrites the permutations to optimize for memory locality and parallelism. The optimized final form of the

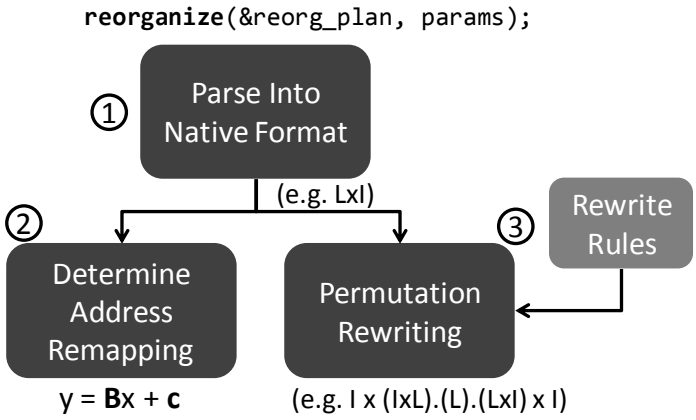


Figure 3.3: Overview the mathematical framework toolchain.

permutation specifies the local permutation, memory access permutations as well as the control flow through labelled formula constructs as discussed previously.

Later in the thesis we will include another goal for utilizing the Spiral based toolchain; determine the accelerator configurations. Key parameters from the labelled constructs such as, local permutation, memory access permutation and index transformation, are packed into configuration words and offloaded to program the HAMLeT unit in memory.

Chapter 4

HAMLeT Architecture for Data Reorganization

The previous chapter presented a mathematical foundation to represent and optimize the memory system performance through transforming data layout, memory access pattern and address mapping. Performing these optimizations dynamically on conventional processing platforms are both difficult and costly. This chapter presents an architectural substrate to perform these optimizations, and their side effects, very efficiently. In particular this chapter presents the HAMLeT (Hardware Accelerated Memory Layout Transform) architecture for highly-concurrent, energy-efficient and low-overhead data reorganization performed in memory. HAMLeT architecture is targeted for integration into a 3D-stacked DRAM, in the logic layer directly interfaced to the local controllers. The architecture design is driven by the fundamental requirements and the implications from the mathematical framework as well as the existing infrastructure, operation and organization of the 3D-stacked DRAM. A portion of the proposed architecture HAMLeT is presented in accompanying publications [22, 23].

The chapter starts with the microarchitectural design of HAMLeT focusing on fundamental components. First, Section 4.1 presents the data reorganization unit (DRU) and discusses the key com-

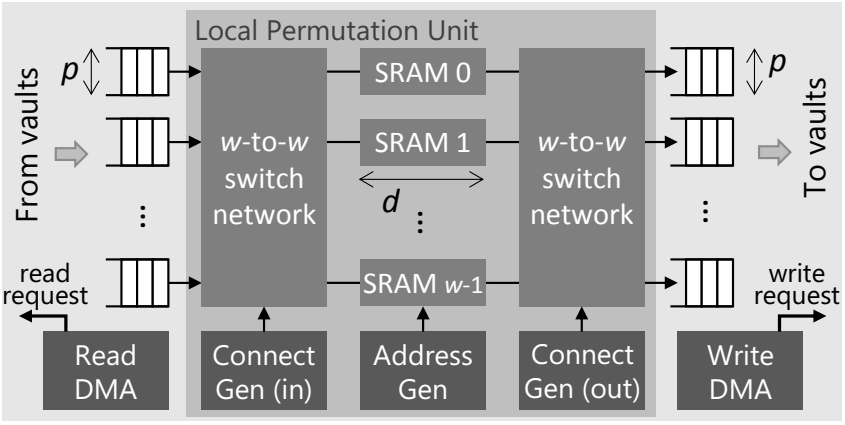


Figure 4.1: Data reorganization unit (DRU).

ponents including DMA units and reconfigurable permutation memory. Then Section 4.2 presents the address remapping unit (ARU) that handles general address remapping which is the essential requirement for completely hardware based data reorganization. Finally, Section 4.3 demonstrates how to leverage the formal framework developed in Chapter 3 as a toolchain for programming the HAMLeT.

4.1 Data Reorganization Unit

Data reorganization unit (DRU) is the main component of the HAMLeT architecture. Briefly, DRU streams data from DRAM, applies a local permutation and streams data back to DRAM in chunks. This operation is repeated to reorganize big datasets. A high level overview of the DRU is shown in Figure 4.1. As discussed in Chapter 3, the data reorganization is captured by two types of permutations, (i) local permutations and (ii) memory access permutations. The local permutation is performed by the reconfigurable permutation memory (RPM) highlighted in Figure 4.1. Whereas, memory access permutations are performed by streaming data to/from DRAM with the help of specialized read/write DMA units. DMA units and RPM control units (i.e. connection and address generators) are programmed to perform various permutations.

4.1.1 Reconfigurable Permutation Memory

The main goal of the DRU is to permute the streaming data at the throughput that matches the maximum internal bandwidth of the 3D-stacked DRAM. Permuting streaming data that arrives in multiple elements per cycle is a non trivial task. The reconfigurable permutation memory (RPM) in DRU mainly adopts the solution from [99] and extends it support runtime reconfigurability, as we will see later.

Exploiting both parallelism and locality within the memory requires permutations both in time and space. The DRU locally buffers and permutes data in chunks. It features w parallel SRAM banks and $w \times w$ switch networks where w is the number of vaults (see highlighted parameters in Figure 4.1). It can stream w words of p bits every cycle in parallel where p is the data width of the TSV bus. Independent SRAM banks per vault utilize the inter-vault parallelism. It also exploits the intra-vault parallelism via pipelined scheduling of the commands to the layers in the vault. In addition to the parallel data transfer, it also exploits the DRAM row buffer locality. Each SRAM bank can buffer multiple DRAM rows. In a reorganization operation, elements within a DRAM row can be scattered to multiple rows, worst case being full scattering. Assuming that the DRAM row buffer holds k elements, each SRAM is sized to hold k^2 elements. This allows transferring k consecutive elements in both read and write permutations exploiting the row buffer locality. We also employ double buffering to maximize the throughput. Hence the SRAM buffer size is given as $d = k^2$ where the total storage is $2wd$.

Spiral Permutation Memory. Spiral supports the automatic hardware generation for the stride permutation family [99] as well as arbitrary permutations [82]. Specifically, [99] supports any local permutation whose index transformation is also a permutation on the bit representation. However, number of stages in the switch network, connection between switches, and SRAM address generation control are configured and optimized specifically for the given permutation at the hardware compilation. Hence, when a permutation memory is generated, it is configured for the target permutation which is limited in supporting different permutations at runtime.

Assume that, we want to perform a local permutation P, where B_P is the bit representation (i.e. index transformation). The overall operation, $y = B_P x$, can be performed in two stages, the write-stage $z = Mx$ and the read-stage $y = N^{-1}z$, where

$$B_P = N^{-1}M.$$

Here, M determines how the streaming data is stored into the SRAMs (the write-stage), and N^{-1} determines how it is read out of the SRAMs into the resulting data stream. Puschel et. al. demonstrated that M and N matrices can be further decomposed [99]. Decomposition for the write stage is shown below.

$$\begin{pmatrix} z_t \\ z_b \end{pmatrix} = \begin{bmatrix} M_4 & M_3 \\ M_2 & M_1 \end{bmatrix} \begin{pmatrix} x_t \\ x_b \end{pmatrix}$$

Here, x_t and x_b correspond to the *stage number* and *where in stage* respectively. In other words, combined $\{x_t, x_b\}$ represents the unique number of the particular element in the permutation order. Similarly, z_t and z_b represent the address in RAM and the corresponding RAM number respectively. Thus, the addresses that each element is written/read (i.e. z_t) and routing of each element to/from RAM banks (i.e. z_b) can be calculated as follows:

$$z_t = M_4 x_t + M_3 x_b \tag{4.1}$$

$$z_b = M_2 x_t + M_1 x_b \tag{4.2}$$

Analogous to the write stage, read stage (i.e. $y = N^{-1}z$) is expanded as well (refer to [99] for details). Spiral framework further processes and decomposes these sub-matrices to derive the connectivity and cost metrics. Ultimately, it derives an optimal (or partially optimal) configuration for the permutation memory and generates its RTL hardware implementation. Hence the hardware implementation is specifically generated for the given permutation and it cannot perform any other permutation.

Runtime Reconfiguration. Our goal is to provide a substrate that can be reconfigured at runtime

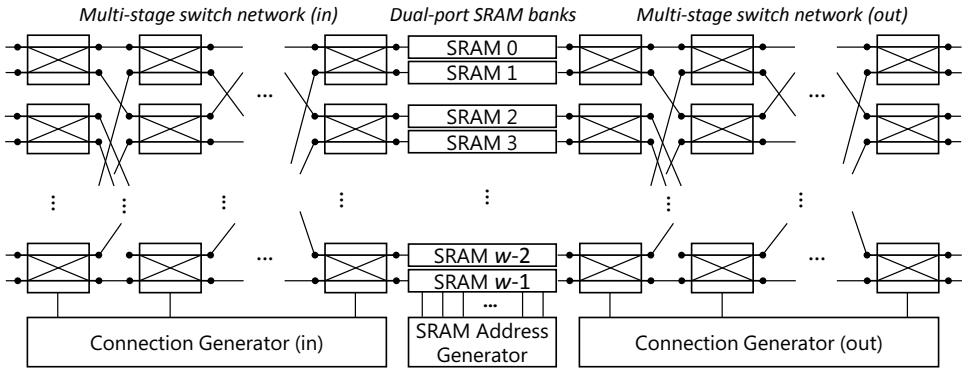
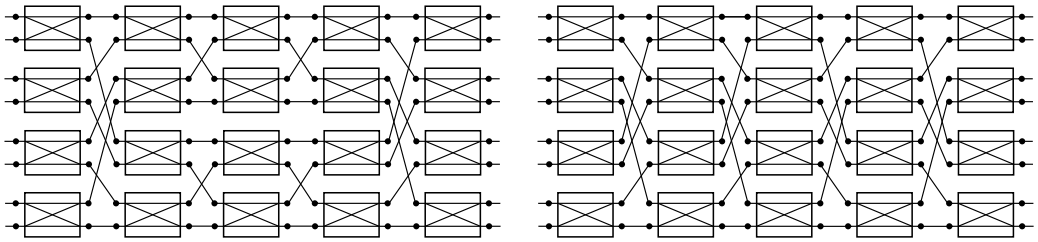


Figure 4.2: Overview of the reconfigurable permutation memory.

to perform any permutation, only constrained by the maximum supported size and streaming width. To this end, we extend Spiral permutation memory via (i) rearrangeably non-blocking Clos network implemented as a multistage network, and (ii) programmable switch connection and address generator. High-level overview of the reconfigurable permutation memory (RPM) is given in Figure 4.2. Moreover, by utilizing the formal framework, we automate the derivation of the RPM configuration parameters. Given a local permutation, Spiral based toolchain derives the configuration for the RPM control units (i.e. connection and address generators). These configurations are offloaded to program the RPM.

First, the reconfigurable permutation memory employs (i) rearrangeably non-blocking Clos networks as the input and output switch networks. As discussed previously, these switch networks implement the routing between input/output ports and SRAM banks which require an all-to-all communication to support any permutation (where the index transform is a bit permutation). For a rearrangeably non-blocking Clos network there exists a switch configuration such that any unused input can be routed to any unused output [31], but some of the existing routings may have to be rearranged. In our case, however, this is not a critical problem since the switch configuration is changed every cycle to meet the streaming data throughput.

In our solution the rearrangeably non-blocking Clos network is implemented as a multi-stage network. Two examples for 8x8 rearrangeably non-blocking multi-stage switch networks are shown in Figure 4.3. Both Benes and omega-flip (i.e. omega and omega-inverse combined) networks provide



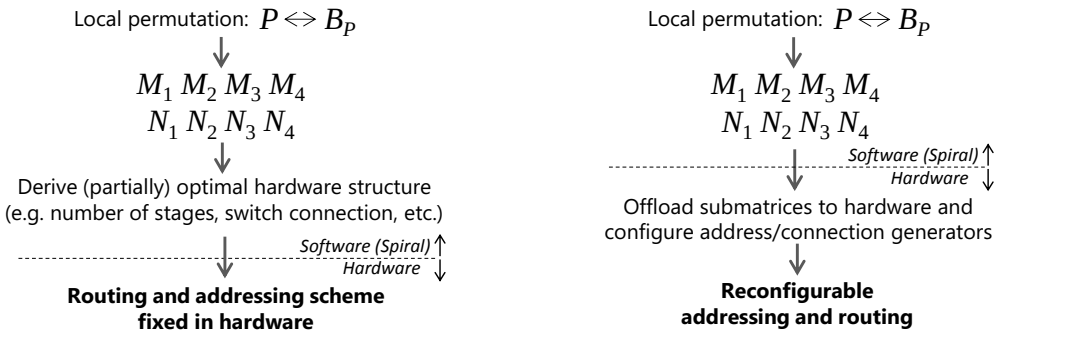
(a) 8x8 Benes network (b) 8x8 Omega & inverse-omega network

Figure 4.3: Two examples for 8x8 rearrangeably non-blocking multi-stage switch networks.

all-to-all connection at a low hardware cost [31]. Such networks have $2\log_2(N)-1$ stages where each stage contains N/2 switches, thus total $N\log_2(N)-N/2$ switches. Asymptotic switch complexity is $O(N\log(N))$. The examples given in Figure 4.3 have $2\log_2(8)-1=5$ stages where each stage has $8/2=4$ switches, adding up to total $5\times 4=20$ switches.

Main disadvantage of using a multi-stage network is the latency. However, in the streaming permutation problem, throughput is the main concern not the latency. Benes (or equivalently omega-flip) network provides the throughput to permute the streaming data every cycle where the switch stages are pipelined if necessary for the large streaming widths. A crossbar switch, on the other hand, can provide strict-sense non-blocking interconnection such that any unused input can connect to any unused output without any rearrangement on the existing connections. However, as the number of input/output nodes increase, the hardware complexity increases dramatically. Asymptotic switch cost is given as $O(N^2)$ for the crossbar switch. Assuming that the switch network is integrated in a 3D-stacked DRAM system with several vaults (e.g. HMC 2.0 has 32 vaults), a crossbar switch becomes too costly.

Secondly, the permutation memory datapath is controlled by the (ii) connection and SRAM address generator units as shown in Figure 4.2. These units generate the required switch connection configuration and SRAM addresses every cycle that will route the data elements to and from the SRAM banks. As discussed previously, given a target local permutation, Spiral infrastructure decomposes and processes the permutation matrix to derive an optimal (or partially optimal) structure (i.e. number of network stages, switch connection, address generation scheme, etc.) for the permutation



(a) Current flow in Spiral [99].

(b) Proposed flow for reconfigurable permutation memory.

Figure 4.4: Permutation memory compilation flow.

memory. Then it generates an optimized but fixed hardware implementation. In our framework, we utilize Spiral frontend only to derive the sub-matrices M_1, M_2, M_3, M_4, N_1, N_2, N_3, N_4 from M and N matrices where $B_P = N^{-1}M$ and B_P is the bit representation of the given local permutation P. Then, as opposed to further decomposition for the optimal hardware structure compilation, the sub-matrices are packed and directly offloaded as-is to the RPM to configure the address/connection generators. These control units can be reconfigured to perform different permutations. Current permutation memory synthesis flow and the proposed flow are demonstrated in Figure 4.4(a) and Figure 4.4(b) respectively.

SRAM addressing and switch connection configuration is calculated using the submatrices (N_i and M_i) where equations 4.1 and 4.2 demonstrate the case for the read network. Mathematically, these calculations require two matrix-vector multiplications and one matrix addition. These operations can be achieved very efficiently considering that the matrices are actually bit matrices. The matrix-vector multiplications (i.e. $N_i x$ and $M_i x$) correspond to a bit permutation of the input vector. This is achieved by a single bit crossbar switch where the submatrices (N_i and M_i) determine the crossbar configuration (i.e. set of connections) to permute the input vector ($\{x_t, x_b\}$). Then, an XOR stage handles the addition of the permuted bit vectors. When these structures are put together, they construct the address/connection generators as shown in Figure 4.5. Here, the input vectors x_t and x_b when combined (i.e. $\{x_t, x_b\}$) represent the number of the particular element in the permutation order, which are tracked by specific hardware counters. As the elements progress through the RPM,

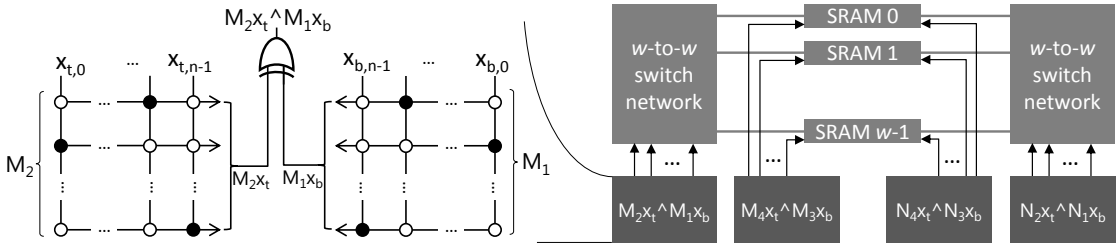


Figure 4.5: Detailed view of the SRAM address and connection generator units.

counters are incremented and new routing/addressing is calculated for each element.

4.1.2 DMA Units

DMA units are responsible for streaming the input data from DRAM to the local permutation unit and the locally permuted data back to DRAM (see Figure 4.1). They ensure continuous flow of data through DRU. Besides interfacing and communication to the DRAM controllers, DMA units perform the memory access permutations.

DRU features two distinct DMA units for read and write. RPM continuously streams data using double buffering. Hence the read and write DMA units must stream the input/output data continuously. Outputs from the formal framework contain separate memory access permutations for reading and writing. These permutations are mapped on the corresponding DMA unit.

DMA units generate the memory addresses based on the targeted memory access permutations. Assuming that it performs a memory access permutation P, the sequence of the addresses are determined by the index transformation of the permutation P. As shown in Figure 4.6, DMA unit features counters coupled with bit shuffle units (BSU) which implement the index transformation. Assuming the index transformation is governed by $y = Bx + c$, B matrix and c vector directly configure the BSUs in the DMA units.

A DMA unit contains multiple counters and BSUs to generate multiple addresses every cycle. Number of counter-BSU couples are determined based on the maximum number of requests DRAM system can service. For a 3D-stacked DRAM, the number of parallel requests serviced is determined by

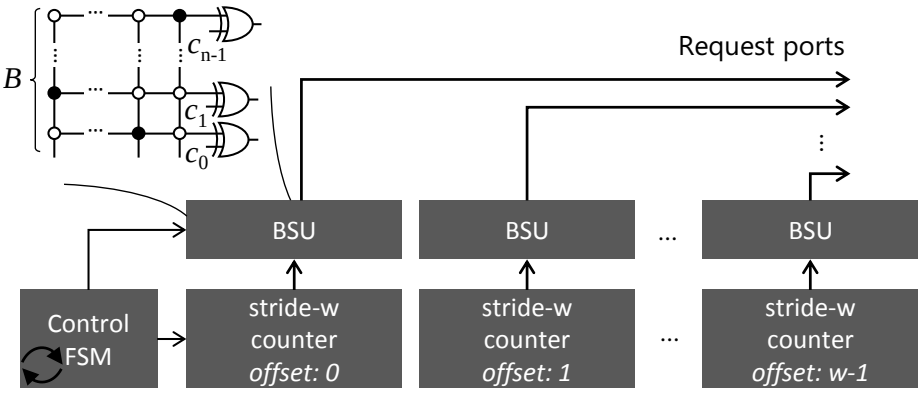


Figure 4.6: DRAM address generator in a w-wide DMA unit.

the concurrency of the vaults/layers in the stack. Multiple requests are generated by stride counters with sequential offsets as shown in Figure 4.6.

Eliminating redundant data movement. Proposed DMA unit supports transferring data for any permutation specified by its bit representation. Following the direct implication from the permutation description, permuting a dataset requires reading and writing all of its elements. However, eventually some of the elements are written into their original locations after the reorganization. For example, in a matrix transpose operation elements in the diagonal do not change. Though the elements that stay at their original locations is a tiny fraction of the large matrix for the transposition, they can constitute a large fraction for some data reorganizations. To understand the details of this let's focus on a data reorganization where its corresponding bit permutation is given in Figure 4.7.

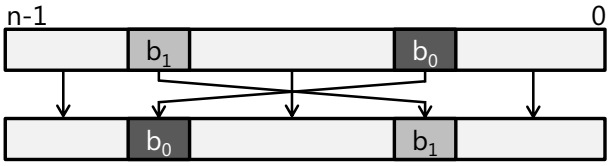


Figure 4.7: Bit permutation of an example data reorganization.

Figure 4.7 demonstrates a bit permutation where only a single bit pair is swapped. When the original and reorganized locations are the same, then the corresponding elements stay at their original locations. Table 4.1 demonstrates the special cases for the bit permutation given in Figure 4.7 and

Table 4.1: Original and remapped locations for the bit permutation given in Figure 4.7.

Original			Remapped			Same?
b_1	b_0	rest	b_1	b_0	rest	
0	0	x	0	0	x	yes
0	1	x	1	0	x	no
1	0	x	0	1	x	no
1	1	x	1	1	x	yes

analyzes them if the original and remapped locations are the same. We observe that half of the dataset is remapped into their original locations. Furthermore, this can be generalized to data reorganizations where s bits are swapped in the index transformation. Without providing a formal proof, we state that when s bits are swapped in the index transformation, $(1/2)^s$ of the elements are mapped into their original locations. Following the matrix transposition example, let's assume an $2^n \times 2^n$ element square matrix. Transposition of this matrix is captured by the permutation $L_{2^{2n},2^n}$ whose bit permutation is given by the circular shift $C_{2n,n}$ where n bit pairs are swapped. Hence, $(1/2)^n$ elements are mapped onto their original locations. In fact, the 2^n diagonal elements of the $2^n \times 2^n$ correspond to the $(1/2)^n$ of its total elements.

This allows further optimizing the data reorganizations by eliminating these redundant data movements. Especially when the reorganization is performed in-place, these redundant movements, where the original and remapped locations are the same, can be completely eliminated. On the other hand, for out-of-place reorganizations these redundant movements can bypass the DRU and directly streamed into their remapped locations. DMA units detect if the bit permutation for the data reorganization has a limited number of bit-swaps. In such cases, it eliminates (or optimizes) a considerable portion of the data movements.

4.2 Address Remapping Unit

This section presents the address remapping unit (ARU) to perform the index transformation of the permutation based data reorganizations. Chapter 3 demonstrates that the index transformation

corresponds to the address remapping for a data reorganization. It also presents set of rules to systematically determine the address remapping as a closed form expression. One crucial implication of this property is that it allows calculating the remapped address on-the-fly. This eliminates the need for costly hardware look-up tables or OS page table updates to keep track of the data remappings. Moreover, if implemented in hardware, it allows a software-transparent data reorganization where the address remapping completely handled in hardware. The goal is to present an efficient and scalable hardware substrate that handles the address remapping by implementing the closed form index transformation expressions.

4.2.1 Bit Shuffle Unit

The address remapping operation (i.e. $y = Bx + c$) consists of two main parts. First, the B matrix permutes the input address bits x, then the c vector inverts the permuted bits. To support that functionality we propose the bit shuffle unit (BSU), a single bit crossbar switch connected to an array of XOR gates, as shown in Figure 4.8. In Section 3 we saw that B is a permutation matrix, thus, there is only a single non-zero element per row or column. The location of the non-zero element in each row of the matrix determines the closed switch location in the corresponding row of the crossbar. The crossbar configuration, i.e. the set of closed switch locations, is stored in a configuration register which can be reconfigured to change the bit mapping. After input address x is permuted via the crossbar, XOR array inverts the bits according to the c vector to generate the remapped address y. Hence, when configured according to a particular address remapping (B and c), BSU will forward the input addresses to the remapped locations. Overall, this unit can implement any address remapping where B is a bit permutation and c is a bit vector, i.e. all combinations of the rules (3.11)–(3.16) from Table 3.1.

4.2.2 Supporting Multiple Remappings

BSU can support any address remapping globally over the entire address space. However, there are cases where only a portion of the address space is remapped or different parts of the address space

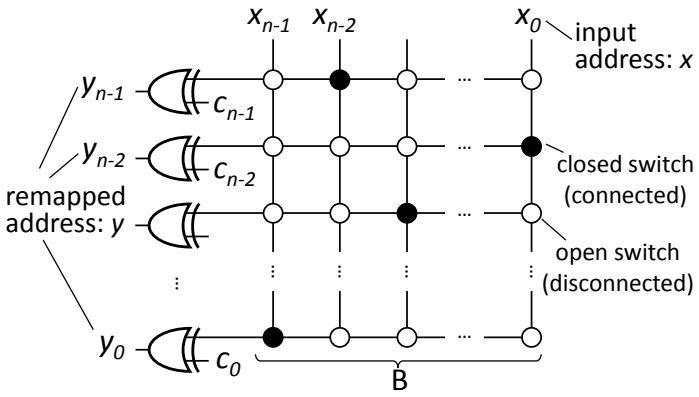


Figure 4.8: Bit shuffle unit (BSU).

have different remappings. Furthermore, as discussed previously there are conditional permutations that change based on the portion of the dataset that the particular element falls into. Our goal is to allow different address mappings, thus different data reorganizations, for different regions. A *region* is a partition in the memory address space. An application can occupy multiple regions if needed. Physically partitioned address space also allows exploiting spatial differences of the memory access patterns.

The address remapping unit (ARU) that extends the BSU as shown in Figure 4.9 supports multiple address remapping schemes simultaneously, including conditional permutations (3.17)–(3.18). The *configuration store* keeps multiple configurations for the BSU. The configuration includes the full state of the BSU (i.e. B and c). When an access is enqueued in the scheduling FIFOs of the memory controller, the *region bits* from the address are used to index the configuration store. Then the corresponding entry is put in the *configuration register* that configures the ARU for that particular access.

The ARU can support all the remapping schemes presented in Section 3.1 via a very simple hardware. The configuration store indexing and the read latency depends on the number of different mappings supported simultaneously and the number of memory regions. As these numbers increase, ARU reconfiguration latency increase as well. The ARU architecture can be restructured to have multiple BSUs as shown in Figure 4.10. This architecture allows actively pre-configured BSUs so that when consecutive accesses are mapped into different regions, remapped addresses can

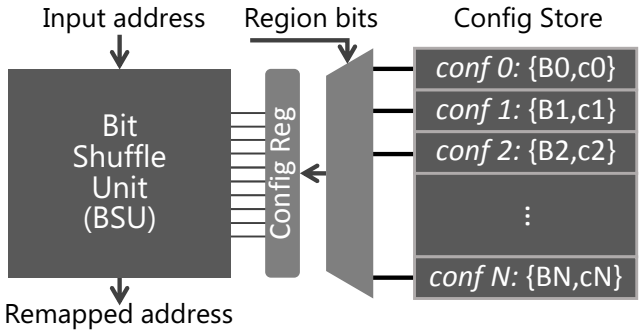


Figure 4.9: Address remapping unit (ARU) supporting multiple address mappings for different memory regions.

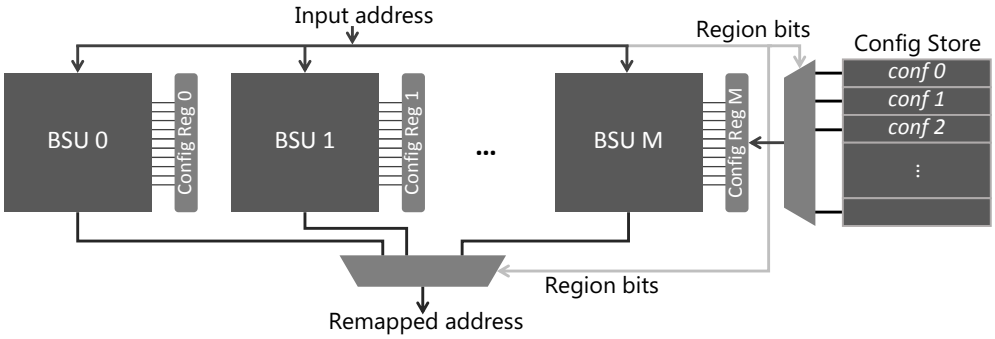


Figure 4.10: ARU with multiple BSUs.

be determined without reconfiguring the ARU. Furthermore, BSU reconfigurations can be scheduled ahead of time and the reconfiguration latency can be overlapped while active BSUs serve other requests. Nevertheless, as we will see the evaluations in 7, for a typical implementation, the overall latency and energy consumption of the combined configuration store indexing, BSU reconfiguration and bit permutation is very low. Furthermore, the timing of the ARU is not on the critical path of the memory access since typically an access spends several clock cycles in the memory controller FIFOs waiting to be scheduled.

4.3 Configuring HAMLeT

The domain-specific architecture of the HAMLeT trades the programmability for energy efficiency. Despite being domain-specific, it serves as a generic data reorganization engine that can be reconfig-

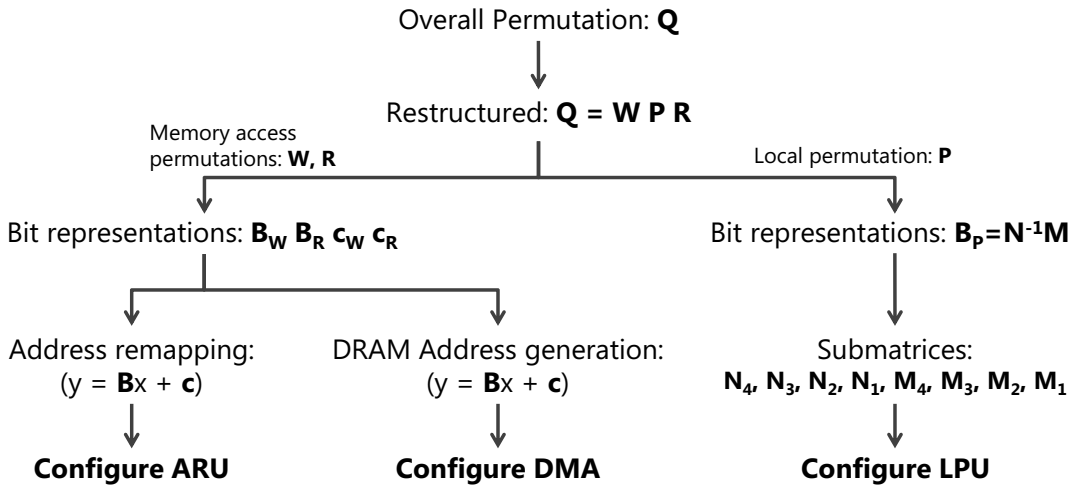


Figure 4.11: From permutation specification to HAMLeT configuration.

ured to perform various combinations of permutations as building blocks. As presented previously, both DRU and ARU units in the HAMLeT can be reconfigured according to the target data reorganization.

Figure 4.11 summarizes the configuration flow starting from a target data reorganization expressed as a permutation to the low-level configuration parameters that will program the main components of the HAMLeT. Synthesis from the data reorganization to the configuration parameters is automated using the Spiral-based toolchain that is introduced in Chapter 3. The frontend is used as described previously to rewrite the permutations and derive index transformation. First, the data reorganization permutation expressed as a formula in SPL and restructured. Then, formula constructs are labelled with the implied functionality. We extend the the toolchain via a customized backend. Memory access permutations are further processed to derive the address remapping and memory address generation parameters. Later these parameters are offloaded to configure the ARU and DMA units. Local permutations, on the other hand, are further decomposed to derive the connection and address generator parameters to program the reconfigurable permutation memory.

Spiral-based toolchain enables automated synthesis of the configuration parameters from the given data reorganization permutation. However, in order to fully integrate the HAMLeT into the existing hardware/software ecosystem, we will further extend the proposed flow and include additional

hardware/software solutions. System-level hardware and software integration concepts will be elaborated later in Chapter 6.

Chapter 5

Fundamental Use Cases

Combination of the formal framework and the HAMLeT architecture presented in previous chapters serve as an infrastructure to perform efficient data reorganization and address remapping. This section introduces two fundamental modes of operation that will use the developed infrastructure as an enabling technology.

First, Section 5.1 presents the *automatic mode* which enables a software-transparent dynamic data layout reorganization. In this hardware based mode, data layout is transformed physically in memory, where the whole operation and its side effects (e.g. address remapping, coherence, access scheduling) are completely handled in hardware.

Then, Section 5.2 introduces the *explicit mode* where the data reorganization is explicitly offloaded from software and accelerated by the HAMLeT in memory. This technique requires a collaboration between the hardware, software and OS which is handled by a custom software stack.

5.1 Automatic Mode

Motivation. DRAM based memory systems are hierarchically divided into fundamental blocks. A conventional DRAM based main memory system features multiple channels, ranks, banks, rows

and columns. This hierarchy is expanded in the modern DRAM systems with bank groups and sub-arrays [26, 72]. Moreover, 3D-stacked DRAM systems orthogonally extend this hierarchy with multiple layers and vaults. Memory controllers implement an address mapping policy to determine the mapping from the physical addresses to the DRAM "coordinates". Most widely used address mapping policies include row and cache line interleaving schemes. Row interleaving scheme interleave consecutive cache lines first into same row from the same bank, then interleave the rows among banks. Whereas, cache line interleaving scheme places consecutive cache lines into different banks/ranks/vaults/layers. There are various hybrid approaches and sophisticated address mapping techniques, such as permutation based interleaving [118], that aim to exploit the memory parallelism and locality better. Moreover, application-specific efficient address mapping schemes can be determined via profiling.

For conventional systems, however, memory controllers implement a global address mapping scheme that is optimized for the common case. Yet, the best performing address mapping scheme depends on a particular application's data layout and memory access pattern. Moreover, even a single application may exhibit spatially and temporally different memory access patterns. First, it can access disjoint parts of its dataset with varying access patterns. Second, the memory access behavior can change over time as the application progresses through different phases. Hence, global and static address mapping can lead to underutilized memory system.

Our goal is to enable a memory system which can employ multiple different address mappings for different regions of the address space. Further, it can change these address mappings on-the-fly, thus, leading to a non-global and dynamic address mapping. Changing the address mapping at the runtime requires the corresponding data to be physically reorganized in the memory to retain the original program semantics.

Approach. We approach this problem via a combined data reorganization and address remapping in hardware utilizing the HAMLeT architecture. As presented in Chapter 4, HAMLeT architecture features the address remapping unit (ARU) and the data reorganization unit (DRU). ARU handles

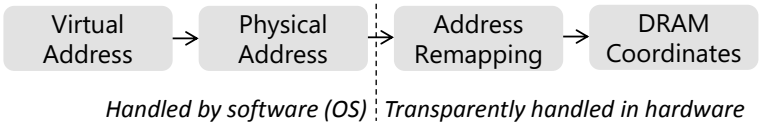


Figure 5.1: Virtual address to DRAM coordinates (i.e. channel, bank, row, etc.) translation stages.

the address remapping for multiple memory regions. Further, DRU is specialized for efficient permutation based data reorganization. In the automatic mode of operation, the HAMLeT architecture is driven by a hardware based memory access monitoring system that profiles the memory access patterns of applications, and configures the ARU/DRU units for address remapping and data reorganization. This enables software transparent dynamic data layout operation, where both the profiling and the reorganization are handled in hardware.

5.1.1 Changing Address Mapping

Changing the address mapping is essentially a bit-permutation on the memory address bits. The bit shuffle unit (BSU) in the ARU can perform all bit permutations. When the address mapping of a memory region changes, ARU stores the corresponding BSU configuration. Given the BSU configuration (i.e. B and c as described in Section 4.2), it can translate the original input addresses to the remapped locations in hardware. This operation is essentially an indirection implemented in hardware. It is completely transparent to the virtual address space—virtual to physical mapping remain the same after the reorganization (see Figure 5.1). Hence, all the entries in the page table, cached translations in the TLB (translation lookaside buffer), and cached copies of the reorganized data remain valid after the reorganization.

5.1.2 Physical Data Reorganization

When the address remapping is a bit permutation, the corresponding data reorganization is also a permutation as shown in Chapter 3. Moreover, it belongs to the class of permutations which are constructed out of stride permutations. DRU executes permutation based data reorganization efficiently.

Hence, in the automatic mode, DRU is configured to perform the data reorganizations.

Data reorganization can be in-place or out-of-place. When the data is transformed in-place, only a constant additional memory, which is smaller than the original dataset, is used. On the other hand, an out-of-place reorganization uses an additional storage that is as large as the original dataset.

In-place permutation is a complex operation where the elements are swapped following the permutation cycles. Naive algorithms either use $O(n^2)$ operations or $O(n)$ extra storage for n element arrays. Furthermore there are algorithms that use $O(n\log(n))$ operations and $O(\log(n))$ extra storage, which require complex index calculations [51, 61]. Whereas, out-of-place permutation requires an additional storage equal to the size of the dataset, i.e. $O(n)$. But it can perform the permutation using $O(n)$ operations.

In the automatic mode, overhead of the data reorganization is a primary concern. Latency and energy overheads need to be minimized to amortize the data reorganization cost. In-place algorithms require additional read-write operations which increase the reorganization latency as well as energy consumption. Furthermore, in-place algorithms require irregular memory accesses to individual elements according to the permutation cycles. This destroys the locality of the memory access patterns which causes low bandwidth and high energy consumption in the memory. Hence, out-of-place operation is a better fit for the automatic mode to minimize the reorganization overhead by eliminating additional memory accesses and by keeping the locality of the memory accesses.

Out-of-place operation permutes the data into a destination memory space which is disjoint from the source space. We call this operation as *move-only*. If the reorganization requires the dataset to stay at its original address span, then the reorganized data is streamed back into the source space. We call this operation as *move-back*. Note that this scheme requires an extra pass over the dataset to stream it back into its original memory space. Finally, the *move-remap* technique provides the best of both worlds. In this scheme, the data is reorganized into the destination memory space. Then instead of moving the reorganized data back into its original place, original memory region and the destination region are remapped such that the pointer to the original memory region now points to the destination region and vice versa. This eliminates the need for extra pass over the dataset while

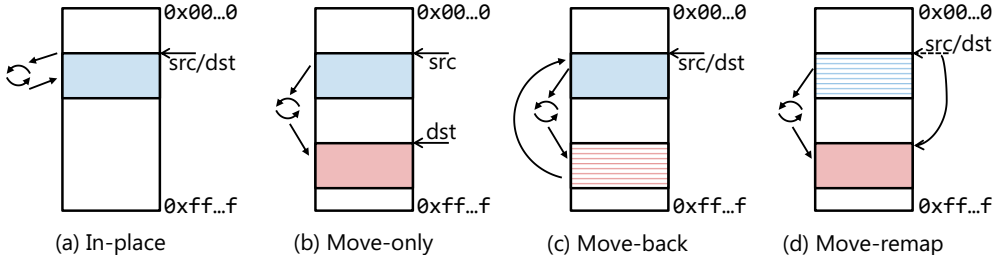


Figure 5.2: In-place and out-of-place data reorganization schemes.

keeping the data at its original memory space. These schemes are summarized in Figure 5.2.

Partitioning the memory address space into smaller regions enables a practical implementation for these techniques. In the automatic mode, permutations across memory regions are not allowed. Data can be permuted within each region and the regions themselves can move and swap with other regions. In this way ARU can easily keep track of the region movements. It would be much more costly to follow freely moving address pointers with no boundaries. The partitioning bits from the DRAM address space is selected smartly to allow the important data permutations across different banks, vaults and layers. Partitioning bits are selected from the mostly stationary row or column bits in the DRAM address. Thus, permutation within a region includes data movements across banks, vaults and layers—only the movements across certain DRAM rows are blocked. As we will see later, automatic reorganization requires permutations that change the address mapping between bank, vault, layer and row bits. Hence, allowing permutations only within regions does not limit the useful data reorganizations.

5.1.3 Memory Access Monitoring

In the automatic mode, memory access stream is monitored to determine a possibly inefficient address mapping where the memory access pattern does not match the underlying data layout, causing underutilized memory parallelism and locality. Memory access monitoring determines the changes in the DRAM address bits. A certain bit in the address is said to be *flipped*, if it is changed compared to its previous value. Monitoring hardware records the bit-flip rates (BFR) of the address bits in the memory access stream. A bit with high BFR implies frequent changes in short time which can be

used to exploit parallelism. Mapping the bits with high BFR onto DRAM address bits that correspond to bank, rank, vault, or layer (which we refer to *parallelism bits*) can improve the parallelism. On the other hand, address bits with low BFR implies stationary behavior. Such bits are better suited to be mapped onto DRAM row bits (i.e. *locality bits*) to minimize row changes, hence row buffer misses, improving locality. Bit flip rate based memory access profiling is a rough approximation for a better memory address mapping to exploit the parallelism and locality. Making robust decisions about the address mapping changes using the BFR metric requires several design decisions which are further elaborated later in Chapter 6. At this stage, the most important aspect of the memory access monitoring is that it detects the underutilized memory system in hardware. It allows issuing an address remapping with data reorganization to improve the memory system performance.

5.1.4 Host Application and Reorganization in Parallel

When the memory access monitoring detects an inefficient address mapping and issues the data reorganization, the entire dataset (or a few memory regions) of the application will be physically reorganized in memory. During the reorganization, DRU utilizes the memory bandwidth efficiently. It achieves close to peak internal bandwidth utilization of the 3D-stacked DRAM, by using its high throughput parallel architecture. However, data reorganization still incurs latency and energy overhead. As we will further analyze in detail, the overhead of reorganization is amortized via the improved memory access performance during the long runtime of the application.

When the reorganization is taking place, the application itself requires accessing the data from the memory. To handle executing parallel host application and reorganization, we employ two main techniques; block on reorganize (BoR) and access on reorganize (AoR).

Briefly, block on reorganize (BoR) locks the memory regions under reorganization such that only DRU requests are serviced. Though, independent processes can still access other memory regions. Main advantage of this method is that it enables a very fast reorganization. DRU streams the data almost at the peak internal bandwidth utilization. Moreover, it eliminates the coherence and scheduling issues since only DRU is allowed to access the data until the reorganization is finished. On the

other hand, the host-side application is halted until the reorganization is finished.

Access on reorganize (AoR) method allows both DRU and the host processor to access the data in memory. In this way, host-side application can make progress while the data is under reorganization. However, this brings up a few side issues such as coherence and memory access scheduling. The original data and the reorganized data have to be kept coherent for correctness. Moreover, accesses from the DRU and from the host processor needs to be interleaved to provide the desired quality of service to the host while minimizing the data reorganization duration. Later in Section 6, we further elaborate on the proposed mechanisms to tackle these issues arise when the host application and the reorganization are performed in parallel.

5.2 Explicit Mode

Motivation. For an application. given an address mapping scheme, data layout in the memory is a fundamental property that determines the memory system performance (bandwidth utilization, energy efficiency, power consumption, etc.). Address mapping schemes aim to improve memory parallelism and locality assuming a generic behavior from the application. However, if the elements that are sequentially needed by an application are placed into different rows of the same DRAM bank, neither address mapping nor access scheduling mechanisms can solve the inevitable row buffer misses. To maximize the memory parallelism and locality, data layout need to be compatible with the application's memory access pattern.

However, several data-intensive applications fail to utilize the available locality and parallelism due to the inefficient memory access patterns and the disorganized data placement in the DRAM. This leads to excessive DRAM row buffer misses and uneven distribution of the requests to the banks, ranks or layers which yield very low bandwidth utilization and incur significant energy overhead. Memory layout transformation via data reorganization in the memory aims the inefficient memory access pattern and the disorganized data placement issues at their origin.

Therefore, for some applications it can be more efficient to explicitly transform the data layout

into an optimized format before the actual computation. Several previous work report improved performance with optimized data layouts [21, 36, 58, 92, 107, 108].

Moreover, several high-performance computing applications such as linear algebra computations, spectral methods, signal processing, and molecular dynamics simulations require data reorganization operations as a critical building block (e.g. matrix transpose, multidimensional dataset rotation, pack/unpack, permute) [12, 30, 55, 58].

Although these operations simply relocate the data in the memory, they are costly on conventional systems mainly due to inefficient access patterns, limited data reuse and roundtrip data traversal throughout the memory hierarchy.

Approach. Near-data processing approach minimizes the data transfer between the DRAM and the processor. We follow a two pronged approach for efficient data reorganization, which combines (i) the DRAM-aware HAMLeT architecture integrated within 3D-stacked DRAM, and (ii) a mathematical framework that is used to represent and optimize the reorganization operations. Explicit mode of operation, as opposed to the automatic mode, exposes the the HAMLeT to the user for explicit acceleration of the data reorganization operations. In this mode, first the data reorganization is expressed in the native language (SPL) of the Spiral based mathematical toolchain. Accompanied by a custom software stack for offloading and memory management, it derives the DRU/ARU configurations and programs the HAMLeT.

5.2.1 Spiral Based Toolchain

Explicit operation starts with the data reorganization expressed in SPL. SPL formula expressions for certain common operations are further wrapped into high-level function interfaces. One can express the data reorganization in the native SPL language or using the high-level wrapped function interfaces which are later translated into SPL. Then, SPL formulas are passed to the proposed Spiral based toolchain. Spiral is used as a compiler for the SPL. It restructures the SPL formulas using the rewrite rules. The optimized final form of the SPL expression specifies the local permutation,

memory access permutations as well as the control flow through labelled formula constructs. Key parameters from these labelled constructs are packed into configuration words and offloaded to program HAMLeT.

5.2.2 Explicit Memory Management

In the automatic mode, HAMLeT reorganizes data in memory and handles all the side effects in hardware. However, offloading the operation from the software requires explicit communication, synchronization and memory management. Firstly, the allocated memory region and the boundaries for the accelerator have to be defined. The memory management software has to make sure that the accelerator only accesses the data within allowed boundaries. A custom memory management software specifically allocates the source and the destination memory spaces for the accelerator. Furthermore, a specific memory space is allocated for synchronization and communication between the host and the accelerator. Host offloads the configuration data including HAMLeT architecture configuration, allocated memory pointers and sizes to this region. HAMLeT and the host processor share the main memory where the actual application data sit, hence data is not offloaded. Implementation of the accelerator memory management software along with the side issues are further elaborated in the next chapter.

Chapter 6

System Architecture and Integration

The previous chapter demonstrated two fundamental processing models that use the developed mathematical framework and the proposed architecture as an enabling technology. However, to achieve a system-level integration there needs to be a few changes at the software and hardware levels as presented in the previous chapter. This chapter presents series of hardware and software mechanisms that assist the HAMLeT architecture to achieve a more flexible system-level integration.

The chapter starts with the design choices regarding the near-data processing architecture in Section 6.1. It discusses accelerator placement in addition to the hardware mechanisms that assist the HAMLeT architecture for system-level integration. Section 6.2 focuses on the issues arise when the host and HAMLeT operate in parallel. Then Section 6.3 discusses the host architecture/software and the near-data processing integration issues. Finally, Section 6.4 puts together all these components and gives an overview of the proposed data reorganization system.

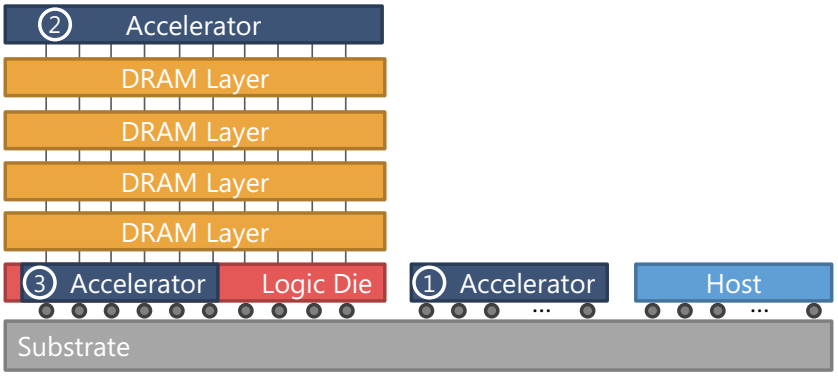


Figure 6.1: Accelerator integration options to a 3D-stacked DRAM based system for near-data processing: (1) Off-stack, (2) stacked as a separate layer, (3) integrated in the logic layer.

6.1 Memory Side Architecture

6.1.1 Interface and Design Space

HAMLeT architecture is designed targeting a near-data processing system where the accelerator is integrated in the logic layer of a 3D-stacked DRAM behind the conventional interface. However, near-data processing can be achieved using several different integration options with different proximity levels between the memory and the accelerator. Taking the processor core and the DRAM modules as different ends of the spectrum, accelerators can be integrated various places in between. These options provide design tradeoffs related with latency, bandwidth, thermal issues, and manufacturing cost. Figure 6.1 demonstrates different integration options for near-data processing accelerators. Note that there are also several places in the processor die itself to integrate the accelerator where they share various levels of cache hierarchy, interconnect, memory controllers, etc. However, this thesis focuses on accelerators integrated near main memory—on-chip accelerators are not included in our taxonomy.

Off-stack. First option is the off-stack integration (① in Figure 6.1). In this scheme, the accelerator is a separate die communicating with the 3D-stacked DRAM via off-chip connections. This minimizes the thermal interaction between the accelerator and the stacked DRAM. Accelerator dissipates the heat almost entirely through its own die surface. Moreover, the 3D-stacked

DRAM design is kept unmodified. Eliminating customized hardware modification minimizes the overall manufacturing cost. However, the accelerator uses the off-chip connections for communicating with the DRAM. Depending on the 3D-stacked DRAM technology this connection could be a high-speed short-reach links or an interposer substrate. Off-chip bandwidth is more limited compared to the internal bandwidth provided by TSV based connections. Moreover, the limited off-chip bandwidth which also serves to the host processor is now also shared with the accelerator. More critically, external communication requires significant energy compared to the internal TSV based data transfers. For HMC, SerDes based off-chip communication accounts for more than half of the total stack energy.

Separate layer. Another integration option, stacking the accelerator as a separate layer to the stacked DRAM, is shown as ② in Figure 6.1. The accelerator is implemented as a separate die which communicates with the rest of the stack using TSV based vertical connections. However, it still needs a load/store interface that handles the DRAM communication. To handle this issue, it can either use the base logic layer to communicate with the DRAM or implement separate memory/vault controllers. Additional memory/vault controller units will consume extra area and power in the accelerator layer. Furthermore, the DRAM layers will be driven multiple controllers. There needs to be a TSV bus arbitration which allocates the TSVs to the commands coming from accelerator layer and the base logic layer. If the accelerator layer uses the base logic layer as an interface to the DRAM, the need for separate memory/controller in the accelerator layer disappears. In this case, the requests coming from the accelerator layer consume TSV cycles while sending the requests/data to the logic layer as well as receiving the data back from the logic layer. Though ② in Figure 6.1 shows that the accelerator is stacked at the top, it could be put on top of the base logic layer. In this case, there can be separate connections between the logic layer and the accelerator layer, possibly using micro-bumps between these two layers. Hence, the bandwidth pressure on the TSVs due to the communication between the accelerator and the logic layer can be relaxed. However, sandwiched between the DRAM layers and the base logic layer, the accelerator layer will potentially suffer from a heat dissipation issue with this approach. Furthermore, integration as a separate layer increases

the manufacturing cost and complexity of the memory stack.

In logic layer. As demonstrated with ③ in Figure 6.1, the accelerator can be moved closer to the memory by integrating it in the base logic layer. In this scheme, the accelerator directly communicates with the logic layer memory controllers. Therefore, it does not require additional data and command transfers through the TSV bus to communicate with the logic layer. Note that both off-stack integration and stacking the accelerator on top of the logic layer while connecting them with separate microbumps allow direct communication between the accelerator and the logic layer. However, these solutions require very high cost hardware modifications such as a separate dies, microbump and interposer connections. Integration in the logic layer is much more cost effective compared to the previous schemes, though it requires simple modifications in the logic layer. Moreover, eliminating multiple hops through the TSVs or external accesses through interposer/SerDes links greatly improve latency, bandwidth and energy efficiency. The main shortcoming of this approach, however, is the limited area and power consumption headroom for a custom accelerator implementation. Logic layer already includes native control units such as an interconnection fabric, memory/vault controllers, SerDes units, etc. It is reported these units leave a real estate to be taken up by custom logic [94]. However, compared to a separate die, the area and power consumption headroom is more limited. This scheme is a better fit for simple accelerators with low power consumption and area overheads.

Integrating HAMLeT. Main motivation of the HAMLeT architecture is to achieve high throughput and bandwidth utilization using the limited area and power budget very efficiently. DRU and ARU units are very area and energy efficient, as we will see later in detail, yet they can effectively utilize the high bandwidth through a simple parallel architecture. This allows us to pursue the option ③ for 3D-stacked DRAM based HAMLeT implementation. Figure 6.2 demonstrates the logic layer architecture for this option. The logic layer resembles a HMC-like 3D-stacked DRAM which features the native control units and an interconnection network that connects these control units. Figure 6.2 also highlights the BIST (built-in self test) unit attached to the interconnect. This

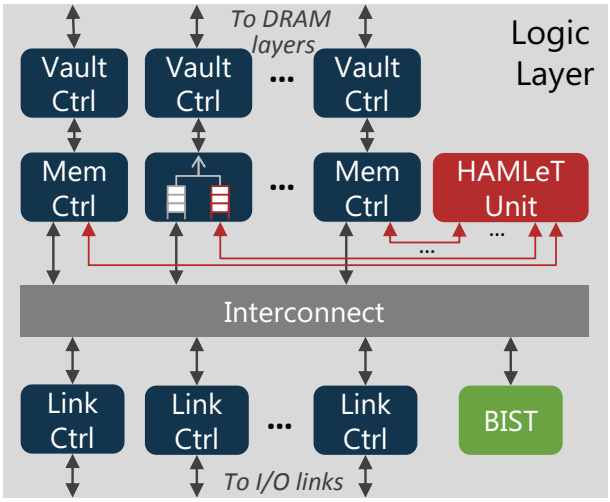


Figure 6.2: Logic layer architecture for integrating HAMLeT into the 3D-stacked DRAM.

demonstrates that heterogeneous components already exist in the logic layer which extends the fundamental operation of the stacked DRAM. There are other proposals that follow a similar approach where the accelerator blocks are attached to the interconnect fabric in the logic layer [88]. When the accelerator is attached to the interconnect which connects all the memory/vault controllers and I/O links, it can communicate with any of the vaults via the corresponding memory controller. This creates a flexible infrastructure for accelerators to communicate with different vaults of the DRAM layers. However, it complicates the interconnect design significantly. For example, the HMC interconnect is a crossbar switch [94]—attaching more accelerators to the fabric will have a quadratically increasing cost.

DRU unit in the HAMLeT features switch networks which allow it to route any input/output to/from any SRAM bank. Furthermore, a rearrangeably non-blocking switch network is sufficient for the purpose of the permutation based data reorganizations, which is much less costly than a crossbar switch. Hence, HAMLeT is integrated even behind the interconnect, directly connected to the memory controllers as shown in Figure 6.2. Each port of the HAMLeT has a single point-to-point connection to a memory controller. This enables consuming high-bandwidth data transferred with each independent vault. Necessary data exchanges are handled in the HAMLeT by the DRU.

Integration scheme given in Figure 6.2 also requires modifications to the memory controllers in

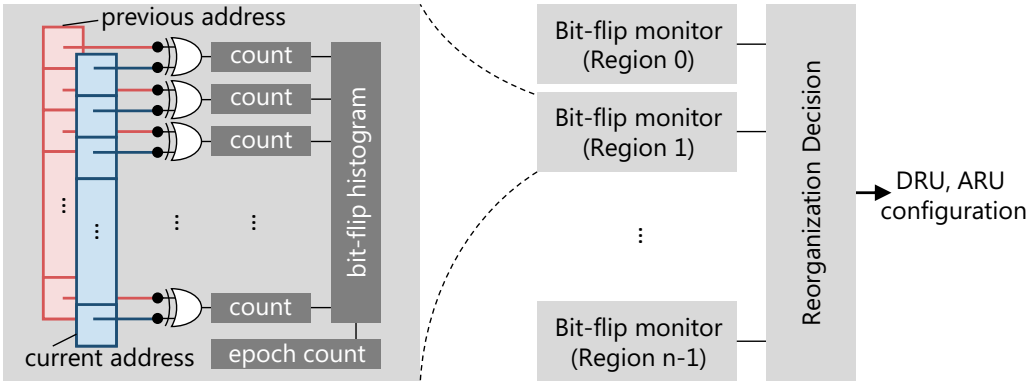


Figure 6.3: Physical address bit-flip rate (BFR) monitoring unit for multiple memory regions.

the logic layers. Now the memory controller needs to serve for the data reorganization requests, in addition to the the regular memory requests coming from the host processor and from the other accelerators. Handling the host requests and the reorganization requests served in parallel efficiently will be discussed next in Section 6.2.

6.1.2 Implementing Memory Access Monitors

As discussed in Chapter 5, in the automatic mode of operation, memory access stream of the host applications are monitored to determine a possible inefficient address mapping where the physical access patterns do not utilize the DRAM parallelism and locality. To determine such inefficient DRAM address mapping cases, we inspect the change rate, or bit-flip rate (BFR) of DRAM address bits, as introduced before. Bits with high BFR in physical address can be mapped onto parallelism bits in the DRAM address (i.e. bank, vault, layer). On the other hand, bits with low BFR should be mapped into the locality bits (i.e. row bits in DRAM address).

Figure 6.3 demonstrates the memory access monitoring hardware to determine the BFR in the physical address stream. The basic operation of the bit-flip monitor is to XOR the current address with the previous reference to determine which bits are flipped. Flipped bits increment a counter that tracks the number of changes of a particular bit. This unit counts the total number of requests in epochs. At the end of every epoch the BFR values of every bit is normalized with the number of total accesses to generate the bit-flip histograms. An example bit-flip histogram is given in Figure 6.4.

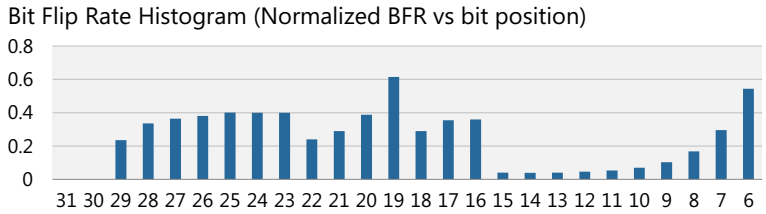


Figure 6.4: An example bit-flip histogram of a memory access stream (facesim from PARSEC [32]).

There are various methods to use the BFR histograms for deciding a DRAM address remapping. In interpreting the BFR values, our goal is to devise decision mechanisms that are robust, easily implementable in hardware, and capture the application behaviour realistically. For that purpose, we have a multi-step decision mechanism that uses the BFR histogram in various ways to find the most efficient address remapping.

BFR value based. First, the decision logic implements BFR value based *stuck* and *streaming* detectors. These are essentially per address bit comparators that check whether the BFR value is larger or smaller than certain threshold values. If a BFR value is larger than a pre-set streaming threshold, that bit is marked as streaming. Similarly, if the BFR value of a bit is smaller than the stuck threshold, it is marked as stuck. Ideally a stuck bit in the physical address should be mapped into one of the locality bits in the DRAM address. A stuck bit mapped into the parallelism part results in underutilized parallelism. Similarly, a streaming bit in the locality part of the DRAM address leads to excessive row buffer misses. If the streaming or stuck behavior of such bits is consistent for a few epochs, then they are great candidates for remapping.

BFR ratio based. For some memory access patterns, there are no extremely high or low BFR values in the physical address stream, in other words, no streaming or stuck bits. In such cases, a BFR ratio based decision mechanism is utilized. The ratio between the highest BFR value from the locality bits and the lowest BFR value from the parallelism bits is calculated. The BFR ratio is compared against a pre-set threshold. If the highest BFR ratio is greater than the BFR ratio threshold consistently for several epochs, the corresponding bit pair considered to be a candidate

for remapping. This operation is repeated to find other bit pairs where the BFR ratio is greater than the threshold to remap.

Both the BFR value based and BFR ratio based thresholds can be reconfigured during the runtime. They are initialized to empirically determined values. These values can be reconfigured explicitly by the user to a desired value when the configuration is offloaded. Furthermore, they can be adjusted dynamically based on the average maximum and minimum BFR rates observed over a period of time.

In both of these techniques, BFR is observed over a period of time, for several epochs, to make a robust decision. This period can become longer to coalesce possibly multiple reorganization into a single one. Data reorganization requires a full pass over the dataset that is reorganized. Multiple separate data reorganizations requires multiple passes. Yet, separate reorganizations can be combined into a single permutation (see rule 3.15) which can be implemented as a single pass over the dataset. Therefore, issuing a reorganization is delayed for a possible coalescing if there is another bit remap candidate being monitored but not yet observed long enough.

6.2 Handling Parallel Host Access and Reorganization

Near-data processing creates a heterogeneous computing system where the DRAM is shared between multiple processing elements. These processing elements generate memory access request streams with very different characteristics. These characteristics include request rate (memory intensity), bandwidth and latency sensitivity, parallelism and locality in the memory accesses, etc. Memory system needs to maximize the overall throughput and provide a fair service to all of the request sources based on these characteristics. We analyze this concept in the context of parallel data reorganization and host memory access.

HAMLeT is integrated in the logic layer of the 3D-stacked DRAM behind the interconnect, directly interfaced to the memory/vault controllers as shown in Figure 6.2. Data reorganization requests are directly submitted to the memory controller where they are buffered in a separate queue.

Memory controller will decide which request to schedule from the regular queue and the reorganization queue. To this end the main goals are; (i) extract highest overall throughput, (ii) devise efficient mechanisms that provide a control over the bandwidth allocation to different sources, and (iii) provide a minimum service guarantee to the host application while the data is transparently reorganized.

6.2.1 Block on Reorganization (BoR)

The first mechanism provides the highest priority to the data reorganization requests. In the BoR technique, when a memory region is under reorganization, all the regular memory accesses to that region are blocked. Independent processes can still access other memory regions. This allows very fast reorganization operation by minimizing the interference from the host memory accesses. Data reorganization requests are highly memory intensive and regular. Regular and intensive memory accesses efficiently exploit the parallelism and locality of the stacked DRAM layers. Hence prioritizing the DRAM-friendly access patterns of the data reorganization requests greatly reduce the latency overhead of the data reorganization. Moreover, the memory regions under reorganization are only accessed by the HAMLeT which eliminates coherence problem. This simplifies the scheduling algorithm and the memory controller design since it does not have to handle any coherence updates. However, this operation stalls the host processor for a period of time. Host processor is not allowed to access the memory while the reorganization is taking place. As we will see later, the latency of the reorganization can be in the order of milliseconds. This overhead is comparable to a potential page fault in the program. Nevertheless, overhead of the reorganization is amortized during the long runtime of the application. However, this can still be problematic for latency-sensitive critical applications.

6.2.2 Access on Reorganization (AoR)

The previous scheme, BoR, aims simplicity and maximum reorganization throughput by blocking the host accesses when data reorganization is taking place. Whereas, AoR allows host requests

and the reorganization requests to be serviced in parallel. Main goals of this technique are to devise efficient mechanisms that provide a control over the bandwidth allocation to different sources and extract maximum overall throughput while providing a minimum service guarantee to the host application.

Fine-grain interleaving. First we go after a fine-grain interleaving scheme to maximize the request issue rate. Abundant parallelism available within the stacked DRAM system creates the opportunity to issue several requests to different banks, vaults and layers. Low-latency and high-bandwidth vertical TSV connections create a substrate that support very high request rates. Furthermore, multiple banks, vaults and layers create a parallelism that can service several requests concurrently. A fine-grain interleaving of multiple request streams can utilize this substrate to issue several requests both spatially and temporally.

However, the main disadvantage of this technique is that it does not preserve the locality in the memory access streams by interleaving them at a fine granularity. As discussed previously, data reorganization requests are highly memory intensive and regular. Furthermore, they rely on efficiently exploiting the parallelism and the locality from the DRAM to achieve high bandwidth. Mostly irregular host accesses interfere with the DRAM-friendly access patterns of the data reorganization. This leads to excessive row buffer misses and wasted TSV cycles while waiting for the DRAM banks to precharge and activate new rows. Hence the increased issue request rate is not necessarily reflected as improved bandwidth utilization.

Request batching. In order to preserve the locality of the access streams request batching is performed where multiple requests from each memory access stream are bundled together to form a batch. Then batches are interleaved instead of individual requests. Within the batch boundaries memory access locality and parallelism of the corresponding access stream is preserved. Interleaving batches themselves, on the other hand, enables sharing the DRAM bandwidth among different sources. Furthermore, the size of the batches determine the bandwidth sharing ratio between the sources. Hence, adapting the size of the batches for different access streams provides an efficient

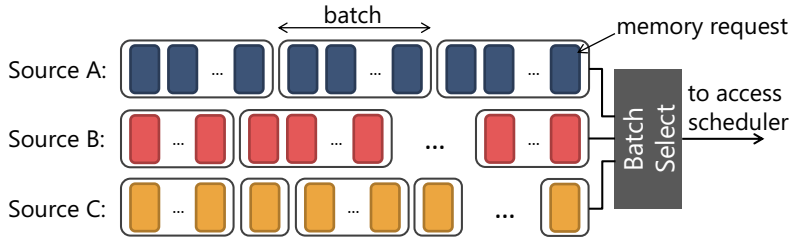


Figure 6.5: Overview of the request batching with three sources.

control mechanism for allocating bandwidth to different sources.

However, course-grain interleaving with batch scheduling does not utilize the request issue rates as efficient as fine-gain interleaving. If there are no available requests that can be scheduled at a given time instant while serving the requests from a batch, no request is scheduled without searching the batches from other request streams. Nevertheless, this tradeoff is beneficial to preserve the locality and parallelism within batches.

Overview of a request batching technique is given in Figure 6.5. In our case there are two main memory access streams; host requests and data reorganizations requests. As given in Figure 6.2, HAMLeT is directly connected to the memory controllers in the logic layer. The memory controller queues the reorganization requests in a separate buffer and continuously makes batches from both of the request streams in-order. Then a batch selection unit arbitrates between the sources and forwards the selected batch down to the memory access scheduler. The access scheduler is decoupled from the batching and it implements an out-of-order DRAM access scheduling. Target batch sizes are configurable. One can allocate more bandwidth to one of the sharers by increasing its target batch size. This provides a very effective knob to adjust for the bandwidth sharing. By making sure that the batch size for the host stays above a certain minimum, the memory controller provides a minimum service guarantee to the host processor during data reorganization.

Note that the host request stream is not as memory intensive and regular as the reorganization request stream. Hence, forming a batch with the targeted number of memory requests for the host access stream can take a very long time since the the memory intensity is low. A strict request-based batching can lead to inefficient batches. Request batching is assisted with a preemptive time-slicing

to avoid these cases. When the batches are being formed, if the allocated time slice expires before bundling the targeted number of requests together, the batch closes with the existing number of requests. This leads to batches with varying number of requests within the access stream which is also exemplified in Figure 6.5.

Overall, request batching with time slicing is a robust technique that provides a efficient control to allocate the shared DRAM bandwidth to different request streams. Compared to the fine-grain interleaving, there can be underutilized request issue slots when there are not enough requests in the batch, especially for the host request batches. However, it preserves the locality and parallelism for each individual request streams. Furthermore, by sizing the batches properly, it can provide a minimum service guarantee to the host application while the data is transparently reorganized

6.2.3 Handling Memory Coherence

Access on reorganization (AoR) allows the data reorganization to take place while the host processor accessing the same data. This allows sharing the DRAM bandwidth for a parallel operation, however it also creates some side effects including memory coherence problem. For the BoR operation, when the reorganization is finished, ARU handles the address remapping of the reorganized data. Hence, it does not create any coherence problem. However, for the AoR operation, while the reorganization is taking place such that the data is read from a source region and written into a destination region in the reorganized form, an update from the host to the source region can create a coherence problem.

In order to handle the coherence issue, any update (i.e. write) to the source region needs to propagate to the target region before the reorganization is finished. Reorganized region will not be visible to the host processor until the reorganization is completed. Hence, the synchronization point is the end of the reorganization. Note that the updates to the other regions will not create any coherence issue and can be committed freely.

Coherence Merge Buffer (CMB). In addition to the regular scheduling queues, the memory controller is modified to implement a coherence merge buffer (CMB). Incoming writes to a source region which is under reorganization are copied into the CMB as a coherence update request. CMB is drained to propagate the coherence update requests to the target region. A data reorganization is considered to be completed when the data reorganization requests and the coherence update requests in the CMB are both finished.

When a coherence update request is inserted into the CMB, first it checks if there is any existing request in this time-ordered buffer that goes to the same target address. The final value of the memory location is determined by the latest written value. This corresponds to a write-after-write (WAW) situation where the initial value is overwritten by the later one. Therefore, in such cases, the new request is enqueued while the older request in the buffer is dropped. This reduces the number of redundant requests.

Furthermore, whenever a data reorganization request is scheduled CMB controller also checks the CMB if there is any coherence update to the same target address waiting to be serviced that is older than the currently scheduled reorganization request. Such coherence updates are also redundant since the scheduled reorganization requests read the most recent copy from the source region. Such requests are also dropped from the CMB.

CMB requests are scheduled first when there is a row-hit to an open bank, and then if the buffer is almost full. This scheduling mechanism tries to minimize the interference between the CMB requests and already existing host and reorganization requests. Also it keeps the CMB generally occupied to maximize the chance of dropping the redundant coherence requests via previously described techniques.

Multi-stage Reorganization Until now, we assumed that the data reorganization is an atomic operation such that the outcome of the data reorganization is made visible via address remapping when the entire operation is completed. Atomicity assumption simplifies handling the address remapping, scheduling the requests and the hardware implementation. However, it has side effects on the coher-

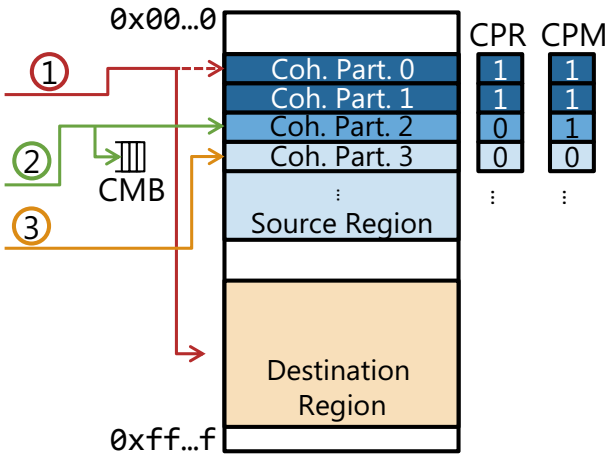


Figure 6.6: Handling the host memory accesses in the multi-stage reorganization scheme.

ence updates. During the data reorganization, any incoming write to the source memory region also generates a coherence update to the destination for coherence. Due to atomicity, coherence updates are required until the data reorganization is completely finished.

In the multi-stage reorganization the goal is to divide the memory region under reorganization into coherence partitions. When a coherence partition is accessed by the DRU it is marked as *modified*. Upon the reorganization of a coherence partition is completed, it is marked as *reorganized*. *Modified* and *reorganized* informations are kept in coherence partition reorganized (CPR) and coherence partition modified (CPM) registers. Figure 6.6 demonstrates three different memory access scenarios in the multi-stage reorganization. Reorganized coherence partitions (i.e. CPR=1, CPM=1) are opened to the host access, although the entire data reorganization is incomplete. In this case (① in Figure 6.6), incoming requests from the host are forwarded via ARU to their remapped locations. In this way, the host processor can access all the reorganized coherence partitions without generating the additional coherence updates. Furthermore, if the host accesses a coherence partition that is unmodified (i.e. CPR=0, CPM=0), than the access is directly transferred to the source region (③ in Figure 6.6). This access will not generate any coherence update since the coherence partition has never been accessed by the DRU. If a coherence partition is modified but not reorganized (i.e. CPR=0, CPM=1), then the access generates a coherence update which is inserted into the CMB (② in Figure 6.6). Overall algorithm for determining the coherence updates and the address forward-

```
// get the configuration for the region of the access
oldConfig = configStore[memAccess.region];
// check if the accessed region is under reorganization
if(oldConfig.underReorganization == true) {
  // get the coherence partition for the access
  coh_part = getCohPart(memAccess);
  // check CPR to see if the reorganization is finished for the partition
  if(CPR[coh_part] == true) // coherence partition reorganized
    BSUconfig = newConfig.BSUconfig;
  else if(CPM[coh_part] == false) // coherence partition unmodified
    BSUconfig = oldConfig.BSUconfig;
  else if(CPM[coh_part] == true) { // coherence partition modified
    BSUconfig = oldConfig.BSUconfig;
    CMB.insert(memAccess); // insert a coherence update
  }
}
else // the accessed region is not under reorganization
  BSUconfig = oldConfig.BSUconfig;
```

Table 6.1: Algorithm for determining the BSU configuration and coherence update for the host memory accesses in the multi-stage reorganization scheme.

ing (i.e. BSU configuration) is summarized in Table 6.1. Multi-stage reorganization reduces the coherence updates significantly, as we will analyze later in detail. However, CMB controller needs to keep track of the reorganization requests and the coherence partitions. Further, ARU needs to support the address mapping based on the coherence partitions.

6.3 Host Architecture/Software Support

For the physical data reorganization driven by the memory controller, user software and OS are kept unchanged. In this use case, the memory controller monitors the memory accesses in the physical domain and issues a physical data reorganization. Hence, the data reorganization only requires a physical to physical address remapping which is completely handled by the ARU.

Physical data migration has been studied in previous work [47, 100, 107]. These approaches use a lookup table implemented in the memory controller to store the address remappings for the data movements. The address remapping lookup table is not scalable to support large scale and fine grain data reorganizations. Hence, these approaches only focus on movement of OS page size data

chunks. Our technique can support data granularity of a single access (e.g. 32 bytes) at much lower hardware cost since it captures the entire remapping information in a single closed form expression which is implemented by the generic ARU.

6.3.1 Memory Management for NDP

For the explicit offload, the host architecture should provide the memory management for both correct and efficient data reorganization in memory. From a system architecture perspective, a host processor is connected to multiple memory stacks where each stack contains HAMLeT units capable of data reorganization in memory. In such configuration, each stack is responsible for reorganization of the local data in the stack. When a data reorganization operation is offloaded, the data for processing should reside in the local stack. Moreover, the processed data should be the most recent copy to avoid coherence problem. Finally, it should stay in the local stack during the reorganization. The NDP memory management system (NMM) should provide these requirements for both correctness and efficiency. In this thesis we utilize the NMM system proposed in [59].

First, NMM should make sure that the data for processing reside in the local stack. For that purpose, a physically contiguous memory region is allocated in the kernel. Than, NMM maps the physically contiguous region into the virtual memory space via *mmap* system call. Hence, the accelerator configuration, commands as well as data can be written from the user side to the allocated memory space. Furthermore, the status of the accelerator and the generated results can be read from this memory space. Hence both host and the accelerator share the same virtual address space. During the mmap call, allocated pages are pinned into memory so that when the operation is offloaded they will not be swapped out. Directly mapping contiguous virtual memory regions into contiguous physical memory regions for specific data structures also have been studied in different contexts [29].

To alleviate the accessibility from the user software, library based NMM functions are also provided. These functions include *malloc*, *free*, *plan* and *execute*. Memory allocation functions *malloc* and *free* only allocate the memory space from the local stack corresponding to the accelerator. Ac-

celerator descriptor, in our case the data structures for DRU and ARU configuration, is represented by the *plan*. Finally, *execute* writes the configuration and commands to the specific command memory space via *memcpy* call. By using these user level memory management functions, configuration data including HAMLeT architecture configuration, allocated source/destination memory pointers and sizes are transferred to the memory. Host does not offload the data for processing since it already resides in the local stack.

Avoiding Cache Flush. In [59], the host also issues a cache flush to ensure that the accelerator accesses the most recent copy in the memory stack to avoid coherence problems. This is a necessary operation for correctness in general NDP-based hardware acceleration. On the other hand, a data reorganization only rearranges the data without using the actual values. Therefore it does not require the most recent copy of the particular data. This implies that while data is being reorganized in the main memory, local caches can have dirty data. Hence, the costly cache flush operation can be avoided for the in-memory data reorganization. HAMLeT features the ARU which automatically forwards the access to the remapped locations. Hence any read access, that is not filtered by caches, is forwarded to the corresponding remapped location which holds the valid value of the data. Dirty cache line write-back requests, however, can change the content of a reorganized data. These requests are properly propagated to the original and/or remapped location(s) by the coherence merge buffer (CMB) as discussed in Section 6.2.

Explicit offload for near data processing is known to be costly where cache flush is the longest operation. It writes any dirty cache line back to the main memory and invalidates the valid cache lines so that the future references go to the main memory. Modern ISAs such as x86 provide instructions for flushing or invalidating specific cache lines (CLFLUSH) or the entire cache hierarchy (WBINVD, INVD) [17]. Nonetheless, searching or keeping track of the cache lines from the source and target regions of the NDP requires additional complexity. Overall, flush and invalidate operations result in a long latency that is on the critical path of the NDP. Furthermore, they consume bandwidth and energy. To handle the coherence easily by avoiding the invalidate and flush operations [50] makes the NDP memory region non-cacheable. However, in this case after the in-memory accelerator fin-

ishes processing, the resulting data needs to be moved to a cacheable region, otherwise exploiting the locality is sacrificed. In our case, on the other hand, costly cache flush operation is eliminated while keeping the NDP memory regions cacheable.

Finally, for the automatic mode of operation, all the side effects of the data reorganization is handled completely in memory via HAMLeT. However, fast out-of-place data reorganization requires a free memory region as discussed previously. The reorganized data is first mapped to the free region. Then, after the data reorganized, the source region is labelled as free region. Hence, the free region is allocated once in the beginning and it is passed around transparently by the HAMLeT. NMM system allocates the free memory region for the automatic mode of operation and passes it to HAMLeT before it is enabled for automatic reorganization.

6.4 Putting It Together

Figure 6.7 summarizes the fundamental use cases where the discussed components are put together.

Automatic Mode. In automatic mode of operation, HAMLeT architecture is driven completely by hardware monitors. Memory access monitors determine a possible inefficient address mapping where the physical access patterns do not utilize the DRAM parallelism and locality. Once the address remapping is determined by the hardware monitors, HAMLeT is directly configured bypassing the Spiral rewrite engine. Along with the target address mapping scheme, the hardware monitors transport the number of contiguous memory regions and their IDs to HAMLeT. Then the DRU performs the corresponding permutation to achieve the target address mapping scheme. This mechanism also collaborates with the NDP memory management system to ensure that it operates within allowed memory boundaries. Furthermore, proposed batch scheduling allows both host access and reorganization take place in parallel, while multi-stage reorganization with coherence partitioning reduces the coherence traffic in the memory.

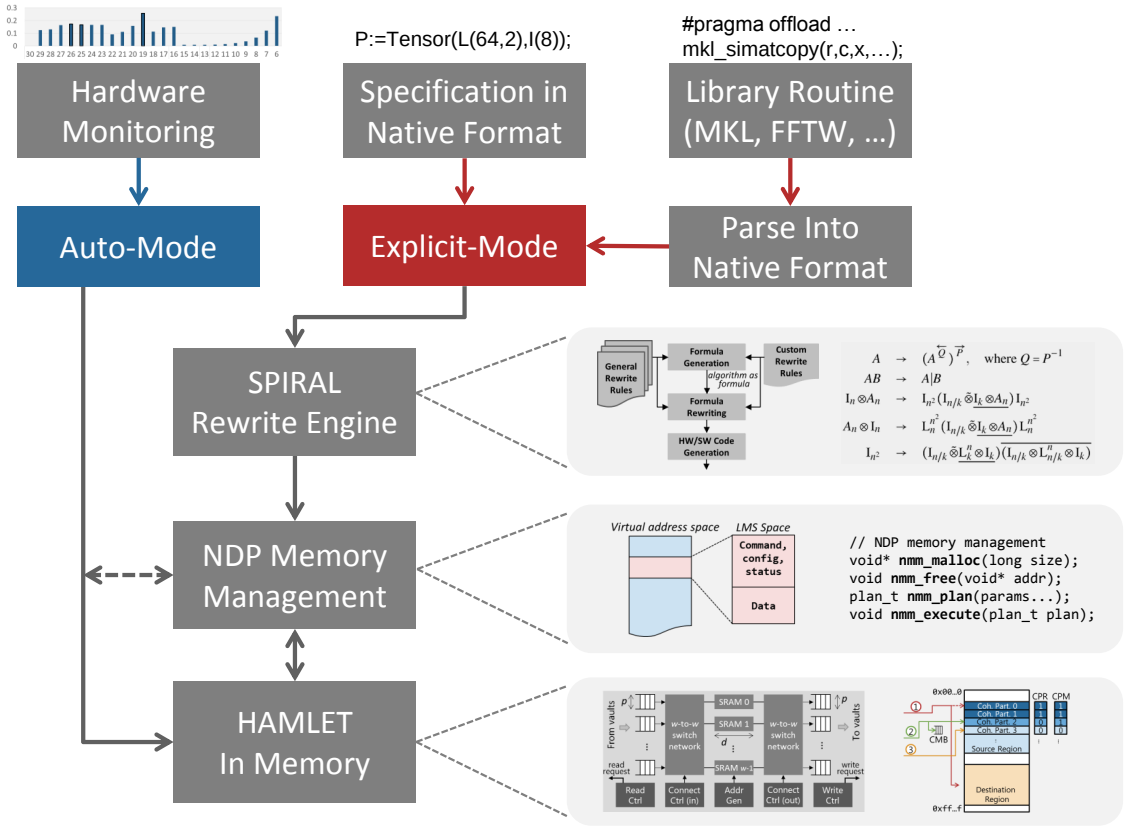


Figure 6.7: Overall view of the fundamental use cases: From specification to in-memory reorganization.

Explicit Mode Explicit mode, on the other hand, is driven by the user level function calls either using the native interface of the SPL or through high-level function wrappers. These operations are then passed to the Spiral rewrite engine. At this stage, Spiral rewrite engine restructures SPL expressions to derive local permutations, memory access patterns and index transformations. Furthermore, it determines the corresponding hardware configuration parameters to program the HAMLeT. Then the NDP memory management handles the explicit memory allocation and offloading the ARU/-DRU configurations. With the help of CMB handling in-memory coherence and ARU implementing hardware based indirection, explicit cache flush is removed from the offload operation.

Chapter 7

Evaluation

This chapter provides an experimental evaluation for the fundamental use cases of the 3D-stacked DRAM based implementation of the HAMLeT architecture. The chapter starts with the details regarding modeling and simulation of the 3D-stacked DRAM based hardware acceleration. Then, it focuses on automatic and explicit use cases individually. In the explicit use case, software transparent dynamic data reorganizations, address remapping efficiency, design space details, multi-programmed workloads, as well as parallel data reorganization and host processor execution issues are discussed. For the explicit use case, it focuses on accelerating commonly used data reorganization routines selected from a widely adopted HPC software library, Intel Math Kernel Library (MKL). Explicit use case evaluations also include important design choices such as placement of the DRU unit and overhead of the offloading from software. The evaluations are concluded by hardware synthesis results to analyze power and area cost of the fundamental components of the HAMLeT architecture.

Table 7.1: 3D-stacked DRAM low level energy breakdown.

Parameter	Value (pj/bit)	Reference
DRAM access	2 - 6	[38, 65, 111]
TSV transfer	0.02 - 0.11	[13, 38, 117]
Control units	1.6 - 2.2	[6, 65]
SERDES + link	0.54 - 5.3	[44, 65, 69, 78, 96]

7.1 Experimental Procedure

7.1.1 3D-stacked DRAM Modeling

We use a custom trace-driven, command-level simulator for the 3D-stacked DRAM. It features a simple CPU front-end similar to the USIMM DRAM simulator [35].

Unless noted otherwise, it models per-vault request queues and FR-FCFS scheduling. Low level timing and energy parameters for DRAM and TSV are faithfully modeled using CACTI-3DD [38] and published numbers from literature [13, 65]. Logic layer memory controller performance and power are estimated using McPAT [6]. Finally, the SerDes units and off-chip I/O links are modeled assuming a high speed short link connection [44, 78, 96]. We assume 4 pj/bit for the total energy consumption of combined SerDes units and off-chip links at 32nm technology node. To demonstrate the relative cost of each operation in the 3D-stacked DRAM, we summarize a typical energy breakdown of various operations in Table 7.1. However, these values change depending on the particular configuration of the memory.

Table 7.2 provides four memory configurations, namely high (HI), medium-high (MH), medium-low (ML) and low (LO), that will be used in our later simulations. For each configuration, it specifies the 3D-DRAM architecture parameters, i.e. number of vaults (banks), layers, TSVs, links, etc. It also provides aggressive and conservative model estimations for the internal bandwidth and power consumption. These are the maximum bandwidth that the stack can reach internally when all the vaults/banks/layers are saturated, and the power consumption at that bandwidth respectively. The 3D-stacked DRAM simulator is driven by the low-level timing and energy estimations from

Table 7.2: 3D-stacked DRAM configurations.

Parameter	HI	MH	ML	LO
Vault (#)	16	8	4	2
Layer (#)	8	4	4	2
Total TSV (#)	2048	2048	1024	512
Agsv. Internal BW (GB/s)	860	710	360	90
Agsv. Internal Power (W)	37	30	15	4
Agsv. Internal En. Eff. (pj/bit)	5.38	5.28	5.21	5.55
Cons. Internal BW (GB/s)	790	520	280	65
Cons. Internal Power (W)	34	23	13	3
Cons. Internal En. Eff. (pj/bit)	5.38	5.53	5.80	5.78
Link (#)	8	8	7	1
Link BW (GB/s)	60	40	40	40
External BW (GB/s)	480	320	280	40
Power (Watt)	45	30	25	4
External En. Eff. (pj/bit)	11.7	11.7	11.2	12.5

(Agsv.=Aggressive, Cons.=Conservative, En. Eff.=Energy Efficiency, BW=Bandwidth)

CACTI-3DD [38] and published numbers [13, 65]. Aggressive and conservative estimations are based on these low-level parameters selected from the available ranges in these numbers. Furthermore, Table 7.2 provides the bandwidth available to the off-chip by the high speed links and the overall power consumption where these links are fully saturated. DRAM page size is 1 KB for all the configurations. The unusual DRAM page size of 1 KB is a typical value for 3D-DRAM (from 256 byte [65] to 2 KB [114] are reported). Table 7.2 shows that these configurations can reach overall energy efficiency of 11-12 pj/bit. On the other hand, internal operation bypasses the costly SerDes IO which both reduces the energy cost and increases the maximum reached bandwidth. This leads to 5.2-5.8 pj/bit energy efficiency for the internal accesses based on aggressive and conservative estimations. These evaluations agree with the published numbers for HMC which can achieve 10.48 pj/bit overall energy efficiency where DRAM layers spend only 3.7 pj/bits [65]. Throughout the thesis conservative estimations are used unless noted otherwise. Note that conservative estimations are in favor of the baseline systems and disadvantageous for improvement estimations of our solutions.

7.2 Automatic Mode

7.2.1 Dynamic Data Reorganization Overview

First, we focus on software transparent dynamic data reorganization technique. As described in Section 5.1 this mechanism combines data reorganization and address remapping in hardware by utilizing the HAMLeT architecture integrated in 3D-stacked DRAM. HAMLeT architecture features the address remapping unit (ARU) and the data reorganization unit (DRU). DRU is specialized for efficient permutation based data reorganization and ARU handles the address remapping for multiple memory regions. In the automatic mode of operation, HAMLeT architecture is driven by a hardware based memory access monitoring system that profiles memory access patterns of applications. The monitoring determines changes in consecutive DRAM addresses, or bit-flip rate (BFR) as introduced before. Bits with high BFR in physical address should be mapped onto parallelism bits in the DRAM address (i.e. bank, vault, layer) to improve parallelism. On the other hand, low BFR ones should be mapped into the locality bits (i.e. row bits in DRAM address) to minimize the row buffer misses. Once the monitoring unit determines such an address remapping, it configures the ARU/DRU units to achieve the data reorganization and address remapping. This enables software transparent dynamic data layout operation, where both the profiling and the reorganization are handled in hardware.

We evaluate the memory traces publicly available from JILP Memory Scheduling Championship [14]. This trace repository includes memory traces from PARSEC [32], SPEC CPU 2006 [63], BioBench [25] trace suits and some undisclosed commercial workloads. For the dynamic data reorganization experiments, we use the MH (medium high) configuration for the 3D-stacked DRAM (see Table 7.2). Processor configuration is given in Table 7.3.

First, we focus on the effect of address remapping on the DRAM row buffer miss rate. Row buffer hit rate evaluation does not include the statistics of the accesses coming from the DRU to reorganize the dataset, it is only based on the host DRAM accesses. In this evaluation, we allow address remappings with up to 3 bit pair swaps to keep the data reorganization simple. First the BFR value

Table 7.3: Processor configuration.

Parameter	Value
Cores	1-4 cores @ 4 GHz
ROB size	160
Issue width	4-32
Pipeline depth	10
Baseline address map	row:col:layer:vault:byte
Memory bus frequency	1.0 GHz
Memory Scheduling	FR-FCFS, 96-entry queue

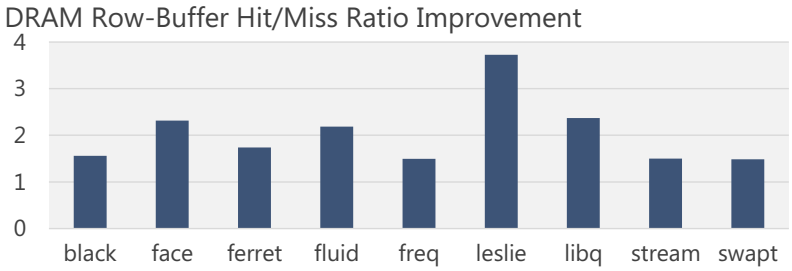


Figure 7.1: Row buffer miss rate reduction via physical address remapping with data reorganization.

based stuck and stream detectors are checked. If there is no BFR value based anomaly detected, then BFR ratios are calculated where the highest BFR in locality region is divided by the lowest BFR in the parallel region to determine the highest BFR ratio. If the highest BFR ratio is smaller than the BFR ratio threshold, BFR ratio calculation is stopped since the highest BFR ratio is already smaller than the threshold. If the BFR ratio is larger than the threshold, BFR ratio calculation continues with the second largest (smallest) BFR values from locality (parallelism) bits until the BFR ratio is smaller than the threshold or it reaches the target bit pair swap goal. The DRAM address stream monitoring is carried out in 50k-cycle epochs. If the detected inefficient BFR behavior is consistent for 4 epochs, a data reorganization is issued to change the address remapping. Before issuing the reorganization, monitoring unit checks if there are any other remapping candidates that are not yet validated for 4 epochs. In that case it delays the reorganization aiming to include that candidate in the address remapping by consolidating them into a single data reorganization. Figure 7.1 shows the improvement in the DRAM row buffer hit to miss ratio with the dynamic data reorganization.

Remapping frequently flipping bits from row address bits to the vault and layer address bits signif-

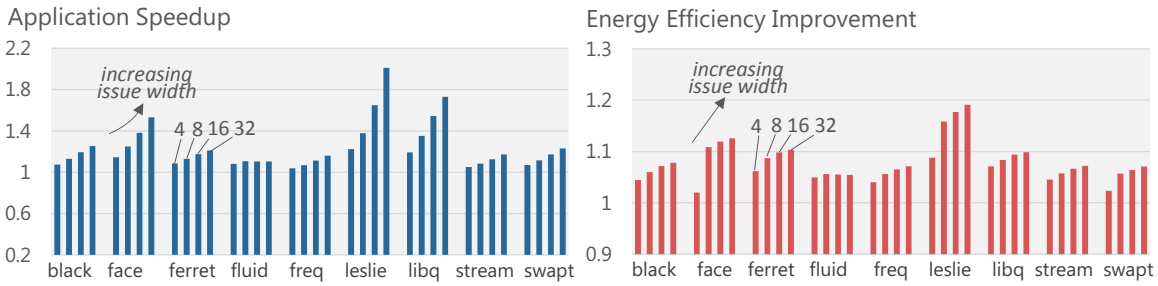


Figure 7.2: Performance and energy improvements via physical address remapping with data reorganization.

icantly reduces the miss rates. It also improves the parallelism by scattering the accesses to vaults and layers more efficiently. This translates into improved performance and energy efficiency as shown in Figure 7.2. This evaluation uses BoR (block on reorganize) mechanism where the host accesses are blocked while the data is under reorganization. We observe that more memory intensive applications (e.g. leslie, libquantum) gain higher improvements since their performance are more sensitive to the memory access performance. To analyze the performance improvement sensitivity to the memory intensity, we also evaluate wider issue width processor configurations (4-32) in Figure 7.2. Wider issue width improves the instruction throughput and memory intensity. With the increased memory intensity, we observe that workloads achieve even higher improvements both in terms of energy and throughput.

Efficient transparent data reorganization requires several design choices including BFR threshold, epoch length, number of memory regions, handling host accesses during reorganization, etc. Provided results for the dynamic data reorganization demonstrates an overview of the possible improvements with this technique. Next, we focus on details of this mechanism and evaluate critical design choices.

7.2.2 Address Remapping: Manual Search vs. Hardware Monitoring

There are as many address mappings as there are bit permutations. However, finding the optimal address mapping can be very difficult. To give an overview of this problem and also the potential, Figure 7.3 demonstrates the improvement achieved by address remapping where only a single bit

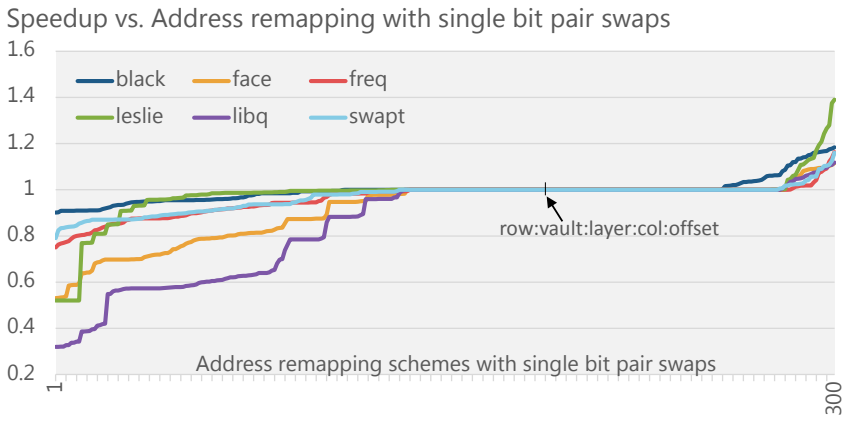


Figure 7.3: Improvements with address remappings where only a single bit pair is swapped.

pair is swapped. Note that, the processor is 32-wide in this experiment to observe the memory improvements more dramatically. We observe that, there exists several schemes that can be achieved by changing the locations of two bits in the address mapping that will lead to performance improvements. However, there are more address remappings that perform much worse than the baseline row interleaving.

BFR based address stream profiling provides an estimation for a more efficient address mapping. This subsection evaluates the effectiveness of the the BFR based monitoring. In this evaluation, first a subset of candidate address mappings are determined. These candidates are only a filtered subset of the entire bit permutations as the entire design space is very large (Assuming 32-bit addresses and 64-byte cache lines, there are 26! different possible address mapping). To determine the candidates, first the whole trace is simulated and group of bits with the highest/lowest BFR values are determined. Furthermore, bits are paired from the high BFR and low BFR groups to create bit pairs with high BFR ratio where the high BFR is from locality bits (DRAM row address) and low BFR is from parallelism bits (vault, layer). Bit pairs not only include highest BFR ratio pairs but also relatively lower BFR ratio pairs and randomly selected pairs from these groups as well. Each address remapping is formed out of up to 6 such bit pair swaps. For each benchmark, we simulate around 50 different address remappings following this procedure. Figure 7.4(a) demonstrates the speedup values achieved with these address remappings.

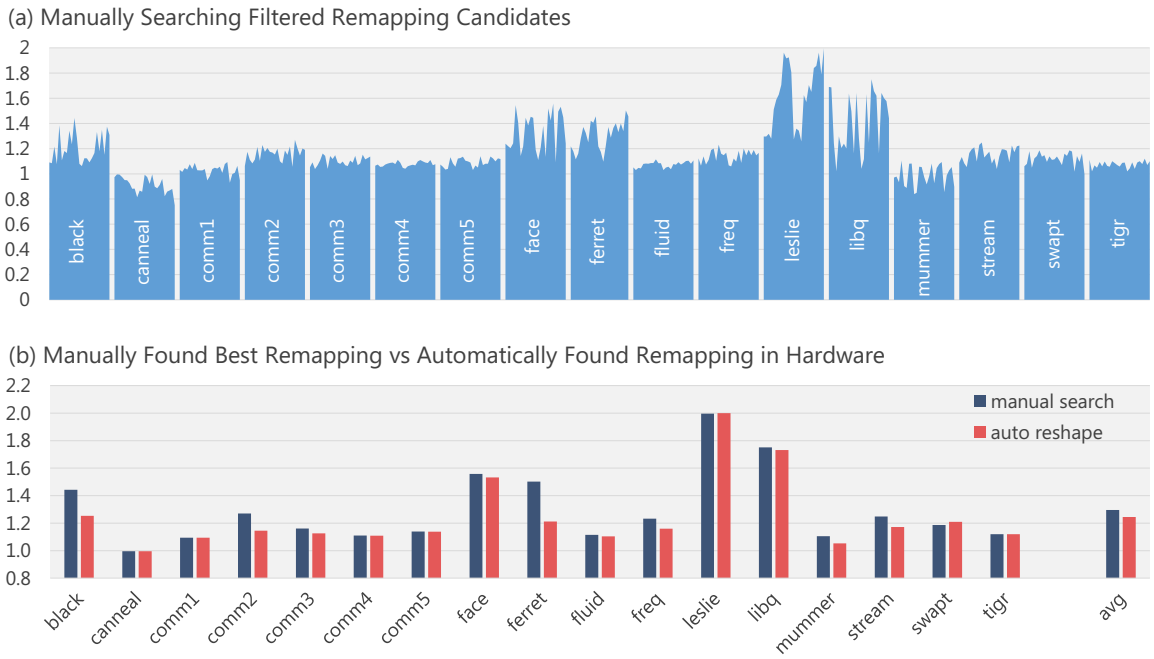


Figure 7.4: Comparing the best address remapping schemes that are manually searched and determined by hardware monitoring.

There is a lot of variation in the achieved improvement based on different address remapping candidates as seen in Figure 7.4(a). We pick the best performing address remapping found by the manual search procedure for each benchmark, and compare it to the automatically found address remapping by the BFR based monitoring in Figure 7.4(b). BFR based monitoring reaches the same improvement for some of the benchmarks (e.g. *leslie, libqantum, facesim*). For some of the benchmarks, however, it cannot find the remappings that perform as high as the manually found ones (*blackscholes, ferret, comm2*). And, for some benchmarks neither of them can find any useful remapping (e.g. *canneal, comm1*). BFR based monitoring usually performs well if the there is a stride pattern that increase the BFR of certain bits. For example *leslie, libqantum,* and *facesim* have relatively regular access patterns. As long as the problematic bit mappings are determined, the potential performance improvement can be mined efficiently. Interestingly, for some of the benchmarks the best performing address remappings include the randomly formed bit pairs as well. BFR based monitoring cannot capture such cases. Nevertheless, hardware monitoring that uses BFR metric can perform on average within 5% of the manually found best remapping where maximum deviation is

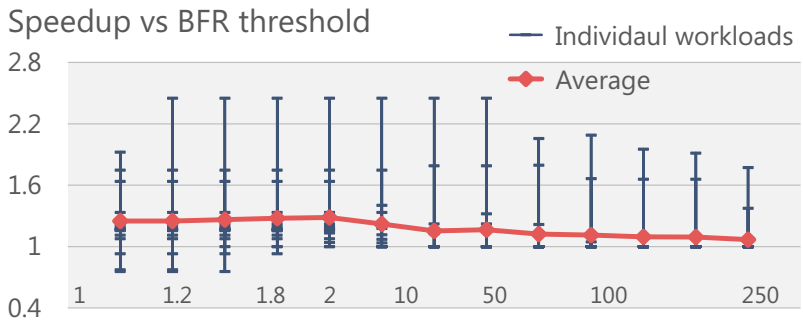


Figure 7.5: Selecting an efficient BFR ratio threshold.

20%.

7.2.3 Tuning the DRU Design Parameters

BFR Threshold. There are three main threshold values to determine a inefficient address remapping. BFR value based stuck and stream detectors compare the BFR of individual bits to the threshold value. Stuck and stream thresholds are set to a very large and small numbers initially. Since the BFR values are normalized every epoch, they can take values between 1.0 and 0. Note that, to avoid floating point arithmetic and minimize the hardware bookkeeping when calculating the statistics they are scaled and quantized to 16-bit registers. Initially, stream and stuck thresholds are set to within 10% of the maximum and minimum values (0.9 and 0.1). Later, these values are adjusted based on the highest and lowest observed BFR. A bit is assigned as a stuck candidate, when its BFR is smaller than the smallest BFR observed or smaller than the initial hard threshold (opposite for the stream candidate).

To improve robustness, BFR ratio based profiling compares the BFR of individual bits from parallelism and locality parts of the DRAM address. As described previously, the highest BFR in locality region is divided by the lowest BFR in the parallel region to determine the highest BFR ratio. If the highest BFR ratio is smaller that the BFR ratio threshold, then BFR ratio calculation is stopped. If the BFR ratio is larger than the threshold, then BFR ratio calculation continues with the second largest (smallest) BFR values from locality (parallelism) bits until the BFR ratio is smaller than the

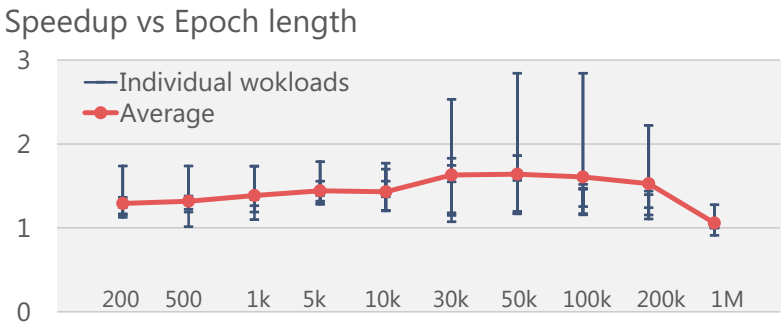


Figure 7.6: Selecting an efficient epoch length.

threshold or it reaches the target bit pair swap goal. Figure 7.5 demonstrates the effect of the BFR ratio threshold to the application speedup. A small BFR ratio threshold can detect even a slight imbalance between parallelism and locality regions of the DRAM address. If an inefficient mapping is detected and fixed early, it greatly improves performance. However, it can select address remappings too early that may not be beneficial in the long run. Towards the left side of the Figure 7.5, we observe that there are some applications that get performance degradation with small threshold values. A large threshold, on the other hand, improves the robustness by allowing the remappings when the BFR ratio of a bit pair is very large. This reduces the chance of inefficient remappings but with a very large threshold it can miss some useful remappings. Threshold values between 1.5 and 3 seem to be the sweet spot for the evaluated benchmarks. We pick a threshold of 2 aiming to minimize early inefficient remappings while maximizing the average speedup.

Epoch Length. Hardware monitoring determines the statistics during every epoch and at the end of each epoch it makes the decisions regarding changing address mapping and data layout. Epoch based approach simplifies the hardware and makes the statistic calculation more tractable. Epoch length needs to be long enough to capture enough statistics and to avoid frequent calculation of histograms. However, if it is extremely long then the reorganization may not be issued in a timely manner.

Figure 7.6 demonstrates the effect of epoch length on the overall application speedup for a set of workloads. We observe that with the very small and very large epochs, overall speedup decreases.

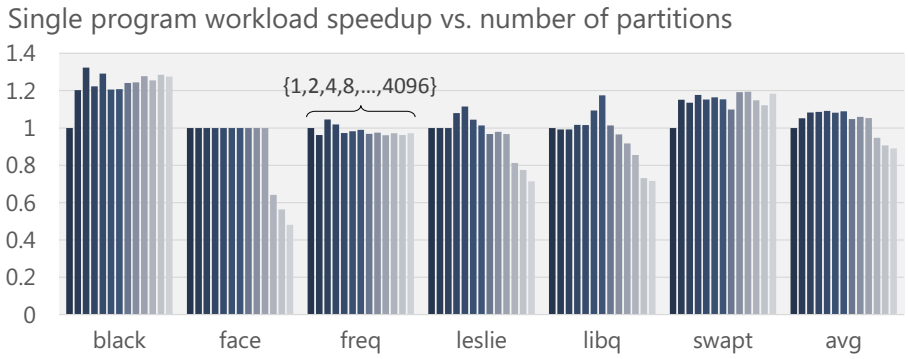


Figure 7.7: Effect of number of memory partitions (1 to 4096) to the speedup for single program workloads. (Normalized to no partitioning)

However, there is a flexibility to pick an efficient epoch length. From 20k to 100k cycle epochs seem to provide the best performance. For our analysis, we chose 50K cycle epochs. Furthermore, statistics are kept for 4 epochs in a moving window fashion.

Number of Memory Partitions. As described in Chapter 4 the memory address space is partitioned into memory regions. ARU supports implementing a different address mapping for each partition. Having multiple memory partitions allow capturing spatial differences in memory access patterns and data layout requirements of different applications which are mapped into different regions in the memory. Moreover, even a single application exhibit varying memory access behavior when accessing data structures that are stored in different parts of the memory. Hence, with multiple independent memory partitions, the whole memory space can have different address mappings optimized for the corresponding access patterns to individual partitions. However, as the size of the memory partition reduces it becomes difficult to identify the data reorganizations per partition. With smaller regions there are reduced number of overall requests to a partition. Furthermore, since data reorganization across the partitions are constrained, improvements with data reorganization become limited.

First, Figure 7.7 demonstrates the effect of increasing number of memory partitions on the speedup for different workloads. Note that the improvements are normalized to the no partitioning case. For some of the worklaods such as *facesim* increaseing the number of partitions does not provide

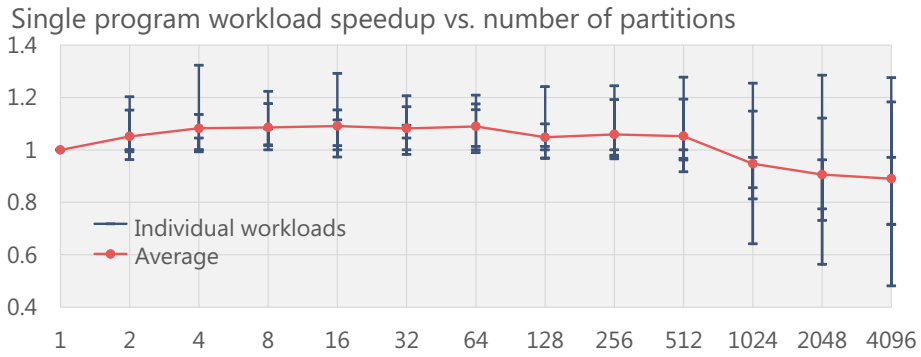


Figure 7.8: Selecting an efficient partitioning scheme.

significant performance improvement. In general, increasing the number of partitions improve the speedup. However, very large number of partitions decrease improvement due to the discussed limitations. Yet, some workloads such as *blackscholes, swaptions* still get improved performance with large number of partitions. Figure 7.8 demonstrates the overall average trend for these set of workloads. Overall, partitioning with 4 to 64 memory regions provide a flexible range to implement efficient data reorganizations.

7.2.4 Multi-program Workloads

Until now we have focused on a single program workloads. In a multi-programmed execution, memory access streams of multiple workloads are interleaved as they share the memory and memory controllers. When memory access streams are interleaved, then the individual access patterns are disturbed which makes the tracking the individual workloads difficult. Though the core/thread/-workload information regarding each memory access can be propagated to the monitoring hardware, it increases the hardware complexity and ties the data reorganization hardware to the specific host architecture.

Instead our hardware monitoring does not distinguish if the memory access stream is coming from a single-program workloads or a multi-program workload. In order to perform efficient data reorganization for multi-programmed workloads it leverages two important observations. First, we observe that when two workloads—which require different address remapping schemes for the best

Table 7.4: Multi-programmed workloads.

Name	2-program		Name	4-program			
w2.01	black	face	w4.01	black	face	libq	fluid
w2.02	fluid	leslie	w4.02	face	leslie	stream	ferret
w2.03	freq	libq	w4.03	freq	libq	leslie	stream
w2.04	stream	ferret	w4.04	stream	ferret	swapt	leslie
w2.05	swapt	leslie	w4.05	swapt	leslie	swapt	leslie
w2.06	libq	leslie	w4.06	libq	leslie	libq	face
w2.07	libq	face	w4.07	libq	fluid	libq	face
w2.08	black	freq	w4.08	black	freq	ferret	swapt
w2.09	ferret	swapt	w4.09	ferret	fluid	black	face
w2.10	libq	fluid	w4.10	libq	swapt	libq	fluid
w2.11	leslie	stream	w4.11	leslie	stream	ferret	freq

performance—are combined, then the resulting address stream generally requires a different third address remapping scheme for the best performance. Hence, instead of trying to isolate different workloads, optimizing for the new access stream formed by the interleave can lead to improvement in the overall performance. Nonetheless, when multiple applications are executed together their accesses are interleaved in time at a fine granularity, but their spatial characteristics are mainly preserved. Each workload works on a separated memory space assuming there is no data sharing between different programs. We also assume a virtual memory system similar to the one proposed in [29] where virtually contiguous large memory spaces are directly mapped to contiguous physical space. Hence, memory partitioning implemented in the hardware monitors can capture the spatial differences in the memory access patterns of different workloads. This enables multiple address mappings for different memory regions that correspond to different workloads to further improve the memory performance for the multi-programmed workloads. Figure 7.9 shows the overall speedup with memory partitioning for (a) 2-programmed and (b) 4-programmed workloads. Full list of the workload combinations are given in Table 7.4. Overall, we observe an increased performance with multiple partitions. In particular, 32 partitions can provide 8% and 9% performance improvements on average for 2-programmed and 4-programmed workloads.

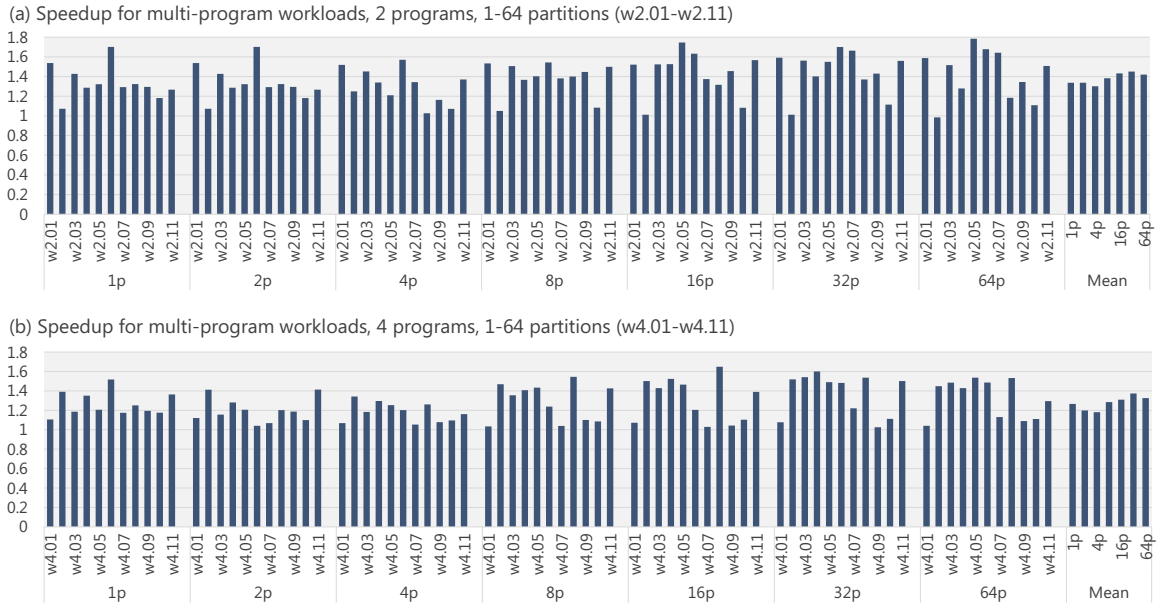


Figure 7.9: Multi-programmed workloads with partitioned memory space.

7.2.5 Host Access and Data Reorganization in Parallel

Until now we assumed the BoR (block on reorganization) mechanism for performing data reorganizations (see Section 6.2). In BoR, memory accesses coming from the host processor are blocked to minimize the interference and to avoid side issues such as coherence and memory access scheduling. BoR is a simple mechanism, however during the data reorganization the host processor can observe significant slowdown. Although this slowdown is amortized in the long runtime with the optimized memory access after the reorganization, for some latency critical applications this can be intolerable. To tackle this issue, AoR (access on reorganization) allows host accesses and the data reorganization accesses to be served in parallel. There are two main issues that arise in the AoR mechanism; (i) coherence between the host and DRU accesses, and (ii) efficient scheduling of the accesses coming from DRU and the host processor. In Section 6.2 we provide several mechanisms to address these issues. In this section, we will evaluate the tradeoffs and effectiveness of these mechanisms.

To support efficient AoR, memory controllers are extended with batch scheduler as discussed previously. The batch scheduler forms batches of accesses coming from the regular host queues and

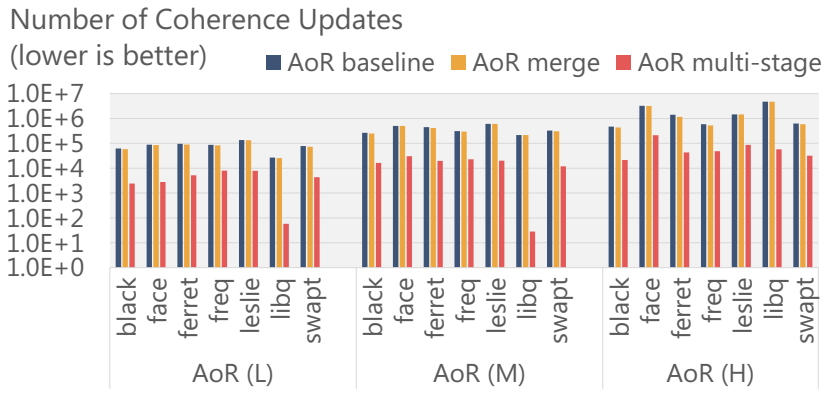


Figure 7.10: Improvements in the number of in-memory coherence updates via merging and multi-stage reorganization schemes for different host priority levels.

the data reorganization request queues. The sizes of these batches determine the interleaving granularity, hence the allocated priority for the sources. In our evaluations we use three batching configurations that will assign low, medium and high priority to the host requests. These three main configurations are labelled with AoR (L), AoR (M) and AoR (H). As we will see later, changing batch configuration provides a very flexible knob to allocate memory bandwidth to different sources (i.e. host and DRU).

First we evaluate the coherence issue for the AoR mechanism. During the AoR, where a source region is reorganized into a target region in the memory, the source serves for the host accesses until the entire region is reorganized. Then the source addresses are remapped such that they point to the target region by ARU. Data reorganization is an atomic operation in the sense that the reorganized data will be available to the host when the entire reorganization is finished. Hence, in the baseline scenario any write access to the region under reorganization will generate a coherence update in the memory to update the data both in the source and the target partitions until the reorganization is fully complete. These updates are handled by the coherence merge buffer (CMB) as discussed previously. CMB merges multiple requests going to the same address. It also drops its outstanding coherence updates, if a data reorganization request with same target address propagates from source to target carrying the latest value. Finally, multi-stage reorganization mechanism is provided which breaks the atomicity into smaller blocks to further alleviate the coherence updates during AoR. Details of these mechanisms are provided in Section 6.2. Figure 7.10 provides the analysis of these

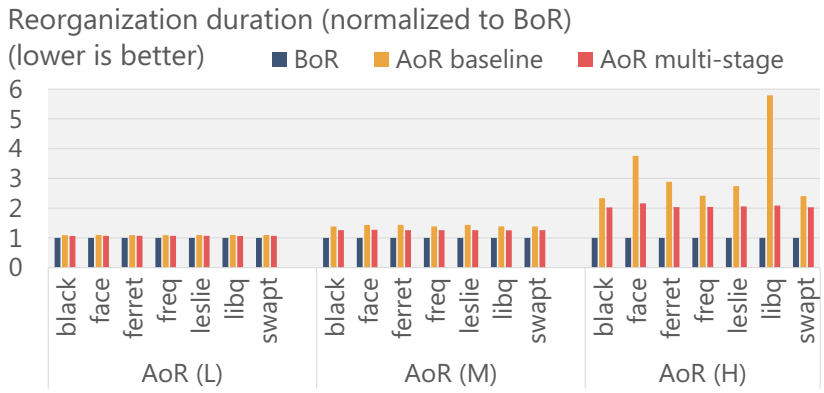


Figure 7.11: Reorganization duration comparison for BoR and AoR. Both baseline AoR and multi-stage reorganization mechanisms are provided.

techniques, namely *merge* and *multi-stage*, over the baseline case for the AoR. First we observe that, as the priority of the host accesses increase (i.e. larger batches for host requests) the total number of coherence updates increase. For the AoR (H), total number of coherence updates significantly larger than the AoR (L). This is mainly due to the fact that when the host accesses are prioritized reorganization takes longer to finish. Until the reorganization is completely finished, any host write access will generate a coherence update to the target region which increases the reorganization duration. Nevertheless, CMB provides a modest reduction in the number of coherence updates by merging and dropping coherence updates when possible. Furthermore, multi-stage reorganization breaks the atomicity into smaller blocks and handles the coherence updates very efficiently which leads to a substantial drop in the total number of coherence updates (see Figure 7.10).

Handling memory access scheduling and the interference are also major components to achieve efficient parallel host and DRU access. Figure 7.11 provides the overall duration of the data reorganization for the BoR and AoR schemes. BoR gives the entire DRAM resources to the DRU by blocking the host accesses. Hence BoR achieves the fastest data reorganization. When the host accesses have low priority in the AoR (L) mode, AoR can achieve a performance very close to the BoR. However, when the host has high priority, AoR scheme takes much longer than BoR due to both coherence updates and the memory interference. Coherence updates are significantly reduced with the multi-stage reorganization which also improves the performance of the AoR data reorganization, especially in AoR (H). Furthermore, batch scheduling minimizes the interference between

host and DRU by preserving their memory access patterns in batches. However, since DRAM bandwidth is shared between the host and the DRU, the reorganization takes longer compared to the BoR. Although it takes longer to reorganize, host accesses are serviced during the reorganization.

In order to provide more insights on BoR and AoR operation with various interleaving granularities, we focus on the details of the host and DRU bandwidth utilization during the data reorganization. Figure 7.12 and Figure 7.13 provide the bandwidth utilization for the benchmarks *blackscholes* and *facesim* during the baseline execution, BoR based, and AoR based data reorganization. Both BoR and AoR schemes start with a bandwidth utilization equal to the baseline execution. During the beginning of the execution memory access stream is monitored for a possible address remapping and data reorganization. This portion can be considered as a training phase. Then the hardware monitor issues a data reorganization. For BoR scheme the entire DRAM is allocated for data reorganization, which immediately drops the bandwidth utilization to zero as shown in instantaneous bandwidth utilization in Figure 7.12 and Figure 7.13. Then, when the data reorganization is finished, the memory controller starts servicing the host accesses. After the data reorganization, the host utilizes the bandwidth much more efficiently which amortizes the cost of the data reorganization quickly. As seen in average bandwidth utilizations both in Figure 7.12 and Figure 7.13, BoR bandwidth utilization drops significantly during reorganization but quickly after the reorganization it crosses the baseline bandwidth utilization.

One shortcoming of the BoR is the substantial drop in the host bandwidth during data reorganization. AoR allows host accesses to be serviced during the data reorganization. Figure 7.12 and Figure 7.13 provide three different priority levels (low, medium and high) for the host accesses during the data reorganization. When the host priority is high, i.e. AoR (H), the host does not experience a significant drop in the bandwidth utilization. However, in this case since the host accesses are prioritized, data reorganization takes much longer. During the AoR operation, both DRU and host utilize lower bandwidth compared to when they are running alone. This delays and limits the bandwidth utilization boost. Different priorities for host/reorganization accesses allow a flexible control to tradeoff between reorganization duration, host bandwidth drop and overall speedup. Next

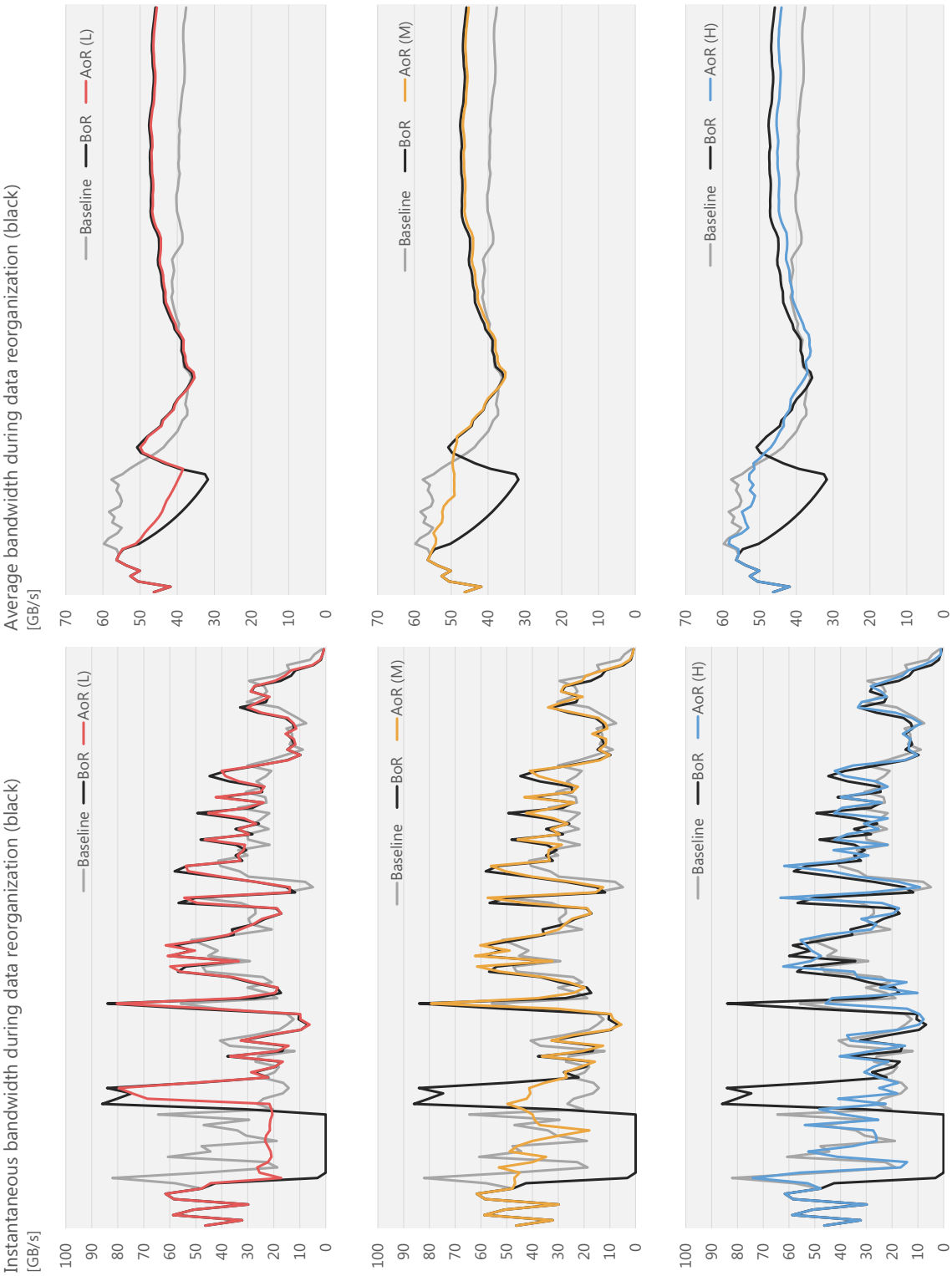


Figure 7.12: Instantaneous and average bandwidth utilization during baseline, BoR and AoR (low, medium and high host priority) for blackscholes.

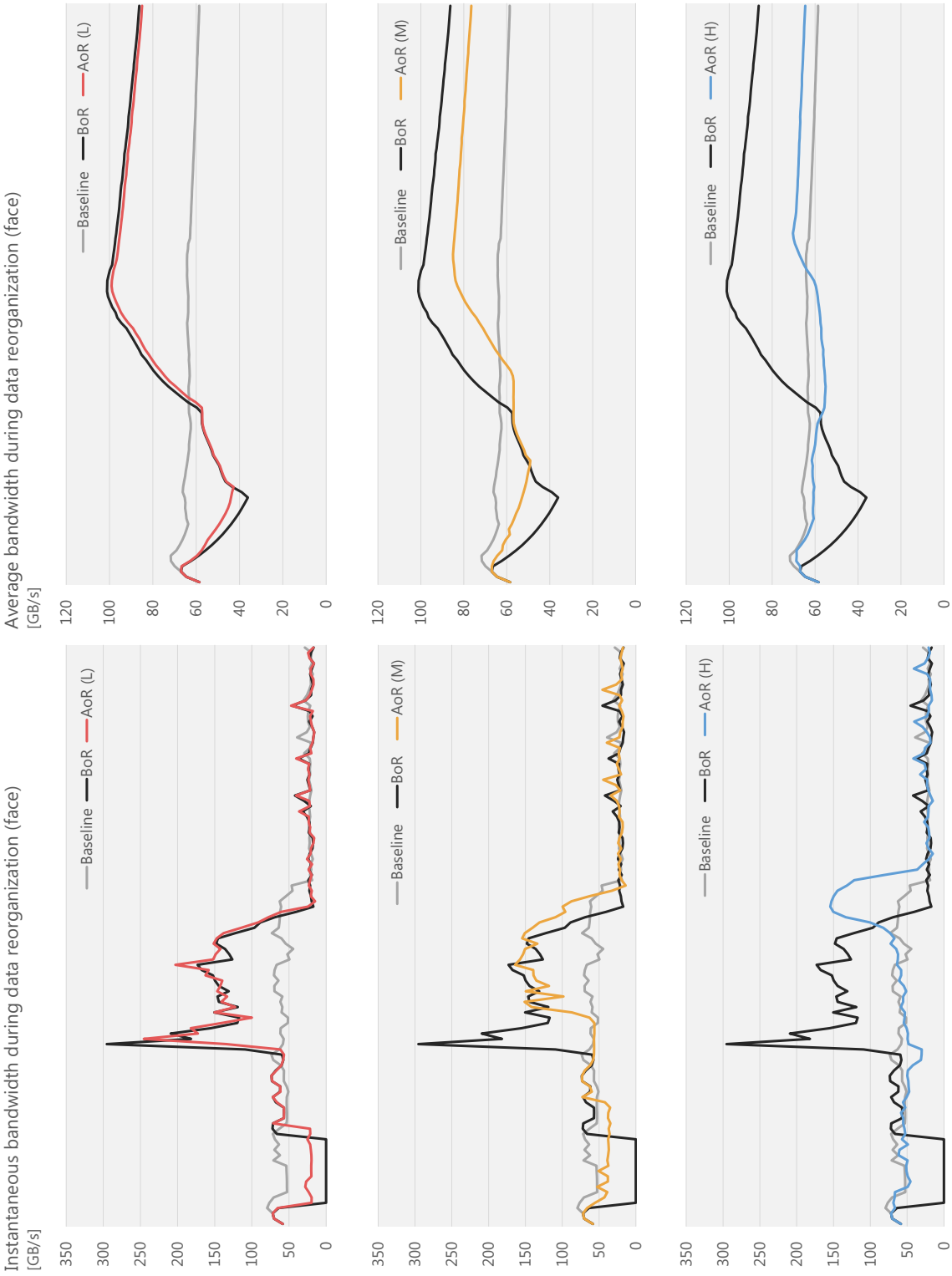


Figure 7.13: Instantaneous and average bandwidth utilization during baseline, BoR and AoR (low, medium and high host priority) for facesim.

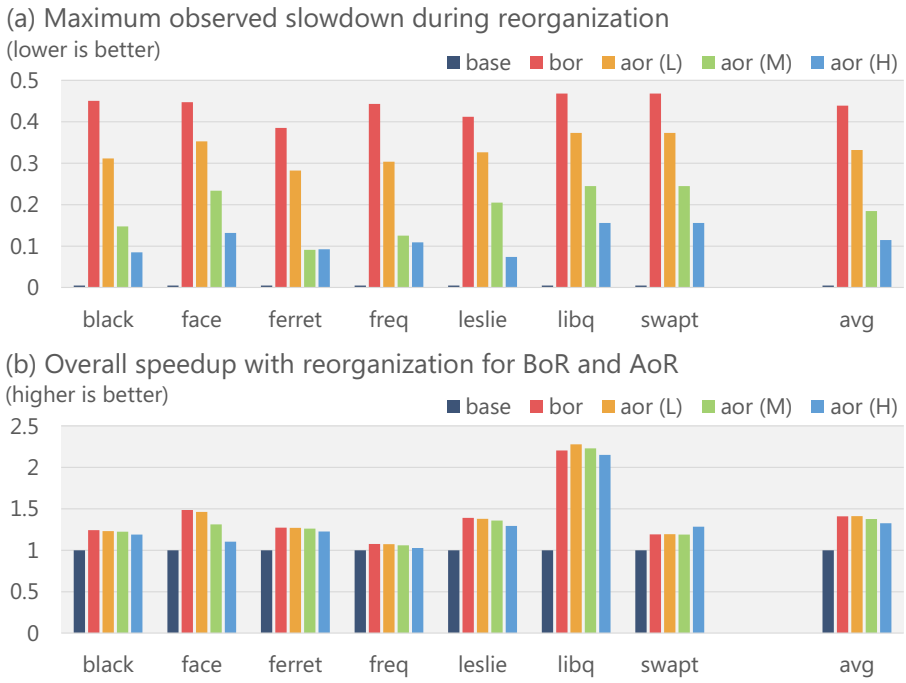


Figure 7.14: Maximum slowdown during reorganization, and the overall speedup for BoR and AoR mechanisms.

we will provide a detailed analysis regarding the effect of AoR on the maximum observed slowdown during the reorganization and the overall speedup.

Figure 7.14(a) provides the maximum slowdowns observed during the data reorganization phase for various workloads. Maximum slowdown is calculated as the drop in the average throughput during the data reorganization. BoR operation leads to the largest slowdown during data reorganization since the host accesses are blocked. On the other hand, AoR operation allows the host accesses to be serviced during the data reorganization which mitigates the host slowdown. As shown in Figure 7.14(a), BoR leads to 45% slowdown on average, whereas AoR improves this up to only 10% slowdown during the reorganization. AoR provides a minimum service guarantee (low, medium and high) to the host processor during the data reorganization. Reduced maximum slowdown during the reorganization implies an improved quality of service.

Overall speedup with the BoR and AoR approaches for these workloads are given in Figure 7.14(b). We observe that on average BoR and AoR (L) performs very close to each other, both around 41%.

Moreover, AoR (L) incurs only 33% maximum slowdown during reorganization whereas BoR has a 44% maximum slowdown. We also observe that AoR (H) improves the overall performance by 34% on average which is only 7% less than the BoR operation. However, AoR (H) causes only a 11% maximum slowdown during data reorganization as opposed to the 44% slowdown for BoR.

7.3 Explicit Mode

As evaluated previously, the automatic mode enables a software transparent technique to assist the host processor by reorganizing the data during the runtime that will result in more efficient memory access. 3D-stacked integration of the HAMLeT architecture allows a fast, energy-efficient and low-overhead data reorganization completely in memory. The HAMLeT architecture integrated in the 3D-stacked DRAM can be utilized to reorganize data in memory via an explicit offload from software. This section evaluates the performance of the explicitly accelerated data reorganization routines. It first focuses on common reorganization routines selected from Intel Math Kernel Library. Then it evaluates the overhead of software offload operation. Finally, it discusses the effectiveness the HAMLeT unit placed in memory and on chip.

7.3.1 Accelerating Common Data Reorganization Routines

First we focus on explicitly accelerated reorganization routines. The list of the reorganization benchmarks are given in Table 7.5. These are commonly used reorganization routines selected from the Intel Math Kernel Library (MKL) [5]. Benchmarks include in-place and out-of-place matrix transpose, vector pack/unpack via increment, scatter/gather indexing, and vector swap with varying stride. Dataset size for these benchmarks ranges from 4 MB to 1 GB. Stride of the accesses ranges from 1 to 8192 elements. Data precision is 32-bit float.

As a reference, we also report the high performance implementations on CPU and GPU systems. Multi-threaded implementations of MKL routines are compiled using Intel ICC version 14.0.3 and run on an Intel i7-4770K (Haswell) machine using all of the cores/threads. GPUs provide substantial

Table 7.5: Benchmark summary.

Number	MKL function	Description
101-108	simatcopy	In-place matrix transpose
201-210	somatcopy	Out-of-place matrix transpose
301-308	vs(un)packi	Vector (un)pack via increment
309-316	vs(un)packv	Vector (un)pack via gather
401-404	cblas_sswap	Vector swap via stride

memory bandwidth which is crucial for high performance data reorganization. As a GPU reference we use a modified version of the implementation from Nvidia [103] using CUDA 5.5 on a GTX 780 (Kepler) platform. However, due to the limited memory size of the GPU we could not run a few of the benchmarks with large dataset sizes (≥ 1 GB).

The benchmarks in Table 7.5 are expressed in SPL and implemented on HAMLeT where the DRU performs the data reorganization. Energy and runtime of the operations implemented on DRU is simulated using the 3D-stacked DRAM simulator. The 3D-stacked DRAM models are based on the conservative estimations from Table 7.2. Reported numbers also include the energy overhead of the HAMLeT implementation in the logic layer which will be analyzed later in detail. For the CPU, we use PAPI to measure the performance and power consumption of the processor as well as the DRAM via Running Average Power Limit (RAPL) interface [9, 17]. Finally, the GPU power consumption is measured on the actual board via inductive current probes using a PCI riser card. The results comparing the CPU, GPU and DRU performance in terms of throughput and energy efficiency are given in Figure 7.15 and Figure 7.16 respectively. Note that these results do not include the host offload overhead neither for GPU nor for DRU–here we report the results only for the individual platforms. We evaluate the host offload overhead for the DRU later in detail.

It is observed that the DRU integrated in the 3D-stacked DRAM can provide substantial performance and energy efficiency improvements when compared to the optimized implementations on the state-of-the-art CPUs and GPUs. DRU integrated in 3D-stacked DRAM benefits not only from the eliminated off-chip roundtrip data movement and the parallel streaming data reorganization architecture, but also from the high available bandwidth. Next, we provide further analysis to under-

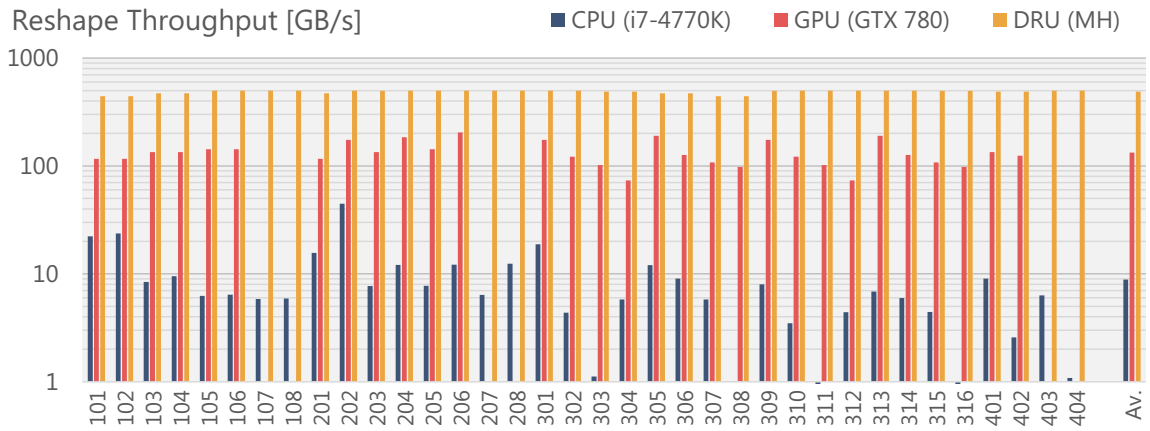


Figure 7.15: Performance of the 3D-stacked DRAM based DRU (HAMLeT) is compared to optimized implementations on CPU and GPU.

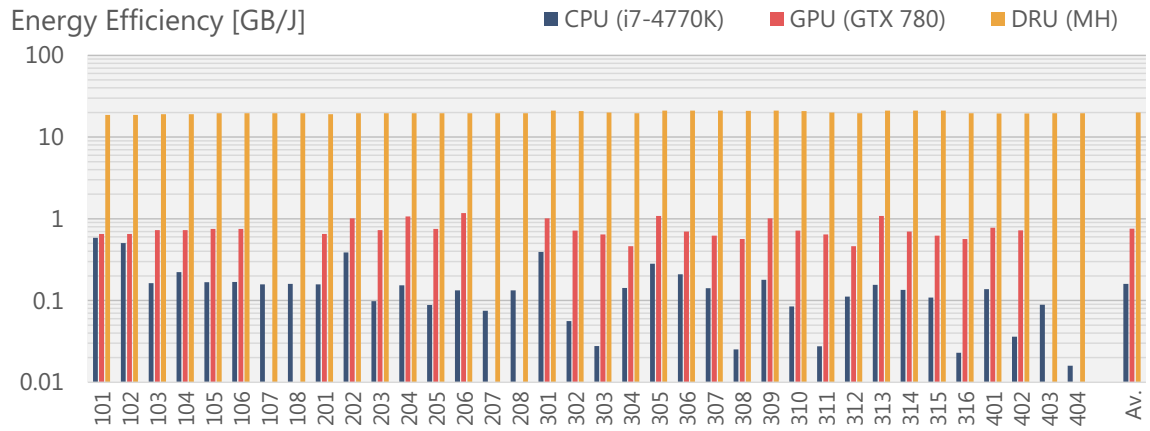


Figure 7.16: Energy efficiency of the 3D-stacked DRAM based is DRU (HAMLeT) compared to optimized implementations on CPU and GPU.

stand the actual contribution provided by eliminating the roundtrip data movement and the parallel DRU architecture.

Evaluations in Figure 7.15 and Figure 7.16 use the medium-high (MH) 3D-stacked DRAM configuration. The bandwidth provided by the MH configuration (i.e. 320 GB/s external, 520 GB/s internal) is much higher than what is available to the CPU and the GPU. We would like to evaluate the case where the same memory bandwidth is provided both to CPU and GPU.

We scale the DRU design down to the level of CPU and GPU individually. We provide individual comparisons where the bandwidth of the 3D-stacked DRAM is very close to each individual

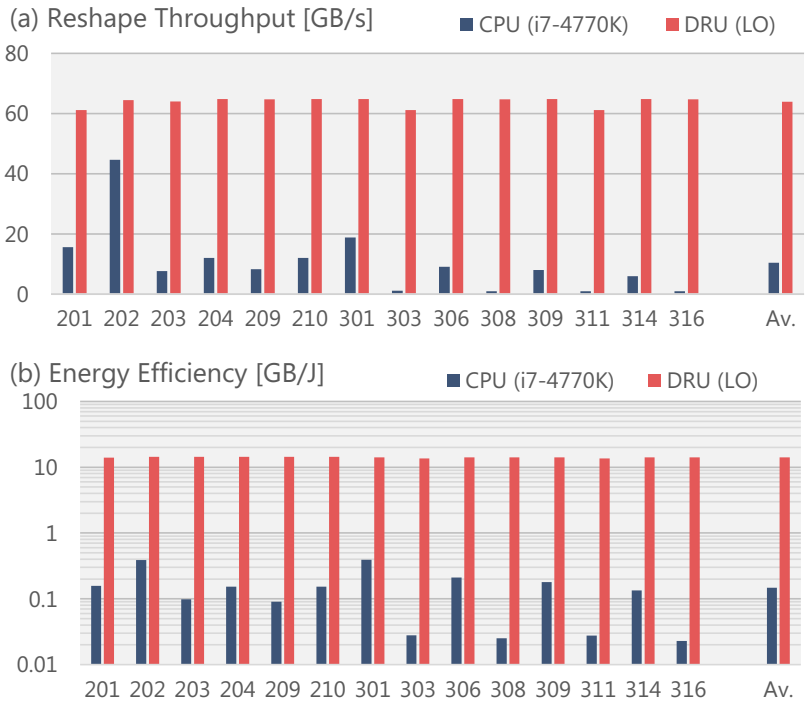


Figure 7.17: DRU (HAMLeT) in LO configuration is compared to the CPU.

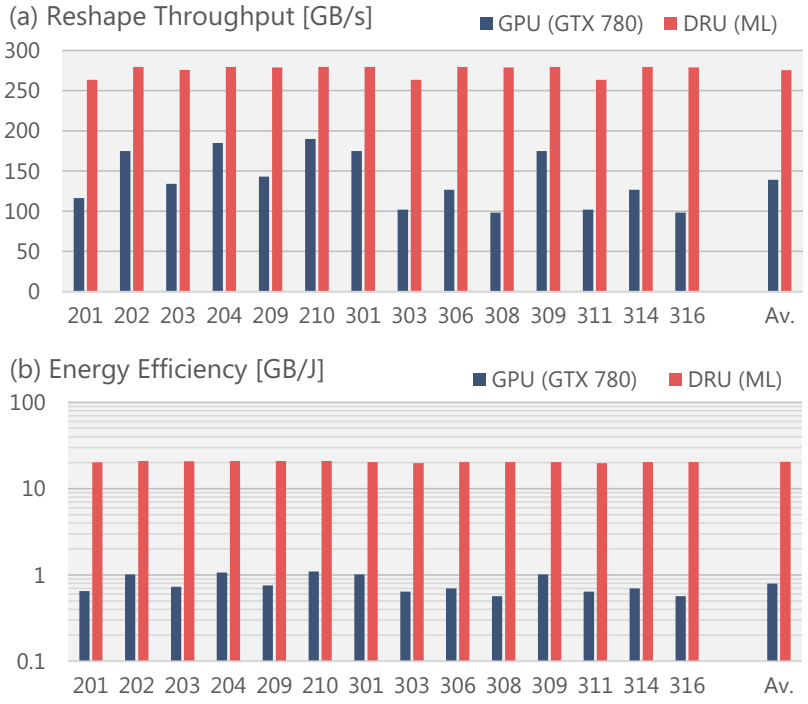


Figure 7.18: DRU (HAMLeT) in ML configuration is compared to the GPU.

platform's bandwidth. Figure 7.17 demonstrates the case where the DRU integrated in the LO configuration (single link, 40 GB/s external, 65 GB/s internal) is compared against the CPU (two channels, 25.6 GB/s). Also Figure 7.18 compares the DRU integrated in the ML configuration (7 links, 280 GB/s both internal and external) with the GPU (288 GB/s). Given the same level of bandwidth, in other words same achievable peak throughput, DRU provides 6.5x/2x better throughput compared to CPU/GPU. Furthermore, it provides 95x/25x better energy efficiency compared to CPU/GPU.

Note that we compare three platforms when executing only the data reorganizations. When the data is being reorganized, other programs running in CPU/GPU will observe substantial slowdowns. But DRU integrated in 3D-stacked DRAM enables a parallel data reorganization completely in memory to assist the host processor. Hence, it has a minimal slowdown effect on the other programs running in the host processor. Here, we emphasize that the 3D-stacked DRAM based DRU is not an alternative to CPU and GPU. Instead, it can be integrated into their memory subsystem to unlock substantially high-throughput and energy-efficient data reorganization capability.

3D-stacked DRAM technology provides a substantial bandwidth at a low power consumption. Naturally, DRU is dependent on these advantageous characteristics of the 3D-stacked DRAM. However, the uniqueness is how to exploit them. Pursuing a simple processing mechanism, data reorganization, with specialized parallel architecture allows us to achieve high throughput using a very low area and power consumption budget. This enables integration within the logic layer of a 3D-stacked DRAM which unlocks the substantial internal bandwidth and energy efficient data access capabilities. Furthermore, the dedicated parallel DRU architecture can utilize these capabilities very efficiently by operating at 94-99% of the streaming bandwidth with a very low overhead hardware as we will analyze later.

7.3.2 Near-memory vs. On-chip

Integration within the stack, behind the conventional interface, opens up the internal resources such as abundant bandwidth and parallelism. We also evaluate the effect of moving the accelerator within

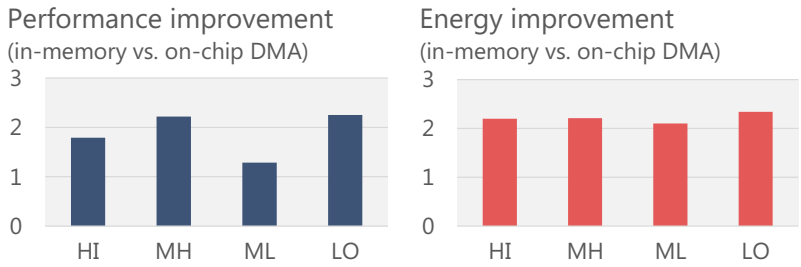


Figure 7.19: Comparison between the accelerator in-memory and on-chip as a memory controller based DMA.

the stack to exploit these resources. For this purpose, we compare the in-memory accelerator to a CPU side accelerator that is integrated on-chip as a DMA engine. When the accelerator is integrated on-chip to the memory controller as a DMA engine, it uses the limited off-chip bus facing large latency and energy overheads. For the in-memory integration, on the other hand, DRU accesses the memory directly from the logic layer bypassing the external interface. Figure 7.19 shows up to 2.2x throughput and 2.3x energy improvements for the in-memory accelerator compared to on-chip DMA accelerator.

7.3.3 Offload and Reconfiguration Overhead

As discussed in Section 6.3, a custom software stack for memory management handles offloading the processing to the HAMLeT where DRU is responsible for the data reorganization and ARU is responsible for address remapping. A baseline offload mechanism, such as the one proposed in [59], requires the transfer of DRU/ARU configurations to the corresponding memory region and a cache flush operation to ensure coherence. This software mechanism is proposed for generic near-data processing where the accelerator requires the most recent copy of the data for correct operation. As discussed previously in Section 6.3, for the data reorganization in memory, HAMLeT does not require the most recent copy to exist in the memory. It handles the coherence in the memory by using series of mechanisms such as multi-stage reorganization and coherence merge buffer (CMB). Furthermore, it allows incoherence between the host caches and the DRAM until the dirty cache lines from the reorganized region are written back in the DRAM. Both during and

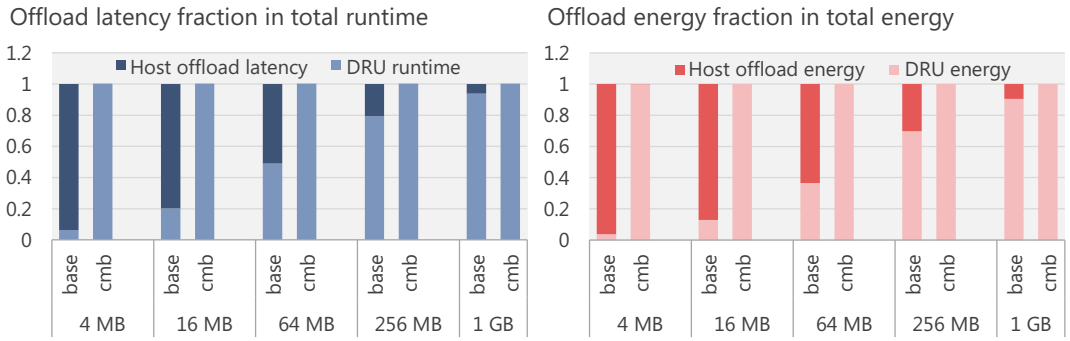


Figure 7.20: Offload latency and energy fraction in the overall operation for baseline and coherence optimized (using CMB) cases.

after the reorganization, CMB controller propagates the written back dirty cache lines to the correct addresses. Once all the dirty lines are written back, host caches and the DRAM become coherent again. For the write-through caches, on the other hand, this issue never arises.

To give a better insight about the overheads for the baseline offload mechanism and the coherence optimized offload mechanism that uses the CMB, we focus on time and energy spent on the host during the offload operation. For this experiment we focus on a subset of the benchmarks (101–210) with varying dataset sizes (4 MB to 1 GB) targeting the MH configuration from Table 7.2. Figure 7.20 provides the fraction of the latency and energy of the offload operation within the entire data reorganization both for baseline (base) and coherence optimized (cmb) mechanisms.

For the baseline mechanism, we observe that for small datasets (< 64MB) the offload operation takes much longer than the accelerated operation itself. For these small sizes, a large portion of the dataset is actually stored in the host side caches which makes them more suitable for host-side execution. Whereas, for the large datasets, the offload overhead is amortized with the long data reorganization. Hence these large datasets can mostly benefit from the high-throughput and energy-efficient processing. Nevertheless, flushing the caches constitutes a large portion of the overall offload operation for the baseline mechanism. As discussed previously, for the coherence optimized mechanism, this overhead is eliminated. Hence, even for small datasets the offload operation is very lightweight as shown in Figure 7.20.

7.4 Hardware Synthesis

HAMLeT architecture introduces ARU and DRU units in the logic layer of the 3D-stacked DRAM. These units increase the hardware complexity in the logic layer. In this section we provide a hardware synthesis based evaluation to analyze the area and power consumption cost of the introduced components. First, we focus on the reconfigurable permutation memory that is at the center of the DRU, performing the local permutations. Then, we analyze the overall area and power consumption overhead in the logic layer. This section demonstrates that with a small hardware overhead, HAMLeT can unlock substantial throughput and energy efficiency for data reorganization in memory.

7.4.1 Reconfigurable Permutation Memory

The reconfigurable permutation memory (RPM) is a generalized version of the permutation memories proposed in [99] which are used in Spiral. Spiral generated permutation memories are specifically optimized for the target permutation. During the hardware compilation, Spiral determines the optimal number of switch stages and the interconnection between them. Then it connects the optimized address generators to the switch network. This creates a highly efficient and compact hardware optimized for the specified problem. However, it generates a permutation memory hardware per problem. Instead, with the RPM the goal is to generate permutation memory configurations and program the hardware to perform the specified problem.

In order to support generalized streaming permutations, whose bit representations are affine transformations on the bits, RPM extend the streaming permutation memories proposed in [99] via two main approaches. First, it extends the switch stages through a rearrangeably non-blocking Clos network implemented as a multistage Benes network. Second, it introduces programmable switch connection and address generator units. When put together, RPM provides a reconfigurable substrate that can perform any permutation whose bit representation is an affine transformation. The maximum throughput and the maximum permutation size are limited by the target frequency, num-

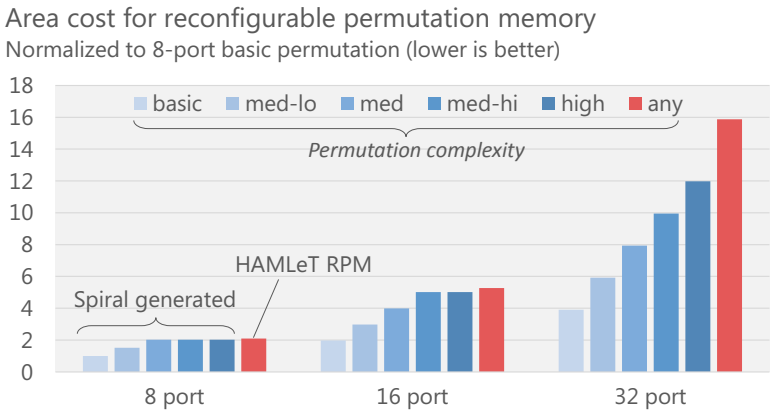


Figure 7.21: Switch network and connection control hardware complexity for Spiral generated and reconfigurable permutation memory.

ber of ports (streaming width), and total SRAM buffer size.

Figure 7.21 evaluates the hardware cost for the local permutation unit implemented as a Spiral generated permutation memory and reconfigurable permutation memory (RPM). Note that the SRAM buffer sizes are the same for both Spiral generated fixed permutation memory and the RPM. Hence, the hardware cost provided in Figure 7.21 only includes the switch network and connection control units. Figure 7.21 provides different parallelism (i.e. number of ports) and permutation complexities. Spiral optimizes the switch networks during the hardware compilation. Hence simple permutations lead to simple hardware as shown in Figure 7.21. As the permutation complexity increases, switch network and the connection control becomes more complicated which increases the hardware complexity. However, with a modest increase in the switch network and connection control hardware complexity, RPM can perform permutations with any complexity through a fixed hardware.

7.4.2 Overall System

We also present hardware cost analysis for the HAMLeT implemented in 32nm technology node. We synthesize the HDL implementation targeting a commercial 32nm standard cell library (typical corner, 0.9V) using Synopsys Design Compiler following a standard ASIC synthesis flow. We use

Table 7.6: HAMLeT power consumption overhead. HDL synthesis at 32nm.

	HI	MH	ML	LO
DMA	16.0 mW	11.9 mW	9.9 mW	8.9 mW
ARU	6.3 mW	3.5 mW	2.1 mW	1.4 mW
DRU (Switch + control)	22.4 mW	16.6 mW	11.2 mW	8.5 mW
DRU (SRAM)	291.9 mW	179.9 mW	89.9 mW	45.0 mW
HAMLeT TOTAL	336.5 mW	211.9 mW	113.1 mW	63.7 mW
3D-DRAM power envelope	45 W	30 W	25 W	4 W

CACTI [2], to model the SRAM blocks. Detailed ARU and DRU synthesis results for the example HAMLeT unit integrated in MH configuration demonstrate that they can reach 238 ps and 206 ps critical path delay (i.e. $>$ 4 GHz operation). However we chose 2 GHz clock frequency to reduce the power consumption. Overall hardware power consumption analysis results are given in Table 7.6.

Overall power consumption overhead ranges from 64 to 337 mW, including leakage and dynamic power. In Table 7.6 maximum power consumptions of the 3D-stacked DRAM configurations at their peak bandwidth utilizations are also given. Overhead coming from the HAMLeT corresponds to only a small fraction of the power envelope of these 3D-stacked DRAM configurations.

Chapter 8

Co-optimizing Compute and Memory Access

Discussions until here mainly focus on the efficient memory access only. As described in Chapter 5, automatic mode of operation focused on transparent data reorganization agnostic to the actual computation. Whereas, explicit mode transforms the data layout into an efficient format for an improved memory access. Both of these techniques exploit the separation of concerns concept where memory accesses and the actual computation are handled independently. However, for certain types of problems this separation is very difficult due to the interdependent nature of both entities. Achieving high-performance and energy-efficiency requires a co-optimization of the computation and the memory access. In this chapter, we will analyze an example for such type of problem, fast Fourier transform (FFT).

Due to the intrinsic properties of FFT, computation and memory accesses are tightly coupled to each other. There exists several alternative algorithms to restructure the the compute flow to exploit the parallelism, locality, regularity, or symmetry of computation stages where different alternatives exhibit varying memory access behavior [53]. Achieving the optimal performance requires an algorithmic co-optimization of the memory access and computation dataflow targeting an architectural

machine model. This chapter will demonstrate an optimization framework for large size FFTs focusing on the whole algorithm including memory access and computation. Tensor algebra based mathematical framework, which is particularly utilized in this thesis for expressing data reorganization, address mapping, memory access patterns and data layout can also represent FFT algorithms. Hence, we will use the proposed mathematical language as a common abstraction to capture the compute flow and memory access optimizations focusing on FFT algorithms. The key optimizations are centered around memory access and data layout transformation. In particular, adopting a block data layout and optimizing the dataflow of FFT algorithms for the block data layout enables reshaping inefficient strided memory accesses.

This chapter starts with describing single/multi dimensional FFT algorithms and an abstract machine model with a two-level memory hierarchy requiring block data transfers. Then 1D, 2D, and 3D FFT algorithms optimized for block data layouts are derived mathematically. The chapter than focuses on the automation capabilities for algorithm optimization and design space exploration. Finally, it analyzes HAMLeT architecture for explicitly performing the required data layout transformation that enables blocked data layout FFT algorithms.

8.1 Fast Fourier Transform

First we describe the fast Fourier transform (FFT) using the Kronecker product based formalism introduced in Chapter 3. Mathematically speaking, the discrete Fourier transform (DFT) of an n-element input vector corresponds to the matrix-vector multiplication $y = \mathrm{DFT}_n x$, where x and y are n point input and output vectors respectively, and

$$\mathrm{DFT}_n = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi i/n}.$$

Computation of the DFT by direct matrix-vector multiplication requires $\mathrm{O}(n^2)$ arithmetic operations. Well-known fast Fourier transform (FFT) algorithms reduce the operation count to $\mathrm{O}(n \log n)$. Using the Kronecker product formalism in [109], an FFT algorithm can be expressed as a factorization of

the dense DFT into product of structured sparse matrices. For example, the well-known Cooley-Tukey FFT [41] can be expressed as

$$\mathrm{DFT}_{nm} = (\mathrm{DFT}_n \otimes \mathrm{I}_m)\, \mathrm{D}_m^{nm} (\mathrm{I}_n \otimes \mathrm{DFT}_m)\, \mathrm{L}_n^{nm}. \tag{8.1}$$

In (8.1), L_n^{nm} represents a stride permutation, I_n is the $n \times n$ identity matrix, and $\otimes$ is the Kronecker or tensor product as we saw in Chapter 3. Finally, D_m^{nm} is a diagonal matrix of *twiddle factors*.

Multidimensional DFTs can also be considered as simple matrix-vector multiplications, such that $y = \mathrm{DFT}_{n\times n\times\cdots\times n}\, x$ where

$$\mathrm{DFT}_{n\times n\times\cdots\times n} = \mathrm{DFT}_n \otimes \mathrm{DFT}_n \otimes \ldots \otimes \mathrm{DFT}_n. \tag{8.2}$$

Similar to single dimensional DFT, multidimensional DFTs can be computed efficiently using multidimensional FFT algorithms. For example the well-known row-column algorithm for 2D-DFT can be expressed in tensor notation by using (8.2) and tensor identities [109] as

$$\mathrm{DFT}_{n\times n} = \left(\mathrm{L}_n^{n^2} (\mathrm{I}_n \otimes \mathrm{DFT}_n)\, \mathrm{L}_n^{n^2}\right)\left(\mathrm{I}_n \otimes \mathrm{DFT}_n\right). \tag{8.3}$$

The constructs in the formulas are performed from right to left on the input data set. Abstracting the input and output vectors as $n \times n$ arrays, firstly n point 1D-FFTs are applied to each of the n rows. Then, taking the results generated by the first stage as inputs, n point 1D-FFTs are applied to each of the n columns. The overall operation of (8.3) is demonstrated in Figure 8.1(a). Here, assuming a standard row-major data layout, the first stage (row FFTs) leads to sequential accesses in main memory, whereas the stride permutations ($\mathrm{L}_n^{n^2}$) in the second stage (column FFTs) correspond to stride-n accesses which is demonstrated in Figure 8.1(b).

Similarly, by using (8.2) and tensor identities in [109] the well-known 3D decomposition algorithm for 3D-DFT can be represented as, where $A^M = M^{\mathrm{T}} A M$,

$$\mathrm{DFT}_{n\times n\times n} = (\mathrm{I}_{n^2} \otimes \mathrm{DFT}_n)^{\mathrm{L}_{n^2}^{n^3}} . (\mathrm{I}_{n^2} \otimes \mathrm{DFT}_n)^{\mathrm{I}_n \otimes \mathrm{L}_n^{n^2}} . (\mathrm{I}_{n^2} \otimes \mathrm{DFT}_n)^{\mathrm{I}_{n^3}}. \tag{8.4}$$

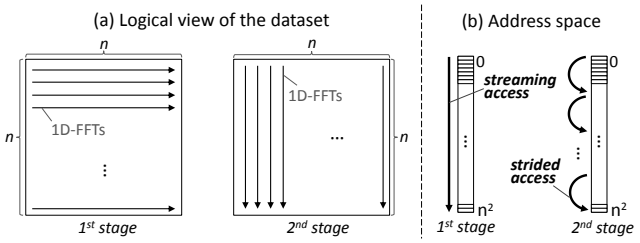


Figure 8.1: Overview of row-column 2D-FFT computation in (8.3).

The overall operation of (8.4) is demonstrated in Figure 8.2(a). If we assume a sequential data layout of the cube in x-y-z direction, first stage (FFTs in x) corresponds to sequential accesses to main memory however, due to the permutation matrices $I_n \otimes L^{n^2}_n$ and $L^{n^3}_{n^2}$, second and third stages (FFTs in y and z, respectively) require stride n and stride n^2 accesses respectively (as shown in 8.2(b)).

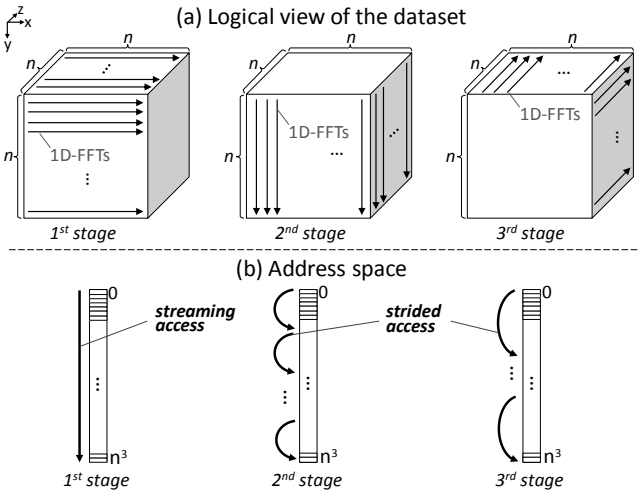


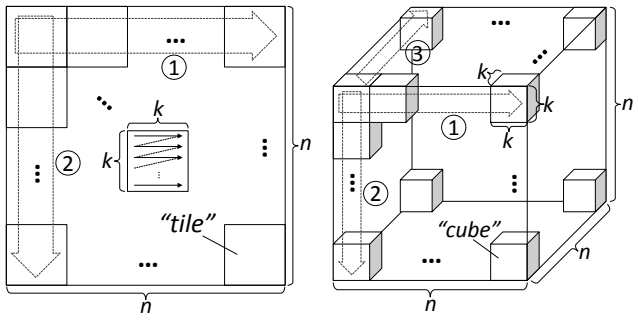
Figure 8.2: Overview of 3D-decomposed 3D-FFT computation in (8.4).

Until now we discussed decomposing large 2D and 3D-FFTs into small 1D-FFT computation stages that fit in the local memory considering the described machine model. A large 1D-FFT whose data set do not fit in local memory requires similar decomposition into smaller 1D-FFT kernels (e.g. Cooley-Tukey decomposition in (8.1)). In (8.1), we observe the same stride permutations as the row-column 2D-FFT algorithm. Hence, from a memory access pattern point of view, overall large size 1D-FFTs are handled very similar to the 2D-FFT computation (see Figure 8.1).

In summary, conventional large size FFT algorithms that use standard data layouts require strided accesses. These strided access patterns correspond to accessing different data blocks in the main memory. Continuously striding over data blocks does not allow amortizing the high latency cost of the main memory accesses, which yields very low data transfer bandwidth and high energy consumption. While there are FFT algorithms like the *vector recursion* [52] that ensure block transfers, they require impractically large local storage for data block sizes dictated by the main memory.

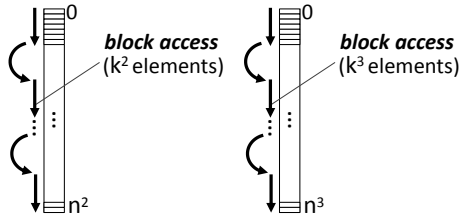
8.2 Machine Model

For optimizing the FFTs we target a high level abstract machine model that has three main components: (1) Main memory and data transfer, (2) local memory, and (3) compute. *Main memory*, represents the S_M-size large but slow storage medium (e.g. DRAM, disk, distributed memory, etc.) which is constructed from smaller S_B-size *data blocks* (e.g. DRAM rows, disk pages, MPI messages, etc.). Accessing an element from a data block within the main memory is generally associated with an initial high latency cost (A_M^{miss}). After the initial access, accessing consecutive elements from the same data block has substantially lower latency (A_M^{hit}) where $A_M^{\text{miss}} = A_M^{\text{hit}} + C$ and C is an overhead cost whose value depends on the particular platform. Hence the initial high latency cost of accessing a data block in the main memory can be best amortized by transferring the whole contiguous chunk of elements of the accessed data block. Algorithm design thus needs to make sure data is transferred in chunks of the native block size to minimize the overhead. In contrast, *local memory*, is S_L-size small buffer used for fast access to the local data (e.g. cache, scratchpad, local cluster node, etc.). We assume that local memory can hold multiple data blocks of the main memory i.e. $S_M > S_L > S_B$. Finally, *compute* represents the functional units that actually process the local data (e.g. vector unit, ALU, etc.). Various high-performance and parallel computing platforms ranging from embedded processors up to distributed supercomputers fit into this high-level machine model. FFT algorithms should be carefully fitted to these architectures to achieve high performance and power/energy efficiency.



(a) Tiled representation. (b) Cubic representation.

Figure 8.3: Logical view of the dataset for tiled and cubic representation.



(a) Tiled access (b) Cubic access

Figure 8.4: Block accesses are demonstrated in address space.

8.3 Block Data Layout FFTs

Strided accesses required by the conventional FFT algorithms lead to precisely the worst-case behavior in main memory (Figure 8.1, Figure 8.2). Changing the spatial locality of the memory accesses by adapting a customized block data layout in the main memory enables avoiding inefficient strided access patterns. By avoiding strided accesses and transferring large contiguous blocks of data one can amortize the latency of the main memory accesses and often also a large energy overhead. However, the overall dataflow of the FFT algorithm has to be restructured to map on the block data layout memory accesses. We focus on tiled and cubic data layout schemes. The resulting block data layout FFT algorithms will be transferring tiles or cubes of contiguous data blocks to and from the main memory in all stages of the computation.

Tiling is basically a block data layout, where n^2 element vectors are considered as $n \times n$ element matrices which are divided into $k \times k$ element small tiles (Figure 8.3(a)). Then the elements within tiles are mapped to physically contiguous locations in the main memory. When the tile size is selected to

match the data block size in the main memory, transferring a tile corresponds to transferring a whole contiguous data block from main memory. The access pattern is demonstrated in Figure 8.4.

As first introduced in [19], given the tiled data layout, one can compute the 2D-FFT while avoiding strided accesses. The main idea is instead of transferring stripe of elements in row and column direction as shown in Figure 8.1, transfer "tiles" in row and column direction (see ① and ② in Figure 8.3(a)).

From a memory access pattern perspective, 2-stage Cooley-Tukey algorithm for 1D-FFT has the same behavior as the row-column 2D-FFT algorithm as mentioned in Section 8.1. Hence the tiled data layout can be used in computing the 1D-FFT to avoid strided accesses.

Similarly in the cubic data layout, n^3 element data set is abstracted as a $n \times n \times n$ element three dimensional cube which is divided into $k \times k \times k$ element smaller cubes (see Figure 8.3(b)). Then each $k \times k \times k$ element small cube is physically mapped into a contiguous data block in main memory. Hence transferring a cube corresponds to transferring a whole data block from the main memory. The access pattern is demonstrated in Figure 8.4.

Cubic data layout enables 3D-FFT computation without strided accesses. Similar to the 2D-FFT scheme, the main idea is instead of transferring stripe of elements in x, y and z direction as demonstrated in Figure 8.2, transfer "cubes" in x, y and z direction (see ①, ② and ③ in Figure 8.3(b)).

A custom data layout comes along with address translation scheme that maps the logical addresses to the physical locations in the main memory. As discussed in Chapter 3, formal representation of a custom data layout scheme is given as $\mathrm{DFT} = ((\mathrm{DFT})^{\overleftarrow{Q}})^{\overrightarrow{P}}$ where $Q = P^{-1}$. Here $\overleftarrow{(.)}$ and $\overrightarrow{(.)}$ represent the address mapping and the data layout respectively. This representation only makes the data layout and the address mapping constructs explicitly labelled, the overall operation is still a natural DFT computation. Block data layouts and their address mapping schemes can be expressed in the proposed formal framework as we will see.

In summary, large size single and multidimensional FFTs when decomposed into smaller stages

require strided memory accesses. These FFTs can be performed while avoiding strided accesses by using block data layouts. (i) Remembering that when each tile (cube) is mapped to a data block in main memory, transferring tiles (cubes) in any order (i.e. ①, ② and ③ in Figure 8.3) corresponds to making use of the whole data blocks. (ii) Once tiles (cubes) are transferred to on-chip memory, they can be shuffled freely since on-chip memory does not incur any extra penalty depending on the access patterns. Hence, by combining the two properties (i) and (ii), one can perform the memory access pattern schemes required by multi-stage FFT algorithms efficiently. Note that these algorithms require full row or column of tiles (cubes) to be held simultaneously in the fast on-chip memory so that the 1D-FFTs can be done locally.

Although conceptually straightforward, capturing the details of the overall operation algorithmically is non-trivial. The discussion above omits the complexity of the data permutations and the details of the local computation. Formal representation in tensor notation allows capturing all the non-trivial details of the algorithms and the machine model in the same framework. Abstracting the algorithm and the machine model in the same framework allows detailed formula manipulations targeting the machine model, which is crucial to achieve high performance implementations.

8.4 Formally Restructured Algorithms

In our approach, we identify set of formula identities that restructure the given FFT formula so that it can be mapped to the data layout scheme and target machine model efficiently. The goals for restructuring the FFT are (i) to make sure that all the permutations that correspond to main memory accesses are restructured to transfer tiles/cubes, and (ii) to breakdown the formula constructs such that the local permutations and local 1D-FFT computations fit in the local memory.

Rewrite Rules. Rewrite rules are set of formula identities that capture the restructuring of the FFT algorithms. Rewrite rules only restructure the dataflow–the overall computation stays the same. We list the necessary formula identities used in restructuring the algorithms for the tiled data layout in

Table 8.1: Tiled mapping rewrite rules.

$$A \rightarrow (A^{\overleftarrow{Q}})^{\overrightarrow{P}}, \quad \text{where } Q = P^{-1} \tag{8.5}$$

$$AB \rightarrow A|B \tag{8.6}$$

$$\mathrm{I}_n \otimes A_n \rightarrow \mathrm{I}_{n^2}(\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{I}_k \otimes A_n})\,\mathrm{I}_{n^2} \tag{8.7}$$

$$A_n \otimes \mathrm{I}_n \rightarrow \mathrm{L}^{n^2}_n(\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{I}_k \otimes A_n})\,\mathrm{L}^{n^2}_n \tag{8.8}$$

$$\mathrm{I}_{n^2} \rightarrow \underbrace{(\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{L}^n_k \otimes \mathrm{I}_k})}_{R_{8.9b}} \underbrace{\overline{(\mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k)}}_{R_{8.9a}} \tag{8.9}$$

$$\mathrm{L}^{n^2}_n \rightarrow \underbrace{(\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{L}^{nk}_k})}_{R_{8.10b}} \underbrace{\overline{(\mathrm{L}^{n^2/k}_{n/k} \otimes \mathrm{I}_k)}}_{R_{8.10a}} \tag{8.10}$$

Table 8.1. The labels on the restructured formula constructs represent the implied functionality in the implementation. $\overleftarrow{(.)}$ and $\overrightarrow{(.)}$ represent the address translation and the data layout respectively. $|$ represents a memory fence. $\mathrm{I}_\ell \tilde{\otimes}$ corresponds to iteration operator. Finally, $\underline{(.)}$ and $\overline{(.)}$ correspond to the local kernel (permutation or computation) and the main memory access permutation respectively. These labelled formula constructs are restructured base cases, hence an FFT algorithm that consists only of these constructs considered to be final restructured algorithm. Next we will analyze the rewrite rules in more detail through an example.

Tiled 2D-FFT. We now apply the rewrite rules to a given 2D-DFT problem to obtain the restructured FFT. For $\mathrm{DFT}_{n\times n}$, we assume the n^2 element data set size (S_D) do not fit in the local memory, i.e. $S_M > S_D > S_L$, hence the large $\mathrm{DFT}_{n\times n}$ should be decomposed into smaller DFT_n stages as discussed. Further we assume that $k \times k$ tiles match the data block size S_B and local memory can hold a whole stripe of n/k tiles, i.e. $S_L \geq S_B \times n/k$. We now derive a tiled 2D-FFT targeting this machine model. We first demonstrate the algorithm derivation steps, then focus on the structure of the final derived algorithm.

As shown in Table 8.3, the starting point is $\mathrm{DFT}_{n\times n}$. First, as shown in (8.17), $\mathrm{DFT}_{n\times n}$ is expanded into smaller DFT stages by using (8.3). Then the DFT stages are separated via memory fence, as shown in (8.18), by using the rule (8.6). Next, rules (8.7)-(8.8) make the data permutations

Table 8.2: Cubic mapping rules. R_j refers to the right hand side of (j).

$$\mathrm{I}_n \otimes \mathrm{I}_n \otimes A_n \to \mathrm{I}_{n^3}\,(\mathrm{I}_{n^2/k^2} \,\tilde{\otimes}\, \underline{\mathrm{I}_{k^2} \otimes A_n})\,\mathrm{I}_{n^3} \tag{8.11}$$

$$\mathrm{I}_n \otimes A_n \otimes \mathrm{I}_n \to (\mathrm{I}_n \otimes \mathrm{L}^{n^2}_n)(\mathrm{I}_{n^2/k^2} \,\tilde{\otimes}\, \underline{\mathrm{I}_{k^2} \otimes A_n})(\mathrm{I}_n \otimes \mathrm{L}^{n^2}_n) \tag{8.12}$$

$$A_n \otimes \mathrm{I}_n \otimes \mathrm{I}_n \to \mathrm{L}^{n^3}_n\,(\mathrm{I}_{n^2/k^2} \,\tilde{\otimes}\, \underline{\mathrm{I}_{k^2} \otimes A_n})\,\mathrm{L}^{n^3}_{n^2} \tag{8.13}$$

$$\begin{aligned} \mathrm{I}_{n^3} \to & (\mathrm{I}_{n/k} \otimes \mathrm{L}^n_k \otimes \mathrm{I}_{nk})(\mathrm{I}_{n^2/k^2} \,\tilde{\otimes}\, \underline{\mathrm{L}^{nk}_{k^2} \otimes \mathrm{I}_k}) \\ & \overline{(\mathrm{I}_{n^2/k^2} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_{k^2})(\mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k)} = R_{8.14} \end{aligned} \tag{8.14}$$

$$\begin{aligned} \mathrm{I}_n \otimes \mathrm{L}^{n^2}_n \to & (\mathrm{I}_{n/k} \otimes \mathrm{L}^n_k \otimes \mathrm{I}_{nk})(\mathrm{I}_{n^2/k^2} \,\tilde{\otimes}\, \underline{\mathrm{L}^{nk^2}_{k^2}(\mathrm{I}_{n/k} \otimes \mathrm{L}^{k^2}_k \otimes \mathrm{I}_k)}) \\ & \overline{(\mathrm{I}_{n/k} \otimes \mathrm{L}^{n^2}_{n/k} \otimes \mathrm{I}_k)(\mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_{nk})} = R_{8.15} \end{aligned} \tag{8.15}$$

$$\begin{aligned} \mathrm{L}^{n^3}_{n^2} \to & (\mathrm{I}_{n/k} \otimes \mathrm{L}^n_k \otimes \mathrm{I}_{nk})(\mathrm{I}_{n^2/k^2} \,\tilde{\otimes}\, \underline{\mathrm{L}^{nk^2}_{k^2}}) \\ & \overline{(\mathrm{L}^{n^3/k}_{n^2/k^2} \otimes \mathrm{I}_k)(\mathrm{I}_n \otimes \mathrm{L}^n_k \otimes \mathrm{I}_n)} = R_{8.16} \end{aligned} \tag{8.16}$$

Table 8.3: Tiled 2D-FFT algorithm derivation steps. (R_9 and R_{10} are given in (9)-(10) in Table 8.1.)

$$\mathrm{DFT}_{n\times n} = (\mathrm{DFT}_n \otimes \mathrm{I}_n)(\mathrm{I}_n \otimes \mathrm{DFT}_n) \tag{8.17}$$

$$= (\mathrm{DFT}_n \otimes \mathrm{I}_n)|(\mathrm{I}_n \otimes \mathrm{DFT}_n) \tag{8.18}$$

$$= \underline{\mathrm{L}^{n^2}_n(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \mathrm{I}_k \otimes \mathrm{DFT}_n)}\,\mathrm{L}^{n^2}_n\,|\mathrm{I}_{n^2}(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})\,\mathrm{I}_{n^2} \tag{8.19}$$

$$\begin{aligned} = & \overline{(\mathrm{L}^{n^2/k}_n \otimes \mathrm{I}_k)}(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{L}^{nk}_n})(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{L}^{nk}_k})\overline{(\mathrm{L}^{n^2/k}_{n/k} \otimes \mathrm{I}_k)}\,| \\ & \overline{(\mathrm{I}_{n/k} \otimes \mathrm{L}^n_k \otimes \mathrm{I}_k)}(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{L}^n_{n/k} \otimes \mathrm{I}_k})(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{L}^n_k \otimes \mathrm{I}_k})\overline{(\mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k)} \end{aligned} \tag{8.20}$$

$$\begin{aligned} & = \Big(\big(R^{\mathrm{T}}_{8.10a} R^{\mathrm{T}}_{8.10b}(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})R_{8.10b}R_{8.10a}\,|R^{\mathrm{T}}_{8.9a}R^{\mathrm{T}}_{8.9b}(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})R_{8.9b}R_{8.9a}\big)^{\overleftarrow{Q}}\Big)^{\overrightarrow{P}}, \\ & \text{where } P = Q^{-1} = \mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k\,. \end{aligned} \tag{8.21}$$

Table 8.4: Tiled 1D-FFT algorithm derivation steps. (R_9 and R_{10} are given in (9)-(10) in Table 8.1.)

$$\mathrm{DFT}_{n^2} = (\mathrm{DFT}_n \otimes \mathrm{I}_n)\,\mathrm{D}^{n^2}_n\,(\mathrm{I}_n \otimes \mathrm{DFT}_n)\,\mathrm{L}^{n^2}_n \tag{8.22}$$

$$= (\mathrm{DFT}_n \otimes \mathrm{I}_n)|\mathrm{D}^{n^2}_n\,(\mathrm{I}_n \otimes \mathrm{DFT}_n)\,\mathrm{L}^{n^2}_n \tag{8.23}$$

$$= \mathrm{L}^{n^2}_n\,(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})\,\mathrm{L}^{n^2}_n\,|\mathrm{I}_{n^2}\,\mathrm{D}^{n^2}_n\,(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})\,\mathrm{L}^{n^2}_n \tag{8.24}$$

$$\begin{aligned} & = \Big(\big(R^{\mathrm{T}}_{10a}R^{\mathrm{T}}_{10b}(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})R_{10b}R_{10a}\,|\,R^{\mathrm{T}}_{9a}R^{\mathrm{T}}_{9b}\underline{\mathrm{D}^{n^2}_n}(\mathrm{I}_{n/k} \,\tilde{\otimes}\, \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n})R_{10b}R_{10a}\big)^{\overleftarrow{Q}}\Big)^{\overrightarrow{P}}, \\ & \text{where } P = Q^{-1} = \mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k\,. \end{aligned} \tag{8.25}$$

Table 8.5: Cubic 3D-FFT algorithm derivation steps. (R_{14}-R_{16} are given in (14)-(16) in Table 8.2.)

$$\mathrm{DFT}_{n\times n\times n} = (\mathrm{DFT}_n \otimes \mathrm{I}_n \otimes \mathrm{I}_n)(\mathrm{I}_n \otimes \mathrm{DFT}_n \otimes \mathrm{I}_n)(\mathrm{I}_n \otimes \mathrm{I}_n \otimes \mathrm{DFT}_n) \tag{8.26}$$

$$= (\mathrm{DFT}_n \otimes \mathrm{I}_n \otimes \mathrm{I}_n)|(\mathrm{I}_n \otimes \mathrm{DFT}_n \otimes \mathrm{I}_n)|(\mathrm{I}_n \otimes \mathrm{I}_n \otimes \mathrm{DFT}_n) \tag{8.27}$$

$$= \mathrm{L}^{n^3}_n (\mathrm{I}_{n^2/k} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n}) \mathrm{L}^{n^3}_n |(\mathrm{I}_n \otimes \mathrm{L}^{n^2}_n)(\mathrm{I}_{n^2/k} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})(\mathrm{I}_n \otimes \mathrm{L}^{n^2}_n)|\mathrm{I}_{n^3}(\mathrm{I}_{n^2/k} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})\mathrm{I}_{n^3}$$

$$= \Big(\big((\mathrm{I}_{n^2/k^2} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})^{R_{16}}|(\mathrm{I}_{n^2/k^2} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})^{R_{15}}|(\mathrm{I}_{n^2/k^2} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})^{R_{14}}\big)^{\overleftarrow{Q}}\Big)^{\overrightarrow{P}},$$

$$\text{where} \quad P = (\mathrm{I}_{n^2/k^2} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_{k^2})(\mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k) \tag{8.28}$$

explicit and label the kernel computation as shown in (8.19). Rules (8.9)-(8.10) restructure the data permutation so that they correspond to tile transfer operations (see (8.20)). Finally, in (8.21), the rule (8.5) defines the data layout and the corresponding address mapping. The result is given in (8.21). Note that rewrite rules only apply formula identities. Hence, the overall operation is still the same $\mathrm{DFT}_{n\times n}$ computation with a different dataflow. In (8.21), we observe that the local permutation and computation kernel size, $k \times n$ elements, fit in the local memory. However, derived algorithm is restricted to the problem sizes for which a whole stripe of tiles can be held simultaneously in the local memory so that the kernels can be processed locally (remember $S_L \geq S_B \times n/k$ where $S_B = k^2$ elements). Inspection of the (8.21) shows that all of the formula constructs are labelled base cases of the tiling rewrite rules, hence this formula corresponds to a final restructured algorithm.

We now analyze the final restructured algorithm by interpreting the Equation (8.21). Considering the DFT computation is a matrix multiplication, the constructs in the resulting algorithm (8.21) are performed from right to left on the input data set. First, $R_{8.9a}$ reads tiles, i.e. whole contiguous data blocks, from the main memory. Then, $R_{8.9b}$ shuffles the local data to natural order and then 1D-FFTs are applied to the local data. Finally, $R^{\mathrm{T}}_{8.9b}$ re-shuffles the local data after FFT processing and $R^{\mathrm{T}}_{8.9a}$ writes the local data back into the main memory as tiles. These operations are executed in a pipelined parallel way for the entire dataset which completes the first stage. The algorithm consists of two stages separated by a memory fence (|). Overall operation in the second stage is very similar to the first stage except the permutations. The second stage has the permutations $R_{8.10a-b}$ instead of $R_{8.9a-b}$ where $R_{8.9-8.10}$ are given in (8.9)-(8.10).

In addition to the tiled 2D-FFT algorithm, we use the set of formula identities (8.5)-(8.10) given in Table 8.1 to derive optimized tiled algorithms for large 1D-FFT (see (Table 8.4)). Further, we use formula identities (8.11)-(8.16) shown in Table 8.2 to derive 3D-FFT algorithms using cubic data layout (see (Table 8.5)). Similar to the 2D-FFT case, rewrite rules are applied to restructure the dataflow and the derivation steps are shown in Table 8.4 and 8.5. We provide the final optimized algorithms in (8.21), (8.28) and (8.25) for 2D, 3D and 1D respectively.

Until here we focused only on 1D, 2D and 3D FFTs using tiled and cubic data layouts, yet the mathematical framework can easily be extended to higher dimensional FFTs using higher dimensional hypercube data layouts.

8.5 Algorithm and Architecture Design Space

Formal representation in tensor notation allows capturing all the non-trivial details of the algorithms and the machine model in the same framework. Abstracting the algorithm and the machine model in the same framework allows detailed formula manipulations targeting the machine model. In this section we demonstrate a design space exploration methodology for an efficient FFT hardware accelerator implementation exploiting the formal framework.

8.5.1 Automated Design Generation via Spiral

We use Spiral [98] formula generation and optimization framework to automatically derive details of the block data layout FFT algorithms and generate hardware implementations. Spiral uses formal representation of FFT algorithms in tensor notation and restructures the algorithms using rewrite rules. We included the formula identities shown in Table 8.1 and Table 8.2 into Spiral's formula rewriting system so that it drives the optimized algorithms automatically (e.g. (8.21), (8.28) and (8.25)). Lastly, we build a custom backend that translates the hardware datapath formula to the full system described in Verilog.

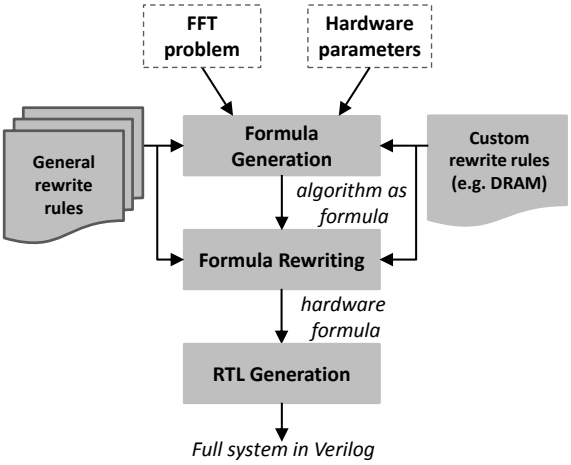


Figure 8.5: Overview of the design generator tool.

The overall view of our compilation flow is shown in Figure 8.5. First, the input FFT problem is expanded as a formula tagged with hardware parameters. At this point the formula is a high-level generic representation that does not have explicit implications for the hardware. Then this formula is restructured using our block data layout rewrite rules as well as selected Spiral default rules. After this step we reach a final structured formula where each formula construct is labelled with its targeted hardware module. Lastly, the custom backend generates the hardware components for the labelled formula constructs in Verilog targeting the parameterized architecture shown in Figure 8.6. In addition to the inferred hardware structures, the backend generates necessary wrappers, glue logic and configuration files that will interconnect all the modules and configure the full system.

8.5.2 Formula to Hardware

The architecture that we target (shown in Figure 8.6) is highly parameterized and scalable which can be configured for the given problem/platform parameters.

DRAM Controllers. In Figure 8.6, without the loss of generality, we provide a dual channel architecture featuring two DRAM controllers. Throughout the computation, one of the DRAM controllers is used for reading the inputs and the other one is for writing the outputs to DRAM in

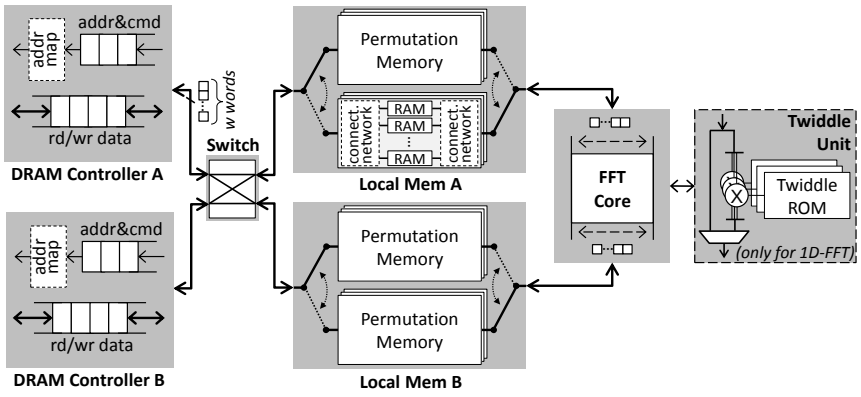


Figure 8.6: Overall view of the targeted architecture.

parallel. When a stage is completed, the two DRAM controllers switch their read/write roles for the next stage repeatedly until the computation is finished. Note that dual channel architecture is an abstraction of the actual underlying hardware, multiple DRAM channels can be bundled together or a single DRAM channel can be split into two to fit in this abstraction.

Depending on the optimizations that are used, a memory mapping scheme is represented in the generated hardware formula. The RTL generator translates this representation and configures the address mapping module in the DRAM controller (see Figure 8.6).

Local Memories. DRAM-optimized algorithms used in this work require holding multiple tiles (cubes) on-chip at once to apply data permutation and 1D-FFT on the local data. Local memories are local fast buffers constructed from SRAM and connection networks serving for that requirement. Considering that multiple elements arriving each cycle, these data shuffle operations become non-trivial to achieve by using minimal storage and by avoiding bank conflicts. Our tool first labels the formula constructs that correspond to these local data shuffle permutations. Then the formal representation of the permutation along with the hardware directives is used to automatically generate permutation blocks based on the techniques described in [82].

Local memories also construct the interface between the FFT core and the DRAM controller and allow the decoupling between the two. We employ double-buffering technique in the local memories. Hence, the computation and data transfer operations are overlapped to maximize the system

throughput.

FFT Core. We generate the streaming 1D-FFT core automatically using [83]. Control over the FFT core parameters (radix r, streaming width w) allows us to adjust the computation throughput according to the data transfer bandwidth to create balanced designs.

1D-FFT algorithm for large problem sizes requires a separate twiddle multiplication step when decomposed as discussed in Section 8.1. The FFT core is augmented by a separate twiddle factor multiplication unit for that case as shown in Figure 8.6.

8.5.3 Design Space Parameters

There are several ways of architecting a system to achieve a given goal of performance or power efficiency for a given FFT problem and hardware resources. Control over the adjustable design parameters allows walking within the space of various design possibilities. Some of the important design parameters can be categorized as follows:

- Throughput: FFT radix (r), streaming width (w), frequency (f).
- Bandwidth: Type of tiling (2D, 3D), tile size (T)
- Algorithm: Algorithmic choices provided by Spiral (A)

In addition to the adjustable design parameters, there are also given problem and platform constraints:

- Problem: FFT type (1D, 2D, 3D), size (n), precision (p)
- Platform: DRAM banks (b), row buffer size (R), max bandwidth (B), max power (P), max on-chip SRAM (S)

As we will see, exploring such a design space constructed by the given problem/platform constraints and design parameters is extremely difficult considering the relations between them and the costs

associated with adjusting every parameter. An automated design generation and exploration is essential to cover this wide design space and decide the most efficient parameters.

8.6 Experimental Results

8.6.1 Machine Model Based Evaluation

Algorithms derived in this work can be implemented in hardware or software on various platforms since we target a generic machine model. However, for machine model based evaluations we use hardware implementations on an Altera DE4 FPGA platform. DE4 FPGA platform comes with two channels of total 2 GB DDR2-800 DRAM which corresponds to the main memory considering the machine model described previously, so S_M = 2 GB. DRAM rows are the data blocks in the main memory and the DRAM row buffer size is 8 KB so S_B = 8 KB. Strided accesses to different data blocks (i.e. DRAM rows) yield 1.16 GB/s DRAM data transfer bandwidth whereas transferring contiguous DRAM row buffer size data chunks results in 11.87 GB/s bandwidth out of given 12.8 GB/s theoretical peak. We have measured the penalty of non-contiguous access to a different data block as approximately, $A_M^{\text{miss}} - A_M^{\text{hit}} = C$ = 20 clock cycles at 200 MHz. DE4 further provides 2.53 MB of on-chip SRAM which we consider as the local memory, hence S_L = 2.53 MB. Finally, floating point units correspond to the compute element in the machine model. For example, a single precision 4K×4K 2D-FFT has a total data set of S_D = 128 MB where n = 4096. Further we need 32×32 single precision complex valued element tiles to match the S_B = 8 KB, so k = 32. This overall configuration fits in the described machine model assumptions, i.e. $S_M > S_D > S_L \geq S_B \times n/k$.

Our evaluation results are summarized in Table 8.6. We report performance in Gflop/s, which is calculated as $5n\log_2(n)/t$ for DFT_n where t is the total runtime, thus higher is better. In Table 8.6, we provide (i) bandwidth bounded *theoretical peak* performance for Altera DE4 where we assume zero latency but limited bandwidth DRAM and infinitely fast on-chip processing (hence it is calculated as the total Flops divided by the time it takes to transfer the entire data at the peak DRAM bandwidth), (ii) actual results from *Spiral* generated block data layout implementations on Altera

Table 8.6: Performance results from Altera DE4. (GF = GFLOPS, TP = Theoretical peak)

FFT	Prec. [bits]	TP [GF]	Perf (% of TP) [GF]	Model (Error) [GF]
256×256	32	32	23.2 (72.6%)	23.1 (-0.4%)
512×512	32	36	29.2 (81.2%)	29.3 (+0.2%)
$1k \times 1k$	32	40	34.4 (86.0%)	34.7 (+0.9%)
$2k \times 2k$	32	44	38.3 (87.2%)	39.5 (+3.1%)
$4k \times 4k$	32	48	42.1 (87.7%)	43.8 (+4.2%)
256×256	64	16	12.4 (77.9%)	12.5 (+0.4%)
512×512	64	18	14.9 (83.2%)	15.1 (+0.8%)
$1k \times 1k$	64	20	17.1 (86.0%)	17.4 (+1.1%)
$2k \times 2k$	64	22	19.2 (87.4%)	19.5 (+1.4%)
$128 \times 128 \times 128$	32	28	23.4 (83.6%)	23.6 (+1.2%)
$256 \times 256 \times 256$	32	32	26.8 (84.0%)	27.2 (+1.3%)
$512 \times 512 \times 512$	32	36	30.3 (84.2%)	30.7 (+1.3%)
$64k$	32	32	23.3 (72.7%)	23.7 (+2.1%)
$256k$	32	36	29.2 (81.2%)	29.7 (+1.6%)
$1M$	32	40	34.4 (86.1%)	34.9 (+1.4%)
$4M$	32	44	38.4 (87.2%)	39.6 (+3.2%)
$16M$	32	48	42.1 (87.7%)	43.8 (+4.1%)

DE4, and finally (iii) a *realistic peak* performance estimated by our model where we include DRAM latency cost and bounded on-chip processing. Implementations generated by Spiral always transfer contiguous data blocks from main memory (i.e. whole DRAM rows). Optimized use of the DRAM row buffer leads to efficient DRAM bandwidth utilization, hence Spiral generated implementations reach on average 83% of the theoretical peak performance and 97.5% of the realistic peak performance.

Performance Model Verification. We will use the realistic peak performance estimations as our performance model in the design space explorations. Given the platform and FFT problem parameters, the model gives performance estimations. When we take the Altera DE4 as the target platform, the estimations for 1D, 2D and 3D FFTs are given in Table 8.6. When the performance estimations and the actual hardware results are compared, the error is found to be less than %4.25 for the designs that are generated and implemented on DE4.

8.6.2 Design Space Exploration

Experimental Setup. Before going into the details of design space exploration we first explain our experimental methodology. We use Spiral to generate designs for a given problem. Given the FFT problem and hardware parameters, our tool generates the HDL as well as simulation and synthesis scripts. Then, we synthesize the generated HDL targeting a commercial 32nm standard cell library using Synopsis Design Compiler following the standard ASIC synthesis flow. In addition to the standard ASIC synthesis flow, for non-HDL components, we use tools such as: CACTI 6.5 for on-chip RAMs and ROMs [2], McPAT for DRAM memory controllers [6], and DesignWare for single and double precision floating point units [3]. We target both off-chip DDR3-DRAM and 3D-stacked DRAM as main memory. For off-chip DRAM we use DRAMSim2 [102] and Micron Power Calculator [8] to estimate DRAM performance and power consumption. For the 3D-stacked DRAM model we use CACTI-3DD [38]. Lastly, for the overall performance estimation, we use a custom performance model backed up by cycle-accurate simulation. All of the tools are integrated resulting in an automatic push-button end-to-end design generation and exploration tool.

Exploration. First, we present an example design space for a selected 2D-FFT hardware implementation using 3D-stacked DRAM. In this scheme, a 2D-FFT accelerator hardware is stacked as a separate layer in the 3D-stacked DRAM. Given problem parameters are 8192×8192 point complex single-precision ($2 \times 32 - bits$ per complex word) 2D-FFT, and the platform parameters are 4 layer, 8 banks per layer, 512 TSVs, 8192 bit row buffer ($N_{\text{stack}} = 4$, $N_{\text{bank}} = 8$, $N_{\text{TSV}} = 512$, $R = 1\text{KB}$) fine-grain rank-level 3D-stacked DRAM. Further we set $S = 8\text{MB}$ and $P = 40\text{W}$.

Problem and platform parameters are the basic inputs to the generator. Spiral handles the algorithmic design choices (A) and prunes the algorithms which are determined to be suboptimal at the formula level. Then, for the given input configuration, it generates several hardware instances varying the design space parameters. A subset of the design space is shown in Figure 8.7 in terms of performance (GFLOPS) and power consumption (Watt) for various streaming width ($w = 2, 4, 8, 16$), frequency ($f = 0.4\text{GHz} \rightarrow 2\text{GHz}$), radix ($r = 2$) and tile size ($T = 0.125\times \rightarrow 2 \times \text{row buffer size}$) pa-

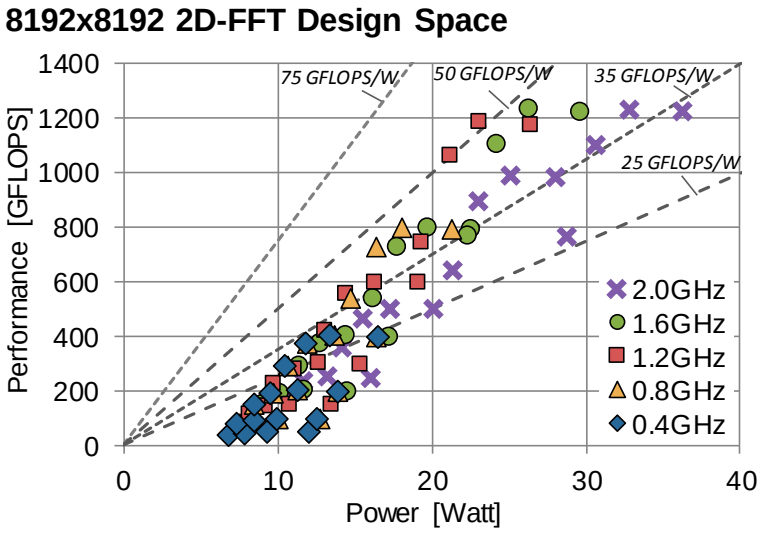


Figure 8.7: Design space exploration for 2D-FFT with 3D-stacked DRAM. Isolines represent the constant power efficiency in Gflops/W (see labels).

rameters.

Figure 8.7 also provides isolines for constant power efficiency, in other words achieved performance per consumed power (GFLOPS/Watt). We observe that there are multiple design points on the constant power efficiency lines, which suggests that the same power efficiency can be achieved with different parameter combinations. There are also several suboptimal designs that are behind the pareto frontier. Hence, it is not obvious which parameter combinations will yield the most efficient system at the design time. This highlights the complexity of the design space.

To further elaborate on the design space tradeoffs, we first illustrate the effects of the tile size in tiled FFT algorithms on the performance and power efficiency for different system configurations in Figure 8.8 (see Table 8.7 for memory configurations). (i) DRAM and (ii) on-chip resources are two main components of the overall system: (i) Increasing the tile size improves the spatial locality in DRAM accesses through efficient use of the row buffer. Efficient use of the row buffer leads to minimal activate/precharge operations which improve the bandwidth utilization and energy efficiency in DRAM, and consequently improve overall system performance and energy efficiency. (ii) On the other hand, from the on-chip resources viewpoint, local memory needs to be large enough to hold larger tiles which increases power consumption. Additionally, larger local memory needs to

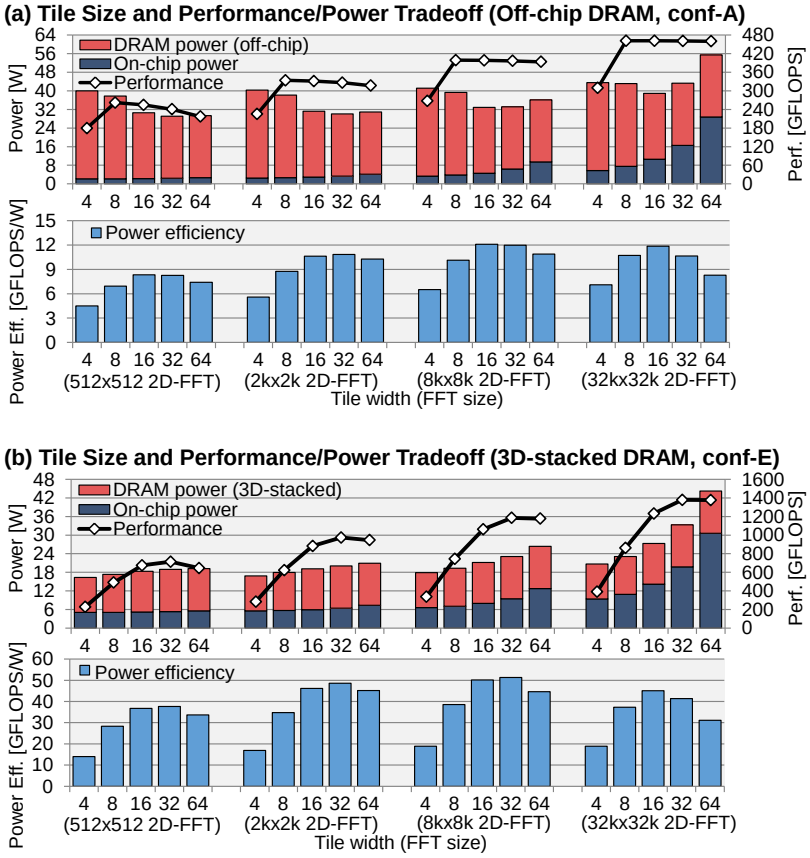


Figure 8.8: Effect of tile size on performance and power efficiency for off-chip and 3D-stacked DRAM systems. (Rest of the parameters are fixed.)

be filled and emptied in the beginning and at the end of the overall computation pipeline which decrease the performance. Conflicting tradeoffs in determining the tile size construct an optimization problem.

Moreover, different platform and problem configurations have different optimization curves. For example, power consumption of smaller problem size configurations are heavily dominated by the DRAM power consumption hence it is more desirable to improve the DRAM power consumption with larger tiles. Whereas larger problem size configurations prefer smaller tile sizes to save the on-chip power consumption (see Figure 8.8). Different platform configurations can also have different tradeoff relations. For example, BLP (bank-level parallelism) allows overlapping row buffer miss delays with data transfer on different banks. Therefore one can maximize the performance even if

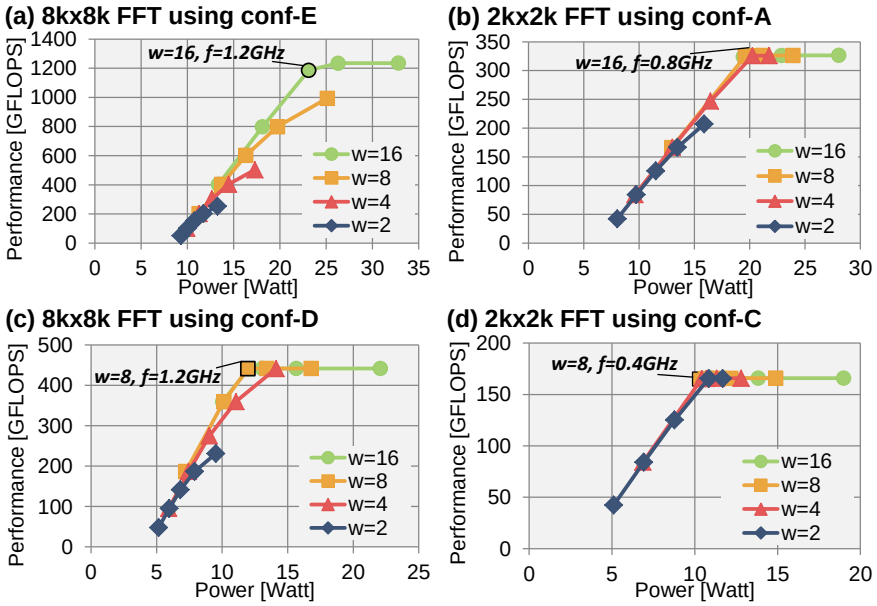


Figure 8.9: Frequency (f) and streaming width (w) effects on power and performance for various problem/platform configurations (fixed tile size). Parameter combinations for the best design (GFLOPS/W) are labelled.

the RBL is not fully utilized, with the extra energy cost of activate/precharge operations in DRAM (see Figure 8.8(a)). However, in 3D-stacked DRAM, banks within a rank contributes additively to the aggregate bandwidth exploiting the high bandwidth TSV bus as discussed previously. Therefore, unlike off-chip DRAMs, low RBL utilization (i.e. small tiles) reduces the performance significantly as shown in Figure 8.8(b).

Determining the memory subsystem configuration, particularly tile size in our case, is a key component in achieving high performance and energy efficiency, but computation configuration poses a great importance as well to achieve a balanced design. In a balanced design, computation throughput needs to match the estimated bandwidth utilization. One can pick different parameter combinations that will achieve the same throughput matching the given DRAM bandwidth. In Figure 8.9, given a fixed fixed tile size and platform/problem parameters (i.e. fixed DRAM bandwidth), we demonstrate design instances with various frequency and streaming width parameters. As highlighted in Figure 8.9 the most efficient parameter combinations show variations for different problem/platform configurations.

Table 8.7: Main memory configurations.

Name	Configuration (off-chip DRAM) Chan/Bank/R[Kb]/width/Type	Max BW [GB/s]
conf-A	8 / 8 / 64 / x8 / DDR3-1600	102.4
conf-B	8 / 8 / 64 / x8 / DDR3-1333	85.36
conf-C	8 / 8 / 64 / x8 / DDR3-800	51.2
Name	Configuration (3D-stacked DRAM) $N_{stack}/N_{bank}/N_{TSV}/R$[Kb]/Tech[nm]	Max BW [GB/s]
conf-D	4 / 8 / 256 / 8 / 32	178.2
conf-E	4 / 8 / 512 / 8 / 32	305.3
conf-F	4 / 8 / 512 / 8 / 45	246.7
conf-G	4 / 8 / 256 / 32 / 32	229.5

Finding the best system configuration given the problem/platform constraints establishes an optimization problem. Simply chasing the highest performance or lowest power consumption is not sufficient to get the most efficient system. As it is shown in Figure 8.8 and Figure 8.9, careful study of the design space is necessary to understand the tradeoffs. It is difficult to determine the crossover points where one of the dependent parameters becomes more favorable to the others and there is no structured way of finding the best parameter combinations. This highlights the importance of an automatic design generation and exploration system–it would be extremely difficult to complete such design exploration by hand.

Pareto-optimal Designs. We automatically generate pareto-optimal DRAM-optimized implementations and evaluate their performance and energy/power efficiency for the main memory as well as for the overall system. Our experiments include various main memory configurations which are shown in Table 8.7. Firstly, the overall system performance and power efficiency comparison of the naive baseline and the DRAM optimized implementations for 1D, 2D and 3D-FFTs of various sizes with regular off-chip DDR3 DRAMs and 3D-stacked DRAMs are demonstrated in Figure 8.10. DRAM-optimized implementations are generated by Spiral for the given problem and platform parameters. The same on-chip hardware resources and the same memory configurations are provided both for the baseline and the DRAM-optimized implementations. However, baseline implementa-

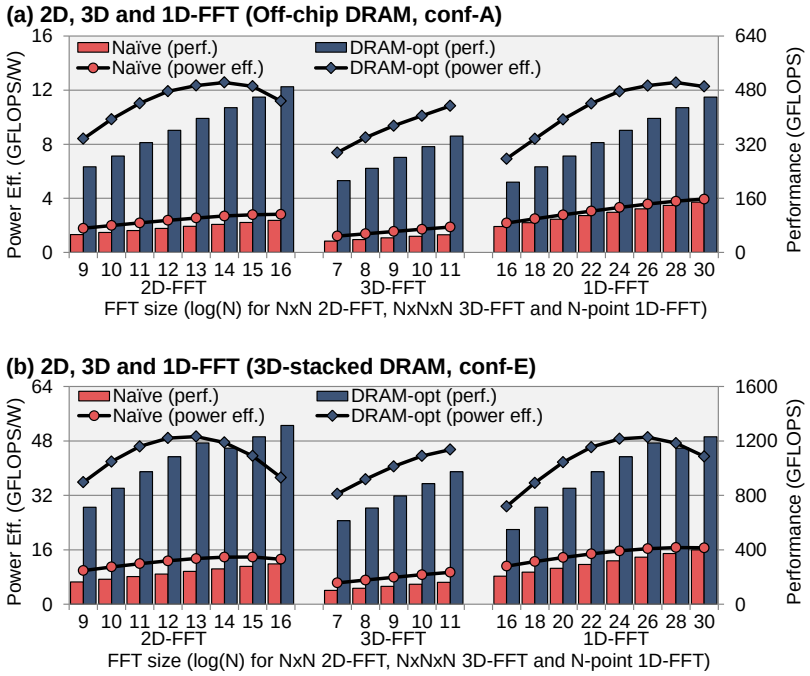


Figure 8.10: Overall system performance and power efficiency comparison between naive and DRAM-optimized implementations for 1D, 2D and 3D FFTs using memory configurations conf-A and conf-D respectively.

tions have naive unoptimized DRAM access patterns in contrast to DRAM friendly access patterns of the optimized implementations. Further, DRAM-optimized implementations are “pareto optimal" such that they are specifically fitted to the target platform for the best performance per power (i.e. GFLOPS/Watt) by the design generator tool. Although the best designs in terms of power efficiency vary depending on the problem and platform configurations as shown in Figure 8.10, generated DRAM-optimized designs can achieve up to 6.5x and 6x improvements in overall system performance and power efficiency respectively over naive baseline implementations. Also, to provide a point of reference, modern GPUs and CPUs can achieve only a few GFLOPS/W for the problem sizes that we are concerned with [19, 40]. For example, cuFFT 5.0 on the recent Nvidia K20X reaches approximately 2.2 GFLOPS/W where machine peak is 16.8 GFLOPS/W [15]. On the other hand, our DRAM-optimized hardware accelerators achieve up to nearly 50 GFLOPS/W (see Figure 8.10).

Improvements particularly in the main memory is the core of our framework. In Figure 8.11, pairs

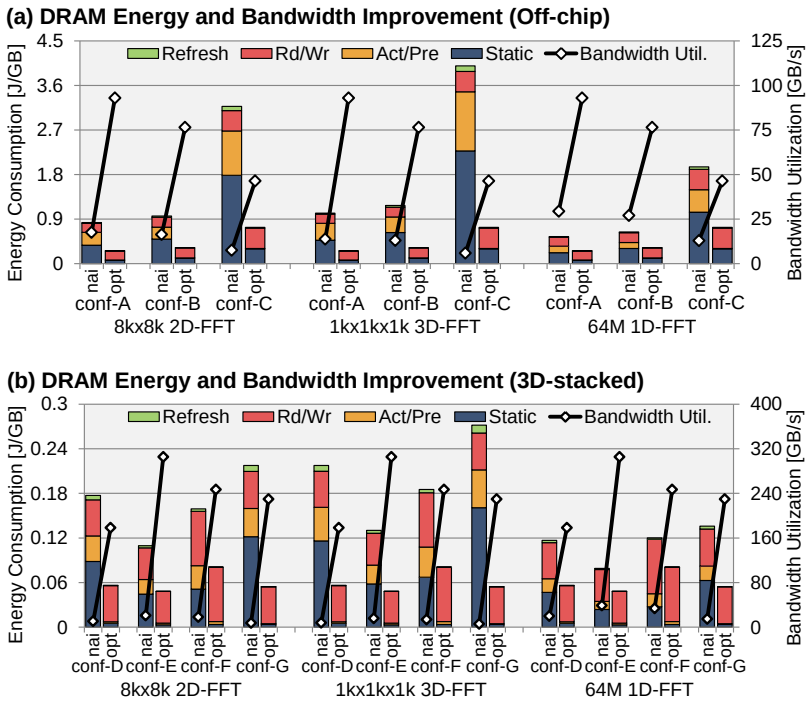


Figure 8.11: DRAM energy and bandwidth utilization for naive (nai) and DRAM-optimized (opt) implementations of selected FFTs and memory configurations.

of bars compare DRAM-optimized (opt) and naive baseline (nai) implementations for various memory configurations (see Table 8.7). The results show that the DRAM-optimized accelerators for 1D, 2D and 3D FFTs can achieve up to 7.9x higher bandwidth utilization and 5.5x lower energy consumption in main memory for off-chip DRAM, and up to 40x higher bandwidth utilization and 4.9x lower energy consumption in main memory for 3D-stacked DRAM. Due to their vulnerability to the strided access patterns, 3D-stacked DRAMs achieve better improvement through optimized access patterns. Another interesting point is that the generated 1D, 2D and 3D FFT designs have very similar efficient tiled access patterns which allow them to achieve a bandwidth and energy efficiency near theoretical maximum. Consequently, in Figure 8.11 we observe that optimized implementation for different FFTs with the same memory configuration reach almost the same DRAM bandwidth and energy consumption.

In Figure 8.11, the bars representing DRAM energy consumption are broken into segments as refresh, read/write, activate/precharge and static energy consumption. An important observation is

that the activate/precharge energy is significantly reduced, almost eliminated, compared to the baseline. Effective usage of DRAM row buffer in our DRAM-optimized implementation leads to very low row buffer miss rate which significantly reduces the total activate/precharge energy consumption. As expected, read/write energy stays the same since the same amount of data is read/written in both naive and optimized implementations. DRAM-optimized systems have higher bandwidth utilization which allows them to finish the data transfer quicker, saving total refresh and static energy consumption. There is also notable reduction in the refresh and static energy consumption. Therefore, in summary, effective usage of DRAM row buffer not only directly reduces the activate/precharge energy and improves achieved maximum bandwidth, but also reduces the total static and refresh energy by transferring the same amount of data faster.

8.6.3 Explicit Data Reorganization

Discussions and evaluations demonstrate best performance and energy efficiency can only be achieved by a co-design framework where memory access, compute dataflow and hardware datapath are co-optimized for a target architecture model. In this co-optimization framework, efficient memory access is an essential part where the block data layout is the key to unlock optimized implementations. Until here, we focused on the improvements that can be achieved with the optimized data layout. In this section, we will evaluate the overhead of such data layout transformation. In particular we will demonstrate efficient data layout transformations achieved by HAMLeT integrated in 3D-stacked DRAM.

Figure 8.12 and Figure 8.13 demonstrate the data reorganization overhead for transforming the row-major data layout into tiled and cubic data layouts for 2D and 3D FFTs respectively. Baseline implementations use row-column algorithms where the datapath is directly mapped onto the target 3D-stacked architecture model. Optimized implementations, on the other hand, are selected from pareto-optimal design points which employ a blocked data layout algorithm. Optimized implementation latency and energy consumption is broken down to demonstrate the overhead of the explicit data reorganization handled by HAMLeT in the logic layer. We observe that, especially for large

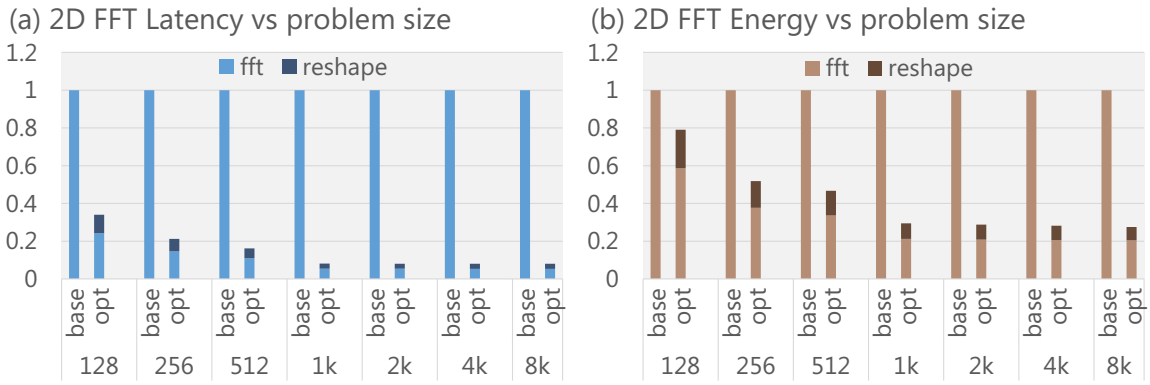


Figure 8.12: Overall latency and DRAM energy improvement with data reorganization for 2D FFT using HAMLeT in the logic layer.

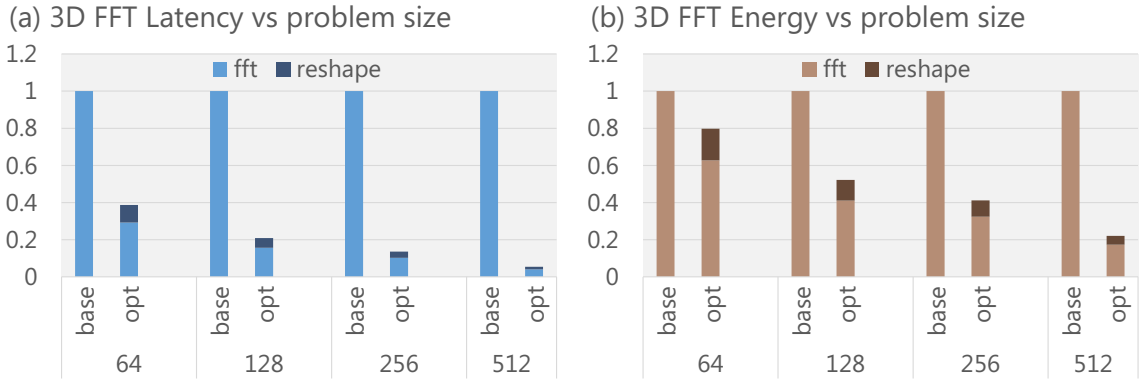


Figure 8.13: Overall latency and DRAM energy improvement with data reorganization for 3D FFT using HAMLeT in the logic layer.

problems both the energy and latency overhead of data reorganization is amortized with optimized memory accesses.

Chapter 9

Related Work

9.1 Planar Near Data Processing

Integrating processing elements near memory have been studied in the past under various technology contexts with different proximities between processing elements and the memory.

PIM and DIVA. Both PIM (processing in memory) [57] and DIVA (data intensive architecture) [48] make the observation that the data movement between the processing elements and the memory becomes a critical bottleneck. PIM architecture integrates a serial bit processor interfaced at the output path of an SRAM block. The processor executes simple ALU and data movement commands in SIMD fashion. DIVA builds upon the PIM idea to create a scalable system that features multiple PIM units. DIVA targets DRAM based memory, however its prototype is manufactured using SRAMs. DIVA includes caches, support for virtual address translation, pipelined control units as well as scalar/vector ALUs. This architecture allows it to target both simple and complex irregular applications ranging from matrix transpose to database query.

Active-pages, IRAM and FlexRAM. IRAM [93], FlexRAM [67] and active-pages [90] mainly focus on planar DRAM based near data computing. Berkeley IRAM (intelligent RAM) project seeks

a technology in which both the processor and the DRAM can be implemented together. The goal is to tackle the Von Neumann bottleneck via tightly integrated DRAM and processor on the same die. Vector IRAM (V-IRAM) combines DRAM with a vector processor integrated on the same chip targeting 0.13 μm technology [74]. FlexRAM [67], on the other hand, is mainly motivated by the general purpose computing near DRAM. Proposed architecture features processing arrays finely interleaved with DRAM macros. Overall system features both plain DRAM and NDP DRAM which are utilized based on the application. Finally, active-pages [90] features reconfigurable logic elements integrated near DRAM. An active Page consists of a page of data and a set of associated functions which can operate upon that data.

Main limitation of the planar NDP approaches are manufacturing costs and resulting performances. These approaches target implementing processing elements and the DRAM cells using the same technology. Due to different process technologies of DRAM and logic, it requires advanced fabrication technologies to integrate DRAM and logic on the same die. Nevertheless, the performance of the logic elements fabricated in DRAM technology will be much more limited compared to the conventional ones. On the other hand, DRAM implemented on-chip, such as embedded DRAM (eDRAM) [28], is limited with the memory capacity.

9.2 3D-stacking Based NDP

Recent 3D stacking technology integrates different process technologies of DRAM and custom logic [16, 65, 94] as separate dies using vertical TSV connection. This enables heterogeneous dies closely integrated where the logic elements are implemented highly efficient conventional transistor process technology and DRAM dies are implemented in the DRAM process technology. This technology gives a rise to the NDP concepts that has been proposed decades ago but suffered mainly from manufacturing difficulties, quality, and cost.

3D-stacking technology is an attractive option for variety of NDP architectures [20, 22, 23, 50, 59, 64, 71, 76, 88, 97, 114, 117, 120]. These architectures can be divided into two main categories

general purpose processing (e.g. [50, 64, 76, 88, 97, 114, 117]) and application-specific approaches (e.g. [20, 22, 23, 59, 120]).

General Purpose. General purpose processing is mainly achieved by programmable CPU [64, 76, 97, 114], vector processors [88] and GPU units [117]. There are also proposals that target reconfigurable logic integrated on top of DRAM via vertical TSVs [50]. These approaches require a separate layer to implement the area and power hungry programmable processors [50, 71, 76, 114]. On the other hand, area and power budget in the logic layer is extremely limited to implement general purpose processors [64, 88, 97, 117]

[117] reports that due to limited compute capability in 3D-stacked DRAM based NDP, workloads with high compute intensity and low memory intensity can even get slowdowns with NDP. Furthermore, [64] demonstrates that in order to utilize the available bandwidth provided by 3D-stacked DRAM, more than 200 PIM processing elements are required which exceeds the TDP power consumption constraint. To tackle the power consumption and thermal impact constraints [97] features low-EPI programmable processors architected as many-core. Further, to stay in the original power envelope it disables some of the SerDes links sacrificing the off-stack communication bandwidth.

Domain-Specific. Domain specific hardware is more effective in providing high throughput from the limited area and power budget. However, the scope of the domain-specific hardware is limited compared to programmable processors. There are several work integrating specialized hardware with 3D-stacked DRAM [20, 22, 23, 59, 120]. Most these approaches focus on separate accelerator dies for accelerating data-intensive applications including FFTs, synthetic aperture radar imaging and sparse matrix computations [20, 59, 120]. Separate die for the accelerator relaxes the area consumption constraints and minimizes the possible thermal hotspots. However, integration in the logic layer unlocks accessing the peak internal bandwidth [22, 23].

9.3 Software Based Data Layout Transformation

Compiler Based. There exists extensive compiler based program transformation frameworks aiming to optimize data locality [66, 80, 112, 113]. Frameworks focusing on loop transformations provide several methods to restructure the loop nests to exploit data locality and parallelism better [80, 112, 113]. Solely, loop transformation based optimizations are limited with the data dependencies. [66] extends the loop transformation (permutation, tiling) with data layout optimizations. Proposed algorithm transforms the loop nests and changes the memory layouts of multidimensional arrays in a unified framework where data layout transformation does not impose any data dependencies. A limitation of the static compiler based approach is that it does not capture the dynamic runtime information (ambiguous memory dependence problem).

GPU-centric. Furthermore, there exists GPU-centric data layout transformation frameworks [36, 108]. Memory coalescing, combining multiple memory accesses into a single transaction, is key for efficient processing with GPUs. Dymaxion and DL ([36, 108]) provide APIs for data index transformation mechanism that allows users to reorganize the layout of data structures such that they are optimized for localized, contiguous access. In this way they aim to coalesce thread accesses efficiently. Given a PCIe connected GPU with high bandwidth graphics memory on the device, this framework can overlap the data layout transformation with slow data transfer over PCIe in a software pipelined manner.

9.4 Hardware Assisted Data Reorganization

There are various proposals for hardware assisted data migration for better data placement [47, 100, 105, 107]. Main idea is to migrate or reorganize the data in the memory system to exploit the locality or tradeoffs between different memory technologies efficiently. [47, 100, 105] focuses on migrating data in a heterogeneous memory systems such PCM+DRAM or DRAM+3D-DRAM. But [107] focuses reorganization within memory to exploit the row buffer efficiently. These techniques

rely on the memory controller for hardware based memory access monitoring through counters. For example, [107] keeps counters for various pages which facilitates hashing to keep track of several pages. [105] keeps competing counters.

A key problem in hardware based migration is that it chagnes the physical location of the data which invalidates the current virtual to physical mapping in the page table or TLB. To handle this problem, these techniques either employ a hardware based indirection mechanism or update the page table entries via OS calls. Hardware based indirection is implemented via lookup tables that store the original and the remapped locations of the data. Address translation look-up table is not scalable to support large scale and fine grain data reorganizations. To increase the data migration granularity, [107] proposes micro-pages, i.e. smaller pages than the conventional OS pages. Furthermore, [100] keeps the lookup table small and updates the OS page tables periodically, or when the lookup table become full. This thesis focuses on permutation based data movement schemes where the address translation is calculated on-the-fly. This enables very fine-grain data movements (e.g. cache line level). Furthermore, it also features a lookup table based solution to enable multiple translations for different partitions in the memory. This creates a flexible hardware substrate that can handle fine-grain data reorganizations at a low-cost.

There exists specialized hardware solutions for data movement and reorganizations [22, 23, 45, 104, 119]. In [104, 119] authors focus on bulk copy and movement operations. [119] targets a DMA-like on-chip copy engine to accelerate the data movement in server platforms. On the other hand, [104] brings the bulk copy operations closer to the memory by modifying the internal structure of the DRAM. Moreover, [34] provides a hardware based indirection in the memory controller through address remapping. Address remapping without physical relocation can consolidate accesses via indirection but it does not solve the fundamental data placement problem. FPGA based data reorganiztion engines are studied in the previous work as well [45]. Hardware accelerators still suffer from memory latency, though data reorganization operations are ideally memory-to-memory such that the roundtrip data movement between the CPU and DRAM can be eliminated. This thesis, along with accompanying publications [22, 23], proposes data reorganization accelerator integrated

in 3D-stacked DRAM to minimize the roundtrip latency.

9.5 Kronecker Product Formalism for Hardware Design

The tensor (Kronecker) formula language used in this thesis has been previously used in various contexts to describe transform algorithms. These include FFT hardware algoirthms [83] and software based optimized implementations targeting multicores [54], SIMD vector units [79] and distributed memory architectures [37]. This tensor formula language is the basis for the SPL language and the Spiral project [98, 116].

From the hardware design perspective, [81, 82, 83, 99] demonstrates a compilation and optimization framework from mathematical representation to a hardware datapath. This framework takes high level hardware descriptions expressed as formulas, automatically generates an algorithm then maps the algorithm to a datapath. It uses the tensor based formula language aiming to capture the hardware datapath reuse.

Furthermore, [19, 20, 21, 24] uses the formal tensor framework to capture the memory accesses and data layout. [21, 24] extends the mathematical framework to efficiently optimize the memory access patterns by using block data layouts. [20] uses the formal framework to explore the tradeoffs in the design space of DRAM-optimized hardware implementations.

Specifically, [82, 99] demonstrate hardware implementations for streaming permutations facilitating the tensor based formal framework. However, this thesis considers the permutations as the basic building blocks for data reorganizations. The work presented in this thesis (as partly shown in [23]) is first to utilize the permutations and tensor formula language to manipulate data reorganizations for efficient DRAM access and in-memory hardware based operation.

Chapter 10

Conclusions and Future Directions

This thesis presents a formal framework that expresses key memory optimizations such as transforming data layout, reordering memory access pattern and changing address mapping through permutations. Permutations represented as matrices are systematically restructured to derive various implementation alternatives exploiting parallelism and locality in memory.

Driven by the implications from the formal framework, this thesis presents the HAMLeT architecture for highly-concurrent, energy-efficient and low-overhead data reorganization performed completely in memory. Memory access optimizations derived by the formal framework are directly mapped onto the HAMLeT architecture. HAMLeT pursues a near-data processing approach exploiting the 3D-stacked DRAM technology. Parallel streaming architecture can extract high throughput via simple modifications to the logic layer keeping the DRAM layers unchanged.

Enabled by the efficient dynamic data reorganization capability, this thesis demonstrates solutions for the conflict between memory access patterns and data layouts through several fundamental use cases. Data reorganization in memory provides an efficient substrate to improve memory system performance transparently. Explicit operation, on the other hand, gives an ability to offload and accelerate common data reorganization routines. Explicit data reorganization exposes the key memory characteristics to the algorithm design space and creates opportunities for algorithm/architecture

co-design.

10.1 Future Directions

The ideas presented in this thesis are preliminary and open up many new questions. There needs to be further investigation in the following directions:

10.1.1 Data Reorganization for Irregular Applications

We make the key observation that permutations can represent memory access optimizations such as transforming data layouts, changing address mapping or reordering the access patterns. Although the range of operations include variety of useful data reorganizations as demonstrated in this thesis, there needs to be more investigation for irregular applications. As demonstrated, dynamic data reorganization can improve the memory access performance for general purpose workloads. However, this improvement is much less when compared to explicitly optimized regular applications such as MKL routines or FFTs. There exists applications that exhibit certain levels of regularity despite having a complex dataflow. For example, stencil computation, convolutions and wavefront propagation type of workloads which suffer from inefficient memory system performance can be good candidates that worth further investigation.

10.1.2 General Purpose NDP

Data reorganization in-memory is a special case for near-data processing (NDP). This thesis provides hardware/software mechanisms to handle the system integration issues that relies on several features of the data reorganizations. Reorganization does not require the actual values, it only relocates the data to improve the memory performance when it is accessed. Hence it gives flexibilities to handle the coherence. Furthermore, address remapping for permutation based data reorganizations are handled in hardware which eases the virtual memory integration. However, general

purpose near-data processing requires a tighter integration to the host processors shared memory system.

Cache Coherence. First, NDP implementations needs to ensure that the operated data is a valid most recent copy. To achieve that, NDP can be treated as a cache coherent processor integrated in the system. Hence, the memory accesses are handled by the memory management unit (MMU) which requires directory access, page translations, coherence update/invalidations. Depending on the type of workloads this can create significant coherence traffic to flush the dirty copies of the data from on-chip caches to the memory. For a highly irregular dataset, this can involve several associative look-ups to clean up the cache. If the application's entire dataset is relatively small where a large portion fits in the caches, NDP can even degrade the overall performance due to coherence traffic.

In our analysis, we provided a software-issued cache flush mechanism as a baseline. This mechanism can be improved such that the operands of the NDP that exist in the caches are flushed if dirty and invalidated if valid. For a simple dataset this can improve the overall coherence traffic. There are other hybrid hardware/software approaches that should be investigated further to enable efficient coherent NDP.

Virtual Memory. Operating on large contiguous memory spaces can be handled relatively easily. Providing a large contiguously allocated physical memory space to the NDP through base address and total allocated size can be sufficient for regular workloads. NDP implementation can be treated as a memory mapped I/O device. NDP and host processor can communicate through a pre-allocated memory region. However, pointer chasing or indirect gather/scatter type of applications require a virtual to physical address translation very frequently. This can be achieved via hardware page walkers and TLBs implemented at NDP side. This also brings the issue of page-fault handling at NDP side. NDP can also be handled through an IOMMU mechanism. There are certain advantages and disadvantages of these approaches that needs to be researched.

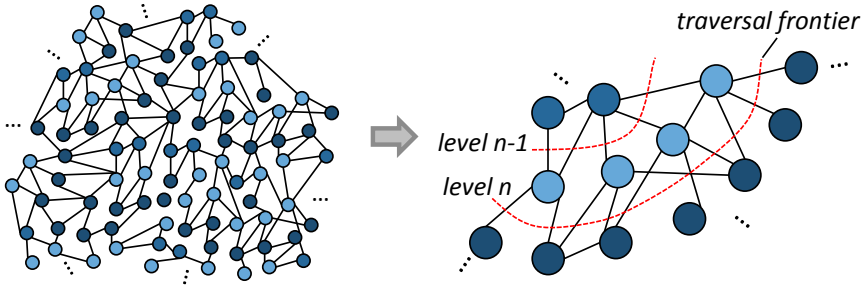


Figure 10.1: Graph traversal progress in breadth first search.

10.1.3 Graph Traversal in 3D-stacked DRAM

Graphs are fundamental data representations that have been used extensively in various domains. Exploring large graphs efficiently is a challenging task due to irregular memory access patterns, limited data reuse and long chain of pointer chasing issues. Cache-based memory hierarchies and prefetchers in modern processors cannot successfully address these problems. Furthermore, the traversal throughput is mainly limited by the memory bandwidth [73]. As a part of this thesis we also present a preliminary work on graph query operations via DRAM-optimized hardware accelerators within 3D-stacked DRAM. Abundant internal parallelism and high bandwidth within the 3D-stacked DRAM enables efficient, fast, and parallel traversal. Large number of nodes are traversed utilizing high internal bandwidth, only a fraction of the nodes are filtered and transferred via slow off-chip bus. Further, the roundtrip data movement between the memory and the processor due to the long chain of pointer chasing is minimized via in-memory traversal.

Analysis of Graph Query Operations. There is a great variety of graph applications but graph processing algorithms are mainly divided into two groups [75]: graph computation and graph queries. Graph computation involves vertex centric computation of the whole graph typically several iterations (e.g. belief propagation, matrix factorization, community detection). Graph queries, however, are reduction operations where only a fraction of the vertices and/or edges are filtered (e.g. search, shortest path, connected component). Hence graph query can be a potential candidate for near memory acceleration.

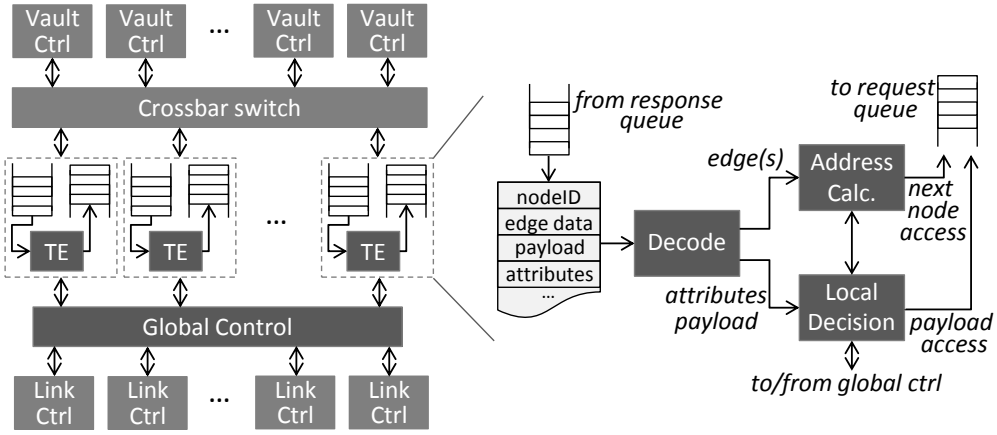


Figure 10.2: Architectural overview of the traversal engine.

For graph data structures, traversal is the fundamental operation that enables structured exploration and manipulation of the dataset. Graph traversal is the fundamental building block for various graph algorithms such as connected component analysis, A*, shortest path, centrality calculation, breadth first search [18, 95, 106]. Graph traversal efficiency, in terms of traversed edges per second (TEPS), is an important metric that is used to rank supercomputers [4]. Hence, graph traversal operations can be considered as the fundamental primitive for the hardware acceleration.

Architectural Design of a Traversal Engine. Graph traversal by nature consists of series of simple operations: fetch node(s), decode and determine the next node(s), request next node(s). As the traversal progresses, several child nodes are discovered (e.g. BFS) which allows an embarrassingly parallel search (see Figure 10.1). Therefore multiple branches of the graph can be traversed concurrently. Overall architecture includes multiple traversal engines (TE) to exploit the concurrency and increase the traversal throughput (see Figure 10.2).

Each traversal engine (TE) features queues for incoming nodes and outgoing requests (i.e. response and request queues). These queues are orchestrated by a global scheduling unit that can issue the operations to the TEs and to the DRAM layers to maximize the bandwidth utilization and traversal throughput. Data structure information is be offloaded to the TEs such that when a node is fetched, TEs determine the payload, attribute, and next node pointers. Each TE also features a decision logic

that checks if a predefined condition is met to stop or continue the traversal. The overall architecture is shown in Figure 10.2.

Our main goal is to utilize the internal bandwidth and parallelism of a 3D-stacked DRAM. To exploit the inter-vault parallelism of independent TSV buses, overall architecture features per-vault scheduling queues. Crossbar switch in the logic layer is utilized if a request in a queue will be scheduled to a different vault. Further the scheduling unit exploits the intra-vault parallelism of multiple vertically stacked banks through carefully scheduling the requests generated by multiple TEs to different layers. Since the locality of the requests are limited, various DRAM scheduling policies can be explored (e.g. closed page) to achieve the best performance and energy efficiency.

Graph Storage Format and Mapping to the Architecture. Storage format and layout of the graph is critical for traversal performance and energy efficiency. A potential storage format as discussed in [110] is to separate the structural information from the payload data. Structural information, or connectivity information, stores the node IDs, source/sink relations, some attributes and the pointers to the payload. Often the structural information, several hub nodes or hot sub-graphs of large graphs can be stored in-memory which enables in-memory graph exploration. Separating the structure information is also useful since graph queries are mostly structural such that they do not need payload access (e.g. count the number of reachable nodes to a certain depth, find the highly connected nodes, find the shortest path between two nodes etc.).

Although graph traversals exhibit limited spatial locality, the temporal locality can be exploited via specialized stacked memory hierarchies. We will explore mapping the frequently used structure information to SRAM, cache, and eDRAM based memory hierarchy options.

Future Work. As a first step, mapping efficient BFS, SSSP and CC algorithms to the developed parallel architecture with the specialized memory hierarchy can be evaluated through real world graph examples such as social networks, WWW, communication networks [10].

High performance and energy efficient exploration of the large graphs using DRAM-optimized hard-

ware accelerators within 3D-stacked DRAM will potentially create opportunities for in-memory traversal of in-memory graph databases.

Bibliography

[1] AMD HBM. https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf.

[2] CACTI 6.5, HP labs. http://www.hpl.hp.com/research/cacti/.

[3] DesignWare library, Synopsys. http://www.synopsys.com/dw.

[4] The graph500. http://www.graph500.org/.

[5] Intel math kernel library (MKL). http://software.intel.com/en-us/articles/intel-mkl/.

[6] McPAT 1.0, HP labs. http://www.hpl.hp.com/research/mcpat/.

[7] DDR3-1600 dram datasheet, MT41J256M4, Micron. http://www.micron.com/parts/dram/ddr3-sdram, .

[8] DDR3 sdram system-power calculator, Micron. http://www.micron.com/products/support/power-calc, .

[9] Performance application programming interface (PAPI). http://icl.cs.utk.edu/papi/.

[10] Stanford network analysis project (SNAP). http://snap.stanford.edu/.

[11] Tezzaron semiconductor, diram4 3d memory. http://www.tezzaron.com/products/diram4-3d-memory/.

[12] Gromacs. http://www.gromacs.org, 2008.

[13] Itrs interconnect working group, winter update. http://www.itrs.net/, Dec 2012.

[14] Memory scheduling championship (MSC). http://www.cs.utah.edu/ rajeev/jwac12/, 2012.

[15] CUDA toolkit 5.0 performance report, Nvidia. https://developer.nvidia.com/cuda-math-library, Jan 2013.

[16] High bandwidth memory (HBM) dram. JEDEC, JESD235, 2013.

[17] Intel 64 and ia-32 architectures software developers. http://www.intel.com/content/dam/www/public/us/en/documents /manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, October 2014.

[18] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010. ISBN 978-1-4244-7559-9.

[19] Berkin Akin, Peter A. Milder, Franz Franchetti, and James C. Hoe. Memory bandwidth efficient two-dimensional fast fourier transform algorithm and implementation for large problem sizes. In *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012, 29 April - 1 May 2012, Toronto, Ontario, Canada*, pages 188–191, 2012.

[20] Berkin Akin, Franz Franchetti, and James C. Hoe. Understanding the design space of dram-optimized hardware FFT accelerators. In *IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18-20, 2014*, pages 248–255, 2014.

[21] Berkin Akin, Franz Franchetti, and James C. Hoe. FFTS with near-optimal memory access through block data layouts. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014, Florence, Italy, May 4-9, 2014*, pages 3898–3902, 2014.

[22] Berkin Akin, James C. Hoe, and Franz Franchetti. Hamlet: Hardware accelerated memory

layout transform within 3d-stacked DRAM. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, pages 1–6, 2014.

[23] Berkin Akin, Franz Franchetti, and James C Hoe. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 131–143. ACM, 2015.

[24] Berkin Akin, Franz Franchetti, and James C. Hoe. Ffts with near-optimal memory access through block data layouts: Algorithm, architecture and design automation. *Journal of Signal Processing Systems*, 2015.

[25] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 2–9. IEEE, 2005.

[26] JEDEC Solid State Technology Association et al. Ddr4 sdram standard, jesd79-4a, 2013.

[27] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 176–186. IEEE, 1991.

[28] J. Barth. edram to the rescue. 2008.

[29] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 237–248. ACM, 2013.

[30] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Qingda Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, Feb 2005.

[31] Vaclav E Benes. *Mathematical theory of connecting networks and telephone traffic*, volume 17. Academic press New York, 1965.

[32] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[33] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[34] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: building a smarter memory controller. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 70–79, Jan 1999.

[35] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, S Pugsley, A Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. Usimm: the utah simulated memory module. 2012.

[36] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proc. of Intl. Conf. for High Perf. Comp., Networking, Storage and Analysis (SC)*, pages 13:1–13:11, 2011.

[37] Srinivas Chellappa, Franz Franchetti, and Markus Püeschel. Computer generation of fast fourier transforms for the cell broadband engine. In *Proceedings of the 23rd international conference on Supercomputing*, pages 26–35. ACM, 2009.

[38] Ke Chen, Sheng Li, N. Muralimanohar, Jung-Ho Ahn, J.B. Brockman, and N.P. Jouppi. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Design, Automation Test in Europe (DATE)*, pages 33–38, 2012.

[39] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102. ACM, 2013.

[40] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proc. of the 43th IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, 2010.

[41] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[42] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of parallel and distributed computing*, 22(3):462–478, 1994.

[43] Robert H Dennard, VL Rideout, E Bassous, and AR Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5): 256–268, 1974.

[44] Timothy O Dickson, Yong Liu, Sergey V Rylov, Bing Dang, Cornelia K Tsang, Paul S Andry, John F Bulzacchelli, Herschel A Ainspan, Xiaoxiong Gu, Lavanya Turlapati, et al. An 8x 10-gb/s source-synchronous i/o system based on high-density silicon carrier interconnects. *Solid-State Circuits, IEEE Journal of*, 47(4):884–896, 2012.

[45] Pedro C. Diniz and Joonseok Park. Data reorganization engines for the next generation of system-on-a-chip fpgas. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 237–244, 2002. ISBN 1-58113-452-5.

[46] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the*

2013 ACM SIGMOD International Conference on Management of Data, pages 1221–1230. ACM, 2013.

[47] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[48] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, et al. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing*, pages 14–25. ACM, 2002.

[49] Ronald G. Dreslinski, David Fick, Bharan Giridhar, Gyouho Kim, Sangwon Seo, Matthew Fojtik, Sudhir Satpathy, Yoonmyung Lee, Daeyeon Kim, Nurrachman Liu, Michael Wieckowski, Gregory Chen, Dennis Sylvester, David Blaauw, and Trevor Mudge. Centip3de: A many-core prototype exploring 3d integration and near-threshold computing. *Commun. ACM*, 56(11):97–104, November 2013. ISSN 0001-0782.

[50] A. Farmahini-Farahani, Jung Ho Ahn, K. Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 283–295, Feb 2015.

[51] Faith E Fich, J Ian Munro, and Patricio V Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

[52] Franz Franchetti and Markus Püschel. *Encyclopedia of Parallel Computing*, chapter Fast Fourier Transform. Springer, 2011.

[53] Franz Franchetti and Markus Püschel. *Encyclopedia of Parallel Computing*, chapter Fast Fourier Transform. Springer, 2011.

[54] Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[55] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Platform Adaptation"*, 93(2):216–231, 2005.

[56] Sameh Galal and Mark Horowitz. Energy-efficient floating-point unit design. *Computers, IEEE Transactions on*, 60(7):913–922, 2011.

[57] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 28(4):23–31, Apr 1995.

[58] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008. ISSN 0098-3500.

[59] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C. Hoe, and Franz Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design. In *In the Workshop on Near-Data Processing (WoNDP) (Held in conjunction with MICRO-47.)*, 2014.

[60] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G Friedman. Ac-dimm: associative computing with stt-mram. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 189–200. ACM, 2013.

[61] Fred Gustavson, Lars Karlsson, and Bo Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38 (3):17, 2012.

[62] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding

sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 37–47. ACM, 2010.

[63] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[64] M. Islam, M. Scrback, K.M. Kavi, M. Ignatowski, and N. Jayasena. Improving node-level map-reduce performance using processing-in-memory technologies. In *7th Workshop on UnConventional High Performance Computing held in conjunction with the EuroPar 2014*, UCHPC2014, 2014.

[65] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88, June 2012.

[66] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 285–297, 1998.

[67] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. Flexram: Toward an advanced intelligent memory system. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 5–14. IEEE, 2012.

[68] Stephen W. Keckler. Life after dennard and how i learned to love the picojoule. In *Keynote at the 44th Intl. Symposium on Microarchitecture*, 2011.

[69] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011. ISSN 0272-1732.

[70] G. Kestor, R. Gioiosa, D.J. Kerbyson, and A. Hoisie. Quantifying the energy cost of data movement in scientific applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 56–65, Sept 2013. doi: 10.1109/IISWC.2013.6704670.

[71] Dae Hyun Kim, K. Athikulwongse, M. Healy, M. Hossain, Moongon Jung, I. Khorosh, G. Kumar, Young-Joon Lee, D. Lewis, Tzu-Wei Lin, Chang Liu, S. Panth, M. Pathak, Minzhen Ren, Guanhao Shen, Taigon Song, Dong Hyuk Woo, Xin Zhao, Joungho Kim, Ho Choi, G. Loh, Hsien-Hsin Lee, and Sung-Kyu Lim. 3d-maps: 3d massively parallel processor with stacked memory. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 188–190, Feb 2012.

[72] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 368–379. IEEE, 2012.

[73] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, 2013. ISBN 978-1-4503-2638-4.

[74] Christoforos E Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, et al. Scalable processors in the billion-transistor era: Iram. *Computer*, 30(9):75–78, 1997.

[75] Aapo Kyrola. *Large-scale Graph Computation on Just a PC*. PhD Dissertation, CS, CMU, 2014.

[76] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proc. of the 35th Annual International Symposium on Computer Architecture, (ISCA)*, pages 453–464, 2008.

[77] Krishna T Malladi, Frank A Nothaft, Karthika Periyathambi, Benjamin C Lee, Christos Kozyrakis, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*.

[78] Mozhgan Mansuri, James E Jaussi, Joseph T Kennedy, T Hsueh, Sudip Shekhar, Ganesh

Balamurugan, Frank O'Mahony, Clark Roberts, Randy Mooney, and Bryan Casper. A scalable 0.128-to-1tb/s 0.8-to-2.6 pj/b 64-lane parallel i/o in 32nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, pages 402–403. IEEE, 2013.

[79] Daniel McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. Automatic SIMD vectorization of fast Fourier transforms for the larrabee and avx instruction sets. In *International Conference on Supercomputing (ICS)*, 2011.

[80] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trns. Prog. Lang. Syst.*, 18(4):424–453, 1996.

[81] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Formal datapath representation and manipulation for implementing DSP transforms. In *Design Automation Conference (DAC)*, pages 385–390, 2008.

[82] Peter A. Milder, James C. Hoe, and Markus Püschel. Automatic generation of streaming datapaths for arbitrary fixed permutations. In *Design, Automation and Test in Europe (DATE)*, pages 1118–1123, 2009.

[83] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2), 2012.

[84] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[85] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[86] O Mutlu and T Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 63–74. IEEE, 2008.

[87] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160. IEEE Computer Society, 2007.

[88] R. Nair, S.F. Antao, C. Bertolli, P. Bose, J.R. Brunheroto, T. Chen, C. Cher, C.H.A. Costa, J. Doi, C. Evangelinos, B.M. Fleischer, T.W. Fox, D.S. Gallo, L. Grinberg, J.A. Gunnels, A.C. Jacob, P. Jacob, H.M. Jacobson, T. Karkhanis, C. Kim, J.H. Moreno, J.K. O'Brien, M. Ohmacht, Y. Park, D.A. Prener, B.S. Rosenburg, K.D. Ryu, O. Sallenave, M.J. Serrano, P.D.M. Siegl, K. Sugavanam, and Z. Sura. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, March 2015.

[89] Mike O'Connor. Highlights of the high bandwidth memory (hbm) standard.

[90] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *ISCA*, pages 192–203, 1998.

[91] Dhinakaran Pandiyan and Carole-Jean Wu. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 171–180. IEEE, 2014.

[92] N. Park, Bo Hong, and V.K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, July 2003.

[93] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *Micro, IEEE*, 17(2):34–44, Mar 1997.

[94] J. T. Pawlowski. Hybrid memory cube (HMC). In *Hotchips*, 2011.

[95] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE*

International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, 2010. ISBN 978-1-4244-7559-9.

[96] John W Poulton, William J Dally, Xi Chen, John G Eyles, Thomas H Greer, Stephen G Tell, John M Wilson, and C Thomas Gray. A 0.54 pj/b 20 gb/s ground-referenced single-ended short-reach serial link in 28 nm cmos for advanced packaging applications. 2013.

[97] Seth Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. NDC: Analyzing the impact of 3D-stacked memory+logic devices on mapreduce workloads. In *Proc. of IEEE Intl. Symp. on Perf. Analysis of Sys. and Soft. (ISPASS)*, 2014.

[98] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93 (2):232–275, 2005.

[99] Markus Püschel, Peter A. Milder, and James C. Hoe. Permuting streaming data using rams. *J. ACM*, 56(2):10:1–10:34, April 2009. ISSN 0004-5411.

[100] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[101] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory access scheduling. In *Proc. of the 27th Int. Symposium on Computer Architecture (ISCA)*, pages 128–138, 2000.

[102] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comp. Arch. Letters*, 10(1):16–19, 2011.

[103] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in CUDA. Nvidia CUDA SDK Application Note, 2009.

[104] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO-46, pages 185–197, 2013.

[105] Jaewoong Sim, Alaa R Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent hardware management of stacked dram as part of memory. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 13–24. IEEE, 2014.

[106] Steven Skiena. *The Algorithm Design Manual (2. ed.)*. Springer, 2008.

[107] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. In *Proc. of Arch. Sup. for Prog. Lang. and OS*, ASPLOS XV, pages 219–230, 2010.

[108] I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11, May 2012. doi: 10.1109/InPar.2012.6339606.

[109] Charles Van Loan. *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[110] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, 2010. ISBN 978-1-4503-0064-3.

[111] C. Weis, I. Loi, L. Benini, and N. Wehn. Exploration and optimization of 3-d integrated dram subsystems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(4):597–610, April 2013.

[112] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings*

of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91, pages 30–44, 1991.

[113] Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):452–471, 1991.

[114] Dong Hyuk Woo, Nak Hee Seong, Dean L Lewis, and H-HS Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[115] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[116] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.

[117] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7.

[118] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *In Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 32–41. ACM Press, 2000.

[119] Li Zhao, R. Iyer, S. Makineni, Laxmi Bhuyan, and D. Newell. Hardware support for bulk data movement in server platforms. In *Proc. of IEEE Intl. Conf. on Computer Design, (ICCD)*, pages 53–60, Oct 2005.

[120] Qiuling Zhu, B. Akin, H.E. Sumbul, F. Sadi, J.C. Hoe, L. Pileggi, and F. Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7, Oct 2013.