Data Reorganization in Memory Using 3D-stacked DRAM

Berkin Akin Franz Franchetti James C. Hoe
Carnegie Mellon University
{bakin,franzf,jhoe}@ece.cmu.edu

Abstract

In this paper we focus on common data reorganization operations such as shuffle, pack/unpack, swap, transpose, and layout transformations. 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. This paper presents a two pronged approach for efficient data reorganization, which combines (i) a proposed DRAM-aware reshape accelerator integrated within 3D-stacked DRAM, and (ii) a mathematical framework that is used to represent and optimize the reorganization operations.

We evaluate our proposed system through two major use cases. First, we demonstrate the reshape accelerator in performing a physical address remapping via data layout transform to utilize the internal parallelism/locality of the 3D-stacked DRAM structure more efficiently for general purpose workloads. Then, we focus on offloading and accelerating commonly used data reorganization routines selected from the Intel Math Kernel Library package. We evaluate the energy and performance benefits of our approach by comparing it against existing optimized implementations on state-of-the-art GPUs and CPUs. For the various test cases, in-memory data reorganization provides orders of magnitude performance and energy efficiency improvements via low overhead hardware.

1. Introduction

Motivations. Data reorganization operations often appear as a critical building block in several scientific computing applications such as signal processing, molecular dynamics simulations and linear algebra computations (e.g. matrix transpose, pack/unpack, shuffle, etc.) [6, 16, 27, 29]. High performance libraries generally provide optimized implementations of these reorganization operations [3, 27]. Furthermore, data reorganizations are often employed as an optimization to improve the application performance. There are several works demonstrating reorganization of the data layout into a more efficient format to improve performance [29, 11, 44, 21, 55, 54]. Physical data reorganization in memory is free of data dependencies and it preserves the program semantics—it provides a software transparent performance improvement [35].

However, reorganization operations incur significant energy and latency overheads in conventional systems due to limited data reuse, inefficient access patterns and roundtrip data movement between CPU and DRAM. It is shown that a substantial portion of the total system energy is spent on data movement [38, 37]. According to technology scaling trends, the ratio of the data movement energy to the total system energy is further increasing [37].

3D-stacking. Near data processing (NDP) can be an effective solution to reduce the data movement between the processor and the memory. By integrating processing capability into the memory, NDP allows localized computation where the data reside, reducing the roundtrip data movement, energy and latency overheads. NDP has been studied in the past under various technology contexts [28, 43, 45, 36]. However, the performance of these approaches is limited by the logic elements that are manufactured in memory process technology.

Emerging 3D die stacking with through silicon via (TSV) technology gives a rise to a new interpretation of the near data processing (NDP) concepts that has been proposed decades ago. 3D-stacked DRAM, such as Micron's Hybrid Memory Cube (HMC), exploits the TSV based stacking technology and re-architects the DRAM banks to achieve much better timing and energy efficiency at a much smaller area footprint [22, 34]. It substantially increases the internal bandwidth and reduces the internal access latency by eliminating pin count limitations. More interestingly, by integrating different process technologies of DRAM and custom logic, it allows high performance compute capability near memory.

However, this compute capability is limited by the power and thermal constraints within the stack. Complete processing units integrated in the logic layer fall short in sustaining the available internal bandwidth within the allowed power budget. Simpler compute mechanisms with specialized hardware can be an effective means to capture this opportunity. *In this paper, our goal is to enable highly concurrent, low overhead and energy efficient data reorganization operations performed in memory using 3D-stacked DRAM technology.*

Accelerating Data Reorganization. We observe that common data reorganization operations can be represented as a permutation. In this paper, we present a mathematical framework that allows structured manipulation of permutations rep-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'15, June 13-17, 2015, Portland, OR USA
Copyright 2015 ACM 978-1-4503-3402-0/15/06$15.00
http://dx.doi.org/10.1145/2749469.2750397

resented as matrices. This framework provides two important capabilities that we exploit to enable efficient hardware-based in-memory data reorganization. First, it enables restructuring the data flow of permutations. This gives us the ability to consider various alternative implementations to exploit the locality and parallelism potentials in the 3D-stacked DRAM. Second, for a given permutation, it allows deriving the index transformation as a closed form expression. Note that the index transformation corresponds to the address remapping for a data reorganization. We show that, for the class of permutations that we focus on, the index transformation is an affine function. This implies that the address remapping of the data reorganization for the entire dataset can be represented with a single, affine remapping function. We develop a configurable address remapping unit to implement the derived affine index transformations, which allows us to handle the address remapping completely in hardware for the physical data reorganizations.

Driven by the implications of the mathematical framework, we develop an efficient architecture for data reorganization in memory. Integrated within 3D-stacked DRAM, interfaced to the local vault controllers behind the conventional interface, the data reorganization unit takes advantage of the internal resources, which are inaccessible otherwise. It exploits the fine-grain parallelism, high bandwidth and locality within the stack via simple modifications to the logic layer keeping the DRAM layers unchanged. Parallel architecture with multiple SRAM blocks connected via switch networks can sustain the internally available bandwidth at a very low power and area cost. Attempting to do the same by external access (whether using stacked or planar DRAM) would be much more costly in terms of power, delay and bandwidth.

We focus on two main use cases for the in-memory data reorganization. Firstly, we demonstrate a software transparent physical address remapping via data layout transformation in memory. The memory controller monitors the access patterns and determines the bit flip rates in the DRAM address stream. Depending on the DRAM address bit flip rates, it issues a data reorganization and changes the physical address mapping to utilize the memory locality and parallelism better. This mechanism is performed transparent to the software and does not require any changes to the user software or OS, since it handles the remapping completely in hardware.

In the second use case, we focus on offloading and accelerating commonly used data reorganization operations using the 3D-stacked accelerator. We select common reshape operations from the Intel Math Kernel Library (MKL) and compare the in-memory acceleration to the optimized implementations on CPU and GPU platforms. Explicitly offloading the operations to the accelerator requires communication between the user software and the in-memory accelerator. For that purpose we utilize a software stack similar to the one proposed in [31].

Contributions. In [13], we first introduced the basic concepts of a 3D-stacked DRAM based accelerator supported

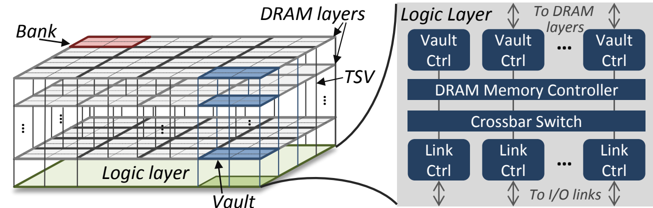


Figure 1: Overview of a HMC-like 3D-stacked DRAM [46].

by the example of regular matrix reorganizations. This paper develops the concept fully to generalized data reorganization efficiently handled by a permutation based mathematical framework for two fundamental use paradigms–explicit offloading and transparent layout transform. The most salient specific contributions are highlighted below.

- We propose an efficient data reorganization architecture that exploits the internal operation and organization of the 3D-stacked DRAM. It keeps the DRAM layers unchanged and requires simple modifications in the logic layer, which yield only a 0.6% increase in power consumption.
- We present a mathematical framework to represent data reorganizations as permutations and systematically manipulate them for efficient hardware based operation.
- We demonstrate a methodology to derive the required address remapping for a given permutation based data reorganization.
- We evaluate the software transparent physical address remapping via data reorganization for various general purpose benchmarks [17, 32] and demonstrate up to 2x/1.2x performance/energy improvements.
- We analyze common data reorganization routines selected from MKL which are performed in memory and demonstrate substantial performance and energy improvements over optimized CPU (MKL) and GPU (CUDA) based solutions.
- For various memory configurations, we compare the in-memory acceleration to the on-chip DMA based operation and show up to 2.2x/7x performance/energy improvements.

2. Background

2.1. 3D-stacked DRAM Overview

3D-stacked DRAM is an emerging technology where multiple DRAM dies and logic layer are stacked on top of each other and connected by through silicon vias (TSV) [46]. 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-stacking technology both from industry such as Micron's Hybrid Memory Cube (HMC) [46], AMD/Hynix's High Bandwidth Memory (HBM) [9], and from academia [25, 39].

Figure 1 shows the overview of a 3D-stacked DRAM architecture. It consists of multiple layers of DRAM where each layer also has multiple banks. A vertical slice of stacked banks forms a structure called *vault*. Each vault has its own

independent TSV bus and vault controller [34]. 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 [22, 57]. 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. [63]). We will refer to this operation as *intra vault parallelism*.

Similar to the conventional DRAM, each bank has a row buffer that holds the most recently accessed DRAM row. If the accessed row is already active, i.e. already in the row buffer, then a *row buffer hit* occurs, reducing the access latency considerably. 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.

3D-stacked DRAM provides high bandwidth and energy efficiency potentials. However these promised potentials are only achievable via efficient utilization of the fine-grain parallelism (intra/inter vault) and locality within the stack.

As shown in Figure 1, the logic layer also includes a memory controller, a crossbar switch, vault and link controllers. 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 [34]. However, the thermal and power constraints limit the complexity of the custom logic that could be introduced.

2.2. 3D Stacking Based Near Data Computing

3D-DRAM offers substantial improvements but bandwidth, latency and energy concerns still exist when connecting the processor and the 3D-stacked DRAM via an off-chip bus. For example HMC spends almost half of its power consumption in the SerDes units transferring data off the stack [48, 34]. Hence it is an attractive option to integrate the processing elements in the base logic layer [48, 63, 13, 60, 33, 40, 12, 26].

However, complete processing elements integrated in the logic layer are limited in taking advantage of the internal bandwidth while staying within an acceptable power envelope. For example, in [48], 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 [33], it is reported that, to sustain the available bandwidth, more than 200 PIM (processing in memory) cores are required, which exceeds the power limit for the logic layer. 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.

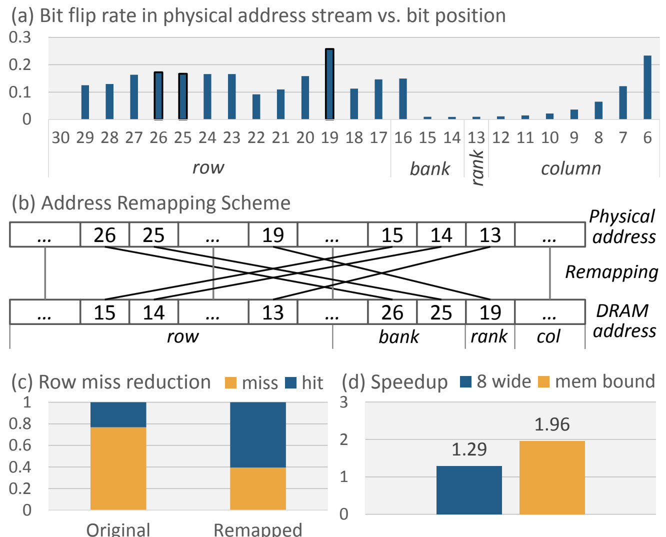


Figure 2: (a) Normalized bit flip rate in the physical address stream, (b) address remapping scheme and (c) row buffer miss rate reduction via the address remapping and (d) the resulting performance improvements for 8-wide and infinite compute power systems for *facesim* from PARSEC.

2.3. Data Reorganization

We refer to an operation that relocates the data in the memory as data reorganization. These operations have several use cases.

In Figure 2 we demonstrate a motivational example for address remapping via data reorganization. For this example, we ran the memory trace for the PARSEC [17] benchmark *facesim* available from [8] on the USIMM DRAM simulator [20] modeling a single channel DDR3-1600 DRAM [2] with open page policy and FR-FCFS (first-ready, first-come first-serve) scheduling. Figure 2(a) demonstrates the normalized average address bit flip rate in the memory access stream. 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. Figure 2(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(c) presents the achieved reduction in the row buffer miss rate via address remapping. Finally, Figure 2(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 show that a global address mapping may be suboptimal for some applications. There exists efficient address mapping schemes such as [61]. Moreover, application-specific address mapping schemes that will lead to higher parallelism and locality can be determined via profiling.

However, simply changing the address mapping at the run-time will result in incorrect program execution–the data need to be reorganized accordingly to retain the original program semantics.

Secondly, data reorganizations also 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.) [6, 16, 27, 11, 29, 14]. Considering the high precision and large data sizes, a significant fraction of the dataset reside in main memory. Therefore, ideally, most of the data reorganization operations are memory to memory. However, on conventional systems the data needs to traverse the memory hierarchy and the processor, incurring large energy and latency overheads.

This paper proposes an efficient hardware substrate to reorganize the data in memory. A mathematical formulation is presented for generalization and optimized implementations of these operations.

3. Mathematical Framework

3.1. Motivation

We make the observation that every data reorganization corresponds to a permutation. A permutation takes an input vector d_{in} and rearranges the elements to produce d_{out}. For example the stride permutation operation, denoted as $\mathrm{L}_{nm,n}$, 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:

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

Permutations can be represented via matrix-vector multiplication such that $d_{out} = \mathrm{L}_{nm,n}.d_{in}$ where $\mathrm{L}_{nm,n}$ is the permutation matrix.

Each permutation has a corresponding index transformation that represents the address remapping for a data reorganization. For example, Figure 3(a) shows the reorganization of an 8-element dataset according to the stride permutation $\mathrm{L}_{8,2}$. After the permutation reorganizes the data, 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(b) an address remapping mechanism that forwards the input addresses (x) to their new locations (y) can solve this problem. Figure 3(b) shows the index transformation of the permutation $\mathrm{L}_{8,2}$, given as $y = Bx$. Following the previous example, this unit will forward the access $x = 001$ to the location $y = 100$ via $y = Bx$ such that the returned data will be b as expected.

The simple example in Figure 3 demonstrates that the index transformation can be used to remap the addresses after a reorganization. Our goals are (i) to develop a systematic way to determine the index transformation for general permutation based reorganization operations and (ii) to design an efficient substrate to implement the index transformation for address remapping.

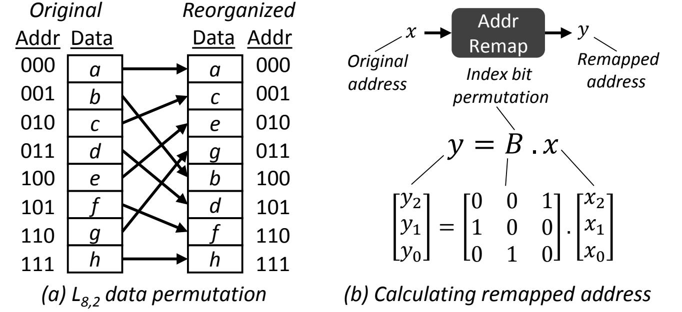


Figure 3: An example data reorganization ($\mathrm{L}_{8,2}$) on 8 elements and the corresponding address remapping scheme.

3.2. Address Remapping

Before going forward, we define a few special matrices and matrix operators. First we define $n \times n$ identity matrix I_n, n-element column vector of 0's μ_n, n-element column vector of 1's ν_n, and finally a cyclic shift matrix $\mathrm{C}_{nm,n}$ which applies a circular shift of n elements in the nm element input vector:

$$\mathrm{I}_n = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}, \quad \mu_n = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \nu_n = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}, \quad \mathrm{C}_{nm,n} = \left[\begin{array}{c|c} & \mathrm{I}_n \\ \hline \mathrm{I}_m & \end{array}\right].$$

Then, we introduce two matrix operators, tensor (Kronecker) product ($\otimes$) and direct sum ($\oplus$). These operators are useful in structured combinations of the matrices. Tensor product is defined as $A \otimes B = [a_{i,j}B]$, where $A = [a_{i,j}]$. The two examples below demonstrate their operation:

$$\mathrm{I}_n \otimes A = \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix}, \quad \text{and} \quad A \oplus B = \left[\begin{array}{c|c} A & \\ \hline & B \end{array}\right].$$

After these definitions, 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:

$$y = Bx + c \tag{1}$$

Here, assuming n bit addresses, x and y are n-element vectors that represent the input and remapped addresses respectively. B is an $n \times n$ matrix and c is an n-element vector. In this form, the index transformation is an affine transformation on the addresses.

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:

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

$$f_\pi(\mathrm{I}_{2^n}) \rightarrow B = \mathrm{I}_n, \quad c = \mu_n \tag{3}$$

$$f_\pi(P \cdot Q) \rightarrow B = B_P \cdot B_Q, \quad c = c_P + c_Q \tag{4}$$

$$f_\pi(P \otimes Q) \rightarrow B = B_P \oplus B_Q, \quad c = \begin{bmatrix} c_P \\ c_Q \end{bmatrix} \tag{5}$$

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. 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:

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

J_n is simply the I_n matrix with the rows in the reversed order. 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 swapping the two consecutive n-element regions in the memory. Address remapping for the swap permutation is given as follows:

$$f_\pi(\mathrm{J}_{2^n}) \rightarrow B = \mathrm{I}_n, \quad c = \nu_n \tag{6}$$

Morton Layout. Morton data layouts, or in general space filling curves, are based on recursive blocking [42]. Large data sets are divided into blocks recursively until the small leaf blocks reach 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 [44, 18]. Z-morton layout can be represented only by stride permutations, so the properties (2)-(5) 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:

$$f_\pi(Z_{2^n,2^m}) \rightarrow B = \mathrm{L}_{n,2} \otimes \mathrm{I}_m, \quad c = \mu_n \tag{7}$$

Conditional Permutations. For some applications, different permutations on separate regions of the dataset are required. 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:

$$f_\pi(P_k \oplus Q_l) = \begin{cases} f_\pi(P_k) & \text{for } 0 \leq i < k \\ f_\pi(Q_l) & \text{for } k \leq i < k+l \end{cases} \tag{8}$$

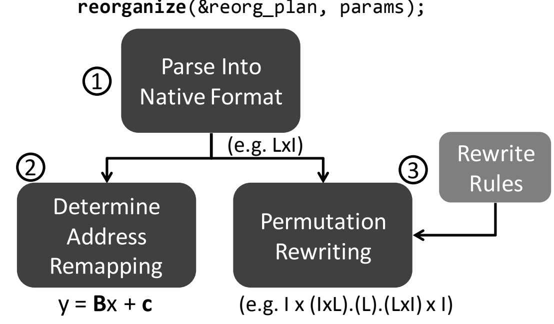


Figure 4: Overview the mathematical framework toolchain.

3.3. Practical Implications

In summary, the properties of the f_π function given in (2)-(8) demonstrate a structured way of deriving the index transformation, or address remapping, for a given permutation. Next, we provide the practical implications of these properties.

Concise Address Remapping. 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.

Address Remapping in Hardware. Analysis of the properties (2)-(8) reveals that 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.

Generalization. The set of properties (5), (6) and (8) demonstrate a systematic way of handling the combinations of various permutations for generalization. Although the developed techniques are limited to permutations only, we demonstrate that they can cover a wide range of reorganization operations (see Section 7).

3.4. A Rewriting System for Permutations

The mathematical formalism used to represent the permutations is well understood and utilized in various domains [49, 56, 59, 50]. It enables us to use the SPL domain specific language to express the data reorganizations as permutations [59]. Furthermore, it enables inclusion of the developed framework into an existing rewriting system for permutations [50]. In [50] there are various well-known permutation identities, referred to as rewrite rules, that restructure the permutation's data flow to optimize for parallelism, SIMD vectorization or hardware implementations [50, 56, 49]. For our purposes, we use this framework to optimize (rewrite) the reorganization

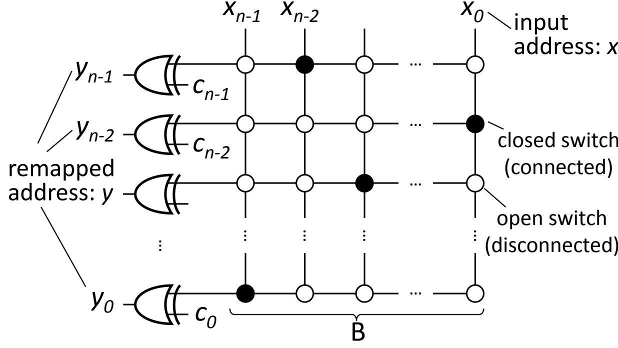


Figure 5: Bit Shuffle Unit (BSU).

operations to exploit memory parallelism and locality.

An overall view of the mathematical framework based toolchain is given in Figure 4. 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. Outputs of blocks ② and ③ will be offloaded to configure the reshape accelerator.

4. Architectural Design

4.1. Address Remapping Unit (ARU)

The address remapping operation (i.e. $y = Bx + c$) consists of two main parts. First, the B matrix shuffles the input address bits x, then the c vector inverts the shuffled 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 5. In Section 3 we saw that B is a permutation matrix, therefore there is only a single non-zero element per row. 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 shuffled via the crossbar, XOR array inverts the bits according to the c vector to generate the remapped address y. This unit can implement any bit permutation, i.e. all combinations of (2)-(7).

Moreover, to support multiple address remapping schemes simultaneously (i.e. conditional permutation (8)) we propose the address remapping unit (ARU) that extends the BSU as shown in Figure 6. The *configuration store* keeps various 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. An application can occupy multiple regions if needed.

The ARU can support all the remapping schemes presented

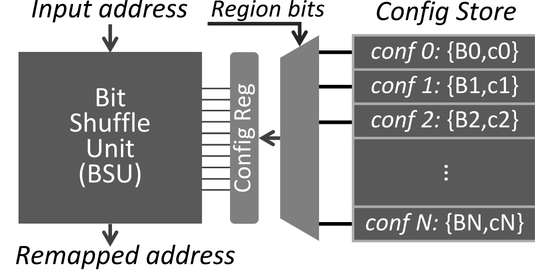


Figure 6: Address Remapping Unit (ARU).

in Section 3 via a very simple hardware. The configuration store indexing and the read latency depends on the number of simultaneous different mappings supported. As we will see later, for a typical implementation, the overall latency and energy consumption of the combined configuration store indexing, bit shuffle and XOR circuit 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.

4.2. Data Reshape Unit (DRU)

The data reshape unit (DRU) shown in Figure 7 executes the final decomposed permutation output from block ③ (see Figure 4). The final decomposed permutation has two main parts, namely global read/write permutations and a local permutation. Read/write permutations are mapped onto read/write controllers in DRU. These are dedicated DMA units for generating DRAM requests based on the read and write permutations. They ensure correct flow of the data between the DRAM layers and the local permutation unit.

The datapath for the local permutation unit within the DRU consists of SRAM banks and two switch networks. 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 local permutation unit in DRU adopts the solution from [49]. 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. 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

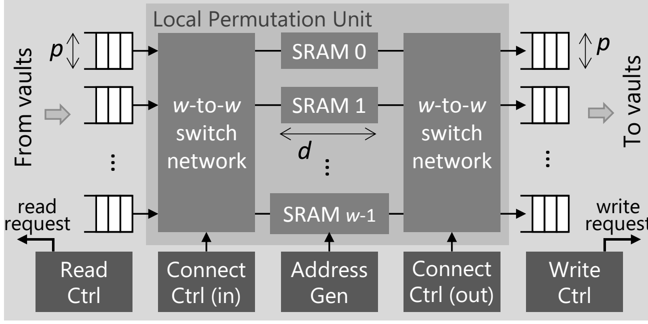


Figure 7: Data Reshape Unit (DRU).

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 $2wd$.

This datapath is controlled by the *connection control* and SRAM *address generator* units. These units generate the required connection configuration and addresses every cycle that will route the data elements to and from the SRAM banks. The local permutation, which is the output of the block ③ in the toolchain (see Figure 4), directly configures these units.

5. Applications

5.1. Software Transparent Data Layout Transformation

First we evaluate the hardware based address remapping via data layout transformation. In this case, the memory controller issues a reorganization by monitoring the memory access stream. Memory access monitoring determines the bit flip rates in the address every epoch. The monitoring hardware XORs the current address with the previous address values and accumulates the number of flips per bit location. At the end of every epoch, it normalizes the number of flips with the number of total accesses. If it observes that the bit flip rate ratio between vault/layer region and row region of the DRAM address is higher than a set threshold consistently over several epochs, it issues an address remapping to swap such bits. The goal is to map the least frequently flipping bits to the row region and most frequently flipping bits to the vault/layer region to increase (intra/inter vault) parallelism and row buffer locality. The physical address remapping requires data reorganization in the memory to retain the original program semantics, which is performed in-memory via DRU. For simplicity, we block the accesses to the memory regions that are being reorganized but independent processes can access other regions. DRU handles the reorganization efficiently (typically under a few milliseconds as we will see in Section 7.3), hence the overhead of the data reorganization is amortized during the course of the long application runtime.

5.2. Accelerating Reorganization Routines

Next we evaluate the acceleration of common data reorganization routines by explicitly offloading the operation to the data reorganization unit (DRU). We focus on commonly used reorganization routines selected from the Intel Math Kernel Library (MKL) [3]. These operations are expressed in the domain specific language (SPL) of our mathematical framework and a custom software stack is used to offload the operation to the DRU integrated in the 3D-stacked DRAM.

6. System Integration

To make the proposed reshape accelerator compatible with an existing system, we need to make a few changes at the software and hardware level. These modifications are based on the use case of the accelerator.

6.1. Hardware Based Approach

For the physical data reorganization driven by the memory controller (see Section 5.1), 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 proposed in Section 4.

Physical data migration has been studied in previous work [54, 24, 51]. 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.

6.2. Custom Software Stack Approach

We provide a custom software stack to give the user a flexible control over the accelerator in issuing an explicit offload (Section 5.2). We follow the same methodology proposed in [31]. In [31] authors provide a memory model similar to NUMA (Non-Uniform Memory Access) where the accelerator can access both local and remote memory stacks. In this work, we further restrict our system such that the accelerator can only access the local memory stack that it is integrated into. Customized memory management functions (malloc/free) only allocate the memory region within the corresponding memory stack. The host offloads the configuration to the allocated region to be read by the accelerator and also uses it to check the status of the accelerator. A device driver maps the allocated physically contiguous region into the virtual memory space via *mmap*. During the mmap call, allocated pages are locked 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

Table 1: 3D-stacked DRAM low level energy breakdown.

Parameter	Value (pj/bit)	Reference
DRAM access (CAS)	2 - 6	[22, 34, 57]
TSV transfer	0.02 - 0.11	[22, 60, 7]
Control units	1.6 - 2.2	[4, 34]
SERDES + link	0.54 - 5.3	[34, 47, 41, 23, 37]

Table 2: 3D-stacked DRAM configurations.

Parameter	HI	MH	ML	LO
Vault (#)	16	8	4	2
Layer (#)	8	4	4	2
Link (#)	8	8	7	1
Link BW (GB/s)	60	40	40	40
Total TSV (#)	2048	2048	1024	512
Intern BW (GB/s)	860	710	360	90
Extern BW (GB/s)	480	320	280	40
Power (Watt)	45	30	25	12

[15]. Finally a cache flush is issued, before the acceleration starts, to ensure that the accelerator accesses the most recent copy in the memory stack to avoid coherency problems.

7. Evaluation

7.1. 3D-stacked DRAM

We use a custom trace based, cycle accurate, command level simulator for the 3D-stacked DRAM. It features a CPU front-end similar to the USIMM DRAM simulator [20]. 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 [22] and published numbers from literature [34, 7]. Logic layer memory controller performance and power are estimated using McPAT [4]. Finally, the SerDes units and off-chip I/O links are modeled assuming a high speed short link connection [47, 41, 23]. 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 1. However, these values change depending on the particular configuration of the memory.

Table 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. 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 [34] to 2 KB [58] are reported). Medium to high end configurations can reach overall energy efficiency of 11-12 pj/bit that is very close to the 10.48 pj/bit energy efficiency of the HMC system [34].

Table 3: Processor configuration.

Parameter	Value
Cores	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

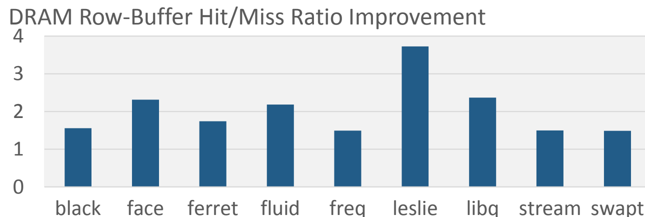
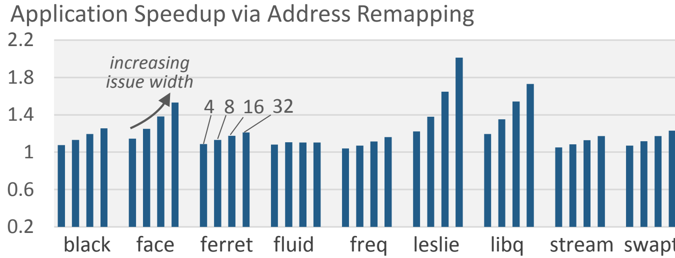


Figure 8: Row buffer miss rate reduction via physical address remapping with data reorganization.

7.2. Software Transparent Data Layout Transform

First, we focus on hardware based data layout transform for address remapping (described in Section 5.1). We evaluate several memory traces from PARSEC [17] and SPEC CPU 2006 [32] suites available from [8]. We use the MH configuration for the 3D-stacked DRAM. Processor configuration is given in Table 3. First, Figure 8 demonstrates the effect of address remapping on the DRAM row buffer miss rate. Remapping frequently flipping bits from row address bits to the vault and layer address bits significantly 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 9. We observe that more memory intensive applications (e.g. leslie, libquantum) gain higher improvements since their performance are more sensitive to the memory utilization. To analyze the performance improvement sensitivity to the memory intensity, we also evaluate wider issue width configurations (4-32) in Figure 9.

Efficient transparent data reorganization requires design choices regarding the DRU parameters such as bit flip ratio threshold, epoch length and number of memory regions. In Figure 10 we observe that with a small bit flip ratio threshold DRU can select bits too early which may not be beneficial in the long run. Whereas, with a very large threshold it can miss some useful remappings but the issued remappings will most likely improve performance. We pick a threshold of 2 to minimize the chance of useless remappings. Similarly, the 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. We chose 50K cycle epochs empirically and



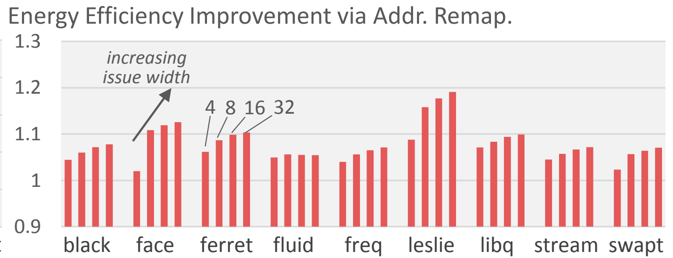


Figure 9: Performance and energy improvements via physical address remapping with data reorganization.

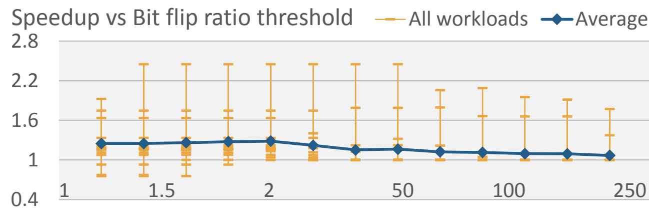


Figure 10: Bit flip ratio threshold sensitivity on speedup.

keep the bit flip rate values in a moving window of 4 epochs. Further, large number of memory regions allow us to capture spatial differences in the access patterns. We chose 16 memory regions in our experiments.

7.3. Accelerating Reorganization Routines

Next, we analyze the use case described in Section 5.2. The list of the reorganization benchmarks are given in Table 4. These are commonly used reorganization routines selected from the Intel Math Kernel Library (MKL) [3]. Dataset sizes for these benchmarks range from 4 MB to 1 GB. 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. As a GPU reference we use a modified version of the implementation from Nvidia [52] using CUDA 5.5 on a GTX 780 (Kepler) platform. However, due to the limited memory size we could not run a few of the benchmarks with large dataset sizes (≥ 1 GB) on the GPU.

Energy and runtime of the operations implemented on the DRU is simulated using the 3D-stacked DRAM simulator. Reported numbers also include the energy overhead of the DRU implementation in the logic layer which will be later analyzed 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 interface [5, 10]. Finally the GPU power consumption is measured from the actual board using a PCI riser card and inductive current probes. The results are given in Figure 11 and Figure 12. Note that the 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

Table 4: 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

DRAM can provide orders of magnitude performance and energy efficiency improvements when compared to the optimized implementations on the state-of-the-art CPUs and GPUs. In these experiments the medium-high (MH) 3D-stacked DRAM configuration is used. The MH configuration provides 320 GB/s of bandwidth that is much higher than what is available to the CPU and GPU. Next, we also provide individual comparisons where the bandwidth of the 3D-stacked DRAM is very close to each individual platform's bandwidth. Figure 13 demonstrates the case where the DRU integrated in the LO configuration (single link, 40 GB/s) is compared against the CPU (two channels, 25.6 GB/s). Also Figure 14 compares the DRU integrated in the ML configuration (7 links, 280 GB/s) with the GPU (288 GB/s). These comparisons demonstrate that given the same levels of memory bandwidth, DRU integrated in the 3D-stacked DRAM can provide much higher performance and energy efficiency, compared to the CPU and GPU platforms. 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 achieve higher performance and energy efficiency.

7.4. Offload Overhead For DRU

As discussed in Section 6, a custom software stack handles offloading the computation to the DRU. Offload operation includes transferring the DRU configuration and a cache flush operation to ensure coherency. To give a better insight, average time and energy spent on the host during the offload operation for different dataset sizes are given in Figure 15. We observe that for small datasets (< 64MB) the offload operation takes much longer than the accelerated operation itself. For these dataset sizes, a significant fraction of the dataset actually resides in the cache, which makes them more suitable for host-side operation. However, more efficient offloading mechanisms can decrease the time and energy spend on the

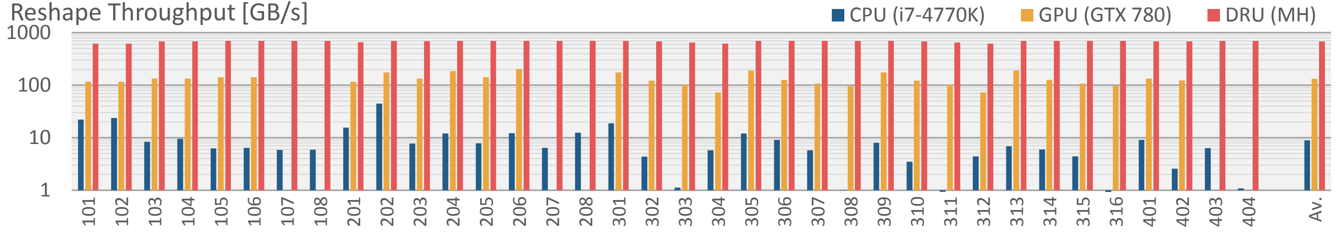


Figure 11: Performance of the 3D-stacked DRAM based DRU is compared to optimized implementations on CPU and GPU.

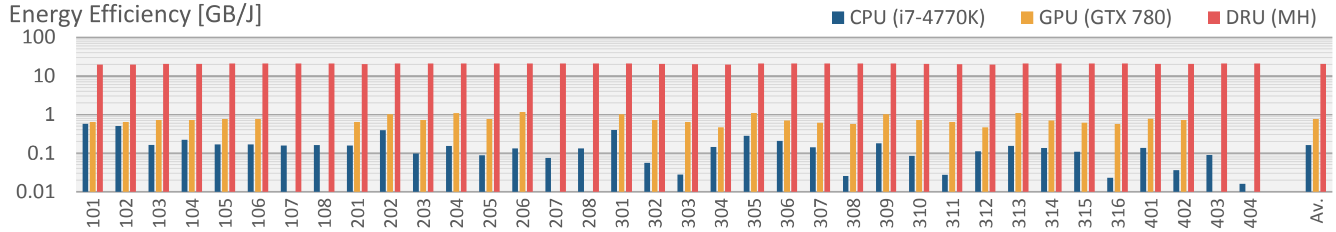


Figure 12: Energy efficiency of the 3D-stacked DRAM based is DRU compared to optimized implementations on CPU and GPU.

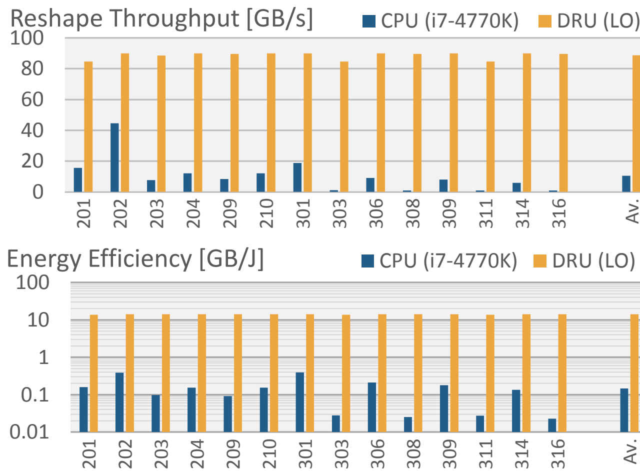


Figure 13: DRU in LO configuration is compared to the CPU.

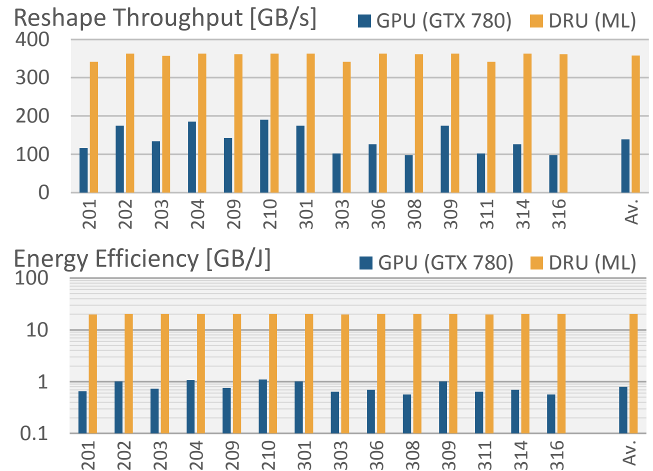


Figure 14: DRU in ML configuration is compared to the GPU.

host, making the accelerator more accessible to various applications. Nevertheless, especially for large datasets, DRU offload (under a millisecond) is much faster compared to the GPU offload which requires the transfer of the actual dataset over slow PCIe (tens of milliseconds).

7.5. In-memory vs. On-chip DMA Accelerator

Integration within the stack, behind the conventional interface, opens up the internal resources such as abundant bandwidth and parallelism. Here, we evaluate the effect of moving the accelerator within the stack to exploit these resources. For this purpose, we take a host system with a CPU-side DMA unit connected to regular 3D-DRAM, and compare it against a system that has the same 3D-DRAM where the logic layer integrates the DRU. Figure 16 shows up to 2.2x performance and 7x energy improvements for the in-memory accelerator compared to on-chip DMA. This comparison provides the lower bound on the improvement due only to DRU since both systems feature the same 3D-DRAM.

7.6. Hardware Cost Analysis

We also present hardware cost analysis for the DRU unit 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 CACTI [1], to model the SRAM blocks. ARU and DRU synthesis results 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. MH has 8 vaults each with 256 bit TSVs. DRAM banks have 1 KB row buffers. Hence we have 16 banks (2×8 due to double buffering) of 32 KB SRAMs where $w = 8$, $p = 256$, $k = 32$, $d = 1024$ as discussed in Section 4. The details of the synthesis results for the full reorganization accelerator integrated into MH configuration are given in Table 5.

Overall, the power consumption overhead of 178 mW, including leakage and dynamic power, corresponds to only a

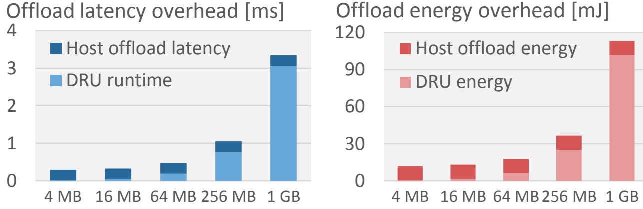


Figure 15: Time and energy spent on the host for offloading the operation to DRU (MH).

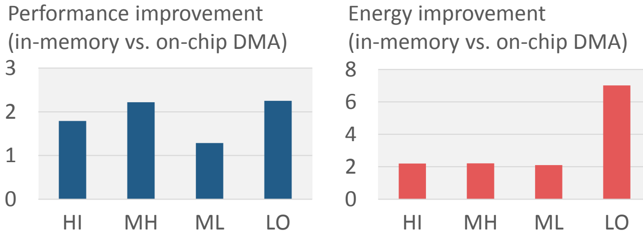


Figure 16: Comparison between the accelerator in-memory and on-chip as a memory controller based DMA.

small fraction (0.6%) of the 30 W power envelope of the 3D-stacked DRAM.

8. Related Work

Near Data Processing. Integrating processing elements near memory has been studied in the past under various technology contexts. PIM [28], active-pages [43], IRAM [45] and FlexRAM [36] are some notable examples for planar NDP. Main limitation of these approaches is the logic and memory elements manufactured in the same process technology.

3D Stacking Based NDP. Recent 3D stacking technology integrates different process technologies of DRAM and custom logic [34]. It is an attractive option to integrate the processing elements in the logic layer [48, 60, 33, 40, 58, 26]. However, complete processing elements integrated in the logic layer are limited in utilizing the internal bandwidth while staying within an acceptable power envelope. There are also several work demonstrating specialized hardware units integrated near DRAM using 3D stacking [63, 13, 12].

Hardware Assisted Data Reorganization. There are various proposals for hardware assisted data migration for better data placement [24, 51, 54]. These approaches simply keep an address translation table in hardware to keep track of the migrated data and update the OS page table periodically. The address translation look-up table is not scalable to support large scale and fine grain data reorganizations.

Direct copy between DRAM rows proposed in [53] changes the DRAM structure for efficient bulk data movement. This technique can be incorporated into the DRAM layers of our system as well, but general data reorganization routines require local buffering and permutation that cannot be addressed only via [53].

Address remapping without physical relocation can consolidate accesses via indirection but it does not solve the fundamental data placement problem [19, 55]. There are also techniques such as specialized copy engines [62], using GPU's high memory bandwidth to overlap the layout transforms with slow PCI data transfer [21], or keeping multiple copies of different layouts [30]. Moreover, in [13] authors demonstrate a 3D-stacked accelerator for regular matrix reorganizations. However, our approach demonstrates generalized data reorganization efficiently handled by a permutation based mathematical framework for two fundamental use paradigms, explicit offloading and transparent layout transform.

Table 5: HDL synthesis results at 32nm.

Unit	Power consumption (mW)
Configuration store (ARU)	0.69
Bit shuffle unit (ARU)	2.9
SRAM banks (DRU)	159
Switch + control (DRU)	16.6
TOTAL	178.6

9. Conclusion

In this paper we present a series of mechanisms that enable efficient data reorganization in memory using 3D-stacked DRAM. In-memory operation, behind the conventional interface, not only minimizes the roundtrip data movement but also opens up resources including high internal bandwidth and abundant fine-grain parallelism, which are inaccessible otherwise.

We make the key observation that the common data reorganization operations can be represented as permutations. We utilize a mathematical framework to manipulate the permutations for efficient operation within the stack. Driven by the implications from the mathematical framework, we develop a low overhead and high performance architecture for data reorganization operations. We also provide hardware/software solutions for system integration.

We evaluate this hardware substrate through two main use cases, (i) physical address remapping via data layout transformation, and (ii) acceleration of the common reorganization operations via explicit offloading. Our results demonstrate that in-memory data reorganization provides orders of magnitude performance and energy efficiency improvements via low overhead hardware.

10. Acknowledgements

We thank the CMU PERFECT team, SPIRAL and CALCM lab members, and the anonymous reviewers for their insightful comments. This work was sponsored by the DARPA PERFECT program under agreement HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred.

References

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

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

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

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

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

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

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

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

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

[10] "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.

[11] B. Akin, F. Franchetti, and J. 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*, 2014, pp. 3898–3902.

[12] ——, "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*, 2014, pp. 248–255.

[13] B. Akin, J. C. Hoe, and F. 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*, 2014, pp. 1–6.

[14] B. Akin, P. A. Milder, F. Franchetti, and J. 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*, 2012, pp. 188–191.

[15] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 237–248.

[16] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. 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*, vol. 93, no. 2, pp. 276–292, Feb 2005.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[18] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. 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*. ACM, 2009, pp. 233–244.

[19] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. 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*, Jan 1999, pp. 70–79.

[20] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module," 2012.

[21] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proc. of Intl. Conf. for High Perf. Comp., Networking, Storage and Analysis (SC)*, 2011, pp. 13:1–13:11.

[22] K. Chen, S. Li, N. Muralimanohar, J.-H. Ahn, J. Brockman, and N. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Design, Automation Test in Europe (DATE)*, 2012, pp. 33–38.

[23] T. O. Dickson, Y. Liu, S. V. Rylov, B. Dang, C. K. Tsang, P. S. Andry, J. F. Bulzacchelli, H. A. Ainspan, X. Gu, L. 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*, vol. 47, no. 4, pp. 884–896, 2012.

[24] X. Dong, Y. Xie, N. Muralimanohar, and N. 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*. IEEE Computer Society, 2010, pp. 1–11.

[25] R. G. Dreslinski, D. Fick, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, D. Sylvester, D. Blaauw, and T. Mudge, "Centip3de: A many-core prototype exploring 3d integration and near-threshold computing," *Commun. ACM*, vol. 56, no. 11, pp. 97–104, Nov. 2013.

[26] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. 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*, Feb 2015, pp. 283–295.

[27] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Platform Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005.

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

[29] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008.

[30] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev, "Sams multi-layout memory: Providing multiple views of data to boost simd performance," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10, 2010, pp. 179–188.

[31] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. 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.

[32] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[33] M. Islam, M. Scrback, K. 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*, ser. UCHPC2014, 2014.

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

[35] 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*, ser. MICRO 31, 1998, pp. 285–297.

[36] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: Toward an advanced intelligent memory system," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE, 2012, pp. 5–14.

[37] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[38] G. Kestor, R. Gioiosa, D. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 56–65.

[39] D. H. Kim, K. Athikulwongse, M. Healy, M. Hossain, M. Jung, I. Khorosh, G. Kumar, Y.-J. Lee, D. Lewis, T.-W. Lin, C. Liu, S. Panth, M. Pathak, M. Ren, G. Shen, T. Song, D. H. Woo, X. Zhao, J. Kim, H. Choi, G. Loh, H.-H. Lee, and S.-K. Lim, "3d-maps: 3d massively parallel processor with stacked memory," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, Feb 2012, pp. 188–190.

[40] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *Proc. of the 35th Annual International Symposium on Computer Architecture, (ISCA)*, 2008, pp. 453–464.

[41] M. Mansuri, J. E. Jaussi, J. T. Kennedy, T. Hsueh, S. Shekhar, G. Balamurugan, F. O'Mahony, C. Roberts, R. Mooney, and B. 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*. IEEE, 2013, pp. 402–403.

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

[43] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *ISCA*, 1998, pp. 192–203.

[44] N. Park, B. Hong, and V. Prasanna, "Tiling, block data layout, and memory hierarchy performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 640–654, July 2003.

[45] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *Micro, IEEE*, vol. 17, no. 2, pp. 34–44, Mar 1997.

[46] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hotchips*, 2011.

[47] J. W. Poulton, W. J. Dally, X. Chen, J. G. Eyles, T. H. Greer, S. G. Tell, J. M. Wilson, and C. T. 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.

[48] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. 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.

[49] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using rams," *J. ACM*, vol. 56, no. 2, pp. 10:1–10:34, Apr. 2009.

[50] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. of IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232–275, 2005.

[51] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 85–95.

[52] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in CUDA," Nvidia CUDA SDK Application Note, 2009.

[53] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture*, ser. MICRO-46, 2013, pp. 185–197.

[54] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: Increasing dram efficiency with locality-aware data placement," in *Proc. of Arch. Sup. for Prog. Lang. and OS*, ser. ASPLOS XV, 2010, pp. 219–230.

[55] I.-J. Sung, G. Liu, and W.-M. Hwu, "Dl: A data layout transformation system for heterogeneous computing," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–11.

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

[57] 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*, vol. 32, no. 4, pp. 597–610, April 2013.

[58] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. 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*. IEEE, 2010, pp. 1–12.

[59] J. Xiong, J. Johnson, R. W. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Programming Languages Design and Implementation (PLDI)*, 2001, pp. 298–308.

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

[61] Z. Zhang, Z. Zhu, and X. 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*. ACM Press, 2000, pp. 32–41.

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

[63] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. 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*, Oct 2013, pp. 1–7.