Self-adaptive Address Mapping Mechanism for Access Pattern Awareness on DRAM

Chundian Li*‡, Mingzhe Zhang*, Zhiwei Xu*‡,Xianhe Sun†
*SKL of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
†Department of Computer Science, Illinois Institute of Technology, Chicago, USA
‡University of Chinese Academy of Sciences, Beijing, China
{lichundian, zhangmingzhe, zxu}@ict.ac.cn, sun@iit.edu

***Abstract*—As DRAM is a considerably slow storage compared to CPU, the long access latency becomes a serious issue and affects the whole execution if fetching data is on the critical path. It is benefical if the data layout on DRAM, which is decided by address mapping, can serve data accesses with either great locality or bank-level parallelism. However for some cases, there exists a huge mismatch between access patterns and data layout of applications, which introduces the difficulty in obtaining locality or parallelism and current general address mapping cannot resolve it well.**

In an effort to overcome this challenge, we present an self-adaptive address mapping mechanism in memory controller to be aware of different access patterns on DRAM without any prior knowledge of applications. Moreover, there is nearly no modification on softwares, including applications, libraries and OS. We take several versions of matrix multiplication as an early verification, since their patterns are regular and simple to control. Impressively, the results reveal that memory performance of naive and tiling versions is improved up to 3.4x and 2.7x at most, 2.1x and 1.7x on average respectively. The whole execution time is reduced by up to 24% and 8% averagely. Even for highly-optimized implementations, execution time is decreased by 7% and memory performance speeds up to 1.6x on average.

I. Introduction

Main memory is regarded as the last level storage in the in-memory computation and its latency is obviously slower than on-core caches. If on-core caches can not cover data requests from CPU well, DRAM will finally serve the requests, where it plays an important role in the whole execution. DRAM serves accesses in two efficient ways, one prefers locality with high row buffer utilization [10]–[12], and the other prefers bank-parallelism with accesses distributed on different banks [13]–[15], [24], [25].

However, there is a worst situation that consecutive accesses fall onto different rows of the same bank. The memory performance degrades and CPU may stall if the execution strictly depends on the data on DRAM [9], [18]. The root cause is due to the inefficient data layout for data fetching on DRAM, which is decided by the address mapping in memory controller (MC) [3]–[7]. However, current static address mappings on DRAM are designed for accesses with better locality or even distribution on banks, instead of such a long distance that spanning one or more rows between adjacent accesses, leading to the worst situation.

As an exploration of this mismatch, we study a classical case with regular access behaviors, the general matrix-matrix multiplication (GEMM), which is common in the scientific and AI fields. The performance of GEMM tremendously depends on the efficiency of the data reuse of vectors or submatrices. Although on-core caches provide friendly locality via programming optimization like tiling technique [33], when data reuse requirements do not match the cache policy of the last level cache (LLC) and key accesses fall onto DRAM, the data reuse problem will be exposed, as the latency of memory access is roughly two orders of magnitude greater than a CPU cycle. Especially, the long distance of consecutive accesses on DRAM, i.e. lower spatial locality, delays the data reuse performance and degrades the whole execution. In addition, the distance varies by the matrix scale, so that an ad-hoc DRAM layout can not be adaptive to different scales. Even for optimized version of GEMM, We observe that the worst situation may be introduced because of mismatch of DRAM data layout and accesses.

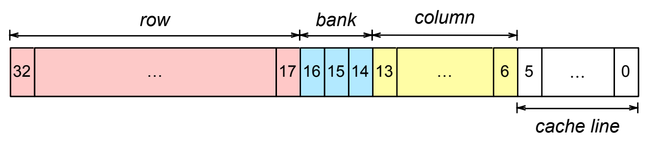
In the paper, we analyze access behaviors of common GEMM versions and highlight the mismatch phenomenon between data layout and access patterns, leading to low DRAM performance. To address the problem, we propose a self-adaptive address mapping mechanism to support data layout reshaping on DRAM to match different access patterns.

The rest of the paper is organized as follows. Section 2 and 3 illustrates background and motivation of our study. Section 4 presents the design and implementation of the self-adaptive address mapping mechanism. Section 5 and 6 describe the evaluation methodology and results. Section 7 reviews related works and section 8 concludes this work.

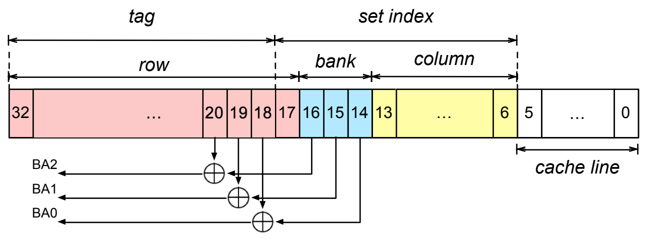
II. Background

A. DRAM Organization

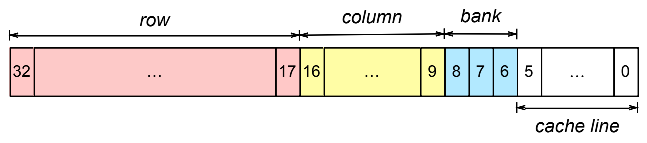
DRAM is organized in a hierarchy, from the channel as the highest level to the column as the lowest level. Every channel works in parallel independently, with an isolated bus suite, which can connect one or more memory modules. In a memory module, from the highest layer to lower one, there are ranks, chips, banks, rows, and columns hierarchically. For every single request of a cache line (64B) coming from LLC, its physical address will be decomposed to a quintuple *(channel, rank, bank, row, column)*, indicating the specific locations for each level. As there are one channel and one



(a) Row-interleaving Address Mapping (Baseline, RI)



(b) XOR Address Mapping (XOR)



(c) Cache-interleaving Address Mapping (CI)

Fig. 1: Typical Address Mappings on 8GB DDR3 DRAM. (1) RI as baseline, (2) XOR, (3)CI.

rank in our testbed (table II), it is simplified to a *triple: (bank, row, column)*.

Each bank only processes at most one data access at the same time. *Row buffer* is a sense amplifier within one bank, and it loads the whole row which the row address selects, before MC issues a *Read/Write* command [9]. In the open-page policy [2], when one access comes into one bank, it will check the row that the row buffer holds before, i.e. the row of the last access within this bank. For an access whose selected row has been in the row buffer, we call the access as a *row buffer hit*, otherwise, it is a *row buffer miss*. Timing constraints [18] tell us the latency of a row buffer miss access is much more expensive than the one of hit access, so the row buffer locality affects the DRAM performance greatly [10]–[12].

Multiple banks can serve in parallel if adjacent accesses are mapped into different banks, so that accesses can interleave to reduce data stalls. The bank-level parallelism (BLP) can potentially improve the DRAM throughput and hide the memory latency if accesses are distributed evenly into the banks [13]–[15], [24], [25]. We use two terms BLP and MLP (memory-level parallelism, in section V) with the same meaning in this paper, if there is no explicit distinction between them.

B. Address Mappings

As the accesses' distribution on banks and rows decides the locality and MLP, affecting DRAM performance mentioned above, there are several considerations about how the data is mapped into the DRAM organization. Although address mappings in commodity DRAM controllers are not public [6], there are a lot of address mappings studied in [3]–[7], involving the row-interleaving one (*RI*, fig. 1a) and the cache-interleaving one (*CI*, fig. 1c), which are two typical ones representing different preferences for locality or parallelism.

The lower column bits ($5 \sim 3$) are ignored in the following paper, as DRAM is accessed in a cache unit. As RI is shown in fig. 1a, the corresponding bits [1] for the triple (bank, row, column) are ($16 \sim 14$, $32 \sim 17$, $13 \sim 6$), from the higher zone to the lower zone. It indicates adjacent physical cache lines are visited in a streaming manner almost in the same row, with a perfect spatial locality of the row buffer. However, for CI in fig. 1c, adjacent lines will be distributed in adjacent banks to utilize the parallelism of banks efficiently. The corresponding bits for the triple are ($8 \sim 6$, $32 \sim 17$, $16 \sim 9$). Basically, these two mappings consider the access performance in orthogonal ways, i.e. the locality and the parallelism respectively. For both mappings, row bits are in the highest zone, which means both mappings are designed to satisfy accesses without too far distance between consecutive ones. However, we will show that RI fails to provides the access locality and CI fails to reveal the MLP on GEMM in the next section III.

[6] shows that XOR mapping is commonly used in modern commodity DDR3 and DDR4. Compared to RI, XOR uses extra row bits to do XOR operations with the bank bits, reducing *bank conflicts* and increasing MLP, through distributing the accesses of distinct rows within the same bank into different banks. As shown in fig. 1b, it uses the lowest three bits of LLC cache tag ($20 \sim 18$) to do XORs with the original bank bits ($16 \sim 14$) to identify the bank, avoiding the bank conflicts coming from cache conflicts of LLC. It performs effectively when two consecutive accesses go into the different rows within the same bank, such as reduction of two large vectors [7]. In addition, XOR mapping preserves the locality of the original accesses, compared to [5].

C. General Matrix-matrix Multiplication (GEMM)

GEMM is one of the most frequent operations among matrix computations, and [34] shows that convolution operations based on GEMM occupy 86% time of the whole execution. The optimization study of traditional GEMM continued for decades and lots of work studied how to accelerate these workloads from various perspectives, including programming model, data organization, compiler or microarchitecture [27]–[30], [33], [34].

There are lots of implementations [27] on different data layouts, loop orders, involving the naive one [31] and the tiling approach [33]. Intel MKL [32] is one of the most efficient implementations of the GEMM, and it is used as the basic library in lots of frameworks and applications.

Intuitively, for the naive version of square matrices, the layout of two matrices are in row-major order, but the right matrix is accessed in column-major order. For example, there is a 64KB stride between consecutive accesses for 8192x8192 matrices. Although tiling and Intel MKL versions of GEMM

[1]Bit represents the significant bit in the binary format and it starts from 0.

TABLE I: Cache Profiles for Different GEMM Implementations on Different Scales of Double-typed Square Matrices. (a) Naive, (b) Tiling, (c) Intel MKL. MR: miss rate. MPKI: missese per kilo instructions.

Version	Scale	L1 MR	LLC MR	LLC MPKI
Naive	4096	44.3%	99.9%	143
	8192	43.8%	99.9%	143
	16384	46.5%	90.4%	142
Tiling	4096	30.3%	70.6%	7
	8192	31.8%	70.5%	7
	16384	32.6%	70.9%	8
Intel MKI	4096	9.8%	61.3%	7
	8192	9.8%	62.7%	8
	16384	10.3%	59.9%	9

is not too DRAM-intensive shown in table I, DRAM performance can affect the whole performance as well, as explained in section III-A. Also, access patterns of GEMMs are so regular that it is easier to show the potential of our novel self-adaptive address mapping mechanism. We conduct a comprehensive DRAM-based analysis on the GEMM in the next section and propose a new design in section IV. Then it shows the benefits of our design using GEMM as a case in section VI.

III. Motivation

As the access pattern of GEMM is regular and easier to be analyzed, we choose three versions of GEMM as a case in our study. In this section, we did a comprehensive DRAM analysis on three dimensions: typical address mappings, GEMM implementations, and matrix scales. RI is regarded as the baseline in all tests in this section.

In our observations, we found the following problems that we would target in the novel design (in section IV).

- Typical address mappings may fail to provide its advantages when they happen to mismatch access pattern on DRAM.
- Performance of XOR conquers one of CI, or the other way around on different patterns.
- All three typical mappings may degrade DRAM performance when consecutive accesses span a long distance that disables both locality and MLP.

A. Mismatch with Access Pattern

Firstly, we measured the IPC, accesses per cycle (APC), locality, MLP on DRAM introduced in section V for tiling GEMM in fig. 2, including 4096x4096, 8192x8192 and 16384x16384 (table I). Although tiling and Intel MKL versions are cache-friendly and not too DRAM-intensive, their performance of the whole execution can be affected by DRAM performance as shown in fig. 2a and fig. 2b. As APC and IPC represent the DRAM performance and whole execution performance respectively, it shows that DRAM performance correlates strongly with the end-to-end execution, especially on a large scale of 16384, with IPCs varying a lot.

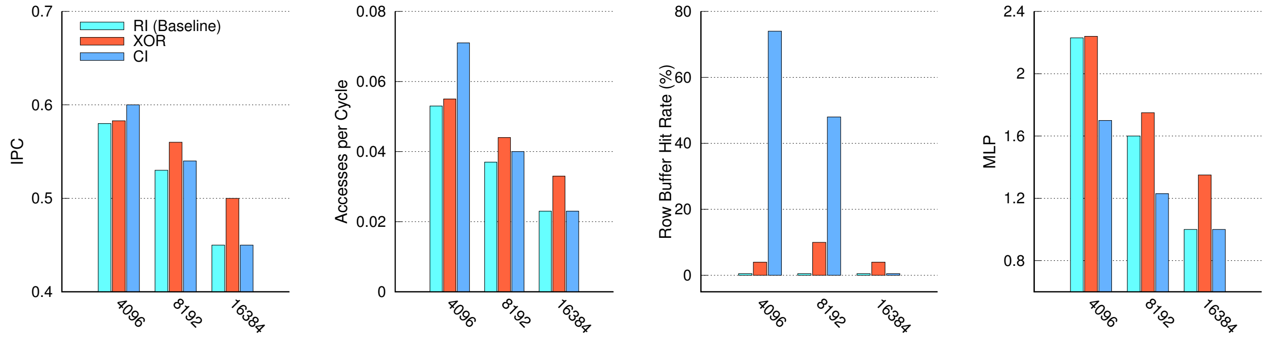
1) Locality or MLP: RI and XOR perform with a tremendously poor locality, nearly zero row buffer hit rate, but much better MLP than CI on 4096x4096 and 8192x8192, as shown in fig. 2c. Instead, CI gain much more locality than RI and XOR, but lower MLP values on the scales of 4096 and 8192, as shown in fig. 2d. It is strange that locality-(RI and XOR) and parallelism-major (CI) mappings suffer the locality and parallelism problems respectively. For 4096 and 8192 scales, as the distances between consecutive accesses on DRAM are roughly 32K and 64K, RI and XOR mappings distribute accesses across several rows with much more bank-parallelism, and CI put them on the same row with high row buffer locality. That is to say, current designed address mappings may fail to exploit their advantages because practical access patterns do not match the data layout decided by address mappings on DRAM.

2) No Perfect Mapping: XOR always gain profits from RI, especially on a large scale, which reveals that RI will introduce bank conflicts for consecutive accesses and XOR can resolve it. However, from fig. 2a and fig. 2b, they show that CI performs better than XOR on 4096x4096 and the other way around on 8192x8192. It reveals that row buffer locality dominates the DRAM performance on 4096x4096 and MLP conquers locality on 8192x8192. Neither of two address mappings is absolutely prominently better, where it depends on access patterns that vary as the matrix scale changes.

3) Worst Situation: It is noticeable that row buffer hit rates of XOR and CI are almost lower than 50%, and it becomes much lower on a large scale (fig. 2c). When it comes to 16384 scale, hit rates are all approximately zero, which reveals that recent accesses within a bank are mapped into different rows because the stride 128K is too large spanning 8 times of row buffer size, regardless of XOR or CI. MLP values of all mappings on 16384x16384 are decreasing compared to ones on small scales. With decreasing of both locality and MLP, APC and IPC degrade a lot. It reproduces the worst situation we mentioned in section I, where locality and MLP are both harmed. Intuitively, it is beneficial if accesses are compacted on one row or distributed onto different banks.

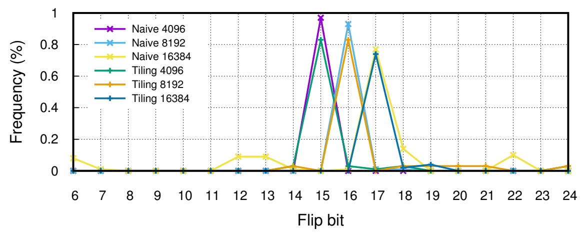
B. Flip on High Bits

In order to observe behaviors of regular access patterns like GEMM, we propose an approach named *Flip Sampling* that approximately illustrates the distance of consecutive accesses, with more details in section IV. In a word, flip sampling is frequency statistics of occurrences for value 1 on every bit of the binary format of distances, and high frequent bit flips reveal outstanding spatial distance of accesses. We use Flip Sampling to analyze three versions of GEMM on the scales of 4096, 8192 and 16384 in fig. 3. It reveals that almost all bit flips happen on a specific bit for naive and tiling version, as shown in fig. 3a. For example, bit 16 is the highest frequent (near 100%) flipping bit in naive and tiling version of 8192x8192, since the distance between adjacent accesses is exactly 64KB (2^{16}). Except for the specific bits, there are

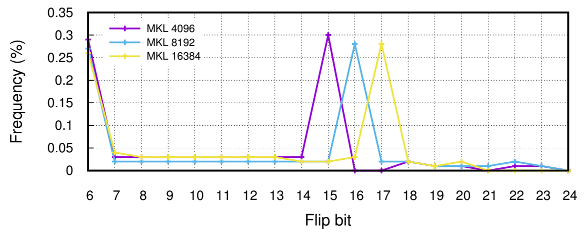


(a) IPC (b) APC on DRAM (c) Locality on DRAM (d) MLP on DRAM

Fig. 2: DRAM Performance Profile for Tiling GEMM on 4096x4096, 8192x8192, 16384x16384. (a) Instructions per Cycle (IPC) , (b) Row Buffer Hit Rate as Locality, (c) Memory-level-parallelism (MLP), (d) Accesses per Cycle (APC).



(a) Bit Flips of Naive and Tiling



(b) Bit Flips of Intel MKL

Fig. 3: Frequency of Bit Flips of GEMM on 4096x4096, 8192x8192, 16384x16384. (1) Naive and Tiling, (2) Intel MKL.

almost 0% flips on the other bits, which means access patterns for naive and tiling are perfectly regular.

For the high-optimized Intel MKL GEMM in fig. 3b, only 25%-30% flips on bit 6 and specific bits (15, 16 and 17), indicating that locality is improved a lot compared to naive and tiling GEMM. In addition, it reveals that adjacent accesses have a large possibility of falling into the same row. However, lots of bit flips still happen on bit 17 for 16384x16384 like tiling one, and adjacent accesses may fall into different rows within the same bank, which is the worst case like 16384x16384 in fig. 2.

For scales of 4096, 8192 and 16384, bit flips on the high zone that is near the row bits of address mappings, which may trigger the row conflicts and degrade DRAM performance tremendously. Both locality- and MLP-major address mappings lose efficiency, as adjacent accesses with long distance may hinder their advantages exactly.

C. Opportunities

That is to say, locality and parallelism are affected by the combination of access patterns and data layout on DRAM, which is decided by address mappings. It concludes that there is a big gap between access patterns and data layout, and this gap introduces long access latency on DRAM, possibly degrading the end-to-end performance. We demonstrate that the state-of-the-art mapping mechanism cannot adaptively resolve the mismatch between the data access and the data layout on DRAM. It gives us potential and opportunities to propose an adaptive address mapping mechanism for access pattern awareness, to cover these problems.

IV. Design

Based on those observations above, we propose an adaptive and pattern-aware memory address mapping mechanism. Firstly, section IV-A gives an overview workflow of the design and highlights the core modules that enforce our mechanism. Then, section IV-B and IV-C present details about the design and its considerations. Afterward, aggressiveness control is shown in section IV-D. Finally, the implementation and the overhead is given in section IV-E and IV-F.

A. Overview

To enforce our adaptive mechanism, we only add two simple tags (@*init* and @*proc*) into the source code taken GEMM as a case, as shown in fig. 4 to distinguish the initial and the computational procedures in compiling and runtime. These two tags will be preprocessed into the calls for cooperating with a loader called *Ctrl Loader* in runtime. Ctrl Loader helps to control the progress of execution through *exec* subprocess and communicating with it. The cost of initialization, indicated by the interval between @init and @proc, can be recorded by Ctrl Loader.

To make the design more flexible and avoid interference from other processes, we use an exclusive DIMM in fig. 4 to support the execution. In the first execution round, when

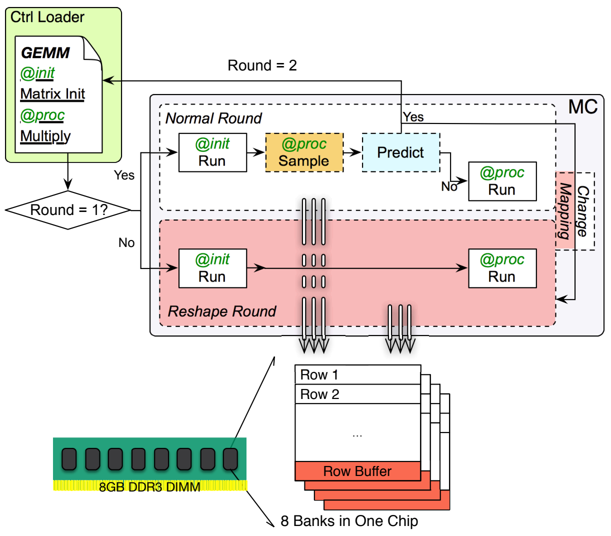
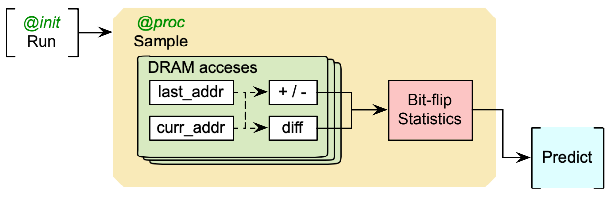
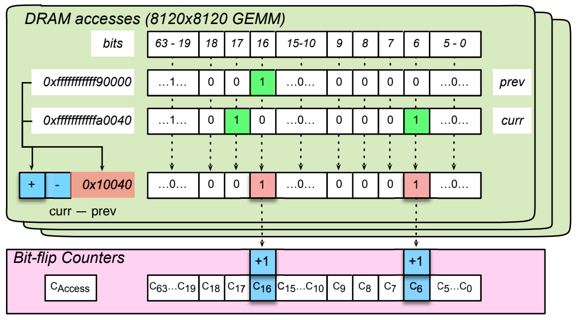


Fig. 4: Design Overview



(a) Sampling Module Design



(b) Implementation of Sampling and Example

Fig. 5: Flip Sampling

encountering @proc, right after initialization, Ctrl Loader informs MC of the computation start, which means the sampling can begin after a short period of warming up. In the sampling stage in fig. 4, accesses are collected and analyzed until a condition is converged, which means the access pattern has been estimated. After that, MC may predict a more efficient mapping based on the policy we prefer and change the address mapping of the specific DIMM to this new one. Simultaneously, MC will throw a hardware fault to CPU to notify Ctrl Loader should kill the current subprocess and then restart a new one again. If there is no better mapping at all, the execution progresses in a normal way and finally return to the Ctrl Loader.

When Ctrl Loader receives the fault signal, it will *exec* the subprocess (here is GEMM) in another round and bind it with the same DIMM, so that the whole execution, including initialization and computation, accesses the specific DIMM with much more efficient changing address mapping. After normal returning from the subprocess, the execution is finished and Ctrl Loader then recovers the specific DIMM to the default address mapping.

B. Flip Sampling

As the access pattern is regular in GEMM, a period of accesses can represent the whole computation. When the initialization of matrices finishes, i.e. meeting with @proc tag, the Ctrl Loader emits a *Sample* signal to notify MC that computation begins. After skip 4096 accesses for warming up, the sampling starts in a window unit of 4096 accesses. The sampling does not stop until the statistics converge a condition that the access pattern we collect keeps stable.

As shown in fig. 5a, the sampling module is composed of two main components: difference calculation, bit-flip statistics. Once the sampling begins, every single access is collected and their physical addresses are processed in sequential order. MC stores the previous access and the current one, and calculate the difference of their addresses for every new access, i.e. the distance of two accesses. An extra sign register indicates whether the difference is positive or negative to avoid involving the access thrashing [7]. Subsequently, the bit-flip statistics updates the number of sampling accesses in total and fliping among the 64 bits from the beginning of sampling, shown in an example (8200x8200 in fig. 5b). One counter of a bit may be updated by $+1$ or -1, which is decided by the signed register is $+$ or $-$.

At the end of every sampling window, it calculates the percentage of the number of flips on bit i based on the total sampling accesses from the very beginning of sampling, notated as ρ_i, $0 \leq i < 64$. Then the sampling *convergence condition* is as follows,

$$Convergence\ Condition: \ \forall i(\rho_i > \lambda) \rightarrow |\rho_i - \rho_i'| < \sigma.$$

ρ_i and ρ_i' indicates the percentage of bit i at the end of current sampling window and previous one. λ and σ are thresholds for frequency of bit flip and interval of convergence, respectively. λ is used to leave the prominent bit flips and filter out neglectable ones. The two parameters are discussed in VI-B, and the default values are 0.15 and 0.01.

Sampling stops until it satisfies the convergence condition, which means the regular access pattern has been extracted in a representation of bit flips. The *access pattern* can be depcited by the following ordered set,

$$Pattern: \{P_i \mid P_i = 1 \ if \ \rho_i > \lambda \ otherwise \ 0\}.$$

C. Pattern-aware Prediction

As presented in section III-A, there is no perfect address mapping that fits all regular access patterns. Like RI, XOR, and CI, one performs better than another, or the other way around in different scales. More importantly, for the high bit-flip shown in section III-B, RI, XOR and CI gain more row

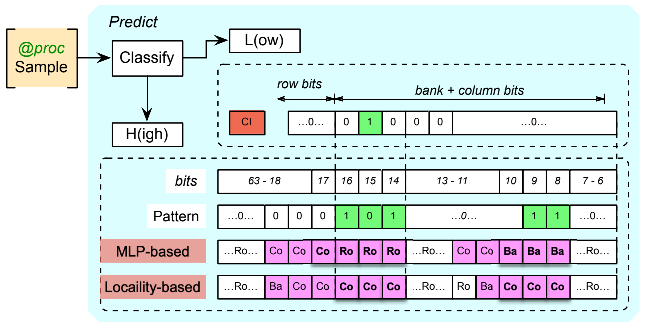


Fig. 6: Prediction Design

conflicts and reduce bank parallelism, since the distance is so far away that two accesses span one or more rows.

The basic idea is to swap the lower bits of banks or columns to row bits of frequent flips, covering inefficiency of locality and MLP due to long distance of two adjacent accesses. However, when bits of frequent flips are distributed randomly, or the number of them is near the number of bank and column bits, the idea can not easily resolve the problem. To address the problem, we propose a strategy-based approach to give a more efficient address mapping for different scales in some conditions.

As shown in section III-B, frequent flips on high bits is a remarkable indicator that the parallelism and locality may be both violated, i.e. the worst situation. Firstly, we classify access patterns with frequent flips into two categories: $\mathbb{H}$(ign) and $\mathbb{L}$(ow). The former owns frequent flips on high bits, otherwise belongs to $\mathbb{L}$. We use an empirical value bit 15 (starts from bit 0) in our testbed, as the boundary between $\mathbb{H}$ (exclusive) and $\mathbb{L}$ (inclusive). As presented in fig. 3, for three versions of 8192x8192 GEMM, their patterns belong to $\mathbb{H}$. However, the pattern of naive 4096x4096 GEMM belongs to $\mathbb{L}$.

As prominent flipping bits are in the lower zone for $\mathbb{L}$, the locality and MLP are not degraded simultaneously. In addition, DRAM performance plays relatively less dominant role in the whole execution when the scale is small. We only choose which one is better between XOR and CI, which is simply decided by the distribution of prominent bit flips in the pattern. If the prominent flips gather together at more significant bits, CI is better than XOR as locality can be better and locality is much important than bank-parallelism in this situation, as shown in fig. 2.

As $\mathbb{H}$ is much more complex, we give two strategies considering how to arrange bank or column bits to achieve better locality or MLP. Locality-based strategy is straightforward and illustrated in an example fig. 6. Columns bits (*Co*) are just placed on the prominent flip bits and bank bits (*Ba*s) can be placed higher near the *Co*s if *Co*s are not enough to cover the flip bits, which means at least we can leverage bank parallelism when two consecutive accesses visit different rows.

The essence of MLP-based strategy is to put all *Ba*s together on the most prominent bits, and then arrange *Co*s onto a zone which is 3 bits higher than the other uncovered prominent bit flips. For example in fig. 6, the three *Ba*s are put on bits 8, 9 and 10, implying that the sequential 8 accesses are distributed on all banks evenly, so there does not exist any bank conflicts at all. Meantime, the *Co*s are placed in two zones starting from bits 11 and 17 respectively, so that the row buffer locality for every 8 accesses (in the same bank) is not zero. Obviously, the more extra *Co*s are used in the aforementioned two zones, the better row buffer locality is. It is fantastic that the MLP-based strategy saves the precious resource of *Ba*s and *Co*s and utilize them to improve MLP and locality as well.

The MLP-based strategy is not better than locality-based one all the time, because it assumes that accesses into the same bank have great locality, and sometimes the probability can not satisfy the strict requirement. We will show the comparison in section VI-B.

Here there is a limitation of our approach, if the number of prominent flip bits is near or more than the number of sum of bank and column bits, it is hard to rearrange data on DRAM. In such a condition, we just continue the execution with the default mapping, i.e. XOR.

D. Aggressiveness Control

As the pattern decided by sampling is used in the predication, the convergence condition of the pattern affects the speed of convergence and precision of the pattern. λ presents how much frequency of bit flips is worthwhile devoting much effort to the access pattern. In the pattern study of GEMM, we found ρ_i shows tremendous difference between each other (in fig. 3), so the boundary of bit flip frequency is clear for every single pattern. We choose 0.15 as the default value of λ, as it can filter out bits of lower frequent flips. The window unit of sampling is fixed in 4096 accesses, and how many window units it needs to get to convergence is also decided by σ, whose default value is set to 0.01, incurring a much high precision of the pattern.

MLP-based strategy reveals better than locality-based one from the mentioned design considerations, but it shows worse performance in a lot of cases. The sensitivity experiments of two parameters and predication strategies are discussed in section VI-B.

E. Implementation

The proposed design is implemented from two layers, i.e. the programming level and the hardware level. We don't change a lot on software including OS and library, except adding two lines of tags and registering the interrupt handler in OS. The two tags, *@init* and *@proc*, are introduced for identifying the program parts semantically, which will be replaced by two functions in preprocessing before being compiled to the binary file. Ctrl Loader uses process communication with the GEMM subprocess to control process spawn and its progress.

As shown in fig. 5, 67 registers including one for the previous address, one for difference sign, one counter for sampling accesses, and 64 counters for bit flips on 64 bits. Address mapping in the logic layer of MC needs a little change to support the arbitrary mapping. Three registers of masks for bank, row, and column can be used to map a physical address to corresponding positions. When a predication is done, they are written with new masks. Meantime, the MC will trigger an interrupt onto a specific pin for notifying CPU that the mapping is changing, and the OS kernel will record it for loop checking of Ctrl Loader.

F. Cost Model

We use *Init, Init'* and *Proc, Proc'* notations to distinguish steps in the first round or the second one (reshaping round). The cost of normal execution using static address mapping is

$$T_{static} = T(Init) + T(Proc).$$

For the adaptive address mapping, as the execution may enter the reshaping round after sampling, the whole cost is

$$T_{dynamic} = T(Init) + T(Sample) + T(Init') + T(Proc').$$

As our mechanism may find a potential and well-suited address mapping to achieve better performance on @proc part in the reshaping round, so we struggle to make it that $T(proc')$ is much less than $T(proc)$. However, we introduce extra overhead, i.e. the sum of $T(Sample)$ and $T(Init')$. As is shown in section VI-B, it is fast to converge a stable pattern in a small sampling window with a neglectable cost, so the overhead can be inferred to,

$$Overhead = T(Sample) + T(Init') \approx T(Init').$$

As is shown above, the whole overhead comes from two sides, sampling cost and the second initialization cost, so if $T(proc) - T(Proc') > Overhead$, we can speed up the execution. The *profit* from our design is defined as the reduction percentage of execution time based on static mapping, taking the overhead into consideration, i.e.

$$Profit = 1 - T_{dynamic}/T_{Static}$$

. It can be transformed to the following formula, in which α is the fraction of initilization time based on the end-to-end excution time using static mapping, i.e. $\alpha = T(init)/T_{static}$.

$$Profit \approx \frac{T(Proc) - T(Proc')}{T_{staic}} - \alpha * \frac{IPC(Init)}{IPC(Init')}.$$

Theoretically, α is inversely proportiona the matrix scale for GEMM [27]. We test scales in group A in table III, and find that α is nearly zero for naive and tiling versions of GEMM, and is 0.02 or less when matrix size is over 8192 using Intel MKL. As matrix initialization accesses the data in a stream and there is no data reuse, the mappings we consider in the paper perform similarily, so that $IPC(Init)/IPC(Init')$ is approximate to 1.

$$Profit(GEMM) \approx 1 - \frac{T(Proc')}{T_{staic}} \approx 1 - \frac{IPC(Proc)}{IPC_{Proc'}}.$$

As mentioned before, $T(Sample)$ is neglectable, so the overhead for one more initilization in reshaping round can be ignored compared to the whole execution, we conclude a simple cost model for GEMM above.

TABLE II: Testbed Configuration

Processor	One core, x86-64, 4-wide issue and 3-wide retire, 1.8 GHz, out-of-order
L1 cache	32 KB D-cache, 32 KB I-cache, 8 MSHRs, LRU policy, 4 cycles
L2 cache	128 KB, 16 MSHRs, LRU policy, 4 cycles
LLC cache	4 MB, 48 MSHRs, LRU policy, 15 cycles
MC	FR-FCFS, 16 KB row size, 400MHz, 32-entry queue, open-page policy
DRAM	DDR3-2133, 1-channel, 1-rank, 8 banks 2^{16} rows, 2^{11} columns, 8B bus

TABLE III: Benchmark Groups.

Groups	Matrix size	Scales
A	2^N. E.g., 8192.	8192, 16384, 32768
B	Tailing zeros. E.g., 24576 (1100...0)	24576, 28672, 30720, 31744, 32256
C	1000x and random zeros. E.g., 10000 ; 17160 (100001100001)	10000, 16000; 17160, 18472

V. METHODOLOGY

Testbed and Method. All the experiments in the paper are evaluated on our testbed, which is based on Champsim [38] and Ramulator [39] as the out-of-order CPU and the cycle-accurate DRAM simulator respectively (table II). For simplicity, we choose DDR3 instead of DDR4, as DDR4 has one more layer bank-group, which is a higher parallel layer and provides more parallelism possibility than banks [1] and it does not affect the insight and the idea in this paper. We use pintool [40] to collect the full instruction traces for every benchmark to verify the design workflow and performance improvement. As we tag the full trace into two parts, i.e. initialization and matrix multiplication, we can evaluate the $IPC(Init)$ and $IPC(Proc)$. Besides, the testbed digests an adequate and representative period of traces, since a part of execution is enough to depict the access pattern and benefits after reshaping. Although the whole accesses of large matrices, like 32768, cannot be served by only one DIMM, the access data in a representative piece evaluated in our experiment can be contained in one DIMM. Therefore, the access pattern and benefits we present can be extended as the same if multiple DIMMs are used.

Benchmarks. The self-adaptive mechanism can support general applications with regular patterns. For fast verification, we choose the GEMM as it has a clear boundary of initialization and computation, and its initialization complexity can be ignored compared to computation. Besides, the cost model for GEMM can be simplified to the IPC improvement of computation shown in section IV-F. Our design supports arbitrary matrices, and we choose the square matrix with scales more than 8192 for benefits exposing. We classify different scales of benchmarks into three groups based on the number for elements of a square matrix, i.e. matrix size. All sizes of matrices in group A are the power of two, from 2^{10} to 2^{14}, which is perfectly regular. A matrix in group B is weakly skewed, the binary format of size has more than half tailing zeros, so intermediate cache conflicts will have much pressure on DRAM and data reshaping will get benefits. The binary format of matrix size in group C is trivial, i.e. zeros are randomly distributed. Particularly, group C includes 1000x, whose decimal format is thousands, such as 10000 and 16000.

Metrics. The DRAM performance is evaluated by accesses per cycle (APC) [36]. In addition, MLP [35] and locality on DRAM are defined as follows.

$$APC(DRMA) = \frac{\#Accesses}{\sum latency}$$
$$MLP(DRAM) = \frac{\sum latency}{Cycle_{active}}$$
$$Locality = \frac{\#Hits}{\#Accesses}$$

$Cycle_{active}$ is the working cycles while processing accesses, excluding the idle cycles on DRAM.

VI. Results

A. Performance Improvement

As XOR improves MLP in some situations without any harm to locality compared to RI [7], and it has been adopted by the commodity memory controller [6], we choose XOR as the baseline in the following experiments. We evaluate naive and tiling versions of all benchmarks, and Intel MKL version of the group A (table III). Besides, two strategies for the pattern-aware predication are analyzed over all cases.

As shown in fig. 7, two strategies improve DRAM performance on almost all the cases, whose normalized APCs are almost above 1 except MLP-based strategy on Intel MKL. APCs are upgraded by 2.1x and 1.4x averagely for naive and tiling versions on MLP-based strategy (fig. 7a and fig. 7b), but it reduce APCs on Intel MKL version of group A benchmarks (fig. 7c). For locality-based strategy, APCs are promoted by 1.9x, 1.7x and 1.6x averagely for three versions. Overall, memory performance of three versions is improved up to 2.1x, 1,7x and 1.6x, by utilizing locality or MLP, or both of them.

As DRAM performance is improved, the whole execution is also accelerated and profits (reduction percent of execution time) of benchmarks are shown in fig. 8. For group A benchmarks, the locality-based strategy reveals 22%, 8%, and 7% profits averagely over three versions, but the MLP-based one shows 20% performance degradation on Intel MKL version (in fig. 8c). Two strategies show similar profits for group B benchmarks, i.e. 24% and 6% profits on average over native and tiling versions. In particular, the execution time of 30720x30720 among group B benchmarks is decreased by 44% and 16% on native and tiling versions. Besides, for group C benchmarks, MLP-based strategy presents 11% and 5% profits and locality-based one presents 8% and 6% profits over native and tiling versions. To sum up, the execution time is decreased by 24%, 8%, and 7% averagely on three versions respectively.

B. Sensitivity Study

In order to enforce a precise predication, a representative access pattern as an input plays an important role for reshaping. To effectively converge on a specific pattern for different scales and versions of GEMM, two parameters in sampling can be controlled. Also, MLP and Latency-based strategies perform well in a different situation, so there is no perfect one defeating another. Next, we show some empirical results and give a piece of guidance on how to decide choices.

1) Sampling and Pattern: The convergence condition mentioned in seciton IV-B affects speed of convergence and precision of the pattern. λ presents how much frequency of bit flips is prominent to the access pattern and σ affects the speed of reaction. As is shown in fig. 9 over the tiling version of 30720x30720, patterns based on flip sampling are approximate to the ideal pattern that extracted from whole accesses. Although it is less precise if σ is 0.01 instead of 0.001, it is fast to converge with lower cost and the prominent bit flips are revealed as almost the same as the ideal pattern. The different effects for λ between 0.15 and 0.03 present significance of bit flips, such as the lower frequency of bit flips on bits 19 and 20 as shown in fig. 9. The frequency below 0.1 is so low compared to the prominent bit flips so that we can neglect it. As shown in fig. 7 and fig. 8, DRAM performance and the whole execution are accelerated a lot, with the default value of λ and σ as 0.15 and 0.01 respectively.

2) MLP or Latency-based strategies: Theoretically, MLP-based strategy is better than locality-based one, as the former can utilize MLP, and save the column bits for the better locality as well. Practically, it may be worse than locality-based one, as sometimes the frequency of bit flips can not satisfy the strict requirement mentioned before. As shown in fig. 7c and fig. 8c, MLP-based strategy behaves worse than locality-based one, even worse than XOR mapping, as the frequency of bit flips are under 30% (in fig. 3b) so that accesses within one bank may drop into different row buffers with high probability, i.e. low row buffer locality.

Aggressively, MLP-based strategy is preferred when the frequency of prominent bit flips is very high, especially above 80%. However, we prefer locality-based one as a conservative mapping predication empirically.

VII. Related Work

To our knowledge, this is the first work using adaptive address mapping without any prior knowledge about access patterns, to exploit both locality and parallelism on DRAM. In this section, we compare our work to prior ones on DRAM performance, address mappings and co-design with access patterns.

DRAM Performance. Scheduling policy is a popular approach to resolve DRAM performance [21]–[24]. A smart scheduler based on previous scheduling decisions and results are proposed in [22]. The machine learning approach is utilized to overcome the static scheduling policy [23]. However, scheduling only can dispatch accesses in a small window, like 32-64 accesses, which can not avoid the inefficiency

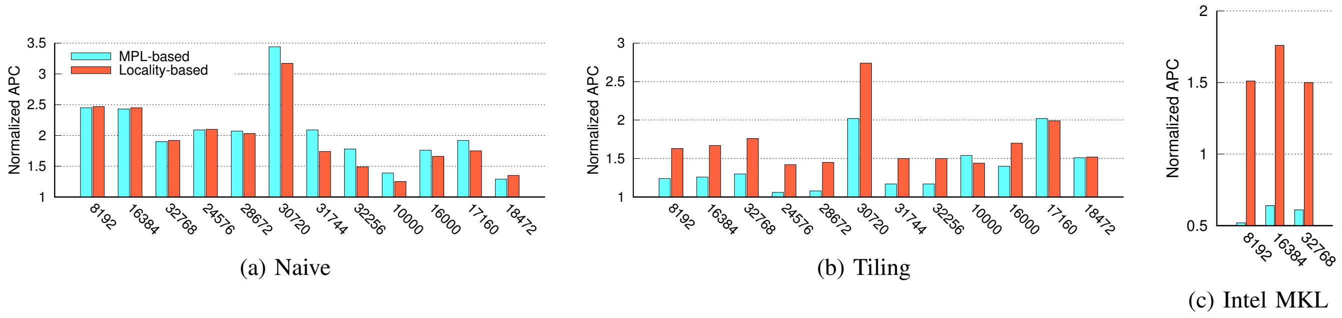


Fig. 7: DRAM Performance Improvement: Normalized APC Based on XOR. (a) Naive (b) Tiling (c)Intel MKL.

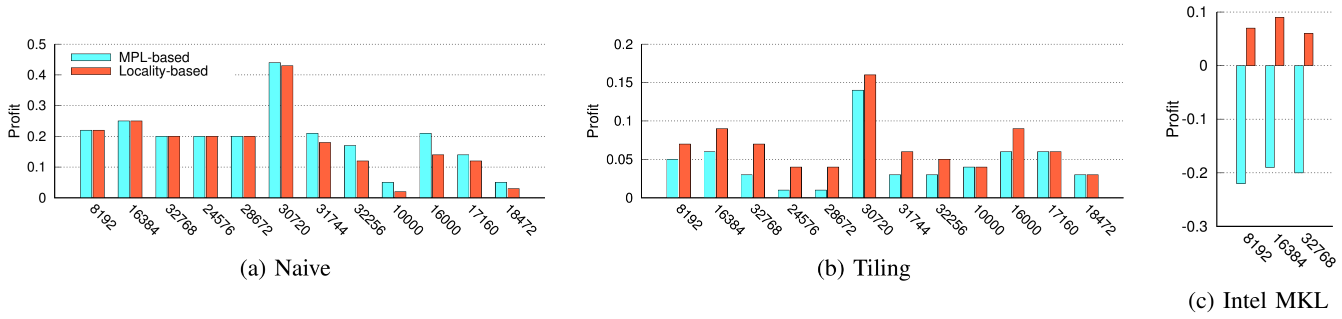


Fig. 8: Profit: reduction percent of execution time (based on XOR). (a) Naive (b) Tiling (c)Intel MKL.

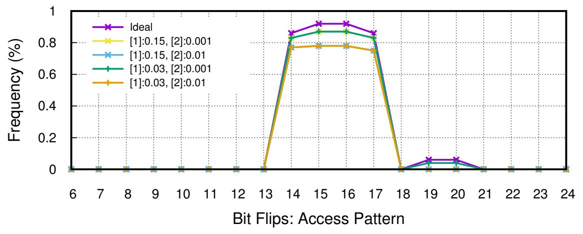


Fig. 9: Sensitivity on Access Pattern Recognition Based on Flip Sampling (30720x30720 as an example). [1]-λ. [2]-σ.

introduced by terrible data layout. Prefetch is adopted to improve MLP [25]. Another promising approach is to modify the DRAM organization [12], [15]–[20]. Parallelism potentials of the subarray, bank and rank are exploited through little changes on DIMMs [15]–[17]. To reduce the data movement cost over the bus between CPU and DRAM, [19], [20] support fast data transfer directly on DRAM. However, these studies do not extract access patterns and address directly the mismatch of data layout and access patterns. Such approaches can be absorbed into our mechanism as an alternative for reshaping instead of restarting the process, as future work.

Address Mapping. Although the address mappings in commodity CPUs are not public [6], there are a lot of address mappings studied in [3]–[7], involving the row-interleaving one (*RI*, fig. 1a) and the cache-interleaving one (*CI*, fig. 1c), which are two typical ones representing different parallelism granularities of access patterns. [7] proposes a general address mapping mechanism with simple XOR operations, addressing row buffer thrashing introduced by LLC conflicts. It preserves spatial locality compared to row-interleaving mapping and upgrades MLP in some cases, for example, reduction of two large vectors. Prior work considers swap bits to cover the conflicts from LLC [5], but it does not consider any knowledge of access patterns. Besides, the above work is all static so that they can not fit the different access patterns, which give us opportunities to present a self-adaptive mechanism to suit patterns. [8] presents a mathematical framework to reshape data in 3D-stack DRAM through data movement, as 3D-stack memory provides high-bandwidth data transfer. It works as a PIM and provides fast data transformation compared to moving data through CPU and DRAM. But the mapping mechanism is not self-adaptive and pattern-aware without prior knowledge of applications, as upper software should notify how to transform data layout. We can deploy our mechanism on the mathematical framework and enable 3D-stack memory aware of access patterns.

Access Pattern Awareness. Hot pages are compacted into a row buffer for better spatial locality [10], which changes OS page management and does not support access patterns without critical data regions, like GEMM. DRAM Chips are utilized to address the non-unit stridden accesses in a gather-scatter way [12], which is limited to small stride less than a cache line. [14] proposes a MLP-based scheduling targeting low-level parallelism of irregular access patterns. Some studies are customized for specific applications [21], [26] without accessing recognition in our work.

VIII. Conclusion and Future Work

In this paper, we present an adaptive address mapping mechanism to be aware of access patterns, bridging the huge mismatch between access patterns and data layout on DRAM. This mechanism is adjustable to different access patterns by adopting suitable mappings to gain either locality or bank parallelism, compared to current general address mappings.

We make the key observation that the inefficiency is coming from the mismatch of access patterns and data layout. The adaptive address mapping can avoid the worst case that both locality and parallelism are harmed. The evaluation results based on GEMM show that our mechanism can improve the DRAM performance several times and reduces execution time even for high high-optimized GEMM.

We have shown early benefits of the adaptive mechanism on different scales of matrix multiplication. It is a pioneering work to fill the gap between access patterns and data layouts. We plan to extend our work in two aspects: one is to dig more profit from other applications with regular patterns; the other is to exploit efficient data movement in 3D-stack DRAM to support fast reshaping on runtime after predicting a suitable mapping.

IX. Acknowledgement

We thank Dr. Xuechen Ding for assistance with writing and ploting that greatly improved the manuscript. We would also like to show our gratitude to the anonymous reviewers for their sincere comments.

References

[1] JEDEC Solid State Technology Association. JEDEC Standard: DDR4 SDRAM. JESD79-4, Sep, 2012.

[2] M. Blackmore. A quantitative analysis of memory controller page policies. 2013.

[3] https://safari.ethz.ch/architecture/fall2018/doku.php.

[4] R. E. Kessler, M. B. Steinman, P. J. Bannon, M. C. Braganza, and G. A. Bouchard. U.S. Patent No. 6,546,453. Washington, DC: U.S. Patent and Trademark Office, 2003.

[5] J. H. Zurawski, J. E. Murray, and P. J. Lemmon. The design and verification of the AlphaStation 600 5-series workstation. Digital Technical Journal, 7(1), 0, 1995.

[6] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. USENIX Security, 2016, pp. 565-581.

[7] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. MICRO, 2000, pp. 32-41.

[8] B. Akin, F. Franchetti, J. C. Hoe. Data reorganization in memory using 3D-stacked DRAM. ISCA, 2016, 43(3), 131-143.

[9] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. ISCA, Vol. 28, No. 2, pp. 128-138, 2000.

[10] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. ISCA, 2010, 45(3), pp.219-230.

[11] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu. ChargeCache: Reducing DRAM latency by exploiting row access locality. HPCA, 2016, pp. 581-593.

[12] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, et al. Gather-scatter DRAM: in-DRAM address translation to improve the spatial locality of non-unit strided accesses. MICRO, 2015, pp. 267-280.

[13] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. PACT, 2012, pp. 367-376.

[14] X. Tang, M. Kandemir, P. Yedlapalli, and J. Kotra. Improving bank-level parallelism for irregular applications. MICRO, 2016, pp. 57.

[15] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A case for exploiting subarray-level parallelism (SALP) in DRAM. ISCA, 2012, 40(3), pp. 368-379.

[16] Y. H. Son, O. Seongil, Y. Ro, J. W. Lee, Ahn, J. H. Reducing memory access latency with asymmetric DRAM bank organizations. ISCA, 2013, pp. 380-391.

[17] W. Shin, J. Jang, J. Choi, J. Suh, Y. Kwon, Y. Moon, and L. S. Kim. Rank-level parallelism in dram. TC, 66(7), pp.1274-1280, 2017.

[18] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-latency DRAM: A low latency and low cost DRAM architecture. HPCA, pp. 615-626, 2013.

[19] K. Chang, P. Nair, D. Lee, S. Ghose, M. Qureshi, and O. Mutlu. Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. HPCA, 2016, pp. 568-580.

[20] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, et al. RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. ISCA, 2013, pp. 185-197.

[21] S. Rixner. Memory controller optimizations for web servers. MICRO, pp. 355-366, 2004.

[22] I. Hur, C. Lin. Adaptive history-based memory schedulers. MICRO, 2004, pp. 343-354.

[23] E. Ipek, O. Mutlu, J. F. Martnez, R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. ISCA, 2008.

[24] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, et al. Parallel application memory scheduling. MICRO, 2011, pp. 362-373.

[25] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving memory bank-level parallelism in the presence of prefetching. MICRO 2009, pp. 327-336.

[26] M. Berezecki, E. Frachtenberg, M. Paleczny, K. Steele. Many-core key-value store. IGCC, 2011, pp. 1-8.

[27] K. Goto, and R. A. Geijn. Anatomy of high-performance matrix multiplication. TOMS, 34(3), 12, 2008.

[28] J. Kepner, and J. Gilbert (Eds.). Graph algorithms in the language of linear algebra. Society for Industrial and Applied Mathematics. 2011.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, et al. Going deeper with convolutions. CVPR, 2015, pp. 1-9.

[30] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. CF, 2006, pp. 9-20.

[31] Benchmarking matrix multiplication implementations. https://github.com/attractivechaos/matmul.

[32] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. High-performance computing on the Intel Xeon Phi. Springer, 5, 2. 2014.

[33] J. Zhao, H. Cui, Y. Zhang, J. Xue, and X. Feng. Revisiting Loop Tiling for Datacenters: Live and Let Live. ICS, 2018, pp. 328-340.

[34] Y. Jia. Learning semantic image representations at a large scale (Doctoral dissertation, UC Berkeley). 2014.

[35] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. ISCA, 2004, pp. 76-87.

[36] D. Wang, and X. H. Sun. APC: A novel memory metric and measurement methodology for modern memory systems. TC, 2013, 63(7), pp.1626-1639.

[37] Y. H. Liu, and X. H. Sun. LPM: A Systematic Methodology for Concurrent Data Access Pattern Optimization from a Matching Perspective. TPDS, 2019.

[38] ChampSim. https://github.com/ChampSim/ChampSim.

[39] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible DRAM simulator. IEEE Computer architecture letters, 2015, 15(1), pp.45-49.

[40] C. K., Luk, R., Cohn, R., Muth, H., Patil, A., Klauser, G., Lowney, et al. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, Vol. 40, No. 6, pp. 190-200, 2005.