3D-DRAM Performance for Different OpenMP Scheduling Techniques in Multicore Systems

Shashank Adavally
Department of Computer Science and Engineering
University of North Texas
Denton, TX, USA
Email: ShashankAdavally@my.unt.edu

Krishna Kavi
Department of Computer Science and Engineering
University of North Texas
Denton, TX, USA
Email: Krishna.Kavi@unt.edu

***Abstract*—Advances in memory technologies including 3D-DRAM memories (such as High Bandwidth Memory (HBM) and Hybrid Memory Cube (HMC) systems), wide I/O memory promise very large bandwidths at lower power consumption to address the needs of high-performance computing as well as emerging big data applications. However, in order to fully benefit from such bandwidths, it is necessary to understand how to optimally organize data across channels, ranks, banks or vaults of the memory structures, how to obtain large volumes of data with fewer accesses and how to schedule threads of multi threaded applications to benefit from these memory organizations. In this paper, we will examine different memory organizations that spread data across channels, ranks, and banks and identify application features that benefit from different organizations. Our study applies to generic DDR memory structures as well as 3D-DRAMs. We will also evaluate scheduling of OpenMP threads (e.g., using static, dynamic and guided) but with emphasis on how different scheduling methods benefit from different memory organizations. Using the best scheduling for the application, proper memory organization, our experiments show, we can achieve up to 16 percent performance gains depending on workload.**

***Index Terms*—OpenMP; Scheduling; Memory Organization; HBM**

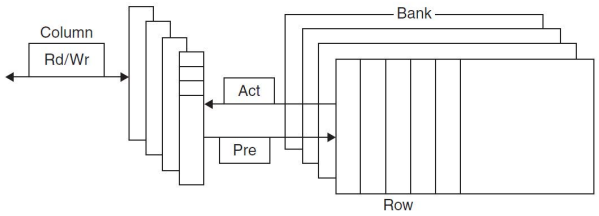

Fig. 1. Row Buffer

```
for (int i = 0; i < 160; ++i) {
    for (int j = 0; j < 160; ++j) {
        for (int k = 0; k < 160; ++k) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Fig. 2. Matrix Multiplication example

I. Introduction

With the introduction of 3D-DRAMs, and wide I/O memories [1] processor-memory bottleneck has been somewhat mitigated. There has been a significant amount of research done on 3D-DRAM usage as a last level cache (LLC) or as part of main memory (see for example [2], [3] [4] [5] [6] [7] [8] [9] [10]). In this paper, we assume 3D DRAM is used as main memory (either alone or in combination with other memories). Our goal is to investigate how the bandwidth of such memories can be utilized by applications. While the two commercially available 3D-DRAM designs, HBM (for example see [11], [12]) and HMC [13] differ in some design details, they still rely on organizations that are similar to DDR technologies. A DRAM-based memory is organized into banks and each bank consists of rows of data as shown in Figure 1 [14].

Larger memories may be organized in banks, bank groups (or vaults in HMC) and ranks, and data may be accessed using multiple channels. It is important to understand the different organizations since they impact memory access latencies and effective bandwidth. Access times (or latencies) involve the activation of a row (which may involve selection of banks and deactivation of previously open rows), accessing the data from the row and then selecting an appropriate amount of data, typically equal to a cache line, from the row, often known as column select. Access latencies depend on if the access falls to the same row as the previous access (in which case there is no need to close current row and activate a new row) or falls to a new row. Consider for example memory accesses of matrix multiplication in Fig 2. We are aware that there are many more efficient implementations of matrix multiplication, but we use this simple algorithm for illustration purposes. The innermost loop will access elements of one matrix row-wise and the element of the second matrix column-wise. These accesses are likely to fall to different DRAM rows and thus cause excessive access latencies. Large caches can alleviate some of these latencies since future accesses may be satisfied by previously cached data. It is still important to understand how data is organized in DRAMs and how an application requests data. In this paper, we will explore the impact of different ways for spreading memory rows across banks, ranks, and channels and

the resulting memory access performance.

In case of multithreaded applications executing on multicore systems, the memory performance depends on how the different threads access data (and how the data is spread across DRAM banks, ranks and channels). The accesses may negatively impact memory latencies if the accesses from different threads fall to different rows. The access behaviors of multithreaded applications, particularly those written using OpenMP [15], can be controlled by deciding on how loop iterations (when using *pragma omp parallel for*) are assigned to threads. For example, consecutive iterations may be assigned to the same thread (chunk scheduling) or cyclically assigned to different threads (cyclical scheduling). And these different approaches can cause threads to access different rows or the same row. In this paper, we will explore how OpenMP scheduling impacts memory access performance when using DRAM OR 3D-DRAM devices.

There have been many research studies on Near-Data Processing (NDP) or Processing In Memory (PIM) (see for example [16] [17] [18] [19] [20] [21]) approaches whereby computation is moved closer to memory to overcome long memory latencies and utilize large bandwidths. When processing elements are embedded in the logic layer of a 3D-DRAM, it may be possible to obtain larger amounts of data on each access (instead of one cache line at a time) since such systems are not limited by bus widths or pin counts. On the other hand, this will be wasteful if consecutive accesses fall to different rows. Also PIM (Processing-In-Memory) elements are likely to be very simple In-Order cores with no or limited cache memories. Thus it is useful to understand if memory accesses from In-Order cores differ from those of Out-Of-Order cores and determine which thread scheduling is better for PIM designs (as compared to scheduling for host CPUs).

In this research, we investigate three related issues that impact memory performance (such as average access times): i) how DRAM rows are distributed across banks, ranks, and channels, ii) how loop iterations are assigned to OpenMP threads and how threads access data, and ii) if PIM designs can benefit from 3D-DRAM technologies.

We conduct experiments using two types of CPU cores: large Out-Of-Order cores with large multilevel caches and simple In-Order cores with only small L1 caches. The second type of cores are the likely choice for processing-in-memory implementations due to power limits on 3D-DRAM devices. As we will report in this paper, which memory organization and which thread scheduling results in the "best" memory performance depending on the type of the core used (Out-Of-Order or In-Order). *Thus, the main contribution of this paper is understanding how different memory organization, different thread scheduling methods and different types of cores play a role in applications' memory access performance.*

The rest of the paper is organized as follows. In section II, we will provide details on how DRAM memories can be organized. We will discuss how threads in OpenMP can be scheduled. This section provides an overview of our work for this paper and describes our experimental approach including the simulation environment and the set of benchmarks we used. In Section III, we will present the results of our experimentation and an analysis of the results. In section IV, we will discuss the potential extension of this work. Section V provides a brief overview of some of the related work that is aligned to this research. Finally, section VI comprises of high-level insights with a conclusion of this work.

II. Methodology

A. Address Mapping

Address mapping describes data distribution across the channels (or vaults in HMC), ranks, banks, rows, and columns in a DRAM memory. If the memory organization is RoBaRaCoCh, shown in Figure 4, then the data is distributed across channels, columns, banks, banks groups, and ranks to comprise a row. That is, a DRAM row of data is spread across the bank, bank groups, ranks, and channels - resulting in very large amounts of data per row access, for example, 256K bytes. This organization is useful if successive accesses fall to the same row, and they can be satisfied from the open row. Consider Fig 2 for example, the innermost loop of matrix multiplication that computes c[i][j] += a[i][k] * b[k][j]. In this case, access to matrix "a" will fall to the same row, however, accesses to "b" fall to different rows (if the matrices are very large). If the organization is ChRaBaRoCo, shown in Figure 3, data is distributed across a series of rows in the same bank, and thus each row is much smaller, for example, 2K bytes. Now it takes 128-row accesses to obtain the same amount of data as in the previous organization; it should be noted that access to a new DRAM row requires closing of previous row (and writing the row back to DRAM) and opening a new row. While this organization may benefit accesses to "b" matrix in matrix multiplication, it may penalize accesses to "a" matrix. Additional architectural optimizations such as prefetching may be relied upon to further optimize memory performance.

B. Types of Memory Organization

Commercial processing systems allow memory organizations based on DIMM sizes. They permit spreading data (or interleaving physical addresses) across multiple DIMMs, channels, and nodes, limiting access to a single DRAM row [22]. We feel it may be possible to utilize some of these address inter-leavings to achieve the two organizations studied in this paper. The two organizations we used represents two extreme cases of data distribution. We now provide more details about these organizations. Although we did not specifically propose organizations covering different 3D DRAMs (such as vaults), the two organizations evaluated are aimed at exploring two extreme cases, one providing very large rows covering all channels, banks, vaults, etc., while the other has very limited row sizes contained within a bank (or within a vault).

1. ChRaBaRoCo:

This is a typical organization where there is no channel level parallelism. We selected this organization since it does not possess any kind of bank or channel level parallelism. Each row is contained within a single bank. For a pictorial

Fig. 3. ChRaBaRoCo.

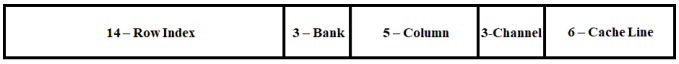

Fig. 4. RoBaRaCoCh.

Processor	values
Core count	4
Type	In-order
Frequency	1.0 GHz
Cache	**values**
L1 size	32K
Shared L2 size	64K
Associativity	4

TABLE I
PIM-LIKE CORE AND CACHE CONFIGURATION.

view, see Figure 3, which shows how a 31-bit address is split into DRAM organization. As can be seen, a DRAM row will be assigned to only one channel and cannot benefit from the existence of multiple channels. 3D DRAM devices are likely to have several channels, in some designs as many as 32. This organization may be better suited for multi-tasking or multi-threaded applications if the accesses from different threads (or tasks) can be spread across different channels. On the other hand, for single threaded applications, this organization does not utilize channel level parallelism and may lead to lower memory performance.

2. RoBaRaCoCh:

This organization spreads rows across multiple channels to exploit channel level parallelism (and obtain data faster), For a pictorial view, see Figure 4, which shows how a 31-bit address is split into DRAM organization. Data from a single DRAM row is now accessed using multiple channels. This organization may cause memory access conflicts for multi-threaded and multitasking workloads. We understand that this organization that uses multiple channels to obtain a large row of data is not currently implemented in commercial systems, but we wanted to explore potential benefits for multicore systems. In a typical commercial system, a channel is responsible for transferring data from one or more ranks and banks (similar to the organization in Figure 3) [22]. However, memory controllers are capable of aligning rows from different banks and ranks so that all channels can be used to transfer the data to achieve effectively the organization shown in Figure 4. Note that our model consists of just one rank, so there is no rank bits are shown in Fig 3 and 4.

Other organizations may be possible resulting in different amounts of data per row access, different amounts of channel level parallelism and rank level parallelism. Some of these organizations may be possible with 3D DRAMs even if they do not make sense with 2D DDR memories. However, we limit our study to the two organizations described here since they represent two extreme cases of data distribution.

C. Scheduling

In this era of multicore processors and many-core GPUs, efficient parallel programming is very crucial to benefit from such systems. For our purpose, we limit our study to homogeneous multicore systems and thus focus on OpenMP style programs. Since the programmers may not be aware of the DRAM organizations (such as those described above), these parallel programs may not achieve optimal levels of performance. For example, not many programmers use the *"schedule"* clause in OpenMP,(i.e., no specific scheduling type specified), relying on runtime system to implement scheduling. This will impact the overall performance up to 3 times when using "schedule" clause compared to not using one. Some programmers use *static chunk* scheduling where a number (i.e., chunk) of consecutive iterations are assigned to the same thread. It is also possible to use other ways of distributing loop iterations to threads, including cyclically assigning iterations to threads (in this case consecutive iterations may be assigned to different threads). In Dynamic scheduling, chunks of loop iterations are assigned to the next idle thread and this may lead to an uneven number of iterations assigned to different threads. In guided scheduling, initially only a portion (typically half) of the work is distributed to threads, and when the assigned work is complete, an additional portion of the iterations are distributed, until all the work is completed. This scheduling type may reduce the chunk size to improve load balancing among the iterations. Finally, when auto is specified, the iteration scheduling assignment is under compiler's control. Although cyclical assignment of loop iterations is not typically used with dynamic and guided scheduling methods, we explore such assignments in this study. It should be obvious that the scheduling used can impact the performance of applications since different scheduling methods result in different memory access patterns. However, our goal is to quantitatively measure the performance differences due to thread scheduling as well as DRAM memory organizations. In some cases accesses from different threads fall to the same DRAM row, while in other cases they fall into different rows of DRAM. And the size and distribution of DRAM rows across channels, banks and ranks impact access latencies. For the purpose of this paper, we focus on two different ways of distributing iterations (chunk and cyclical), three scheduling methods (static, dynamic, guided) and two DRAM organization. While the performance impacts of OpenMP scheduling were studied previously, our emphasis is on the memory accesses resulting from different scheduling methods with different memory organizations. We also study these organizations and scheduling methods for both In-Order and Out-Of-Order cores, since PIMs are likely to use In-Order cores.

D. Simulation

We used Gem5 [23] simulator in Full System mode for our experimental evaluations. Selected programs from benchmarks

Processor	values
Core count	4
Type	Out-Of-Order
Frequency	3.5 GHz
Cache	**values**
L1 size	32K
Associativity	8
Shared L2 size	1M
Associativity	16

TABLE II
HOST-LIKE CORE AND CACHE CONFIGURATION.

HBM	values
Capacity	2 GB
Memory Controllers	1 per Channel
Banks	8
Row Buffer	2 KB
Bus Width	128 bit per Channel
Bandwidth	128GBps

TABLE III
HBM CONFIGURATION.

Benchmark	Labelled as	Benchmark size
Vector Addition	vadd	64000 elements
Matrix Multiplication	mm	200x200 elements
Blackscholes	bh	4096
Streamcluster	sc	Cluster size-1000
Particle Filter	pf	Particles-2000
Hotspot	hs	1024x1024
Computational Fluid Dynamics	cfd	1024 elements

TABLE IV
BENCHMARKS LIST.

suites Rodinia [24], PARSEC [25] and Livermore loops [26] are used in our experiments (see Table IV for information on these benchmarks). We used GCC compiler to compile the benchmarks and simulated each benchmark with different OpenMP Scheduling techniques and two memory organizations described in Section II, using In-Order and Out-Of-Order (OOO) cores. The configuration used in our study are shown in tables I, II, III.

III. ANALYSIS

Here we will describe the results of our experiments and provide an analysis of these results. In all figures, each benchmark is labeled with the scheduling type and a suffix -chunk or -cyclical to indicate how loop iterations are assigned to threads. For example, static-chunk refers to Static scheduling that distributes a fixed number of iterations to each thread, and Dynamic-cyclic refers to dynamic scheduling that distributes iterations cyclically to threads (see Section II).

A. Chunk vs Cyclical Assignment

We now compare the performance differences between chunk and cyclical allocation of iterations (with static scheduling). The size of the chunk is based on the total number of iterations divided by the number of threads. Fig 6 shows the ratio of execution cycles using chunk versus cyclical assignments of loop iterations (Data shown are based on Fig 3 Memory Organization). From Fig 6, it can be noted that some applications like Blackscholes and vector addition perform

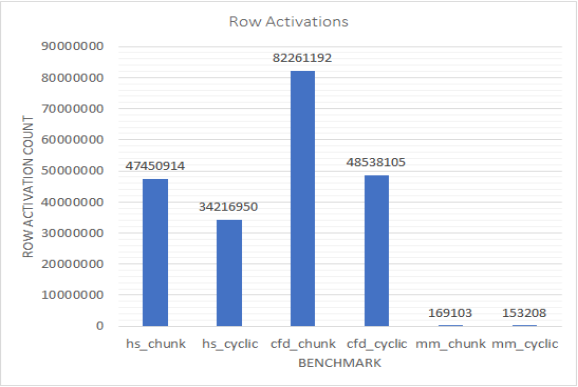

Fig. 5. Row activation count comparison (static scheduling, In order cores)

better with static chunk assignment (rather than static cyclical assignment). This can be understood since these applications access data mostly from vectors or rows of matrices; thus assigning consecutive iterations of loops to the same thread improves spatial locality of data accessed by that thread (and accesses will fall to the same DRAM row). If the cyclical assignment is used for these benchmarks, consecutive iterations will be assigned to different threads and accesses from the same thread may fall to different DRAM row buffers. However, cyclical assignment performs better for other applications, particularly when applications access some data row-wise and some data column-wise. Column-wise accesses do not exhibit spatial localities: thus assigning consecutive iterations to the same thread does not benefit from accessing large amounts of data on each DRAM access. Each thread may need to access multiple DRAM rows to satisfy its data needs. Moreover, the accesses from different threads can lead to opening and closing of DRAM rows, adding to the performance losses. Cyclical assignment can eliminate some DRAM row conflicts. For example, consider Fig 2, which shows a simple matrix multiplication code. To access the first matrix, in chunk mode, 4 threads will divide the workload equally (our experiments use a 4 core system and thus we use 4 threads): each thread processes 40 iterations. Initially, each thread accesses different DRAM row, which causes the activation of several rows, one per each thread. Whereas in cyclical mode, each thread will be assigned one iteration and it is likely that requests from different threads fall into the same row buffer, eliminating the need for multiple row activations to the DRAM. We observed that this can reduce the total number of row activations by up to] 40 percent as shown in Fig 5. We can make the following observation as to when the cyclical allocation performs better: application kernels with nested loops accessing multi-dimensional arrays, where array indexes involve multiple loop indexes. This is the case with Matrix Multiplication, Particle Filter, Hotspot, Computational Fluid Dynamics benchmarks. We can see as much as 4 percent, 6 percent, and 16 percent overall performance gains for Computational Fluid Dynamics, Hotspot and Matrix Multiplication benchmarks respectively with cyclical allocation when compared chunk assignment.

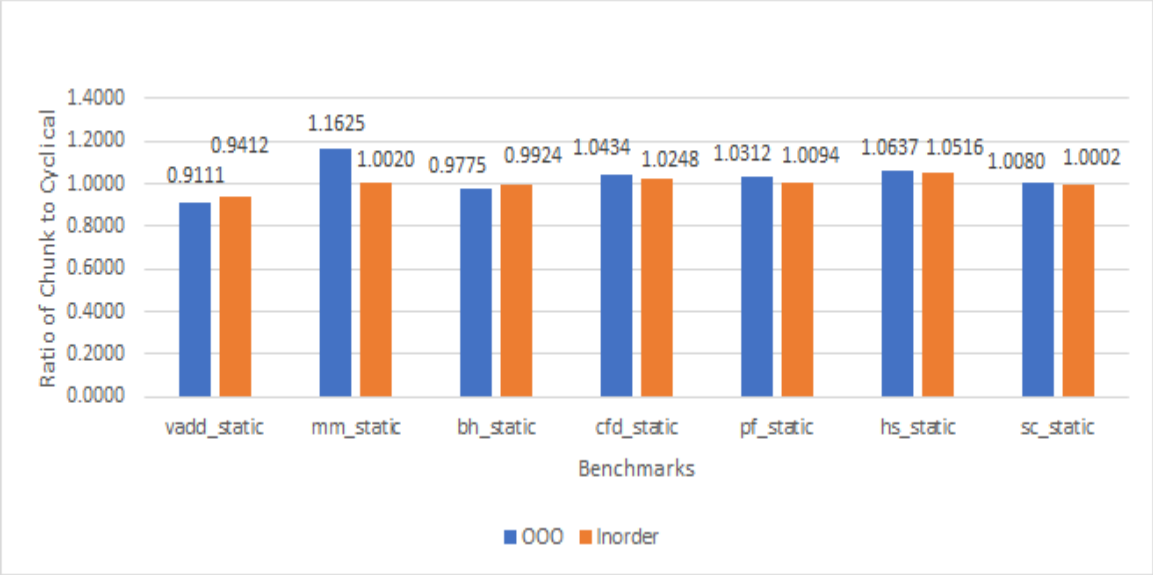

Fig. 6. Chunk vs Cyclic iterations (Cyclical assignment is better when bar value is 1 and above)

Consider Vector Addition where the iterations of the innermost loop (computing c[i]=a[i]+b[i]) are distributed among multiple threads. When cyclical assignment is used, each thread requires one data element per iteration but each cache line contains 64 bytes or holds 16 consecutive single-precision data elements. In this case, each thread will only use one of these 16 elements. Thus for this benchmark (and others like Blackscholes), the cyclical allocation is not good. On the other hand, if chunk assignment is used, consecutive iterations, thus consecutive vector elements are used by the same thread, thus all the 16 elements of a cache line may be consumed by the same thread.

B. Static, Dynamic, Guided Scheduling

In Fig 7, we compare different scheduling techniques with chunk or cyclical assignment (depending on which performed better in our previous experiment shown in III-A and Fig 6) for both In-Order and Out-Of-Order cores using the memory organization that is shown in Fig 3 (bars show the relative performance compared to the best performance achieved. In other words, 1.0 value indicates the best performance and all the other values indicate by how much other organizations differ from the best organization).

We observe that dynamic scheduling technique performed well (more than 6 percent) for benchmarks where the loop bodies contained conditional statements like Hotspot, Stream cluster implying unequal amounts of computations performed in iterations. Even though Computational Fluid Dynamics benchmark has conditional statements, we noticed that this did not cause significant differences in the amount of work done in each loop iteration. Dynamic scheduling allocates work to idle threads thus balancing the workload of the threads. Even though static (cyclical) scheduling performed better for computational fluid dynamics benchmark when using In-Order cores, dynamic scheduling performs as well as the static scheduling with Out-Of-Order cores. We observed that static scheduling allocates a fixed number of iterations to threads, which can lead to load imbalance and impact the overall performance. We noticed that for benchmarks performing better with chunk assignment like Blackscholes, vector addition, guided scheduling performs on par with the best performing scheduling method. Static scheduling works better in benchmarks without conditional statements because of the extra overhead involved in dynamic scheduling.

C. DRAM Organization

For the experiments thus far we used just one DRAM organization viz., shown in Fig 3. In this section, we compare the two memory organizations shown in Fig 3 and Fig 4. We hypothesized that multi-threaded workloads perform better with RoRaBaCoCh organization (Fig 4) since the data is interleaved at channel level and data can be fetched in parallel for the threads. Our experiments show that when applications involve multiple data structures (or arrays), they access the same DRAM row buffers repeatedly (potentially with interleaving accesses by other threads to other rows), which results in higher row activation time and thus the organization shown in Fig 4 did not prove beneficial. However, from Fig 8, with In-Order cores, for Vector Addition and Computational Fluid Dynamics (CFD), the organization shown in Fig 4 performs better. This can be understood because, for these benchmarks, only single dimensional vectors are accessed and the data exhibited spatial localities. For Computational Fluid Dynamics, data items inside the computational kernel is accessed in large strides which span multiple rows in ChRaBaRoCo organization. This results in delays from closing currently open DRAM row and opening the required DRAM row. In RoRaBaCoCh (Fig 4) organization, all the needed data items (in spite of large strides) are contained

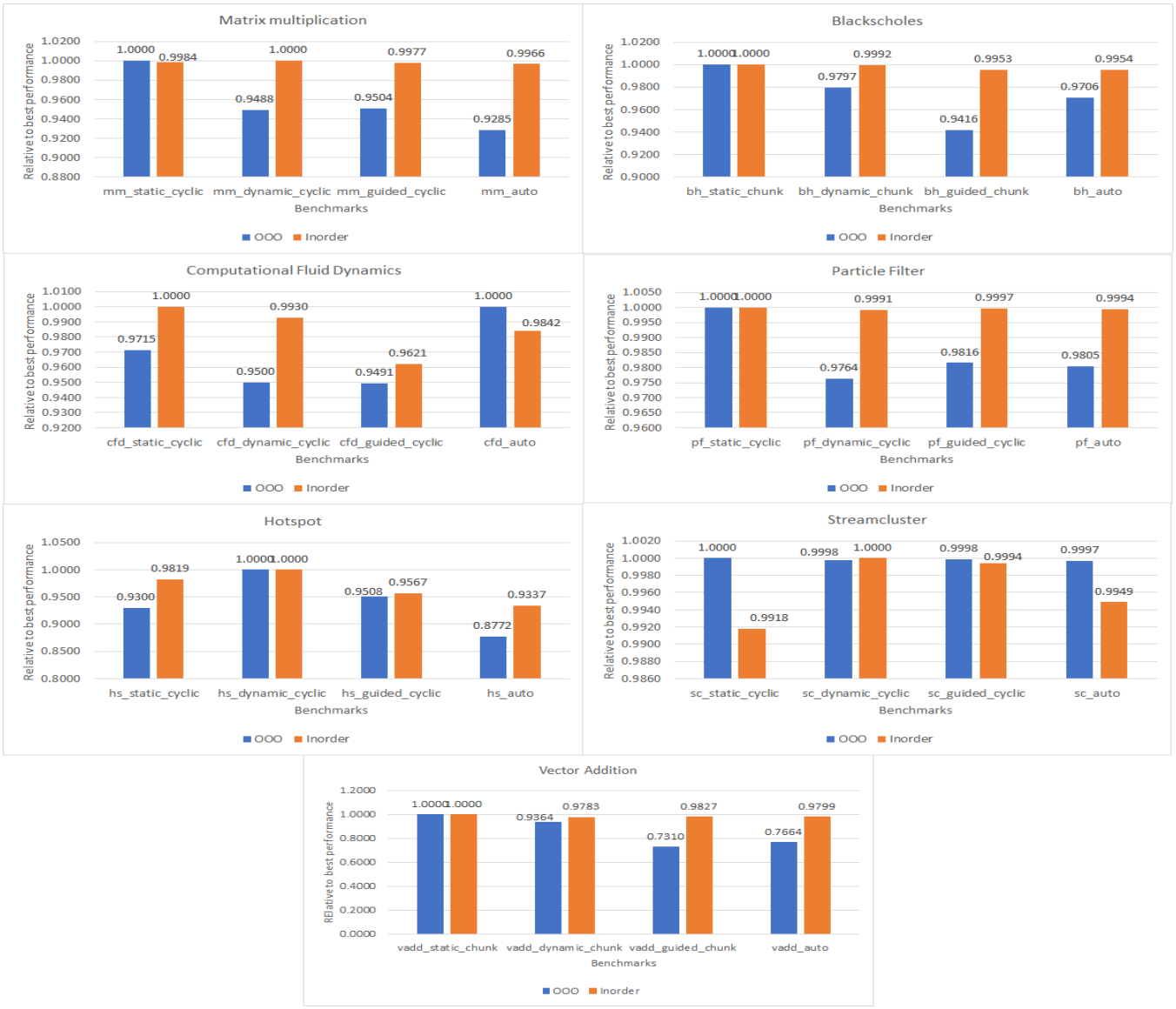

Fig. 7. Comparing different Scheduling Techniques (higher is better)

in the same row but in multiple banks and channels, which can be activated and fetched in parallel. Thus they benefit from channel level parallelism for obtaining very large rows across multiple channels. With Out-Of-Order (OOO) cores, we see similar results. In addition, with OOO cores, Hotspot, Stream cluster, Matrix multiplication also perform better when using RoRaBaCoCh organization (which is not the case with In-Order cores for these benchmarks). We observed similar data access patterns (with large strides) for Computational Fluid Dynamics and Hotspot; and similar access patterns (consecutive data items) for Vector Addition and Stream cluster. For these patterns, with OOO cores, we observe better performance with RoRaBaCoCh organization. For Particle Filter, the access stride length is not large (and does not benefit from large DRAM rows), thus performing better with ChRaBaRoCo organization. We can notice that the Fig 4 organization performs better in most of the benchmarks with Out-Of-Order CPU model.

The experiments thus far provide a better idea in terms of the scheduling technique, assignment of iterations and memory organization that works best for each application. Scheduling can easily be controlled with OpenMP pragmas. Even if DRAM organization cannot easily be modified or fixed by the memory controller, we feel that with a careful physical page allocation one can effectively achieve different DRAM organizations discussed in this paper. Pages assigned to different threads can be aligned to banks, ranks, and channels achieving the organization shown in Fig. 4. The physical page allocation can be controlled by the OS.

D. Out-Of-Order and In-order Cores

One of our aims is to explore which memory organization and which thread scheduling works best for processing-in-memory

Fig. 8. RoRaBaCoCh vs ChRaBaRoCo Organization (Ratio of Fig 4 Organization to Fig 3 Organization, Lower is better for Fig 4 Organization)

cores. As stated before, due to a power limit of 10W [18], PIM cores are likely to be simple In-Order cores with small L1 caches only, and executing at lower (e.g. 1GHz) clock rates. In previous sections and figures, we have shown the performance differences between In-Order cores and large, complex Out-Of-Order cores with multilevel caches and running at higher clock rates (3.5 GHz). Most processing-in-memory (PIM) designs propose to use simple In-Order and low power cores (e.g., ARM) while complex Out-Of-Order cores are used as host computing engines.

From Fig 8, benchmarks like Stream cluster, Hotspot and Matrix Multiplication perform better with Fig 3 organization when configured with In-Order cores but when configured with Out-Of-Order cores, Fig 4 organization outperforms Fig 3. This can be explained as follows, OOO cores may request data out of order (for different iterations which may fall to the same thread or different threads depending on how iterations are distributed) and these requests may fall to different DRAM rows; thus using the DRAM organization shown in Fig 4 can satisfy more OOO memory accesses with a single DRAM row access. On the other hand, In-Order cores request memory in order and may not benefit from large row buffers. When using In Order cores, if the requested addresses are contiguous, like in Vector Addition, Fig 4 performs better with minimum DRAM row activations else Fig 3 works well due to the DRAM row activation conflicts in Fig 4.

Benchmark	Assignment	Scheduling	Organization
Vector Addition	Chunk	Static	RoBaRaCoCh
Matrix Mul.	Cyclical	Dynamic	ChRaBaRoCo
Blackscholes	Chunk	Static	ChRaBaRoCo
Stream cluster	Cyclical	Dynamic	ChRaBaRoCo
Particle Filter	Cyclical	Static	ChRaBaRoCo
Hotspot	Cyclical	Dynamic	ChRaBaRoCo
CFD	Cyclical	Static	RoBaRaCoCh

TABLE V
BEST CONFIGURATION FOR IN-ORDER CORES

Benchmark	Assignment	Scheduling	Organization
Vector Addition	Chunk	Static	RoBaRaCoCh
Matrix Mul.	Cyclical	Static	RoBaRaCoCh
Blackscholes	Chunk	Static	RoBaRaCoCh
Stream cluster	Cyclical	Static	RoBaRaCoCh
Particle Filter	Cyclical	Static	ChRaBaRoCo
Hotspot	Cyclical	Dynamic	RoBaRaCoCh
CFD	Cyclical	Static	RoBaRaCoCh

TABLE VI
BEST CONFIGURATION FOR OUT-OF-ORDER CORES

E. Summary

The differences in the way instructions are executed and order in which data is accessed can result in different memory access behaviors. Tables VI and V show the memory organization, thread scheduling and iteration assignment that results in the best performance for each application studied in this paper, for the two types of processing cores. In many cases the same configurations result in the best performance for these different types of cores, however, there are some differences as can

be seen in these tables. This indicates that PIM designs may require different types thread scheduling and data organization for applications, when compared to executing the applications on a host node.

IV. Future Work

In this paper, we only experimented with a limited set of benchmarks. We will extend this work with more workloads to identify access patterns and determine best memory organization and thread scheduling for a given pattern. We will also explore caching heavily used DRAM rows to minimize access latencies to the same row at a future time. We will explore how different OpenMP scheduling techniques impacts page migration [32] and prefetching [33], [34] techniques used in heterogeneous memory systems.

V. Related Work

There are prior works on analyzing OpenMP scheduling options on only matrix multiplication benchmark [27]; but we analyzed several additional benchmarks, focusing on how memory accesses impact application performance due to both on OpenMP scheduling and DRAM organizations.

In another paper [28], the authors proposed a new mechanism for automatically deciding on scheduling technique at runtime. Their approach changes the way loop iterations are assigned based on observed performance. We feel that the overheads from such dynamic adaptations can defeat any performance gains. We focus on programmer defined scheduling (although dynamic and guided scheduling methods of OpenMP do involve some runtime adaptation of load associated with threads).

In [29], authors tested OpenMP scheduling in hybrid systems (ARM big-LITTLE configuration) and suggested that current scheduling policies are inefficient for heterogeneous systems. Although our study is only limited, we did compare different scheduling methods for simple In-Order and complex Out-Of-Order cores.

Bull [30] talks about overheads with respect to the scheduling chunk size. We investigated best chunk size (either 1 or total-workload-size/NUM-OF-THREADS) for different benchmarks depending on nature of it.

There have been many PIM studies that used simple ARM cores [16], GPUs [18] and specialized ASIC or reconfigurable devices [31]. Our goal is not to evaluate different PIM architecture choices but evaluate how PIMs can benefit from different 3D memory organizations.

VI. Conclusion

Our goal in this paper is to explore the impact of DRAM memories, particularly 3D-DRAMs such as HBM or HMC, on the performance of multi-threaded applications running on symmetric multi-core processors. In this paper, we evaluated memory organization that use multiple channels to access a row of data, which is applicable with 3D DRAMs since they are designed with several channels. We explored how different methods of assigning loop iterations to threads (using OpenMP schedule clause) can impact memory accesses, and in turn impact applications' performance. Since we are concerned with DRAM (particularly 3D DRAM) memories, we also explored if it is beneficial to access large amounts of data relying on high bandwidth through silicon vias (TSV). This can be achieved using different organizations of data in memory. We studied two (extreme) ways of spreading DRAM rows across banks, ranks, and channels. In one case, a row of memory is contained within a bank, and in the other case the row is spread across multiple banks, ranks, and channels: the second organization allows for obtaining large amounts of data (as much as 256KB) on each memory access. In this study, we limited our experiments to 4-core symmetric multiprocessor systems and OpenMP based programming models. We compared static, dynamic and guided scheduling techniques each with chunk and cyclical allocation of loop iterations. We evaluated two types of cores, simple In-Order cores with very small caches (which are likely to be the Processing-In-Memory (PIM) cores embedded in the logic layer or 3D-DRAMs, due to the limits on allowable power), and more conventional Out-Of-Order cores with large multilevel caches. We also varied clock frequencies, with lower frequencies with PIM cores. In general, for applications that access multiple data structures or multi-dimensional arrays using nested loops, and access some arrays row-wise and some column-wise, cyclical scheduling results in better performance; for applications that access single-dimensional arrays, static chunk scheduling performs better. One interesting observation is that when programmers do not specify any scheduling (i.e., do not use schedule clause OpenMP code), default allocation can lead to uneven allocation of loop iterations to threads, sometimes resulting in very poor performance, when compared to using either static chunk or cyclic scheduling approaches. Another interesting observation is that the type of the core (Out-Of-Order versus In-Order) also determines which thread scheduling and which DRAM organization results in best memory performance. We are planning to extend this study with more benchmarks as well as integrate our studies with other 3D-DRAM studies including heterogeneous memory architectures, prefetching DRAM pages and caching heavily accessed row buffers.

Acknowledgment

This research is supported in part by the NSF Net-centric Industry/University Cooperative Research Center and its industrial memberships. The authors would also like to thank the other members of the Computer Systems Research Laboratory at the University of North Texas for their feedback and assistance with some of the simulation tools used in this paper.

References

[1] JEDEC , "Jedec standard wide i/o 2," aug 2014. [Online]. Available: https://www.jedec.org/system/files/docs/JESD229-2.pdf

[2] C. C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 1–12.

[3] U. Kang, H.-J. Chung, S. Heo, S.-H. Ahn, H. Lee, S.-H. Cha, J. Ahn, D. Kwon, J. H. Kim, J.-W. Lee, H.-S. Joo, W.-S. Kim, H.-K. Kim, E.-M. Lee, S.-R. Kim, K.-H. Ma, D.-H. Jang, N.-S. Kim, M.-S. Choi, S.-J. Oh, J.-B. Lee, T.-K. Jung, J.-H. Yoo, and C. Kim, "8gb 3d ddr3 dram using through-silicon-via technology," in *2009 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, Feb 2009, pp. 130–131,131a.

[4] C. Weis, N. Wehn, L. Igor, and L. Benini, "Design space exploration for 3d-stacked drams," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.

[5] F. Sadi, L. Pileggi, and F. Franchetti, "3d dram based application specific hardware accelerator for spmv," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–1.

[6] H. Sun, J. Liu, R. S. Anigundi, N. Zheng, J.-Q. Lu, K. Rose, and T. Zhang, "3d dram design and application to 3d multicore systems," *IEEE Des. Test*, vol. 26, no. 5, pp. 36–47, Sep. 2009. [Online]. Available: http://dx.doi.org/10.1109/MDT.2009.105

[7] G. Loh, "3d-stacked memory architectures for multi-core processors," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, June 2008, pp. 453–464.

[8] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 25–37. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.51

[9] Y. Zhang, L. Li, Z. Lu, A. Jantsch, M. Gao, H. Pan, and F. Han, "A survey of memory architecture for 3d chip multi-processors," *Microprocess. Microsyst.*, vol. 38, no. 5, pp. 415–430, Jul. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2014.03.007

[10] K. Kavi, S. Pianelli, G. Pisano, G. Regina, and M. Ignatowski, "Memory organizations for 3d-drams and pcms in processor memory hierarchy," *J. Syst. Archit.*, vol. 61, no. 10, pp. 539–552, Nov. 2015. [Online]. Available: https://doi.org/10.1016/j.sysarc.2015.07.009

[11] J. C. Lee, J. Kim, K. W. Kim, Y. J. Ku, D. S. Kim, C. Jeong, T. S. Yun, H. Kim, H. S. Cho, S. Oh, H. S. Lee, K. H. Kwon, D. B. Lee, Y. J. Choi, J. Lee, H. G. Kim, J. H. Chun, J. Oh, and S. H. Lee, "High bandwidth memory(hbm) with tsv technique," in *2016 International SoC Design Conference (ISOCC)*, Oct 2016, pp. 181–182.

[12] J. Kim and Y. Kim, "Hbm: Memory solution for bandwidth-hungry processors," pp. 1–24, 08 2014.

[13] "Hybrid memory cube (hmc) consortium." [Online]. Available: http://hybridmemorycube.org

[14] J. L. Hennessy and D. A. Patterson, "Computer architecture, a quantitative approach," in *Text Book*, 2012, p. 98.

[15] OpenMP, "The openmp api specification for parallel programming," aug 2013. [Online]. Available: http://www.openmp.org/

[16] M. Scrbak, M. Islam, K. M. Kavi, M. Ignatowski, and N. Jayasena, "Exploring the processing-in-memory design space," *J. Syst. Archit.*, vol. 75, no. C, pp. 59–67, Apr. 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2016.08.001

[17] M. Scrbak, J. L. Greathouse, N. Jayasena, and K. Kavi, *DVFS Space Exploration in Power Constrained Processing-in-Memory Systems*. Cham: Springer International Publishing, 2017, pp. 221–233. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-54999-6_17

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

[19] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 336–348. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750385

[20] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 105–117.

[21] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski, "A new perspective on processing-in-memory architecture design," in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC '13. New York, NY, USA: ACM, 2013, pp. 7:1–7:3. [Online]. Available: http://doi.acm.org/10.1145/2492408.2492418

[22] AMD, "Bios and kernel developers guide (bkdg) for amd family 15h models 00h-0fh processors," Jan 2013, pp. 211–214. [Online]. Available: http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[24] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274

[25] 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*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[26] Tim Peters, Kendall Square Res. Corp., "Livermore loops coded in c," oct 1992. [Online]. Available: http://www.netlib.org/benchmark/livermorec

[27] N. H. Qun, Z. I. A. Khalib, and R. A. A. Raof, *Performance Analysis of OpenMP Scheduling Type on Embarrassingly Parallel Matrix Multiplication Algorithm*. Cham: Springer International Publishing, 2018, pp. 917–925. [Online]. Available: https://doi.org/10.1007/978-3-319-59427-9_94

[28] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera, "Is the schedule clause really necessary in openmp?" in *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, ser. WOMPAT'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 147–159. [Online]. Available: http://dl.acm.org/citation.cfm?id=1761900.1761916

[29] A. Butko, L. Bessad, D. Novo, F. Bruguier, A. Gamatié, G. Sassatelli, L. Torres, and M. Robert, "Position Paper: OpenMP scheduling on ARM big.LITTLE architecture," in *MULTIPROG: Programmability and Architectures for Heterogeneous Multicores*, Prague, Czech Republic, Jan. 2016. [Online]. Available: https://hal-lirmm.ccsd.cnrs.fr/lirmm-01377630

[30] M. Bull, "Measuring synchronisation and scheduling overheads in openmp," 02 2002.

[31] C. Shelor and K. M. Kavi, "Dataflow based near data computing achieves excellent energy efficiency," in *HEART*, 2017.

[32] M. Islam, K. M. Kavi, M. Meswani, S. Banerjee, and N. Jayasena, "Hbm-resident prefetching for heterogeneous memory system," in *Architecture of Computing Systems - ARCS 2017*, J. Knoop, W. Karl, M. Schulz, K. Inoue, and T. Pionteck, Eds. Cham: Springer International Publishing, 2017, pp. 124–136.

[33] M. Islam, K. M. Kavi, M. R. Meswani, S. Banerjee, and N. Jayasena, "Hbm-resident prefetching for heterogeneous memory system," in *ARCS*, 2017.

[34] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Sept 2012, pp. 337–344.