A Load Balancing Technique for Memory Channels

Byoungchan Oh
University of Michigan
Ann Arbor, Michigan
bcoh@umich.edu

Nam Sung Kim
University of Illinois at Urbana-Champaign
Champaign, Illinois
nskim@illinois.edu

Jeongseob Ahn
Ajou University
Suwon, Korea
jsahn@ajou.ac.kr

Bingchao Li
Civil Aviation University of China
Tianjin, China
bcli@cauc.edu.cn

Ronald G. Dreslinski
University of Michigan
Ann Arbor, Michigan
rdreslin@umich.edu

Trevor Mudge
University of Michigan
Ann Arbor, Michigan
tnm@umich.edu

ABSTRACT

The performance needs of memory systems caused by growing volumes of data from emerging applications, such as machine learning and big data analytics, have continued to increase. As a result, HBM has been introduced in GPUs and throughput oriented processors. HBM is a stack of multiple DRAM devices across a number of memory channels. Although HBM provides a large number of channels and high peak bandwidth, we observed that all channels are not evenly utilized and often only one or few channels are highly congested after applying the hashing technique to randomize the translated physical memory address.

To solve this issue, we propose a cost-effective technique to improve load balancing for HBM channels. In the proposed memory system, a memory request from a busy channel can be migrated to other non-busy channels and serviced in the other channels. Moreover, this request migration reduces stalls by memory controllers, because the depth of a memory request queue in a memory controller is effectively increased by the migration. The improved load balancing of memory channels shows a 10.1% increase in performance for GPGPU workloads.

CCS CONCEPTS

• **Hardware** → **Dynamic memory**; • **Computer systems organization** → *Single instruction, multiple data*;

KEYWORDS

DRAM, HBM, GPU, Memory Controller, Work Stealing

ACM Reference Format:
Byoungchan Oh, Nam Sung Kim, Jeongseob Ahn, Bingchao Li, Ronald G. Dreslinski, and Trevor Mudge. 2018. A Load Balancing Technique for Memory Channels. In *The International Symposium on Memory Systems (MEMSYS), October 1–4, 2018, Old Town Alexandria, VA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3240302.3240306

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS, October 1–4, 2018, Old Town Alexandria, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6475-1/18/10…$15.00
https://doi.org/10.1145/3240302.3240306

1 INTRODUCTION

Graphic Processing Units (GPUs) have developed for 3D graphics, games, and animations, and evolved for general purpose high performance computing [13, 21, 27]. GPU's on-chip computing capability has been improved rapidly in the past two decades [10]. However, the scaling of off-chip memory bandwidth has not followed the increasing computing capability. Thus, the memory bandwidth often becomes a bottleneck limiting application performance [19]. Traditionally, GDDR memories have been used for GPUs. They are throughput-optimized DDR and whose the latest generation is GDDR5. However, GDDR5 has challenges in increasing memory bandwidth, because its interface is narrow (16 or 32 per chip) and fast (up to 7Gbps per pin). Although high date rate is good for high bandwidth, it can only be achieved by consuming high power. In addition, the small number of I/Os provided from GDDR5 requires many memory chips to be accommodated in GPUs to achieve high bandwidth. As a result, the required power and area make GDDR5 prohibitive beyond 1 TB/s of memory bandwidth [8].

High Bandwidth Memory (HBM) has been developed to overcome limited bandwidth of GDDR5 under the given power budget and form factor [1, 17, 26]. HBM is an on-package stacked DRAM and provides high peak bandwidth (∼256 GB/s) through multiple (up to 8) and wide channels (128 I/Os per channel). For the power efficiency, data rate (*i.e.*, double of clock speed in DDR) and thus supply voltage are lowered in HBM, but the increased number of I/Os and channels results in higher peak bandwidth than that of GDDR5-based GPUs. In other words, the high peak bandwidth of HBM stems from a number of memory channels. Therefore, high bandwidth can be achieved in HBM when all HBM channels are utilized well. However, it is hard to evenly utilize all memory channels for all applications because each application has different memory access pattern. Moreover, substantial imbalance on memory channels still remains after

applying an XOR-based address mapping scheme, which randomizes the address mapping to avoid excessive contention on one or few memory channels/banks [35, 36].

To address this issue, we propose a cost-effective technique to improve load balancing for HBM channels. Our technique is conceptually similar with the work stealing technique used in multi-core scheduling, where an idle core steals a work item in a busy core and the load is balanced across multiple cores [4, 25]. Similarly, in our proposed HBM-based memory sub-system, a memory request in a busy channel is migrated to another non-busy channel and issued through that channel. Then, the migrated request is rerouted to its original memory device. However, in traditional GDDR5-based memory sub-systems, this simple load balancing technique is hard to apply because of mainly two reasons. First, memory controllers and their physical channels are placed on the different side of the host processor chip. Thus, the memory request migration requires global interconnection across the whole chip. Second, in order to reroute the migrated requests, extra off-chip interconnections to connect all off-chip GDDR5 chips are needed. Considering the cost to implement extra internal and external interconnections, the load balancing on memory channels by migration and rerouting would be impractical in the traditional GDDR5-based system. However, unlike the GDDR5-based system, the HBM-based system has several advantages in implementing this load balancing technique. First, multiple memory controllers for one HBM are locally placed because they are connected to the same chip having multiple DRAM devices. Thus, the local interconnections can enable the memory request migration. Second, one HBM has 8 channel. The DRAM dies have multiple ports that combine to form the 8 channels. Rerouting of the memory request can be performed inside of HBM. Because each DRAM die for a channel has the physical connection of all TSVs and this connection can be electrically controlled, a simple modification in HBM can implement the rerouting.

In our proposed memory system, if a channel is highly congested whereas other channels are not, the memory request migration is triggered. Then, the migrated memory request is rerouted to its original DRAM device by controlling electrical connection of TSVs in HBM. Through this balancing technique, the imbalance on memory channels is reduced by 7% on average. Moreover, because this request migration effectively increases the depth of memory request queue in a memory controller by occupying other memory controller's queue, the stall by memory sub-system is reduced. These improved load balancing and queue depth, in turn, bring 10.1% of GPU performance improvement (up to 26%).

2 BACKGROUND

2.1 Increasing Demand of Memory Capacity and Bandwidth

The increasing volume of data to be processed by machine learning and big data analytics demands data parallel architectures such as Single Instruction Multiple Data (SIMD)

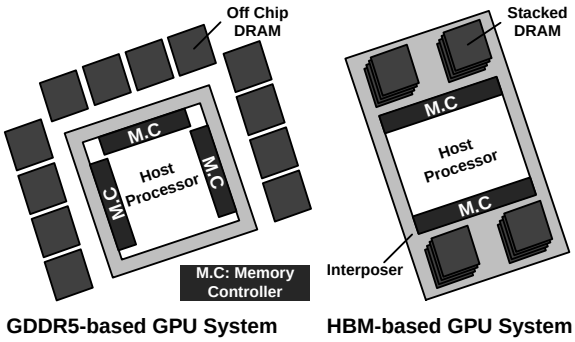


Figure 1: GPU systems with GDDR5 and HBM.

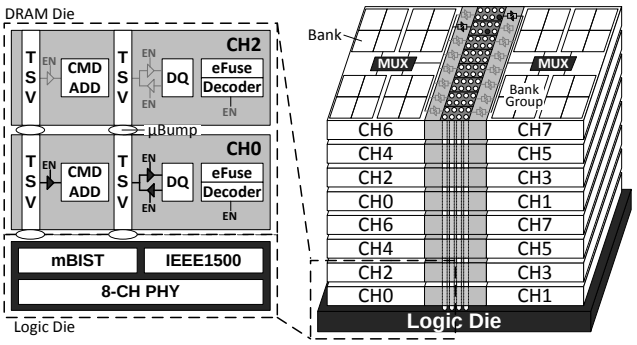


Figure 2: 3D structure of an HBM and a simple example of TSV connections to DRAM dies.

and Single Instruction Multiple Threads (SIMT) architectures [12, 32]. Especially, GPUs become the *de facto* standard and, in turn, the state-of-the-art servers in clouds and datacenters are equipped with GPUs to speed up general purpose computation (*i.e.*, General Purpose computing on Graphics Processing Units (GPGPU)) [27]. Because GPUs have been designed to improve the throughput of applications by spawning many threads simultaneously, the capacity and bandwidth of memory have played an essential role in building high performance applications. For example, in many programming models in GPGPU applications such as nearest neighbor classifiers, decision trees, and neural networks, the size of GPU memory often imposes limitations on the data size, resulting in decreased performance by continually transferring data from the system's memory to the GPU's memory [6]. In addition, because many of GPGPU applications are memory intensive and sometimes exhibit irregularity in their memory access patterns, their performance is significantly affected by memory bandwidth [5].

2.2 High Bandwidth Memory

HBM and HBM-based systems. Memory bandwidth has been continuously increased to meet GPU performance growth. However, in traditional GDDR5-based systems, there are mainly two challenges in increasing memory bandwidth. First,

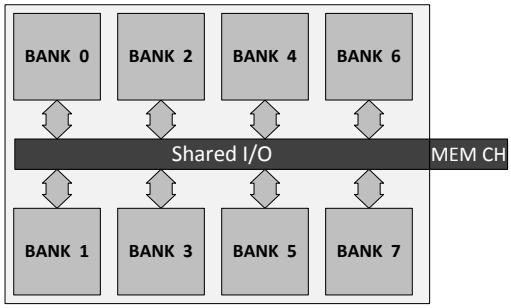
the increased memory bandwidth brings a significant increase in the power budget for memory and this power budget is becoming prohibitive as the bandwidth scales beyond 1 TB/s [8]. Because GDRR5 is connected to a host processor through fast (up to 7Gbps per pin) and narrow (16 or 32 per chip) external I/O interface, its energy-per-bit presents high (∼14pJ/bit). Second, GDDR5 can limit form factors. As shown in Fig. 1(left), GDDR5 requires a large number of memory chips to reach high bandwidth because of its narrow channel. Also, to build a large memory system with a given density of GDDR5, more memory chips are needed. The large footprint by GDDR5 does not only affect form factors, but this also degrades the signal integrity on the memory interface because of long connection distance [15, 23].

HBM, which is an on-package stacked DRAM, has been introduced to overcome power and form factor challenges of GDDR5. Unlike GDDR5, HBM employs a slow (∼2Gbps per pin) and wide (128 per channel) channel and accordingly supply voltage becomes lowered (1.5V -> 1.2V)[1]. In addition, since HBM has multiple stacked DRAM dies and is connected to the host processor via silicon interposer within a package, this system can accommodate a large number of memory devices with a small space as shown in Fig. 1(right).

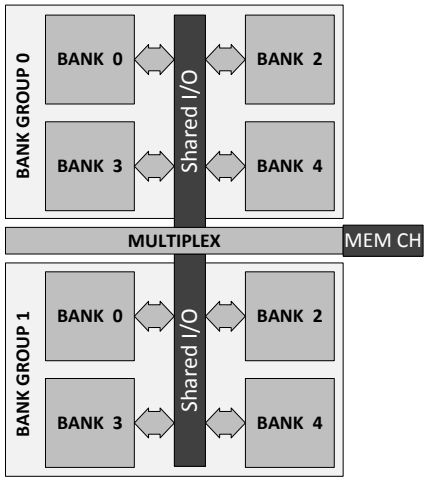
TSV connections. Fig. 2 depicts the internal structure of HBM. An HBM is made with various capacity, the number of stacked layers and channel configurations [17]. In this study, the baseline HBM has 1Gb capacity and 2 half-channels per DRAM die and total 8 DRAM dies (total 8Gb). All DRAM dies are fabricated identically and thus all they are *physically* connected to TSVs for 8 channels. Then, a set of TSVs for certain channels can be *electrically* connected to one of the DRAM dies by using tri-state buffers with the decoder logic shown in the left of Fig. 2. During a manufacturing step, a Stack ID (SID) is programmed to the decoder to enable or disable the tri-state buffers by using electrical fuses (*e*fuses). We describe an example of this *physical* and *electrical* connections between TSVs and DRAM dies in Fig. 2(right), where the set of TSVs have *physical* connections to both DRAM dies but only the bottom DRAM die for `CH0` is *electrically* connected to the TSVs.

Bank group structure. The bank group feature, which is used in GDDR5 and DDR4, is, also, adopted in HBM [17, 18, 24]. We describe the organization of a DRAM device with and without the bank group feature in Fig. 3. As shown in Fig. 3a, all banks are connected to one internal shared data bus. Traditionally, in order to bridge the gap between slow data transfer speed on the shared bus and fast interface speed, data are transferred on the shared bus in parallel and then serialized out the interface with multiple clock cycles (*a.k.a n-prefetch*). In this structure, if the speed gap is increased, the prefetch length and accordingly burst length, which determine memory transaction and LLC line sizes, should be

[1]In this study, we take HBM generation 2 (HBM2) as a baseline. However, we do not differentiate HBM and HBM2 in this study because main features of them, such as wide I/O, multi channels and 3D stacked DRAM dies, are almost same.



(a) Memory organization without bank group



(b) Memory organization with bank group

Figure 3: Comparison between two memory organizations [24].

increased together to keep seamless burst read/write operations. Furthermore, bank-level parallelism in this structure does not improve much with the number of banks because of the limited scalability of the single shared bus. In order to avoid increasing prefetch length and improve the parallelism, the bank group feature has been introduced as depicted in Fig. 3b. In the bank group structure, multiple banks groups (typically, 4 or 8 groups, 4 by default in this study) have their own internal data bus and multiple banks (2 or 4 banks, 4 by default) in a bank group share one data bus. As the result of the separated data bus, multiple sets of data can be concurrently transferred between the interface and bank groups. However, different timing constraints, `tCCDS` and `tCCDL`, are applied when accessing banks in different bank groups and the same bank group, respectively. `tCCDL` is the minimum time between two read commands (or write commands) when accessing the same bank group and determined by the data transfer time on the shared data bus in the bank group. However, `tCCDS` is the minimum time between two read commands (or write commands) when accessing different bank groups and not determined by the data transfer time because two read accesses are served on different buses in different

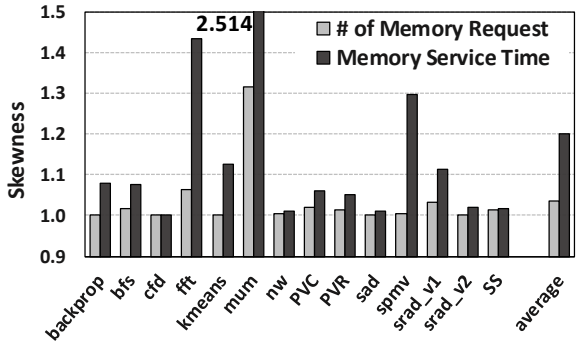


Figure 4: Channel utilization and memory service time.

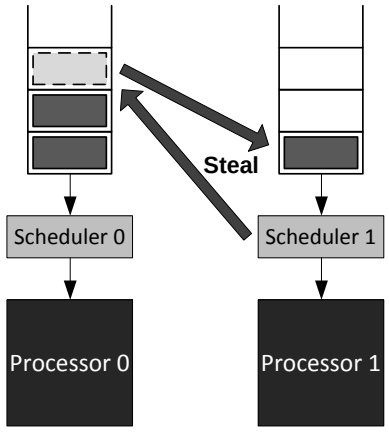


Figure 5: Simple diagram for work stealing.

bank groups. Thus, the bank level parallelism in a bank group is still preserved, but the bank group level parallelism is a higher degree of parallelism.

3 CHALLENGES IN MANY CHANNEL MEMORY SYSTEMS

3.1 Imbalanced Channel Utilization

In general, the memory address mapping scheme is designed considering both spatial locality and parallelism [7]. For example, consecutive cache line accesses are scheduled to the same row in the same bank to take advantage of shorter latency when row buffer hit. On the other hand, accessing blocks of cache line alternates between multiple banks and channels by exploiting bank- and channel-level parallelism. However, depending on workloads memory system can suffer from excessive contention on one or few certain banks and channels. To prevent this situation, a permutation-based mapping scheme (*i.e.*, hashing), in which channel and bank selections are determined by XORing a subset of MSB-side bits, has been proposed [35, 36]. Although this technique partially randomizes memory accesses, it is hard to completely eliminate the imbalanced memory requests on all channels and banks.

Fig. 4 shows the skewness of total memory requests and service time across 8 channels of an HBM. The skewness is defined to the ratio of the minimum value to the maximum value. If the address mapping scheme is ideal and thus all channels receive the equal number of memory requests, the skewness of total memory requests becomes 1. Although the skewness of total memory requests is closed to 1 in many workloads due to XORing applied in the address mapping scheme, some workloads exhibit high skewness. Furthermore, this imbalance on the total number of memory requests is amplified on the service time, which is defined to the total time spent to serve all memory requests in a memory controller, as shown in Fig. 4. Because spatial and temporal locality in each channel can be different with the same number of requests, they can make different scheduling scenario and result in non-equal memory service time in each channel.

The imbalanced memory requests and utilization across the memory channels can negatively affect overall performance by hindering exploiting full capability of all memory channels. Work stealing, which is a well-known scheduling technique for multi-core systems, has been proposed to balance workloads and improve performance [4, 25]. We describe the simplified mechanism of the work stealing in Fig. 5. If a processor is idle (processor 1) is idle, it looks at the queue of another processor (processor 0) and steals its work if there are outstanding works. In this case, because all processors are identical, a work item can be executed in any processor. Therefore, the load balancing technique for memory channels like the work stealing can be considered to a good solution for imbalanced memory channels. However, the load balancing technique cannot be simply applied to the traditional GDDR5-based system, because each memory request (the work item in the work stealing) has its own memory address and it must be served in the preassigned memory device by the address. In other words, if a memory request is migrated to other channels and issued through the other channels, it must be rerouted and served in its initially assigned memory device based on the address.

3.2 Implementation Challenges of Memory Controllers

Having large request queues in the memory controller is generally beneficial to the performance because of mainly two reasons. First, the request queue is the buffer to mitigate the gap between fast input and slow service speeds of memory requests [22]. Thus, the queue depth determines the capability to hold the number of outstanding requests and this can significantly affect the performance when workloads are memory-intensive. Second, there are more chances to make better scheduling decisions (*i.e.*, shorter latency) in larger queues [2]. For instance, a First-Ready First-Come-First-Served (FR-FCFS [29, 38]) scheduler can make more row hits with a larger queue because the scheduler observes more memory requests and this increases the probability to find

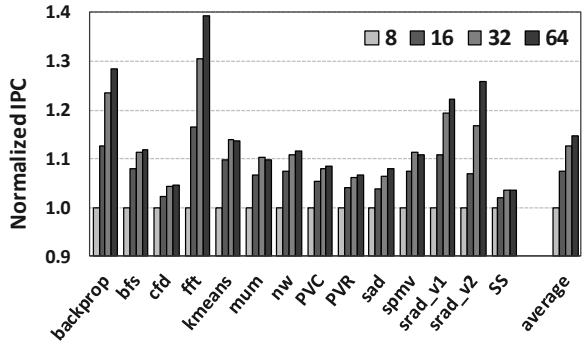


Figure 6: Performance according to the queue depth.

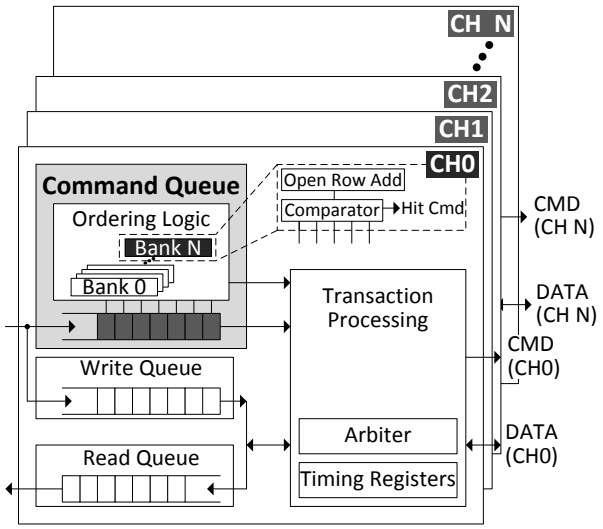


Figure 7: Schedulers of the memory controller in many channel memory systems [16].

memory requests corresponding to the scheduling priority. However, there are fewer row hits with a smaller queue because of limited visibility to memory requests. Fig. 6 depicts the performance improvement according to the number of queue entries with various GPGPU applications. Based on workloads and their memory intensity, the sensitivity of performance improvement to the queue depth varies, but most workloads show higher performance with larger queues.

Unfortunately, in practice, it is hard to implement a sophisticated scheduling policy on a large queue. For example, in order to enable an FR-FCFS policy, the row address of all outstanding requests in the queue should be compared to the address of the already open row per bank every cycle [30, 37]. Such a fully associative search demands Content Addressable Memories (CAMs). The design cost of CAM combining with the scheduling logic super-linearly increases with the increase of the number of queue entries [2, 28]. Furthermore, as depicted in Fig. 7, each memory channel needs its own

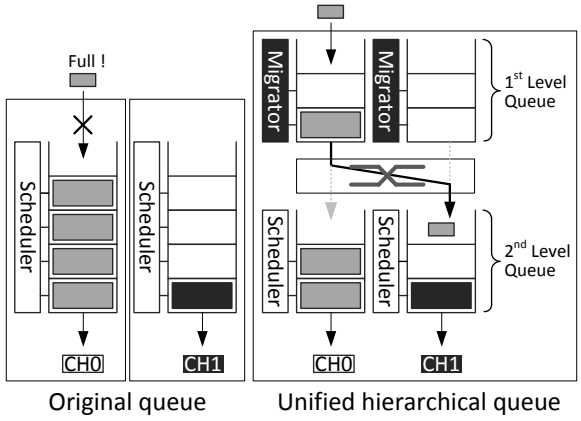


Figure 8: Hierarchical queue structure.

independent memory controller. Therefore, the area of memory controllers has a significant impact on total chip area in many channel memory systems.

4 OVERVIEW OF THE PROPOSED DESIGN

In previous sections, we discussed why all memory channels are not evenly utilized and the request redistribution technique such as work stealing cannot be simply applied to memory systems. In addition, we observed performance improvement with large request queues in memory controllers, but it is hard to implement large queues with a FR-FCFS scheduling policy, because of the super-linearly increasing design cost as a result of the number of queue entries. With such observations, we propose a new memory system design to mitigate the imbalance of channel utilization and effectively increase the queue depth without increasing the actual queue size. In brief, our design allows memory requests to be inserted in, and issued from, any memory controller belonging to the same HBM. Then, the memory request issued through other channels is rerouted inside of the HBM. The bank group feature enables us to concurrently serve multiple requests in the same memory device. There are three key observations which led to the proposed design; (1) multiple memory controllers for one HBM are placed locally, (2) in an HBM, all TSVs have physical connections to all DRAM dies and a set of TSVs constituting a channel can be electrically connected to any DRAM die by the decoder logic, and (3) multiple sets of data can be transferred concurrently inside of DRAM having bank group feature. In the remainder of this section, we first present the memory controller design and scheduling policy for our new design and then introduce the new HBM architecture.

4.1 Re-architecting Memory Controllers

In general, each memory controller for each channel operates independently. In other words, each memory controller does not communicate with one another. There were studies to propose a technique to coordinate all memory controllers by

connecting them to each other [7, 20]. Exchanging the scheduling status of each memory controller or globally applying a single scheduling priority can improve performance, because a single memory channel can be accessed by multiple threads and memory requests issued by a single thread can spread across multiple different channels. However, because memory controllers for different channels are often placed on opposite side of the chip as shown in Fig. 1, it is hard to implement the global interconnection between memory controllers in a traditional GDDR5-based system.

Unified Queue Structure. Unlike GDDR, where one chip provides only 32 I/Os and two chips compose one channel, one HBM provides 8 channels and accordingly the 8 memory controllers for the one HBM can be placed locally as shown in Fig. 1. Therefore, the interconnection between these memory controllers can be, also, implemented locally. With this observation, we propose a hierarchical queue structure as shown in Fig. 8. In our hierarchical queue, one large queue is split into two smaller queues. Because of the super-linear relation between the size of a queue and area, we can save the area by dividing the large queue. The saved area is used to implement crossbars. (Detail area analysis will be discussed in Sec. 5.3.)

In the proposed memory system, a memory request in a channel can be migrated to one of the other channels having room to accept the memory request through the crossbar. In the example of Fig. 8, each channel has a 4-entry request queue and channel 0 (`CH0`) is already full. In this case, the upper level of the memory controller (*e.g.*, last level cache) cannot issue a memory request to `CH0` and thus it is stalled, because there is no entry to accept the memory request in `CH0` as shown in Fig. 8(left). However, if a memory request in `CH0` is migrated to `CH1` and thus one empty entry is created in `CH0`, `CH0` can keep accepting memory requests without incurring stalls in its upper level (right in Fig. 8). Therefore, this technique can effectively increase the queue depth and accordingly reduce the stall of the last level cache.

Channel Borrowing. In addition to the increased queue depth, we allow issuing memory requests migrated from different channels. As a result, the migration reduces overall queuing delay because the memory requests migrated from a busy channel and issued through idle (or less busy) channels do not experience long waiting time in the queue. Note that the memory request issued through a different channel from its original channel must be rerouted to its original channel's DRAM device. Also, a DRAM device should have the capability to handle more than two memory requests at the same time because multiple requests can be sent to the same DRAM device through different channels. To address these issues, we exploit the facts that all DRAM dies (all channels) have physical connections to all TSVs and the bank group structure is capable of serving multiple requests concurrently. The cost-effective implementation inside of HBM will be discussed in the next section.

There are several challenges in scheduling the memory requests migrated from other channels. First, each memory controller must consider the timing constraints and bank status of all other channels to avoid command/data collision and timing violations. DRAM has various timing constraints which must be considered in scheduling memory requests in order to guarantee correct memory operations inside of DRAM (*e.g.*, `tRCD`[2]), provide the power recovery time after high power consumption (*e.g.*, `tRRDS`[3]) and avoid data collision on the memory bus (*e.g.*, `tCCDS`). Hence, the scheduler in a memory controller has to abide by all timing constraints for all other channels as well as that for its channel when issuing memory requests. In Fig. 9, for example, `CH0` has three memory requests requiring an activation command (`ACT`) and `CH1` has no request to issue (❶). In this case, if only `tRRDS` for each channel is considered, the second `ACT` command (`ACT1`) in `CH0` can be migrated to `CH1` and can be issued through `CH1`'s memory bus and `ACT2` can be issued earlier in `CH0` (❷). Although `ACT0` and `ACT1` can be issued through different channels, they will meet in the same DRAM device and make a `tRRDS` violation. In order to avoid the violation, `ACT1` in `CH1` must be issued after `tRRDS` is elapsed from `ACT0` issued in `CH0`. In addition to the delayed `ACT1`, `ACT2` in `CH0` must consider when `ACT1` is issued in `CH1` and cannot be issued earlier as shown in Fig. 9(❸). Therefore, this scheduling example does not have any benefit. Furthermore, this unuseful scheduling is even possible only when each memory controller considers the timing constraints of all channels and bank status of all other memory controllers.

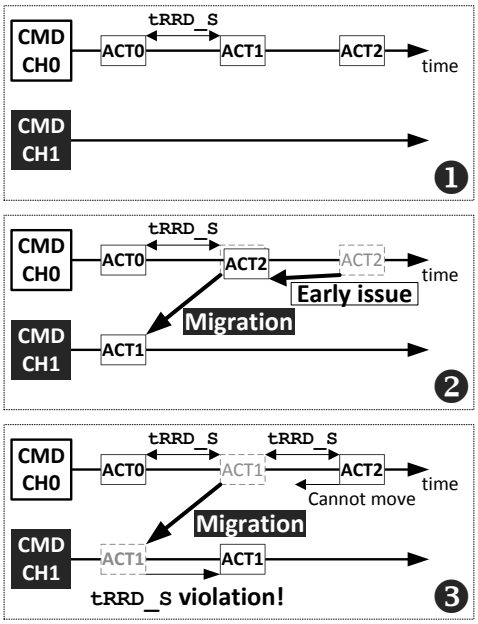


Figure 9: Limited scheduling by the timing constraints.

Second, it is hard to determine the priority of the memory requests with the mixed memory requests having different

[2] minimum time between activation and read/write commands
[3] minimum time between two activation commands for different banks in different bank group

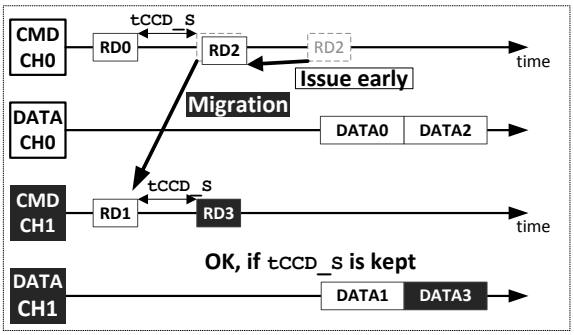


Figure 10: An example of avoiding the timing constraints.

channel addresses. As discussed in Sec. 3.2, an FR-FCFS scheduler compares the address of all outstanding requests to already open row addresses of all banks. Because a bank address of the memory request migrated from other channels should be considered by another independent bank regardless of the same bank address (*e.g.*, `BANK0-CH0` and `BANK0-CH1`), this request migration can effectively increase the number of banks to be considered for the scheduling. Thus, the scheduling complexity and accordingly the design complexity increase by the increased number of banks.

Considering the two challenges described above, the scheduling memory requests including migrated one from other channels is practically impossible. In order to overcome these challenges, we only migrate memory requests which meet the predefined conditions. The first condition is that the memory request has the different bank group address from that of all outstanding memory requests in the second level queue. Because we exploit bank group level parallelism inside of DRAM, the memory requests having different bank group address can be served in DRAM at the same time. In other words, if the memory requests having the same bank group address are issued through the different channel at the same time, DRAM cannot accept all of them because there is only one shared internal I/O for a bank group inside of DRAM. Second, the memory request having currently open row's address is migrated to other channels. In other words, row commands (*i.e.*, `ACT` and `PRE`) have to be issued in the original channel and column commands (*i.e.*, `RD` and `WR`) can be issued in any channel. This second condition enables the avoidance of all timing violations related to internal memory operations (*e.g.*, `tRCD` and `tRRD`). Then, the migrated memory request can be treated as a native memory request in a memory controller. As shown in Fig. 10, `RD1` migrated from `CH0` can be issued through `CH1` unless it violates `tCCDS` of `CH1`, which is `tCCD` for different bank group accesses and thus `RD2` in `CH0` can be issued earlier. Rule 1 summarizes all conditions for the migration.

As we discussed earlier, the migration increases the number of banks to be considered in scheduling requests. However, in the proposed memory system, the memory requests going to the open row are only migrated. In other words, all migrated requests are ready to issue unless they violate `tCCDS`. Rule 2 describes the priorities for the scheduling decision at the second level of queues. Note that FR-FCFS requires the same number of comparators with the number of banks (typically, 16). However, to search the migrated requests only one small comparator is enough, because unlike the comparators for FR-FCFS which compare row addresses (15 bits for the baseline HBM) per bank, the comparator for the migrated requests only compares channel addresses (3 bits for the baseline HBM).

Rule 1: Migration conditions at the 1st level

1. **Full of their 2nd level queue**—Only when the 2nd level queue is full, requests are migrated to other channels. Normally, requests are served in their original channel.
2. **Room of other 2nd level queues**—Only when the 2nd level queue has enough room, where more than half of entries are not occupied, requests are migrated to this queue.
3. **Different bank group**—Requests having different bank group address from that of the outstanding requests in the 2nd level in the same memory controller are migrated to avoid the collision on the internal memory I/O for the same bank group.
4. **Column command**—Requests having no need to issue row commands are migrated to avoid timing violation inside of DRAM.

Rule 2: Scheduling priorities at the 2nd level

1. **Migration**—Migrated requests which are always ready to issue are prioritized over native requests.
2. **Open row (FR-FCFS)**—Row-hit requests are prioritized over row-miss requests. This priority is only applied to native requests.
3. **Arrival time (FCFS)**—Older requests are prioritized over younger requests.

4.2 Re-architecting HBM

As we discussed in Sec. 2.2, all channels in an HBM have physical connections to all DRAM dies and a set of TSVs constituting a channel can be electrically connected to any DRAM die. In addition, the bank group structure enables DRAM to concurrently transfer multiple requests inside of DRAM because each bank group has an individual separated I/O. Motivated by these observations, we introduce alternative paths inside of HBM to serve more memory requests as shown in Fig. 11. In the original design, the DRAM die assigned for `CH7` is only electrically connected to a set of TSVs constituting `CH7` and a memory request coming from the TSVs of `CH7` is transferred to bank group 0 through the 4:1 muxes/demuxes. Because in the original design, only one set of TSVs is connected to this DRAM die and only one memory request is issued to a channel at a time, one set of muxes/demuxes can relay the memory request to a bank group and

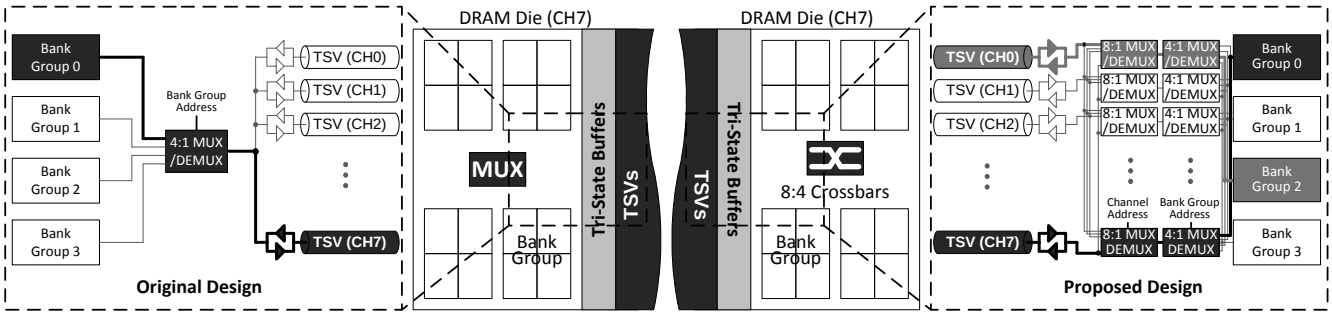


Figure 11: Original HBM and the proposed HBM with crossbars.

Table 1: Configured System

Component	Specification
Number of SM	15
Maximum Threads per SM	1536
L1 Data Cache per SM	16 KB
Number of Memory Channel	8
L2 Cache per Memory Channel	128 KB
Compute Core / Interconnect / Memory Clock	1000/1000/1000 MHz
DRAM Scheduling Policy	FR-FCFS
HBM Configuration per channel	8Gb, 128 I/Os, 2KB pages, 4 bank groups, 4 banks per bank group
HBM Timing Parameters (tCK)	`tRC`=47, `tRCD`=14, `tRP`=14, `tRRDS`=4, `tRRDL`=6, `RL`=14, `WL`=2, `tCCDS`=1, `tCCDL`=2, `tRTPS`=3, `tRTPL`=4, `tWR`=14

Table 2: Workload list

Suite	Benchmark (abbreviation)
MARS	Page View Count (PVC), Page View Rank (PVR), Similarity Score (SS)
Parboil	Fast Fourier Transform (fft), Sum of Absolute Difference (sad), Sparse Matrix Dense Vector Multiplication (spmv)
Rodinia	Back Propagation (bp), Breath First Search (bfs), Computational Fluid Dynamics Solver (cfd), K-means Clustering (kmeans), Needleman-Wunsch Algorithm (nw), Speckle Reducing Anisotropic Diffusion version 1 (srad1), srad version 2 (srad2)

only one bank group can receive a memory request at a time. However, in our proposed HBM, any DRAM die is electrically connected to any set of TSVs by the crossbars. Hence, a memory request coming from any channel can be relayed to any bank group through the 8:4 crossbars, consisting of sets of 8:1 muxes/demuxes and 4:1 muxes/demuxes as shown in Fig. 11(right). Based on the channel address of a request, the tri-state buffers and the first stage multiplexers (8:1 muxes) are controlled. Then, the bank group address is used for the second stage multiplexers (4:1 muxes). In Fig. 11(right), for example, two memory requests are sending to the same DRAM die of `CH7` through `CH0`'s and `CH7`'s TSVs, respectively. Although the two requests are issued from different memory controllers and transferred through different channel's buses and TSVs, they are relayed to the same DRAM die because of their same channel address. However, they have different bank addresses and are eventually arrive in different bank groups. Also, no collision of memory requests occurs in the crossbars (*i.e.*, the requests coming from different TSVs, but going to the same bank group), because memory requests having different bank group addresses can be issued through different channels at the same time in the new HBM.

4.3 Overhead

To enable the migration of memory requests, we introduce extra circuits and storage. In this section, the overhead of our proposed memory system is discussed.

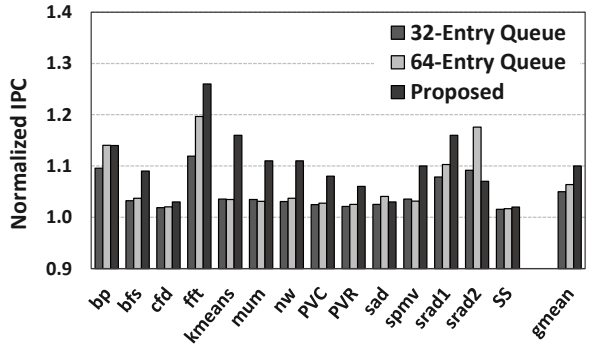
In memory controllers. At the first level of the request queue, in order to search a migration candidate, our design requires similar comparison logic with the FR-FCFS scheduler. In addition, the table keeping track of the bank group status is required. Second, the 8:8 crossbars are introduced to connect memory controllers for all channels for one HBM. Last, in order to differentiate the channel address of a request, 3 bits are added to each entry of the second level queue. However, because the queue depth of each level queue is half of the baseline queue, we, actually, can save the area in spite of the extra circuits and storage. The detail about the area will be discussed in Sec. 5.3 with various queue configurations.

In HBM. In order to reroute migrated memory requests, we use the 8:4 crossbars and extra control signals. The estimated area increase using CACTI-3DD [11] and 20nm DRAM technology information [31] is 1.5% of a DRAM die.

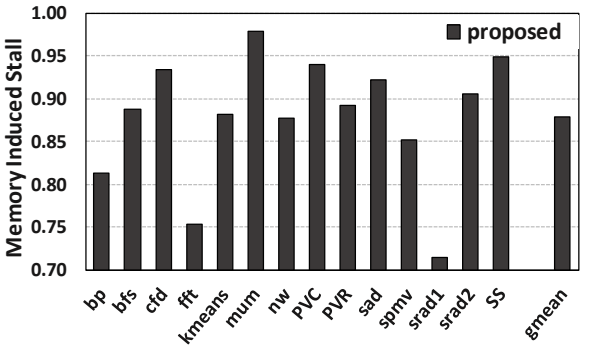
5 EVALUATION

5.1 Methodology

For our evaluation, we use GPGPU-Sim version 3.2.2 and implement our technique in its memory system [3]. The configuration of the evaluated system is summarized in Table. 1. We model HBM based on [8] and present its key parameters

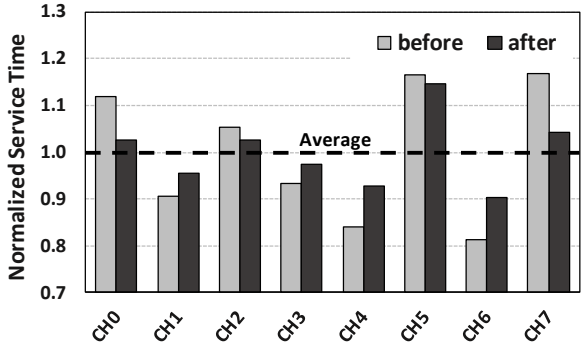


(a) IPC improvement

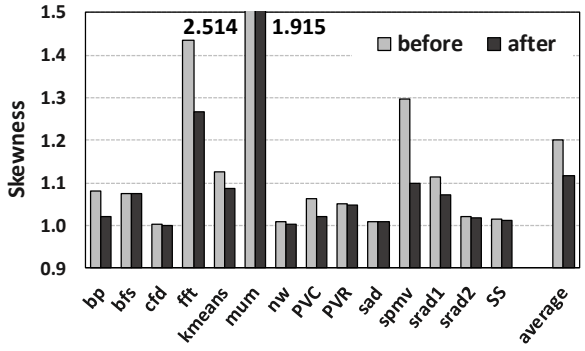


(b) Normalized memory induced stall cycle

Figure 12: Performance improvement after the migration.



(a) Change of channel skewness in *fft*



(b) Change of skewness in memory service time

Figure 13: Change of imbalanced memory service time.

in Table. 1. In order to evaluate our load balancing technique, we use several GPGPU benchmark suites such as MARS [14], Rodinia [9], Parboil [33] and mummerGPU (mum) provided in GPGPU-Sim. The workloads used in the evaluation are listed in Table. 2. We run all the benchmarks for their full length to capture whole characteristics of them. For the baseline, each memory controller has request queues of depth 16. In our memory system, the first level and second level queues have 8 entries, respectively. However, the second level queue can be shared by other channels, if memory requests meet the migration conditions.

5.2 Performance Analysis

Fig. 12a plots the normalized instruction per cycle (IPC) to the baseline after applying our load balancing technique. The 32-entry queue and 64-entry queue are the same systems with the baseline except for the queue depth. As we discussed previous sections, having large queues is good for performance. As shown in Fig. 12a, our memory system outperforms the baseline and large queue systems. The improved IPC over the baseline is 10.1% on average and up to 26.0%. Because the request migration effectively increases the request queue depth, our memory system can hold more memory requests without incurring stall at the upper level. In *bp* and *srad1*, their performance is sensitive to the queue depth and the increased queue depth in the proposed memory system mainly results in their performance improvement. However, in some applications (*e.g.*, *bfs*, *mum* and *nw*), the queue depth does not have a significant impact on the performance. In these applications, our memory system still brings performance improvement. This is because, in the proposed memory system, memory requests can be issued through different channels from their original channel, which reduces overall queuing delay by migrating blocked requests from the busy queue. Fig. 12b shows the normalized stall cycles of memory controllers, in which memory controllers cannot accept memory requests because there is no empty entry in their request queue. Our load balancing technique does not specifically prioritize certain critical requests (*e.g.*, the requests determining the degree of memory divergence), but it reduces overall latency of memory requests. Thus, the reduced stall cycles in memory controllers are mostly reflected in the performance.

In Fig. 13, we present the service time of each channel in *fft* and the skewness of the service time. In *fft*, the skewness of the number of memory requests was ∼1.06, which means there is only a 6% inequality in the number of requests between channels. However, the skewness of the memory service time, which is defined to be the active time of memory controllers to serve the memory requests, was ∼1.44 (Fig. 4 in Sec. 3.1)

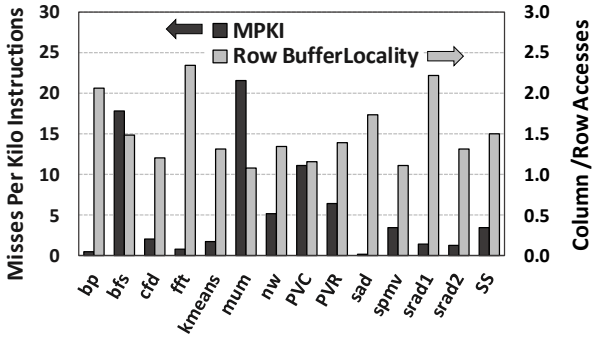


Figure 14: MPKI and row buffer locality.

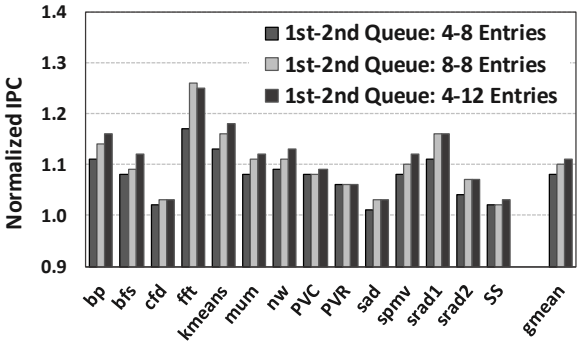


Figure 15: Performance according to different queue configurations.

and is much bigger than the skewness of the number of memory requests because of the difference in spatial and temporal localities between the channels. Because we migrate the memory requests from the busy channel to a non-busy channel, this load balancing results in 12% reduction in the skewness of the service time. As shown in Fig. 13a, the busy channel becomes less busy and the non-busy channel becomes busier in our memory system. Fig. 13b shows the improved load balance in the proposed memory system. Overall, the skewness is reduced by 7% across all workloads, and there is a substantial reduction in *mum* (24%).

As we discussed in Sec. 4.1 (Rule 1), not all memory requests are migrated to other channels in our memory system. In order to obtain benefits from the migration, 1) the workloads should be memory intensive to generate enough congestion in the memory system, 2) the memory channels should be skewed in terms of their request service time and 3) the memory requests should have a certain degree of spatial locality to meet the migration conditions. We present the memory intensity of the workloads as misses-per-kilo-instructions (MPKI) and the spatial locality as the ratio of the number of column commands (`RD` and `WR`) to the number of row commands (`ACT`) in Fig. 14. A weak correlation between performance improvement and MPKI is observed, whereas the row buffer locality of the workloads has a strong correlation with performance improvement.

In Fig. 15, we present performance improvement with various queue configurations. Because we effectively increase the queue depth, 4-8 configuration (4-entry for the first level queue and 8-entry for the second level queue), whose total number of queue entries (12) is smaller than the baseline (16), outperforms the baseline. Also, when the total number of queue entries is the same, the 4-12 configuration shows slightly higher IPC than the 8-8 configuration. Because the migration can happen when memory controllers have more than half of empty entries at the second level queue, having a larger second level queue can permit more migrations and yield higher performance improvement than the smaller second level queue. The difference in performance between 8-8 and 4-12 configurations is 1-3%.

5.3 Area Overhead

In order to estimate the area of the memory controller, we developed Verilog models synthesized with a 45nm design library [34]. For this estimation, we only use standard cells. That is, all memory-components such as CAMs, buffers, and scheduling tables are modeled using flip-flop, but not customized cells, and are therefore conservative. We present the estimated area in Table. 3 Although we introduce crossbars and few extra logic, it is the reduced queue depth that mainly saves the area.

Table 3: Estimated area

Configuration	Area (normalized)
4-8	0.90
8-8	0.95
4-12	0.97

6 RELATED WORK

Multi-Channel Memory Controllers. ATLAS is a scheduling technique proposed for fair scheduling across multiple memory channels [20]. ATLAS periodically orders threads based on the service they have attained from the memory controllers. After a long time quanta, information about the received service is exchanged between memory controllers and a central controller prioritizes the threads that have attained the least service over others in the next epoch. Although ATLAS uses a long time quanta to provide scalability, this approach is not practical for GPUs having thousands of threads.

Chatterjee *et al.* proposed a memory scheduling technique to manage memory latency divergence in GPUs [7]. In order to avoid inter-warp interference, they proposed forming batches of memory requests from a single warp called a warp-group. Also, their scheduling technique coordinates scheduling decisions across multiple memory channels with dedicated point-to-point interconnections between memory controllers.

Although these scheduling techniques can improve performance by coordinating scheduling decisions across multiple

memory channels with given memory requests in a channel, there is no consideration about load balancing for memory channels.

Cost- and Complexity-Effective Memory Controllers. Staged Memory Scheduler (SMS) is a decentralized architecture for application-aware memory scheduling [2]. SMS decouples the memory controller's primary tasks and partitions them across simpler hardware structures in a staged fashion. Because of the decentralized small request queues and simpler scheduling logic, SMS significantly saves the area over FR-FCFS scheduler while improving performance. However, batch formation in SMS occurs an individual queue per thread. Thus, this scheduler is not suitable for GPUs having thousands of threads.

Yuan *et al*. addressed the high complexity of out-of-order scheduling such as FR-FCFS and proposed a complexity-effective solution for achieving the scheduling comparable to that of out-of-order scheduler [37]. Their key observation is that the row locality of the memory requests sent from the shader cores are much higher before they enter the interconnection network compared to when they arrive at memory controllers. By recovering the destroyed row locality in the interconnection network, their simple in-order memory controllers performs comparably with an out-of-order scheduler.

Similar to our design, they strove for reducing the implementation cost of memory controllers. However, both designs mainly focused on implementing an individual memory controller for a channel without consideration about the coordination between multiple channels.

7 CONCLUSION

The performance of memory systems often significantly affects overall system performance. HBM is optimized for high performance by providing a number of memory channels. Specifically, it is adapted to GPUs to meet their demand for high memory bandwidth. We observed that only one or a few memory channels are often highly utilized in GPGPU applications. This imbalance on memory channels hinders exploitation of the full bandwidth of an HBM. To overcome underutilized memory bandwidth, we propose a technique to improve load balancing for HBM channels. Our technique enables memory requests to migrate from a busy channel to other non-busy channels and service it in the other channels. In addition, the proposed technique effectively increases the depth of a request queue in a memory controller, which results in the reduction of the stall cycles by memory controllers. Our load balancing technique mitigates the imbalance of the memory channel utilization and brings 10% of performance improvement for GPGPU applications.

ACKNOWLEDGMENTS

This work is supported in part by Samsung Electronics, NSF CNS-1705047 and Natural Science Foundation of Tianjin, 18JCQNJC00400.

REFERENCES

[1] AMD. 2015. Inside pascal: NVIDIA's newest computing platform. https://www.amd.com/en/technologies/hbm. (2015).
[2] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 416–427.
[3] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.
[4] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.
[5] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 141–151.
[6] Alberto Cano. 2018. A survey on graphic processing unit computing for large-scale data mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 1 (2018).
[7] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonian. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 128–139.
[8] Niladrish Chatterjee, Mike OâĂŹConnor, Donghyuk Lee, Daniel R Johnson, Stephen W Keckler, Minsoo Rhu, and William J Dally. 2017. Architecting an energy-efficient dram system for gpus. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 73–84.
[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.
[10] John Y Chen. 2009. GPU technology trends and future requirements. In *Electron Devices Meeting (IEDM), 2009 IEEE International*. IEEE, 1–6.
[11] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 33–38.
[12] Preeti Gupta, Arun Sharma, and Rajni Jindal. 2016. Scalable machine-learning algorithms for big data analytics: a comprehensive review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 6, 6 (2016), 194–214.
[13] Mark Harris and David Luebke. 2005. GPGPU: General-purpose computation on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2005 Courses: Los Angeles, California*, Vol. 2005.
[14] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. 2008. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 260–269.
[15] Hiroaki Ikeda and Hidemori Inukai. 1999. High-speed DRAM architecture development. *IEEE Journal of Solid-State Circuits* 34, 5 (1999), 685–692.
[16] Intel. 2012. DRAM Controllers for System Designers. https://www.altera.com/solutions/technology/system-design/articles/_2012/dram-controller-system-designer.html. (2012).
[17] JEDEC. 2013. High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/sites/default/files/docs/JESD235A.pdf. (2013).
[18] JEDEC. 2016. GRAPHICS DOUBLE DATA RATE (GDDR5) SGRAM STANDARD. https://www.jedec.org/system/files/docs/JESD212C.pdf. (2016).
[19] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (2011), 7–17.
[20] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance*

Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on. IEEE, 1–12.
[21] David Kirk et al. 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, Vol. 7. 103–104.
[22] John DC Little and Stephen C Graves. 2008. Little's law. In *Building intuition*. Springer, 81–100.
[23] Igor Loi and Luca Benini. 2010. An efficient distributed memory interface for many-core platform with 3D stacked DRAM. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 99–104.
[24] MICRON. 2014. DDR4 SDRAM. https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_dram_2e0d.pdf. (2014).
[25] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.
[26] NVIDA. 2016. Inside pascal: NVIDIA's newest computing platform. https://devblogs.nvidia.com/inside-pascal. (2016).
[27] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. 2007. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, Vol. 26. Wiley Online Library, 80–113.
[28] Subbarao Palacharla, Norman P Jouppi, and James E Smith. 1997. *Complexity-effective superscalar processors*. Vol. 25. ACM.
[29] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 128–138.
[30] Hemant G Rotithor, Randy B Osborne, and Nagi Aboulenein. 2006. Method and apparatus for out of order memory scheduling. (Oct. 24 2006). US Patent 7,127,574.
[31] Samsung Semiconductor. 2016. Research collaboration communications. (2016).
[32] Dilpreet Singh and Chandan K Reddy. 2015. A survey on platforms for big data analytics. *Journal of Big Data* 2, 1 (2015), 8.
[33] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).
[34] Oklahoma State University. 2017. FreePDK: Unleashing VLSI to the Masses. https://vlsiarch.ecen.okstate.edu/flows/. (2017).
[35] Gert-Jan van den Braak, Juan Gomez-Luna, José María González-Linares, Henk Corporaal, and Nicolas Guil. 2016. Configurable XOR hash functions for banked scratchpad memories in GPUs. *IEEE Trans. Comput.* 65, 7 (2016), 2045–2058.
[36] Hans Vandierendonck and Koenraad De Bosschere. 2005. XOR-based hash functions. *IEEE Trans. Comput.* 54, 7 (2005), 800–812.
[37] George L Yuan, Ali Bakhoda, and Tor M Aamodt. 2009. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 34–44.
[38] William K Zuravleff and Timothy Robinson. 1997. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. (May 13 1997). US Patent 5,630,096.