Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures

George L. Yuan Ali Bakhoda Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, CANADA
{gyuan,bakhoda,aamodt}@ece.ubc.ca

ABSTRACT

Modern DRAM systems rely on memory controllers that employ out-of-order scheduling to maximize row access locality and bank-level parallelism, which in turn maximizes DRAM bandwidth. This is especially important in graphics processing unit (GPU) architectures, where the large quantity of parallelism places a heavy demand on the memory system. The logic needed for out-of-order scheduling can be expensive in terms of area, especially when compared to an in-order scheduling approach. In this paper, we propose a complexity-effective solution to DRAM request scheduling which recovers most of the performance loss incurred by a naive in-order first-in first-out (FIFO) DRAM scheduler compared to an aggressive out-of-order DRAM scheduler. We observe that the memory request stream from individual GPU "shader cores" tends to have sufficient row access locality to maximize DRAM efficiency in most applications without significant reordering. However, the interconnection network across which memory requests are sent from the shader cores to the DRAM controller tends to finely interleave the numerous memory request streams in a way that destroys the row access locality of the resultant stream seen at the DRAM controller. To address this, we employ an interconnection network arbitration scheme that preserves the row access locality of individual memory request streams and, in doing so, achieves DRAM efficiency and system performance close to that achievable by using out-of-order memory request scheduling while doing so with a simpler design. We evaluate our interconnection network arbitration scheme using crossbar, mesh, and ring networks for a baseline architecture of 8 memory channels, each controlled by its own DRAM controller and 28 shader cores (224 ALUs), supporting up to 1,792 in-flight memory requests. Our results show that our interconnect arbitration scheme coupled with a banked FIFO in-order scheduler obtains up to 91% of the performance obtainable with an out-of-order memory scheduler for a crossbar network with eight-entry DRAM controller queues.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

Categories and Subject Descriptors

C.1.2 [**PROCESSOR ARCHITECTURES**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms

Design

Keywords

Memory Controller, On-chip Interconnection Networks, Graphics Processors

1. INTRODUCTION

In contemporary many-core compute accelerator architectures, such as today's graphics processing units (GPUs), the DRAM system is shared among many threads and/or cores, each of which generates a stream of independent memory requests. In such a shared memory system, multiple memory access streams interleave and interact in a way that frequently destroys the inherent row access locality present in each individual memory access stream, thereby reducing the efficiency at which DRAM can transfer data across its data bus. Consequently, overall system throughput can suffer.

DRAM efficiency, which we define as the percentage of time that a DRAM chip spends transferring data over its data pins divided by the time when there are memory requests pending, has a direct correlation with the achievable off-chip bandwidth, so maximizing the efficiency is crucial to obtain the full off-chip bandwidth throughput. In order to maximize DRAM efficiency, a variety of memory controllers have been designed for superscalar and multi-core processors, all based around the principle of out-of-order scheduling, where the memory controller will search through a window of pending DRAM requests to make a scheduling decision that will maximize the row access locality and bank-level parallelism [18, 13, 17, 11, 12]. Alternatively, an in-order scheduling memory controller will schedule memory requests in the order in which it receives them from either the processor or the interconnection network if there are multiple processors. Only applications that already have memory access patterns exhibiting high spatial locality will perform well with an in-order scheduler. Recently proposed techniques for effective DRAM scheduling [13, 17, 11, 12] are extensions of First-Ready First-Come First-Serve (FR-FCFS) [18], an out-of-order memory scheduling technique that exploits row access locality among multiple pending requests buffered waiting to be serviced in the memory controller. This is primarily done to minimize the total number

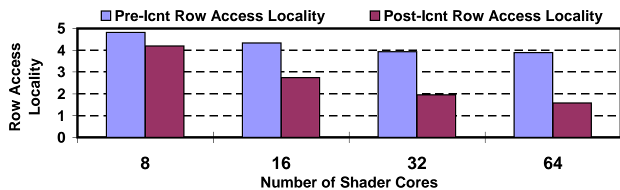


Figure 1: Row access locality versus number of shader cores in a crossbar network

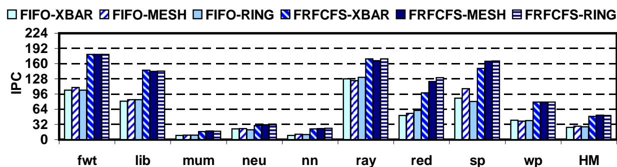


Figure 2: IPC when using FR-FCFS DRAM scheduler vs FIFO scheduler (HM = harmonic mean)

of row buffer switches required by DRAM during the runtime of an application. Each time a DRAM bank switches rows, it must pay a precharge and activate overhead during which time it is unavailable to service pending requests, and thus unable to immediately do any useful work. Poor row access locality results in frequent switching and can drastically reduce DRAM efficiency resulting in a loss of performance.

Modern graphic processing unit (GPU) architectures are now capable of running certain highly parallel general purpose applications (GPGPU) [15, 21, 1]. These GPUs present a different challenge for effective DRAM access since modern GPU architectures support at least an order of magnitude more concurrent threads than CPUs. The NVIDIA GTX 285 has 30 "Streaming Multiprocessors" (which we refer to as shader cores), each capable of supporting 1024 active threads for a total of 30720 threads [15]. Any technique for DRAM scheduling must effectively handle large numbers of threads.

In this paper, we make the key observation that emerging highly-parallel non-graphics applications run on these GPU architectures typically have high DRAM row buffer access locality memory access patterns at the level of individual shader cores. However, the interconnection network that connects the multiple shader cores to the DRAM controllers serves to interleave the memory request streams from multiple shader cores, destroying their row access locality.

We illustrate this in Figure 1, measuring the row access locality of the memory request streams from the individual shader cores before they feed into the interconnection network (*Pre-Icnt Row Access Locality*) and that of the memory request streams received at each memory controller after they come out of the interconnection network (*Post-Icnt Row Access Locality*). In this figure, we fix the number of memory channels to four and increase the number of shader cores to show that, as the ratio of shader cores to DRAM controllers on a GPU chip increases with current trends[1], the gap between the pre-interconnect and post-interconnect row access locality clearly becomes wider and wider.

Because of the extent of the row access locality disruption in GPU architectures, where the number of shader cores greatly outnumber the number of DRAM controllers, FIFO in-order DRAM scheduling performs extremely poorly compared to an out-of-order FR-FCFS scheduler. We show this in Figure 2 for three network topologies, the ring, crossbar, and mesh. Across these topologies, the speedup over a FIFO scheduler when using FR-FCFS is 88.3%, making FR-FCFS the more attractive option. However, the nature of out-of-order scheduling is that it is area-intensive, requiring, in the worst case, associative comparison of all requests in a DRAM controller queue every cycle, therefore requiring a set of comparators for each entry in the queue [8, 19]. Compared to in-order scheduling, which only requires a simple FIFO queue, the area needed to implement out-of-order scheduling can be significant.

We emphasize that the 9 applications shown in Figure 2 are only those that we deemed to be memory-limited out of a total of 26 applications considered. In other words, spending area to implement out-of-order DRAM scheduling will not improve roughly two-thirds of these applications, which are compute-bound. Nevertheless, the memory-limited applications are not negligible, including important applications such as weather prediction, DNA sequence alignment, and raytracing.

In this paper, we propose a complexity-effective solution for achieving DRAM scheduling comparable to that of out-of-order schedulers. Our solution leverages the key finding that, in the GPGPU applications we study, the row access locality of the memory request streams sent from the shader cores are much higher before they enter the interconnection network compared to after they become interleaved inside the interconnection network and arrive at the DRAM controllers. We find that, by using simple in-order memory controllers along with a more intelligent arbitration scheme in the interconnection network routers, we can recover much of the performance lost when a naive in-order memory controller is used in comparison to an out-of-order scheduler.

This paper makes the following contributions:

- It shows that the row access locality inherent in the memory request stream from individual shader cores can be destroyed by the interconnection networks typically used for connecting shader cores to DRAM controllers. We introduce a novel interconnect arbitration scheme that preserves this inherent row access locality, thus allowing for a much simpler DRAM controller design.

- It presents a qualitative and quantitative analysis of the performance of our interconnect arbitration scheme and its various heuristics for both mesh and crossbar networks.

- It also presents a simple solution to deal with the interleaving that can occur due to the interconnection network router having multiple virtual channels, which can improve interconnect throughput [7].

The rest of this paper is organized as follows. Section 2 describes the challenge of DRAM access scheduling in many core accelerator architectures in more detail, to motivate this work. Section 3 introduces our complexity-effective solution for achieving high DRAM performance in GPU architectures. Section 4 describes the experimental methodology and Section 5 presents and analyzes our results. Section 6 reviews related work and Section 7 concludes the paper.

[1] We predict this to occur in correspondence with ITRS [10] predicting the number of transistors on a chip to increase at a faster rate than the chip pin count.

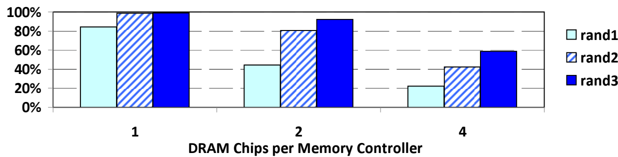


Figure 3: Measured DRAM efficiency for random uniform memory traffic with varying row access locality

2. MANY-CORE ACCELERATOR DRAM ACCESS SCHEDULING CHALLENGES

The scaling of process technology has allowed processor throughput to increase exponentially. However, the properties of packaging prevent the pin count from increasing at the same rate. To alleviate this, chip designers maximize the percentage of pins on a chip used for transferring data. Increasing the amount of off-chip memory supported by the chip will likely require increasing the number of memory channels. Each additional memory controller requires its own set of address pins for communicating the requested memory addresses and control pins for relaying the commands to control the DRAM chips.

One approach to reduce the number of memory channels is to have the memory controller for each channel control multiple DRAM chips, thereby amortizing the address and control pin overhead across multiple DRAM chips [8]. However, there is a limit to how much of this "chip-level parallelism" is available. Consider a case where the typical memory request size is 64 bytes. The CUDA Programming Guide's chapter on performance tuning suggests recent NVIDIA GPUs generate memory requests per *half-warp* of 16 threads and that requests from individual threads that access contiguous 32-bit words can be coalesced into a single 64-byte memory request [15]. If we assume that a single DRAM chip has a burst length of 4 and a bus width of 4 bytes (typical of modern graphics double-data rate (GDDR) memory technology), then the maximum number of DRAM chips that a memory controller can potentially control without wasting data pin bandwidth is four (4 DRAM chips per memory controller $\times$ 4B bus width $\times$ 4 burst length = 64B). Furthermore, increasing the number of DRAM chips per memory controller reduces the number of read/write commands per activated row for each chip. If the memory access pattern of a particular application exhibits low row access locality, then DRAM efficiency can reduce greatly. This occurs when there is a lack of requests to service to the activated rows of other banks when one bank is constrained by the DRAM timing overhead needed to switch rows.

To quantify these effects, Figure 3 shows the DRAM efficiency achieved using FR-FCFS scheduling of uniform random memory access patterns with varying degrees of row access locality measured using a stand-alone DRAM simulator based upon GPGPU-Sim, a publically available massively multithreaded architectural simulator [3]. Figure 3, $rand1$ is an artificially generated uniform random memory access pattern. Since the row designation of each memory access is random, the high number of rows per chip (4096 for the particular GDDR3 memory that we simulate [16]) imply that the chance of two back-to-back requests going to the same row in a particular bank is near-zero, meaning that a new row must be opened after the servicing of each memory request. In $rand2$, we replicate each random memory access once so that there are always two requests back-to-back to the same row. Similarly, in $rand3$, we replicate each random memory access so that there are always three requests back-to-back to the same row. As can be seen, fewer DRAM chips per memory controller effectively means more DRAM read and write commands to the rows of each chip to transfer the same total amount of data, thus increasing the DRAM efficiency.

The example above assumes a memory access pattern that is uniform random. Such a memory access pattern will tend to generate requests to all banks in a chip, thus maximizing "bank-level parallelism". As such, the DRAM efficiency values presented in the above example represent the near-maximum efficiencies obtainable for the given row access locality parameters. On the other hand, non-uniform memory access patterns may generate (a disproportionate amount of) requests to only a subset of the banks of a DRAM chip, which can cause DRAM efficiency to be drastically lower. For example, consider again the previous example where the row access locality is two and the number of DRAM chips per memory controller is two. If all of the requests go to a single bank, the DRAM efficiency plummets from 80.7% to 23.6%.

In order to maximize DRAM efficiency, modern DRAM controllers will schedule DRAM requests out of order in an effort to maximize the row access locality. Without doing so, the performance of in-order scheduling can be drastically worse when simple regular request streams (that would ordinarily schedule very efficiently in DRAM) from multiple sources become interleaved. Figure 4 shows a simplified example that illustrates this. Consider first a single shader core interacting with a single DRAM controller (Figure 4(a)). The request stream of this single shader core consists of two requests to the opened row, Row A. Assuming that Row A has already been loaded into the DRAM row-buffer, both requests can be serviced immediately and no row-switching latency is incurred. Now consider two shader cores interacting with the same DRAM controller (Figure 4(b)). The first shader core issues two requests to Row A while the second shader core issues two requests to Row B. In trying to uphold fairness, a conventional interconnection network router arbiter may use round-robin arbitration, resulting in the two input request streams becoming finely interleaved when they pass through the router. In this case, an out-of-order scheduler would group together the requests to Row A and service them first before switching to Row B and servicing the remaining requests, thereby only paying the row-switching latency once. On the other hand, an in-order DRAM scheduler would service the requests as it receives them, therefore having to pay the row-switching latency three times, resulting in drastically lower DRAM throughput. If an intelligent network router recognized that, in this scenario, it should transfer all the requests of one shader core first before transferring the requests of the second shader core, then the performance obtainable using out-of-order DRAM scheduling could be achieved with a much simpler in-order FIFO DRAM scheduler.

In this paper, we leverage this key observation to design an intelligent interconnection network coupled with a simple DRAM controller that can achieve the performance of a much more complex DRAM controller.

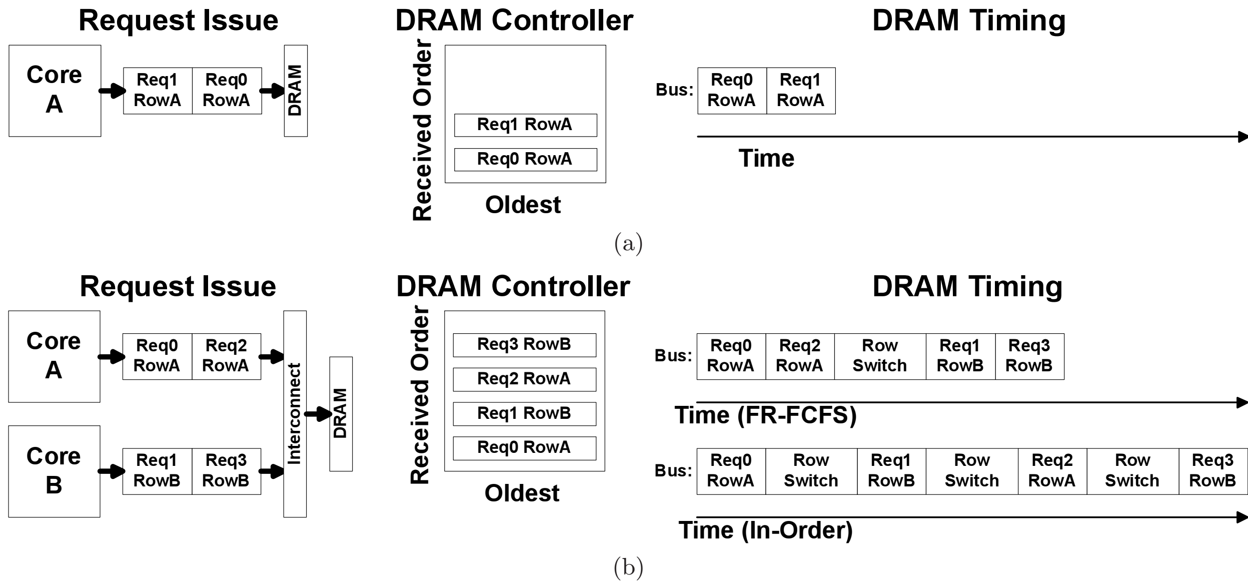


Figure 4: Conceptual example showing effect of request interleaving due to interconnect (all requests to a single bank)

3. ROW ACCESS LOCALITY-AWARE INTERCONNECT ARBITERS

In Section 3.1, we will first quantify the destruction of the DRAM row locality caused by the interconnect. Section 3.2 details our solution for preserving the row locality of the memory request streams as they travel through the interconnect by allowing router inputs to hold grant. In Section 3.3, we describe how having multiple virtual channels can also destroy row locality and our solution to this problem. In Section 3.4, we perform an analysis on the complexity of our proposed interconnection network design coupled with a in-order DRAM scheduler in comparison with a system with a conventional interconnect and out-of-order DRAM scheduling.

3.1 Effect of Interconnect on Row Locality

To determine the effects of memory request stream interleaving due to multiple shader core inputs, we augmented our simulator, which has the architecture shown in Figure 5, to measure the pre-interconnect row access locality versus the post-interconnect row access locality for a set of CUDA applications.

Figure 6 presents the measured pre-interconnect row access locality versus the post-interconnect row access locality (Figure 6(a)) for both mesh and crossbar networks as well as the calculated ratio (Figure 6(b)). As can be seen, the row access locality decreases by almost 47% on average across all network topologies after the interconnect versus the post-interconnect locality across all applications.

Our configuration for Figure 6 has 28 shader cores and 8 memory channels (one DRAM controller per memory channel), which corresponds roughly to NVIDIA's GT200 architecture of 30 streaming multiprocessors and 8 memory channels [4].

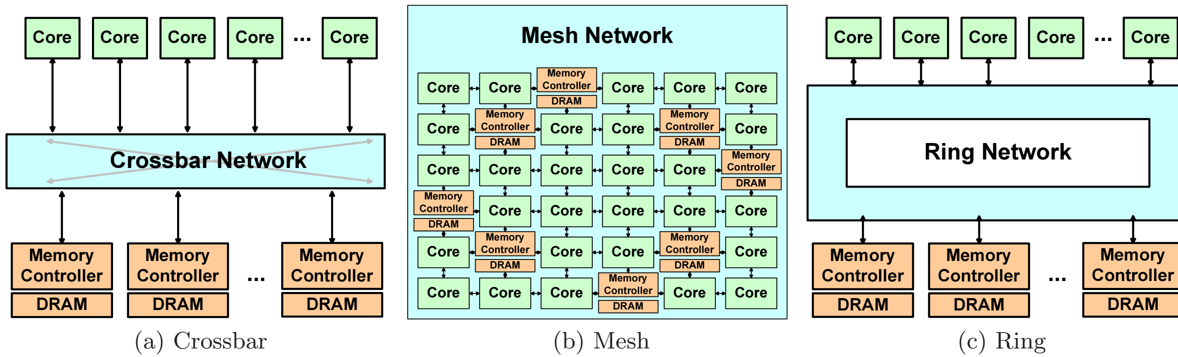
3.2 Grant Holding Interconnect Arbitration

One of the fundamental pillars of conventional interconnection networks and their many arbitration policies is the concept of fairness, such that all input-output combinations receive equal service [7]. Without some degree of fairness in arbitration, nodes in the network may experience starvation. One common method of achieving fairness is to perform round-robin arbitration so that the most recent node that gets serviced becomes the lowest-priority in the next arbitration cycle to allow other nodes to get serviced. As seen in the previous section, such a policy can significantly reduce the amount of row access locality in the memory request stream seen by the DRAM controller. With an in-order memory request scheduler, this can lead to lower DRAM throughput. We therefore propose an interconnect arbitration scheme that attempts to preserve the row access locality exhibited by memory request streams from individual shader cores. To this end, we introduce two simple, alternative modifications that can be applied to any arbitration policy:

"Hold Grant" (HG): If input I was granted output O in the previous arbitration cycle, then input I again gets the highest priority to output O in the current cycle.

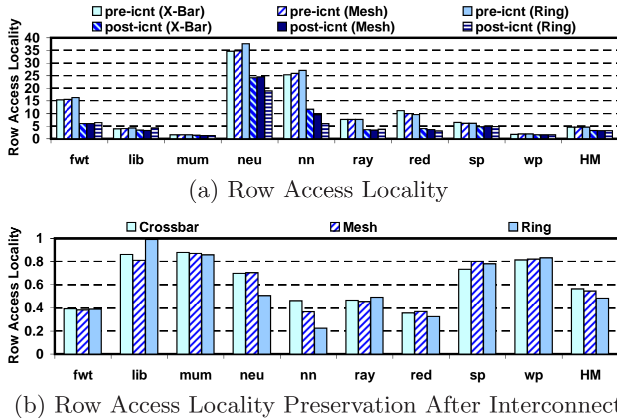
"Row-Matching Hold Grant" (RMHG): If input I was granted output O in the previous arbitration cycle, then input I again gets the highest priority to output O in the current cycle as long as the requested row matches the previously requested row match.

With HG, as long as a shader core has a continuous stream of requests to send to a DRAM controller, the interconnect will grant it uninterrupted access. RMHG has more constraints on when to preserve grant in order be more fair, but may not perform as well since it will not preserve any inherent bank-level parallelism found in the memory request stream, found to be important in reducing average thread stall times in chip multiprocessors by Mutlu et al [12]. To maximize the benefit of using our proposed solution, the network topology and routing algorithm should be such that there is no path diversity. Common examples are a crossbar network or a mesh with non-adaptive routing. (We leave the development of interconnect arbitration schemes that



(a) Crossbar (b) Mesh (c) Ring

Figure 5: Architecture (DRAM chips are off-chip)



(a) Row Access Locality

(b) Row Access Locality Preservation After Interconnect

Figure 6: Measured row access locality before the interconnect (Pre-Icnt) and after (Post-Icnt). HM = Harmonic Mean

preserve row access locality in network topologies with path diversity to future work.) This forces requests sent to the interconnect to leave in the same order. If requests were otherwise allowed to arrive at the DRAM controllers out of order, the row access locality in the memory request stream sent from shader cores may also be disrupted in this way.

In the standard interconnection network router pipeline, there are two stages where our hold-grant policies can be enforced: virtual channel allocation, which happens first, and switch allocation. In our experiments, we enforce our hold-grant policies during virtual channel allocation. If this is not done, virtual channel allocation will interleave packets. In this case, enforcing hold-grant during switch allocation would be ineffective since the packets arbitrated during virtual channel allocation will already be interleaved. To prevent switch allocation from undoing the effects of our hold-grant policy, we restrict the virtual channel allocation as described in the next section.

3.3 Interleaving due to Multiple Virtual Channels

Interconnection networks having multiple virtual channels can reduce network latency by reducing router pipeline stalls [7]. In certain network topologies and with certain routing algorithms, it can also be necessary to prevent deadlock.

In the context of memory request stream interleaving, multiple virtual channels effectively can cause a stream of memory requests sent from one source to one destination to arrive out of order. This occurs since a request that enters a network router gets assigned the first virtual channel that becomes available "on-the-fly". (We refer to this as dynamic virtual channel assignment, or *DVCA*.) When back-to-back requests from one source shader core are sent to one destination DRAM controller, they may get assigned to different virtual channels, where each virtual channel is essentially a different path. This is akin to having path diversity, which was explained in Section 3.2 to be detrimental in preserving row access locality. With requests taking multiple paths to the destination, order cannot be enforced.

To handle the case when multiple virtual channels are used in the interconnect, we propose a static, destination-based virtual channel assignment (*SVCA*). While requests from one input node can still use multiple virtual channels in this assignment, all requests from any shader core (input node) to one specific DRAM controller (output destination node) are always assigned the same virtual channel in each router. In the case where there are virtual channels VC_0 to VC_{v-1} for each router and M DRAM controllers, then all requests to DRAM controller M must use VC_n and only VC_n, where $n = M \mod v$. Doing so allows for a fair virtual channel allotment across all input-output combinations.

Interleaving of requests during switch allocation would still occur, but only between requests to different memory controllers, which does not destroy row access locality.

We evaluate the performance impact of using SVCA versus DVCA in Section 5.3 and show that applications obtain a speedup of up to 17.7% in a crossbar network with four virtual channels per port when using SVCA over DVCA.

3.4 Complexity Comparison

Since our proposed changes are not only limited to the DRAM controller circuitry, but also to the interconnection network, we attempt to define the complexity with a general-enough method that can be applied to any network topology. To this end, we quantify the complexity of our design based on the differences in two metrics, the amount of storage required in bits, and the number of bit-comparisons.

With FR-FCFS, whenever a new command is issued, every request in the DRAM controller queue must be checked to determine the best candidate for issuing [8, 19]. For a queue size Q, this translates to Q times the number of row

Table 1: **Calculated complexity (HMHG4 = Hash-Matching Hold Grant using 4-bit hashes)**

	Bits Stored in ICNT	Bits Compared in ICNT	Bits Compared in DRAM Scheduler	Total
FRFCFS	0	0	M*Q*($log_2(n_R)$+$log_2(n_B)$)	3584 bits compared
BFIFO+HG (XBAR)	C*M	C*M	0	224 bits stored 224 bits compared
BFIFO+HG (MESH)	20*(C+M)	20*(C+M)	0	720 bits stored 720 bits compared
BFIFO+RMHG (XBAR)	C*M + $log_2(n_R)$*n_B*M	C*M + $log_2(n_R)$*M	0	608 bits stored 320 bits compared
BFIFO+RMHG (MESH)	20*(C+M) + (C+M)*$log_2(n_R)$*n_B*M	20*(C+M) + (C+M)*$log_2(n_R)$*5	0	14,544 bits stored 2880 bits compared
BFIFO+HMHG4 (XBAR)	C*M + 4*M	C*M + 4*M	0	320 bits stored 320 bits compared
BFIFO+HMHG4 (MESH)	20*(C+M) + (C+M)*4*5	20*(C+M) + (C+M)*4*5	0	1440 bits stored 1440 bits compared

bits + the number of bank bits. In the GDDR3 technology that we study, there are 4096 rows (n_R) and 4 banks (n_B) , which is equal to 12 row bits ($log_2(n_R)$) and 2 bank bits ($log_2(n_B)$). For a GPU system with M DRAM controllers, this means a maximum number of bit comparisons of $M * Q * (log_2(n_R) + log_2(n_B))$ every DRAM command clock. This number represents an upper bound and can be much lower in certain scenarios. First, the average occupancy of the DRAM controller queue may be very low, especially if the application is not memory-intensive. Second, the DRAM request search may be performed in a two step process where only requests to available banks are selected to check for row buffer hits. Available banks include those that have pending requests and are not in the process of precharging (closing) a row or activating (opening) a row.

Since a DRAM controller that implements FR-FCFS needs to perform fully-associative comparison, it does not need to be banked based on the number of DRAM banks. This allows the controller to maximize its request storage capability if, for instance, all requests go to a single bank. For our complexity-effective solution, we use a banked FIFO queue with one bank per DRAM bank. Doing so allows us to consider requests to all banks simultaneously, thus promoting bank-level parallelism. With this configuration, the DRAM controller considers up to n_B requests each cycle, checking whether the row of the request matches the opened row for each bank to determine whether the DRAM controller can issue the request immediately or whether it needs to first open a new row. Furthermore, when requests are first inserted into the DRAM controller queue, the DRAM controller must check to which of the banks to insert the request, requiring an additional $log_2(n_B)$ comparisons. This adds up to $M * (n_B * log_2(n_R) + log_2(n_B))$ bit comparisons summed across all DRAM controllers.

Our interconnect augmentations to the interconnect, HG and RMHG, also require additional bit comparisons. In this paper, we perform our study using parallel iterative matching (PIM) allocation, a simple algorithm easily realizable in hardware [2], which is important in high-speed interconnects. In PIM allocation, a request matrix of input rows by output columns is first generated to determine which inputs require which outputs. In the context of a crossbar, the number of inputs is the number of shader cores C and the number of outputs is the number of DRAM controllers M. PIM is a type of separable allocator, where arbitration is done in two stages, which is advantageous where speed is important [7]. In our study, we perform input-first allocation. Figure 7 shows a simple example of an input-first

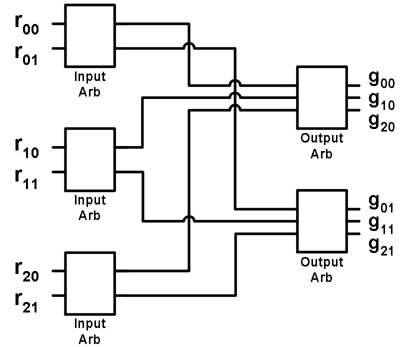


Figure 7: Block diagram of a Parallel Iterative Matching (PIM) separable allocator for 3 inputs and 2 outputs [7]. Each input arbiter (Input Arb) chooses from one of its input request signals r_{IO} to assert, then each output arbiter (Output Arb) asserts one grant signal g_{IO}. (Where I is the input and O is the output).

allocator with three inputs and two outputs. First, the input arbiters select from a single request per input port. In other words, each input arbiter chooses to assert up to one output, depending on whether there are requests from the corresponding input to the corresponding output. The output of this first stage of input arbiters is then fed to the output arbiters, which choose from the different requests to each output. This guarantees that a single input port will not be sending to multiple output ports and a single output port will not be receiving from multiple input ports in the same cycle.

As mentioned earlier, we implement our grant-holding policy at the virtual channel allocator. If we are using HG, then we check whether input-output combinations granted in the previous arbitration cycle are also pending in this cycle. If so, we force allocation to choose the same input-output combinations again. This requires knowing which output was chosen in the last arbitration cycle for each input arbiter and, if there is a request to that output again this cycle, then it is chosen again. To do this, we store one bit for each input-output combination and set the bits of the combinations that were previously granted. We then compare these bits to the ones in the request matrix and, if there is a match, we clear the other request bits to the other inputs for the corresponding output and the other outputs for the corresponding inputs. This essentially forces the allocation algorithm to choose the same input-output combination for

the next cycle because no other options are available. For a crossbar with one router, this totals to C*M bit comparisons and additional bits of storage required. In the context of a mesh, the number of routers is equal to C+M and an upper bound of nine total input and output ports for a two dimensional mesh. This is because we do not care about the output port in a shader core node to itself (since the shader core cannot sink memory requests) and the input port in a memory node from itself (since the memory controller cannot generate memory requests). (Some nodes on the edge or corners of the mesh will also have fewer ports.) This sums up to (C+M)*(20) bits compared and stored since each router will either have 4 inputs and 5 outputs or 5 inputs and 4 outputs for a total of 20 input-output combinations.

An interconnect allocation policy that uses RMHG requires more bit comparisons since, in addition to the bit comparison and storage overheads incurred using HG, it must also check that back-to-back requests are indeed to the same row, thus requiring $log_2(n_R)$ comparisons for each entry in the request matrix. Compounding this problem for the crossbar and, to a much greater extent, the mesh is that requests sent to the same output may be to ultimately different destinations, such as to different banks or, in the case of the mesh, to different chips. One solution to account for this is to have each router keep track of the rows of the most recently granted request to the different possible destinations for each output. Therefore for a single router, which is the case of the crossbar, this would require a total of $log_2(n_R)$*n_B*M additional bits of storage for the interconnect and $log_2(n_R)$*M bit comparisons. Accordingly, each router of a mesh would require $log_2(n_R)$*n_B*M additional bits of storage and $log_2(n_R)$*5 bit comparisons, $log_2(n_R)$ for each output. Summed across all routers in a mesh, the total requirements is (C+M)*$log_2(n_R)$*n_B*M additional bits of storage and (C+M)*$log_2(n_R)$*5 bit comparisons.

Compared to using HG, a mesh network using RMHG can incur significant bit-comparison and area overheads, even more than FR-FCFS. However, it is possible to reduce both bit comparison and bit storage overheads by matching a hash of the row instead of finding exact matches and storing fewer row entries than the maximum required of n_B*M. We call this heuristic *Hash-Matching Hold Grant (HMHG)*. In our experiments, we match and store only a 4-bit hash of the row bits for the last granted request for each output for each router (HMHG4) instead of using RMHG. We found that using more bits did not improve performance. HMHG4 has an overhead of (C+M)*4*M bit comparisons and (C+M)*4*5 bits stored for the mesh and 4*M bit comparisons and 4*M bits stored for the crossbar. Table 1 summarizes our findings. The right-most column shows the calculated complexity for our baseline configuration with 28 shader cores, 8 memory controllers, and each DRAM chip having 4096 rows and 4 banks.

In the next section, we will describe our experimental methodology. In Section 5, we will quantify the effects of using our complexity-effective DRAM controller design in comparison to conventional in-order and out-of-order scheduling memory controllers.

4. METHODOLOGY

To evaluate our design, we used GPGPU-Sim [3], a publicly available cycle-accurate performance simulator of a GPU, able to run CUDA applications. We implemented our changes to the interconnection network simulator, booksim (introduced by Dally et al. [7]), that is used in GPGPU-Sim. Our interconnect configurations, such as router cycle time, network topology, and routing protocols, were chosen based on findings by Bakhoda [3] that performance was relatively insensitive to them and, as such, as we did not spend extraneous effort in tuning them. We evaluated mesh, crossbar, and ring network topologies to show that destruction of the row access locality due to interleaving occurs universally.

Table 2: Microarchitectural parameters (bolded values show baseline configuration)

Parameter	Value
Shader Cores	**28**
Threads per Shader Core	**1024**
Interconnection Network	Ring, **Crossbar**, Mesh
Maximum Supported In-flight Requests per Multiprocessor	**64** [22]
Memory Request Size (Bytes)	**64**
Interconnect Flit Size (Bytes)	**16, 64 (for Ring)**
Interconnect Virtual Channels	**1**,2,4
Interconnect Virtual Channel Buffer Size (Packets)	**8**
DRAM Chips	**16**
DRAM Controllers	**8**
DRAM Chips per Controller	**2**
DRAM Controller Queue Size	8,16,**32**,64
DRAM Controller Scheduling Policy	**First-Ready First-Come First-Serve (FR-FCFS)**, Naive FIFO, Banked FIFO [18]
GDDR3 Memory Timing	t_{CL}=**9**, t_{RP}=**13**, t_{RC}=**34** t_{RAS}=**21**, t_{RCD}=**12**, t_{RRD}=**8**

Table 3: Benchmarks

Benchmark	Label	Suite
Fast Walsh Transform	fwt	CUDA SDK
LIBOR Monte Carlo	lib	3rd Party
MUMmerGPU	mum	3rd Party
Neural Network Digit Recognition	neu	3rd Party
Ray Tracing	ray	3rd Party
Reduction	red	CUDA SDK
Scalar Product	sp	CUDA SDK
Weather Prediction	wp	3rd Party
Nearest Neighbor	nn	Rodinia

Table 2 shows the microarchitectural parameters used in our study. Using GPGPU-Sim allows us to study the massively multi-threaded architectures of today's GPUs, which stress the DRAM memory system due to the many simultaneous memory requests to DRAM that can be in-flight at any given time. In our configuration, we allow for 64 in-flight requests per shader core, amounting up to 1792 simultaneous in-flight memory requests to DRAM. In comparison, Prescott only supports eight [5] and Willamette supports four [9]. We evaluate our design on the 9 different applications shown in Table 3. In order to have results that are more meaningful, we chose our applications by pruning from a much larger set based on whether or not they met all three of our selection criteria: First, the total processor throughput of the application must be less than 75% of the peak IPC for the baseline configuration. Second, the DRAM utilization, percentage time spent of a memory channel transferring data over all time, averaged across all DRAM controllers must be over 20%. Finally, the DRAM efficiency averaged across all DRAM controllers must be less than 90%. Following these three criteria ensures that we

study only applications that are memory-sensitive[2]. We use 3 applications from NVIDIA's CUDA software development kit (SDK) [14], 5 applications from the set used by Bakhoda et. al [3], and one application from the Rodinia benchmark suite introduced by Che et al [6]. We simulate each application to completion.

5. RESULTS

In Section 5.1, we will first show how the interconnect modifications of our proposed solution as described in Section 3.2 performs in comparison to naive FIFO and FR-FCFS. Trends across different network topologies are similar so we only present results for the crossbar network in some places due to space limitations. Section 5.2 provides an in-depth analysis on how our interconnect modifications effect the memory request address streams seen at the DRAM controllers. In Section 5.3, we compare static destination-based virtual channel assignment to dynamic virtual channel assignment. In Section 5.4, we perform a sensitivity analysis of our design across different microarchitectural configurations.

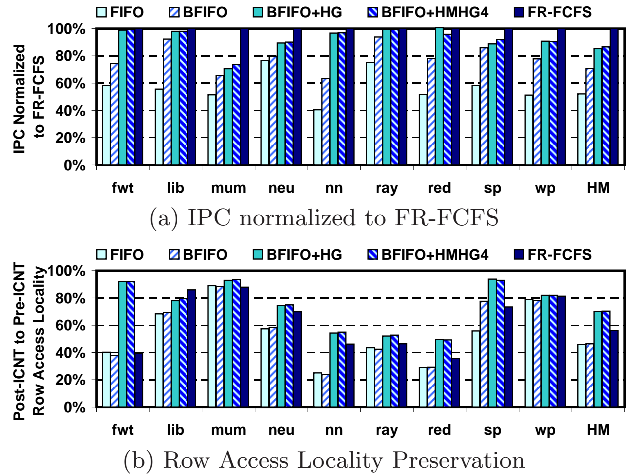
5.1 Complexity Effective DRAM Scheduling Performance

We begin by presenting in Figure 12 the performance of an unbanked FIFO DRAM scheduler, a banked FIFO dram scheduler, and a banked FIFO DRAM scheduler with our interconnection network modifications relative to an out-of-order FR-FCFS scheduler for mesh, crossbar, and ring networks. Since RMHG was found in Section 3.4 to be relatively area intensive, we only show results for the Hold-Grant (HG) modification and Hash-Matching Hold-Grant modification using 4 bit hashes (HMHG4).

We present the per-application IPC normalized to FR-FCFS for the crossbar network in Figure 8(a). The banked FIFO controller outperforms the unbanked controller across the board, indicating that the banked FIFO controller's better proficiency at exploiting bank-level parallelism drastically outweighs the additional potential queueing capacity of the unbanked controller. The row access locality preservation in Figure 8(b) is calculated by dividing the post-interconnect locality of the various configurations by the pre-interconnect locality for FR-FCFS. Our interconnect modifications help preserve over 70% of the row access locality while the baseline interconnect (along with FR-FCFS scheduling) results in only 56.2% preservation.

Figure 9 shows the effect of the HG and HMHG4 interconnect modifications on average memory latency and DRAM efficiency for the crossbar network. Figure 9(a) shows that naive scheduling of memory requests, without our interconnect modifications, in a system that supports thousands of in-flight memory requests can cause the memory latency to increase dramatically (FIFO). Our HM and HMHG4 is able to reduce the latency of an in-order, banked DRAM controller (BFIFO) by 33.9% and 35.3% respectively (BFIFO-HG and BFIFO-HMHG4). Furthermore, Figure 9(b) shows that both HM and HMHG4 improve the DRAM efficiency of BFIFO by approximately 15.1%.

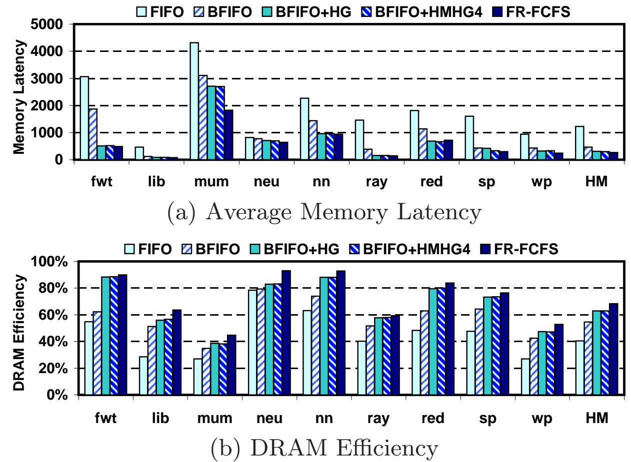
[2]In general, we found that applications that did not match our criteria were insensitive to out-of-order DRAM scheduling optimizations, which can only make a complexity-effective memory controller design for this type of architecture more appealing.



(a) IPC normalized to FR-FCFS

(b) Row Access Locality Preservation

Figure 8: Baseline configuration results for crossbar network, queue size = 32 (HM = Harmonic mean)



(a) Average Memory Latency

(b) DRAM Efficiency

Figure 9: Memory latency and DRAM efficiency (HM = Harmonic mean, Queue size = 32)

5.2 Row Streak Breakers

In order to gain a better understanding of what causes the performance gap between our complexity-effective design and FR-FCFS, we perform an in-depth analysis of the scheduled memory requests in relation to the DRAM controller queue contents for the banked FIFO scheduler: First, we define a sequence of row-buffer hits to any single bank as a single *row streak*. Whenever a DRAM bank switches from an old Row X to a new Row Y (e.g. when a row streak ends), we search backwards through the FIFO queue corresponding to this DRAM bank starting from the pending request to Row Y, looking for any other requests to Row X. We define these requests that we are looking for as *stranded requests* since, in a FR-FCFS scheduler, these stranded requests would also be serviced before the row switches. If we find a stranded request, we look at the requests between the stranded request and the pending request to Row Y. In an ideal interconnection network, these requests, which we call *row streak breakers*, would be prioritized to arrive at the DRAM controller after the stranded request, thus maximiz-

ing row access locality. Furthermore, a FR-FCFS scheduler would prioritize these row streak breakers lower than the stranded request.

Figure 10 characterizes the source of the row streak breakers for the banked FIFO configuration with and without our interconnect augmentations. Each time a stranded request is found using our method described in the previous paragraph, we increment the counter for each category *once* if *any* requests in the corresponding category are found in the set of row streak breakers. In this way, for any set of row streak breakers, more than one category may be credited. We categorize row streak breakers into three possibilities, those originating from a different shader core, those from the same core but from a different cooperative thread array[3] , or *CTA*, and those from the same CTA. Row streak breakers originating from different shader cores dominate the distribution. Furthermore, our interconnect augmentations significantly reduce the number of these particular row streak breakers. Note that the bars in Figure 10 do not add up to 100% because sometimes there are no stranded requests. In these situations, FR-FCFS would make the same scheduling decision as our banked FIFO scheduler, oldest-first. In other words, shorter bars indicate that the performance gap between the corresponding configuration and FR-FCFS is also smaller.

In virtually all cases, our interconnect augmentations reduce the number of row streak breakers due to different shader cores to less than 10% of the total row streaks. One exception is *neu*, where our interconnect augmentations can reduce the row streak breakers to no less than 30% of the total number of row streaks. An in-depth analysis of the requests that comprise row streaks showed that for most applications, the requests from most row streaks originate from a single shader core, explaining why an input-based hold-grant mechanism works so well in this architecture. However, for *neu* the requests of any row streak when using the FR-FCFS scheduling policy were from more than four different shader cores on average (4.18). In this case, FR-FCFS will be able to capture row access locality that our interconnection augmentations cannot.

5.3 Static Virtual Channel Assignment

In this section, we compare the performance results of using static destination-based virtual channel assignment (SVCA) to more conventional dynamic virtual channel assignment (DVCA) in the case when there are multiple virtual channels per router port.

Figure 11 shows the IPC harmonically averaged across all applications for different virtual channel configurations: one virtual channel per port (*vc1*), two virtual channels per port with SVCA (*svc2*), two VCs with DVCA (*dvc2*), four VCs with SVCA (*svc4*), and four VCs with DVCA (*dvc4*). Here, we increase the number of virtual channels while keeping the amount of buffering space per virtual channel constant. When using our complexity-effective DRAM scheduling solution, *BFIFO+HG* and *BFIFO+HMHG4*, performance decreases by 21.3% and 21.7% respectively as the number of virtual channels is increased from one to four while using

[3]A cooperative thread array (CTA) is a group of threads run on a single core that have access to a shared memory and can perform barrier synchronization. One shader core can potentially run multiple CTAs concurrently if there are enough on-chip resources [15].

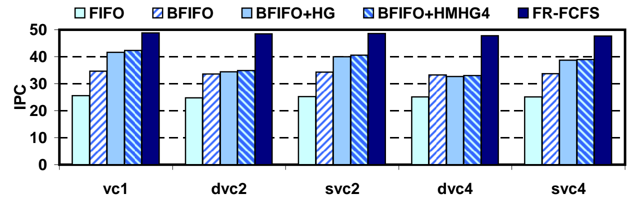


Figure 11: Harmonic mean IPC for different virtual channel configurations (Crossbar network)

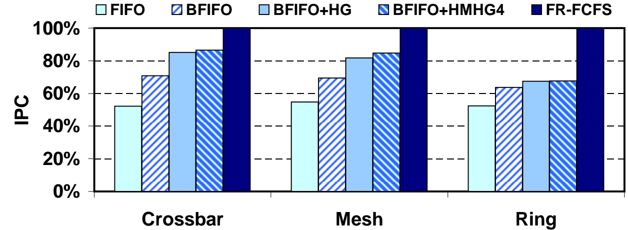


Figure 12: Harmonic mean IPC across all applications for different network topologies. (Queue size = 32)

DVCA. Using SVCA (*svc*4) recovers most of this performance loss, improving upon *dvc*4 by 18.5% and 17.7%. In general, we find that adding virtual channels does not improve IPC, which matches one of the findings of Bakhoda et al. [3] that CUDA workloads are generally insensitive to interconnect latency.

5.4 Sensitivity Analysis

To show the versatility of our interconnect modifications, we test them across two major microarchitectural parameters: the network topology and the DRAM controller queue size.

As Figure 12 shows, our interconnect modifications perform fairly similarly for two very different network topologies: a crossbar and a mesh. For these topologies, HMHG4 achieves 86.0% and 84.7% of the performance of FR-FCFS for memory-bound applications while requiring significantly less bit comparisons than FR-FCFS. For comparison, we also present the results for a ring network, which requires a minimum of two virtual channels for deadlock avoidance [7]. The deadlock avoidance algorithm for the ring imposes restrictions on which virtual channels can be used by any input-output combination to remove cyclic dependencies. To provide adequate bandwidth, we provision the ring network with a 512-bit wide datapath similar to the one in Larrabee [20]. While a ring network has the same number of nodes as a mesh, each router has only three input and output ports, one to the left, one to the right, and one to the shader core or DRAM controller that it is connected to. In comparison, each mesh router has 5 input and output ports, so the complexity as detailed in Section 3.4 of the ring will be 60% (or 3/5ths) of that of the mesh. As shown, our interconnect modifications do not work as well for the ring network due to the interleaving of memory request streams from multiple virtual channels. In-depth analysis showed that the ring network had more than six times as many row streak breakers as the mesh and crossbar networks, thus achieving only 6.3% speedup over a banked FIFO memory controller with no interconnect modifications. We leave the development of

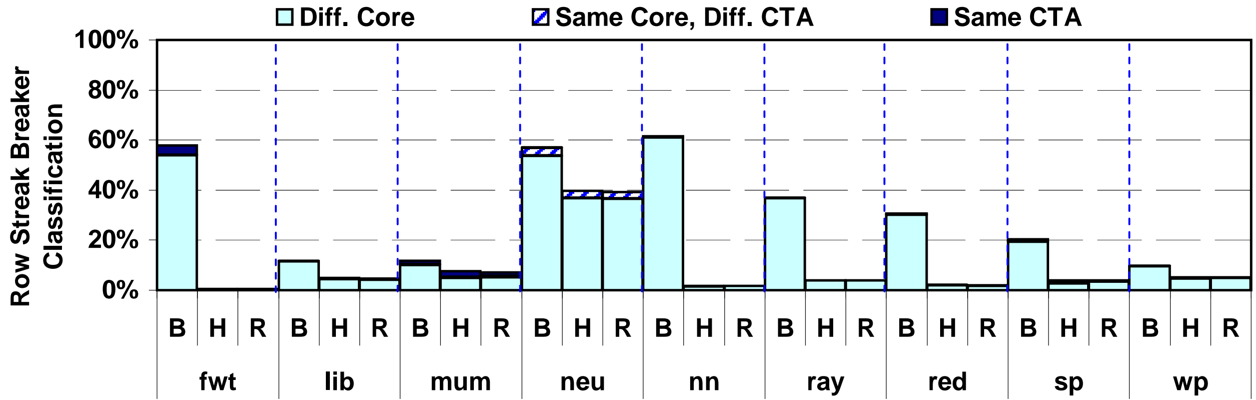


Figure 10: Row streak breakers normalized to row streaks in Baseline (B = Banked FIFO; H = Banked FIFO + Hold Grant; R = Banked FIFO + Hash-Matching Hold Grant with 4-bit row hashes)

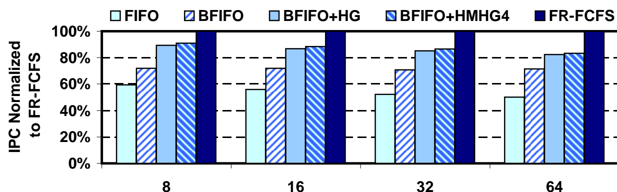


Figure 13: Harmonic mean IPC normalized to FR-FCFS for different DRAM controller queue sizes. (Crossbar network)

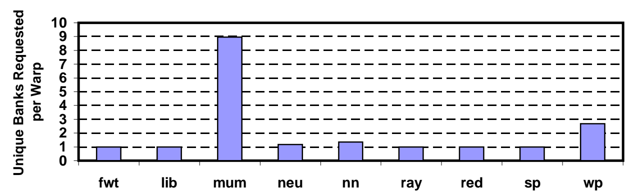


Figure 14: Average number of unique banks requested per warp memory operation

an interconnect modification that works better for this case to future work.

Figure 13 shows the performance of our interconnect modifications harmonically averaged across all applications in a crossbar network and with different DRAM controller queue sizes. For the smallest configuration of DRAM controller queue size 8, BFIFO+HMHG4 achieves 91% of the IPC of FR-FCFS. While the performance relative to FR-FCFS decreases as the DRAM controller queue size increases, the complexity increases since the number of comparisons per cycle scales for FR-FCFS [8, 19] but remains constant for a banked FIFO controller with our interconnect modifications.

6. RELATED WORK

There exist many DRAM scheduler designs proposed for multi-core systems [13, 17, 11, 12]. The primary focus of these designs revolve around the principles of providing fairness or Quality-of-Service (QoS) for different threads and cores competing for shared off-chip bandwidth [13, 17, 11]. When multiple applications run simultaneously on a system, the memory access traffic of one application can cause a naive DRAM controller to starve out the requests of another. To the best of our knowledge, this is the first work that examines the problem of efficient DRAM access scheduling in a massively multithreaded GPU architecture with tens of thousands of threads.

Mutlu et al. [12] present a parallelism-aware batching scheduler (PAR-BS) that coordinates the servicing of multiple requests from a single thread, particularly those to different banks in a DRAM chip, to reduce the average memory latency experienced across all threads. We anticipate the design of such a scheduler to support requests from tens of thousands of threads, which is possible in GPU architectures, to be highly complex and area-intensive. Furthermore, threads stall at the warp-level, such that all threads in a warp must be ready before the warp can be issued to the shader core pipeline. We measure the average number of unique banks to which a warp sends requests, shown in Figure 14. Five out of the nine applications always send requests to only a single bank per warp. In our hold-grant interconnect arbiter, the requests in a warp will be kept together as they traverse the interconnect and they will all arrive at the same bank, so batching is already done. In *mum*, each warp sends requests to at least three different memory controllers per warp memory operation (since each DRAM chip has only 4 banks). We expect PAR-BS to perform poorly in this case as well if there is no communication among the different memory controllers.

There already exist a circuit design to implement grant-holding for arbiters [7]. In contrast to our usage of such a circuit to optimize performance, the original proposal is to hold grant if a flit transmission takes multiple cycles to perform (which would be required for correct transmission of a flit).

7. CONCLUSIONS

We introduced a novel, complexity-effective DRAM access scheduling solution that achieves system throughput within up to 91% of that achievable with aggressive out-of-order scheduling for a set of memory-limited applications. Our solution relies on modifications to the interconnect that relays memory requests from shader cores to DRAM controllers. These modifications leverage the key observation

that the DRAM row buffer access locality of the memory request streams seen at the DRAM controller after they pass through the interconnect is much worse than that of the individual memory request streams from the shader core into the interconnect. Three such modifications are possible: either holding grant for a router input port as long as there are pending requests to the same destination (HG), holding grant for a router input port as long as there are pending requests to the same destination and the requested row matches the requested row of the previous arbitrated request (RMHG), and holding grant for a router input port as long as there are pending requests to the same destination and the hash of the requested row matches the hash of the requested row of the previous arbitrated request (HMHG). These modifications work to preserve the inherent DRAM row buffer access locality of memory request streams from individual shader cores which would otherwise be destroyed due to the interleaving of memory request streams from multiple shader cores. In doing so, it allows for a simple in-order memory scheduler at the DRAM controller to achieve much higher performance than if the interconnect did not have such modifications across different network topologies and microarchitectural configurations.

Acknowledgments

We would like to thank John H. Edmondson and the anonymous reviewers for their valuable feedback on this paper. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

8. REFERENCES

[1] Advanced Micro Devices, Inc. *ATI CTM Guide*, 1.01 edition, 2006.

[2] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, 1993.

[3] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pages 163–174, April 2009.

[4] Beyond3D. NVIDIA GT200 GPU and Architecture Analysis. http://www.beyond3d.com/content/reviews/51.

[5] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The Microarchitecture of the Intel® Pentium® 4 Processor on 90nm Technology. *Intel® Technology Journal*, 8(1), 2004.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008. General-Purpose Processing using Graphics Processing Units.

[7] W. J. Dally and B. Towles. *Interconnection Networks*. Morgan Kaufmann, 2004.

[8] R. E. Eckert. Page Stream Sorter for DRAM Systems, Assignee NVIDIA Corporation, United States Patent 7,376,803, May 2008.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel® Technology Journal*, 5(1), 2001.

[10] Int'l Technology Roadmap for Semiconductors. 2007 Edition. http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[11] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, pages 146–160, 2007.

[12] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proc. 35th Int'l Symp. on Computer Architecture*, pages 63–74, 2008.

[13] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proc. 39th IEEE/ACM Int'l Symp. on Microarchitecture*, pages 208–222, 2006.

[14] NVIDIA Corporation. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

[15] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 2.2 edition, 2009.

[16] Qimonda AG. 512Mbit GDDR3 DRAM, Revision 1.1 (Part No. HYB18H512321BF). http://www.alldatasheet.com/datasheet-pdf/pdf/207161/QIMONDA/HYB18H512321BF.html, September 2007.

[17] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. In *Proc. 16th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 245–258, 2007.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proc. 27th Int'l Symp. on Computer Architecture*, pages 128–138, 2000.

[19] H. G. Rotithor, R. B. Osborne, and N. Aboulenein. Method and Apparatus for Out of Order Memory Scheduling, Assignee Intel Corporation, United States Patent 7,127,574, October 2006.

[20] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 1–15, 2008.

[21] The Khronos Group. *OpenCL 1.0 Specification, http://www.khronos.org/registry/cl/.*

[22] J. Tuck, L. Ceze, and J. Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *39th IEEE/ACM Int'l Symp. on Microarchitecture*, pages 409–422, 2006.