2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10070957

Mitigating GPU Core Partitioning Performance Effects

Aaron Barnes
School of ECE
Purdue University
West Lafayette, IN
barnes88@purdue.edu

Fangjia Shen
School of ECE
Purdue University
West Lafayette, IN
shen449@purdue.edu

Timothy G. Rogers
School of ECE
Purdue University
West Lafayette, IN
timrogers@purdue.edu

***Abstract*—Modern GPU Streaming Multiprocessors (SMs) have several warp schedulers, execution units, and register file banks. To reduce area and energy-consumption, recent generations divide SMs into sub-cores. Each sub-core contains a distinct warp scheduler, register file, and execution units, sharing L1 memory and scratchpad resources with sub-cores in the same SM. Although partitioning the SM into sub-cores decreases the area and energy demands of larger SMs, it comes at a performance cost. Warps assigned to the SM have access to a fraction of the SM's resources, resulting in contention and imbalance issues. In this paper, we examine the effect SM sub-division has on performance and propose novel mechanisms to mitigate the negative impacts. We identify four orthogonal effects caused by sub-dividing SMs and demonstrate that two of these effects have a significant impact on performance in practice. Based on these findings, we propose register-bank-aware warp scheduling to avoid bank conflicts that arise when instruction operands are placed in the limited number of register file banks available to each sub-core, and randomly hashed sub-core assignment to mitigate imbalance issues. Our intelligent scheduling mechanisms result in an average 11.2% speedup across a diverse set of applications capturing 81% of the performance lost to SM sub-division.**

***Index Terms*—GPU, Scheduling, Register File, Bank Conflict**

I. Introduction

The latest GPU architectures from both Nvidia and AMD partition large cores into sub-units referred to as sub-cores [3], [23] and dual compute units [2] respectively. Each sub-core contains its own scheduler, Single Instruction Multiple Data (SIMD) execution units, operand collectors, and register file; however, sub-cores within the same SM share the same memory system. This hard-divisioning of the core can result in imbalance and contention issues that are not seen in a fully-connected monolithic SM with the same thread and compute capacity. Figure 1 shows the speedup of a of hypothetical, fully-connected Volta SM over the four sub-core SM found in contemporary machines.

Comparing a fully-connected core to a partitioned one, four potential performance issues are observed. The first is increased register file bank conflicts. Each sub-core has access to $1/4$ of the register file banks on the SM, increasing the likelihood of bank conflicts, which decreases throughput in the operand read stage. Second, each warp scheduler can only issue instructions to one sub-core, if there is an imbalance in the work assigned to each scheduler, SIMD execution units can go unused. Third, in workloads where warps assigned to the same SM have diverse execution unit demands (i.e. some warps makes heavy use of the tensor cores while others do not), lacking access to all the core's execution units decreases utilization. Finally, if warps assigned to an SM have diverse register-file capacity demands, which can occur when SMs execute concurrent kernels, a lack of register space on one sub-core may prevent others with capacity from accepting work.

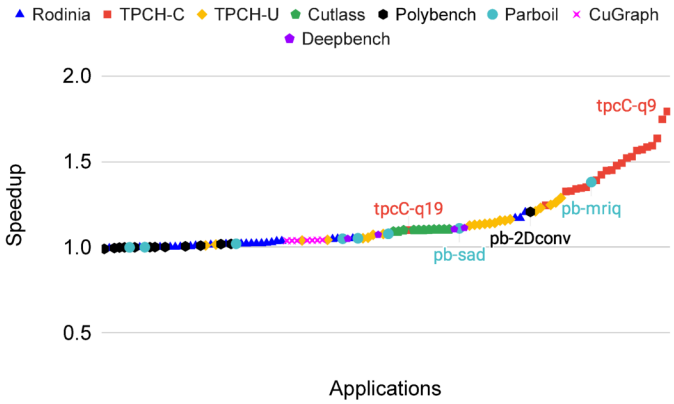


Fig. 1: Simulated speedup of a hypothetical fully connected SM over a four-way partitioned Volta SM for 112 Applications

Although these four factors can potentially harm sub-core performance, we measured the effect of a hypothetical monolithic core on 112 workloads from 8 benchmark suites [10], [12], [13], [16], [22], [30], [55] and observe that register bank conflicts and issue imbalance are responsible for the majority of sub-core performance loss in practice. To mitigate these factors, we develop a novel warp scheduling mechanism to mitigate the effects of additional bank conflicts and propose hashed sub-core warp assignment to reduce imbalance. To the best of our knowledge, we are the first work to study the effects of the GPU sub-core, develop scheduling techniques to mitigate bank conflicts in GPUs, and examine sub-core assignment policies.

A major side-effect of partitioning the SM is a reduction in the number of register file banks available to each warp.

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

In Volta/Ampere for example, each SM has eight register file banks, but each sub-core has access to only two [34], [35]. Even with compiler-orchestrated register swizzling among banks, GPUs still need to make use of operand collectors [24], [47] to keep multiple warps in the operand read stage and maximize register file bandwidth. Increasing the number of warps requesting operands each cycle is a straightforward way to improve register bank utilization. However, doing so involves scaling the number of collector units in each sub-core, which adds area and energy to the SM (evaluated in Section VI-B2), decreasing the gains made by moving to a sub-core design in the first place. Instead of blindly scaling collector units, we develop a novel Register Bank Aware (RBA) warp scheduling mechanism to make better use of the limited banks and collector units available in GPU sub-cores. Although warp scheduling has been extensively studied for other purposes [37], [39], [42], [46], [48], [49], [52], [53], we are the first paper that uses it as a mechanism to mitigate register bank conflicts, which are exacerbated by the partitioned cores in recent GPU generations.

Imbalances across the work assigned to a sub-core can lead to under-utilization and reduced performance. Through extensive microbenchmarking (see Section III-B), we have determined that contemporary Volta and Ampere GPUs assign warps to sub-cores in a simple round-robin fashion. This assignment balances the work across sub-cores, provided warps execute a similar number of instructions. However, not all workloads have well-balanced warps. In particular, highly-optimized workloads that make use of warp specialized programming [11], [14], [19], [20], [31] can exhibit severe imbalance. Warp specialized programming is a technique used to reduce the impact of control-flow divergence by grouping threads with a similar control-flow path into the same warp. Although the SIMD efficiency [27] of warp-specialized programs is high, there can be significant variations in the number of instructions executed by warps in the same thread block. This variation is easily smoothed in a monolithic core where every warp has access to all the SM's resources, but can lead to severe imbalance in sub-core based SMs.

The assignment of warps to sub-cores adds an additional layer to the GPU thread-scheduling hierarchy. Although these sub-cores act as independent units, thread blocks are assigned at the SM granularity. Once a thread block is assigned to an SM, each sub-core is tasked with a subset of the warps that make up the thread block. Table I contrasts the different levels of thread scheduling in a contemporary GPU.

Sub-core warp assignment shares similarities with warp scheduling and thread block scheduling (Table I), but its position in the hierarchy makes it fundamentally different. Sub-core scheduling operates at a warp granularity, however, the assignment of a warp to a sub-core happens only once in the lifetime of the warp and cannot be changed. As a result, decisions are made less frequently than warp scheduling and each individual decision has a greater impact. Sub-core scheduling frequency is more similar to thread block scheduling, however the granularity is different. Thread block scheduling can do nothing about disparities among the warps within a thread block, since it only decides which SM a thread block will be assigned to.

This paper makes the following contributions:

- The first study to improve GPU sub-cores' effect on performance, quantifying the two primary sources of performance degradation: increased register file bank conflicts, and sub-core issue imbalance.
- A novel bank-aware warp scheduler to mitigate the effects of increased bank conflicts in sub-core partitioned register files. A quantitative cost-benefit analysis of the design compared to scaling the number of collector units is presented.
- A hashed sub-core scheduler to eliminate pathological issue imbalances.
- Evaluation of the novel scheduling techniques resulting in an average speedup of 11.2% across all applications and 19.3% on applications sensitive to SM partitioning.

II. Motivation and Background

A. The Streaming Multiprocessor Sub-Core

There has been a trend in scaling not only the number of SMs in each new generation of GPUs, but also the capability of each SM. For example, in 2006 Nvidia introduced their first unified graphics and parallel compute architecture with the Tesla [45]. The Tesla architecture SM contained a single multi-threaded issue unit, eight Streaming Processor (SP) cores, and two Special Functional Units (SFUs). In contrast, each SM in the Ampere A100 (released in 2020) contains four warp schedulers / issue units, four tensor cores, 64 FP32 units, 64 INT32 Units, and 32 FP64 units [3]. As the number of schedulers and functional units within each SM has increased, the need to subdivide the SM to reduce area and power requirements arose. The Maxwell architecture was the first Nvidia design to partition the SM into sub-cores, and all subsequent Nvidia architectures have used a similar SM partitioning. The Maxwell architecture achieved twice the performance per watt of the previous generation, Kepler, while using the same 28 nm process [5]. Sub-core partitioning reduced the die area of each SM while also improving power efficiency. By decreasing the area per SM while providing about 90% of the performance of the Kepler SM, Nvidia was able to increase the total number of SMs from two in the Kepler GK107, to five in the Maxwell GM107, with only a 25% increase in die area.

AMD has also adopted a partitioned Computational Unit (CU) in the RDNA architecture, which it refers to as the dual compute unit. Each dual compute unit contains four wavefront schedulers, one for each of the four SIMD units [2]. Similar to Nvidia's partitioned SM architecture, the two sub-cores share a common L1 data cache and shared memory scratchpad. Each of the four SIMD units in the dual compute unit has its own 20-entry wavefront buffer. Likewise, the GCN architecture also has 4 SIMD units per CU with a 10-entry wavefront buffer per SIMD unit [1]. As work-groups are sent to a dual compute unit the wavefronts within the work-group must be moved into the

Characteristic	Warp Schedulers [37], [39], [42], [46], [48], [49], [52], [53]	Sub-Core Schedulers	Thread Block Schedulers [25], [40], [50], [56]–[60]
Scheduling Granularity	Warp (32 threads)	Warp (32 threads)	Thread Block (1-1024 threads)
Scheduling Frequency	Every Cycle	Upon Thread Block Assignment	Upon Kernel Launch
Deallocation Granularity	Thread Block	Thread Block	Thread Block

TABLE I: Comparison of GPU hardware schedulers

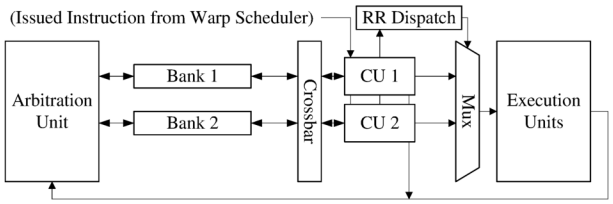


Fig. 2: Operand collector block diagram

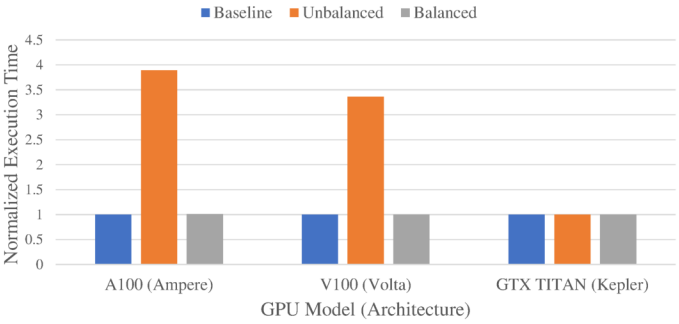


Fig. 3: In-silicon (ran on hardware) GPU microbenchmark test: FMA performance degradation due to sub-core issue imbalances

SIMD unit wavefront buffers, which can lead to a sub-core issue imbalance that cannot be remedied by modifying the wavefront scheduling algorithm.

III. Identifying Sub-Core Performance Effects in Hardware

A. Register File Bank Conflicts

GPUs require huge register files to support the large amount of threads that can be scheduled each cycle. Such large structures have high area and power costs and it can be prohibitively expensive to add additional ports. Several techniques have been proposed to improve GPU register file throughput and reduce power and area requirements [29], [17], [41], [33], but register file banking is the simplest and most commonly used. Register file banking increases maximum bandwidth in a more efficient manner than adding ports, however bank conflicts reduce effective throughput. A bank conflict occurs if a warp instruction's operands are resident in the same bank, and therefore cannot be accessed in a single cycle. The compiler can reduce bank conflicts through carefully selected register assignment, however register access requests from other warps on the sub-core compete for register bank access, and their issue ordering is unknown at compile time.

The Operand Collector is a GPU hardware structure described in Nvidia patents [47] [24] to reduce bank conflicts by

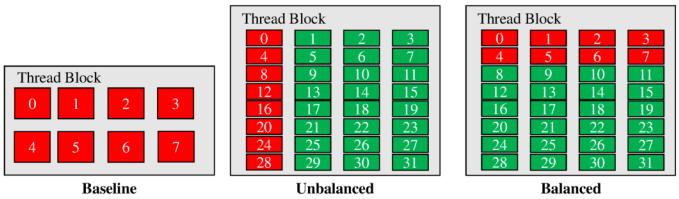


Fig. 4: FMA microbenchmark thread block warp layout (red : compute, green : empty)

processing register file accesses from several warps simultaneously. By processing multiple instructions at once, the Operand Collector hides bank conflicts from a single instruction by processing register file requests for other instructions. Figure 2 shows a block diagram of a baseline Operand Collector which can process two instructions from different warps at once, similar to the Nvidia V100 model. Each Collector Unit (CU) holds state for a single warp instruction and is allocated upon warp scheduler issue. The CU maintains an operand entry for each operand necessary to execute the instruction. Each operand entry contains a ready bit, valid bit, register id, and 32 four-byte data fields (one for each thread in a warp). Upon allocation, CU source operand read requests are sent to the arbitration unit which maintains a queue of read requests for each of the banks. The arbitration unit grants one request for each bank per cycle. When all of the operand entries of a CU are ready, the CU instruction is dispatched to the appropriate SIMD execution unit. The CU is then deallocated and can be filled with a new instruction.

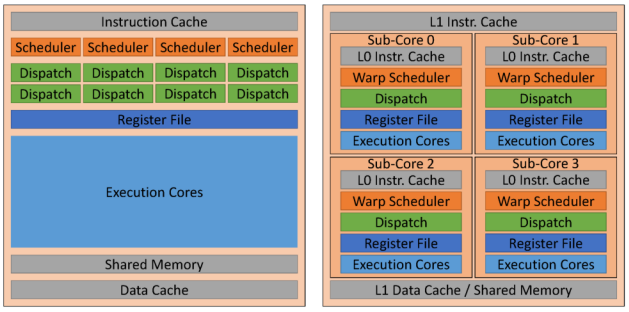
Modern GPUs partition the SM register file into separate structures to improve power and area efficiency. These smaller register file structures typically have a small number of banks. For example, the Ampere A100 and Volta V100 have only two register file banks per sub-core, while previous fully connected architectures had four [34], [35]. Decreasing the number of banks increases the likelihood of bank conflicts. The operand collector, and its associated CUs, are also partitioned among the sub-cores which decreases its capacity to hide bank conflicts. We found several applications highly sensitive to register file throughput, and evaluated the effects of CU scaling and a novel bank-aware warp scheduler in section VI-B.

B. Issue Imbalance

In this work we identify two key characteristics of the sub-core warp assignment mechanism used in modern GPU hardware. First, a simple round-robin warp assignment scheme is used to assign warps to sub-cores. Second, the assignment of warps to sub-cores is static; once a warp is assigned to a sub-core it cannot be re-assigned to execute on a different sub-

core. Resources allocated to a warp can only be freed once the entire thread block the warp belongs to has completed because the programming model guarantees access to a common on-chip shared memory scratch-pad for the entire thread block. Hardware underutilization occurs when a warp assigned to a sub-core has completed execution but cannot be deallocated because it belongs to a thread block still executing on other sub-cores. If all of the warps a sub-core is assigned are in such a state the entire sub-core will stall. In this manner, static assignment leads to substantial performance loss whenever warps are improperly balanced across sub-cores.

Figure 3 shows the performance loss due to sub-core imbalance of a microbenchmark run on three generations of Nvidia GPU hardware. Figure 4 shows the thread block warp layout of each of the three applications. A warp is a group of 32 threads that execute in lock-step, and $warpID = \lfloor threadID/32 \rfloor$. The baseline application includes 256 threads (8 warps) per thread block. Each thread performs 4096 fused multiply add (FMA) instructions on data resident in the register file then waits at a thread-block-wide barrier before exiting. The balanced and unbalanced applications also have 8 warps per thread block performing fused-multiply-add instructions in the same manner as the baseline application, however 768 "empty" threads (24 warps) are additionally included in the thread block. Each empty thread simply waits at the barrier and then exits, without performing any calculation. Warps from a thread block are assigned to a sub-core in sequential order using round robin scheduling, therefore each column in figure 4 will be assigned to a different sub-core on architectures with 4 sub-cores per SM.



(a) Kepler **(b)** Ampere

Fig. 5: Nvidia SM architectures

The results in figure 3 are normalized to the baseline application and indicate that the A100 takes 3.9 times longer to execute when the computation is imbalanced to a single sub-core. The balanced application achieves performance equal to the baseline application which indicates the performance difference is solely due to differences in sub-core balancing. Figure 5 shows the SM block diagrams for Ampere and Kepler architectures. The Ampere SM architecture contains 4 sub-cores per SM [3] while the Kepler architecture has no partitioning [9]. Due to the lack of partitioning in the Kepler architecture no performance difference is seen across the three applications in figure 3. Hopper, Ampere, Turing,

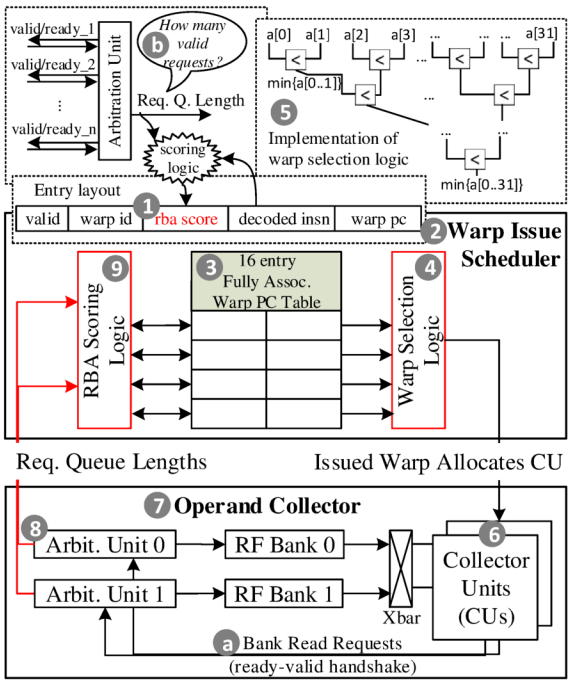


Fig. 6: Register bank aware (RBA) warp scheduler

Volta, Pascal, and Maxwell architectures all feature 2 to 4 sub-cores per SM [3]–[8], and are therefore susceptible to sub-core imbalances. Although real GPU applications may not have empty warps to cause such a stark imbalance, computational variations between warps within a thread block can cause them to finish at significantly different times [58]. We refer to this imbalance between warp finish times within a thread block as *inter-warp-divergence*. Inter-warp-divergence is not to be confused with *intra-warp-divergence*, which refers to threads within a warp taking different control flow paths.

IV. Scheduling Architectures

A. Register Bank Aware Warp Scheduling

Sub-core partitioning has led to the reduction in the number of CUs and register banks available to each warp instruction [34]. These effects have increased pressure on the compiler to avoid register bank conflicts, causing some applications to become limited by the read operand stage (see section VI-B). Increasing the number of CUs is an effective way to improve effective throughput and register file utilization, but comes at a high cost of area and power. Scaling the amount of register file banking is similarly expensive for both area and power [36]. In light of these factors, we propose a Register-Bank-Aware (RBA) warp scheduler, which prioritizes warps that access register file banks with the fewest pending access requests.

Figure 6 outlines the key structures of the RBA design. The additional or modified hardware required to support RBA scheduling is shown in red. Non-red structures are part of the baseline design. For each ready warp instruction, we define its RBA score (①) as the sum of queue lengths of each of its operands. The RBA score is 5 bits wide and

stored in the Warp PC Table(③). The warp scheduler (②) is responsible for arbitrating between decoded ready warps from the instruction buffer and selecting one to issue to an empty CU each cycle. Similar to numerous works studying the warp scheduler [27], [28], [36], [42], [43], [46], we modelled the Warp PC Table (③) as a fully associative table. The warp to issue is found by warp selection logic (④) implemented as a hierarchical comparator network (⑤) that selects the lowest (or highest) scoring entry by comparing a specific field of the entries.

In the baseline design, warps are ordered according to greedy then oldest (GTO) priority, so the warp selection logic compares entries based on their age. Our design modifies the warp selection logic such that it compares warps on the concatenated field of {RBA score, $\overline{\text{warp age}}$}. Here, the overline means taking the complement of age, so older warps score lower. The warp with the lowest RBA score is selected, with older instructions selected in the case of a tie. The selected warp is then issued to an available CU (⑥) in the Operand Collector (⑦).

CUs stage warp instructions while their operands are being retrieved from the register file. Upon allocation, the CU sends bank access requests to the arbitration unit (⑧) for each of its operands. The arbitration unit maintains a separate request queue for each bank of the register file. Priority logic within the arbitration unit then grants bank accesses to CUs each cycle. The only modification necessary to the Operand Collector to support RBA is to send the number of requests in each bank's request queue to the warp scheduler. The scoring logic (⑨) then uses these request queue lengths to compute scores for each ready warp instruction and updates the results in the warp PC table. Scores are computed by summing the length of each queue for each operand of the instruction. For example, if an instruction had three operands, two stored in bank 0, and one stored in bank 1, the score would be computed as: $score = 2 \times length(queue_0) + length(queue_1)$. Two pieces of information are necessary for this computation: the bank ID of each operand of the instruction (present in the baseline design), and the length of each bank's request queue (sent by the modified operand collector). The arbitration unit has dedicated ports per operand per collector unit, exercising a ready-valid handshake protocol (ⓐ), hence the queue length can be conveniently found by summing the number of ports that have a valid request (ⓑ). RBA can tolerate delays in score updates, and does not need to increase the critical path. We evaluated RBA with a 0-20 cycle score update latency and observed $< 0.1\%$ performance degradation (Section VI-B4).

B. Hashed Sub-Core Warp Assignment

1) Hardware Support for Hashed Multiplexing: Kernels with large amounts of inter-warp-divergence are highly sensitive to sub-core imbalances. The assignment of warps to scheduler tables should be designed to mitigate any programmatic warp imbalances. To reduce the likelihood of pathological imbalances the sub-core assignment of a warp should be abstracted from its $warpID$. Replacing the round robin sub-core scheduler used in hardware today with a balanced hashing function can effectively accomplish this goal. Figure 7 shows the hardware structures necessary to support hashed sub-core warp assignment on each SM. The baseline round-robin structures are shown in black, and the additional structures to support hashing are marked in red. The thread block scheduler (①) allocates and frees resources on the SMs as they become available at a thread block granularity. It contains logic which determines which thread block will be scheduled on which SM and sends and receives messages through the interconnect to communicate with each SM. Four key operations must occur when a thread block is scheduled on an SM:

1) Allocate register file space
2) Allocate shared memory space
3) Write kernel arguments to constant memory
4) Load warp program counters (PCs) into warp scheduler PC tables

In order to perform these operations the thread block scheduling unit encodes information about the thread block (such as shared memory size, kernel arguments, number of threads, etc) and sends it over the interconnect to the SM being scheduled. Upon receiving the data, the SM Interconnect Arbiter (②) fills the corresponding storage structures within the SM. Likewise, when a thread block completes execution the SM sends a message through the interconnect to the thread block scheduling unit to notify it of the freed resources. Each sub-core has its own warp scheduler and each warp scheduler contains a warp PC table (⑧). In the baseline architecture a thread block is allocated on an SM by filling the sub-cores' warp scheduler PC table through a round robin multiplexer. The method in which the SM loads warp PCs into the warp scheduler tables forms the crux of the sub-core assignment balancing problem. The baseline round robin assignment can be implemented by a 4:1 multiplexer (③) with select lines driven by a 2-bit binary up-counter (④) that increments each time a new warp in the thread block is assigned to a sub-core.

To support hashed sub-core assignment we add a small 4-entry hash function table (⑤), an additional 4:1 multiplexer

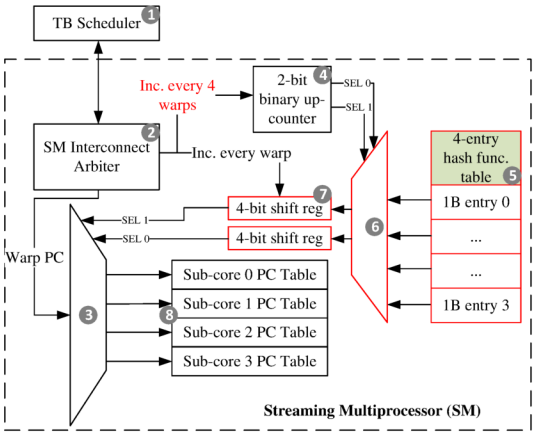


Fig. 7: Modifications to sub-core multiplexer to support hashed warp assignment

(⑥), and two 4-bit shift registers (⑦). Each entry of the hash function table contains a 1 byte entry which determines the assignment of 4 consecutive warp assignments. The upper 4 bits are used to drive select line 0 of the sub-core multiplexer (③) while the bottom 4 bits are used to drive select line 1. Rather than incrementing the 2-bit counter upon each warp allocation, as is done in the baseline round-robin design, the interconnect arbiter instead shifts a bit from the hash function table into the shift register each time a new warp PC is written, while incrementing the 2-bit counter every 4th warp. In this manner, a new entry of the hash function table is selected every four warps. Note that the hash function table supports direct assignment for up to 16 warps, but a thread block may have up to 64 warps. The table simply wraps around, so the 17th warp would re-use the assignment pattern encoded in entry 0.

2) Hash Function Sensitivity: While a particular sub-core assignment hashing function may be optimal for certain kernels, it may perform sub-optimally on applications with a different inter-warp-divergence distributions. We propose and evaluate two different hashing functions, Skewed Round Robin (SRR) and Random Shuffle. The SRR hashing function is shown in equation (1) below, where N is the number of sub-cores and $W \geq 0$ is a count of the number of warps previously allocated to this SM.

$$subcoreID = (W + \lfloor \frac{W}{N} \rfloor) \mod (N) \quad (1)$$

We created the SRR hashing function to keep the number of warps per sub-core even, while shifting the $subcoreID$ index by one every N warps. This shifting factor was determined by examining the TPC-H benchmark and observing that most kernels exhibit a single long running warp every 4 warps. SRR was then crafted to evenly distribute the long-running warps among the sub-cores for applications that match this specific warp balance distribution.

The Shuffle design randomly distributes incoming warps to sub-cores while ensuring that the number of warps assigned to each sub-core does not differ by more than one. Shuffle eliminates pathological imbalances but is suboptimal for highly imbalanced kernels. As the amount of inter-warp divergence of an application grows, an optimized sub-core warp assignment becomes increasingly critical. Figure 8 shows simulated performance of the unbalanced FMA application (described in Section III-B) as the amount of inter-warp divergence is scaled. Like the TPC-H benchmarks, the FMA application has one long running warp every four warps, and therefore is balanced ideally by the SRR hashing function. Note that the suboptimal assignment of Random Shuffle is increasingly detrimental as imbalance scales.

3) Design Cost: The additional hardware required for hashed assignment is minimal. Each SM is augmented with a 4 byte table, an additional 4:1 multiplexer and two 4-bit shift registers. The table can be filled arbitrarily for each SM to support different assignment hashing functions. Warp sub-core assignment only occurs on kernel launch, not every cycle, and the additional multiplexer and shift register have a negligible increase on overall assignment latency.

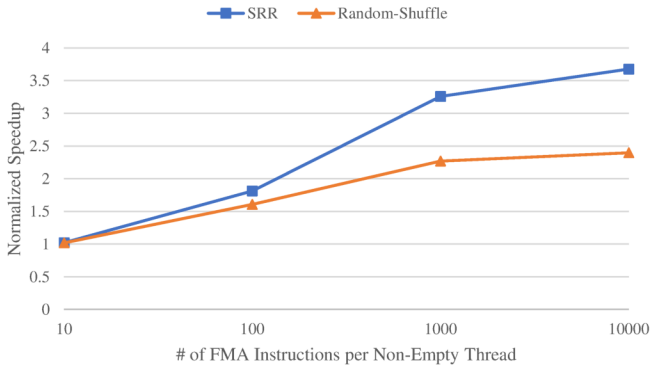


Fig. 8: Simulated GPU microbenchmark - design performance of unbalanced FMA as imbalance scales

The V100 SM holds state for a maximum of 64 warps, and each entry encodes the assignment of 4 warps, so a 16 entry hash table is necessary to encode a unique assignment for each warp. However, a 4-entry table can be used to fully represent the SRR hashing function because the SRR pattern repeats every 16 warps. A full-sized 16-entry table enables the Random Shuffle hashing algorithm to avoid repeating permutations, but provides minimal performance benefit over a 4-entry table. We found a 16-entry table using Random Shuffle to be within 2% of the execution time of the 4-entry table across all benchmark suites studied.

V. Experimental Methodology

All designs were evaluated using the trace-based cycle accurate GPU simulator, Accel-Sim [38], with Nvidia Volta V100 performance models from GPGPUSim 4.0 [18]. Accel-Sim was executed in trace-driven SASS mode such that accurate compiler register allocation and bank mappings were reflected in simulation. Each application was simulated to completion or a minimum of 5 billion instructions. The simulator was updated to accurately reflect the sub-core structures used in hardware for each stage of the pipeline. The Volta V100 model has 80 SMs per GPU, but the number of SMs was limited to 20 while evaluating the TPC-H benchmarks to more accurately model the load seen at larger scale factors. Database Management Systems (DBMS) applications typically run on databases with scale factors of 10 TB or more, but simulating queries with such a large scale factor would take prohibitively long, so a scale factor of 100 GB was used. Moreover, as the sub-core balancing problem is concerned with warp completion times, it is desirable to have the entire query simulate to completion in order to fully capture its behavior.

Table II shows the baseline simulator configurations used for all performance modeling in this study. In order to validate the number of collector units in each SM sub core, we constructed a set of seven microbenchmarks that stress register file bank conflicts. We correlated the cycle count for these microbenchmarks reported by Accel-Sim using 1 to 4 CUs/sub-core with

Number of SMs	80 (20 for TPC-H)
Sub-Cores per SM	4
Warp Scheduler Algorithm	Greedy Then Oldest
Max Warps per SM	64
Device Memory	32 (8 for TPC-H) GB HBM
Shared Memory Banks	32
Register File per Sub-core	64 KB
RF Banks per Sub-core	2
CUs per Sub-core	2
L1 / Shared Memory Cache	128 KB
L1 Instruction Cache	128 KB
Constant Cache	64 KB
L2 Cache	24-way 6MB

TABLE II: Baseline simulator configuration parameters

the silicon cycles from a Volta V100. We find that 2 CUs/sub-core yields the lowest mean absolute error at 16.2%, compared to 43% for the worst performing configuration. As a result, we use 2 CUs/sub-core (8 per SM) in our baseline GPU performance, area, and power results. 112 applications from 8 diverse benchmark suites were used for evaluation in this study. Table III summarizes a few key applications from each suite that are particularly sensitive to SM core partitioning.

TPC-H is a database benchmark standardized by the Transactional Processing Performance Council (TPC) [10]. TPC-H includes 22 SQL queries and operates on data tables generated from a set of templates. SQL queries and data tables can be directly input to Apache Spark through the SQL API. The spark-rapids plug-in [31] and Rapids C++ libraries [11] are open-source projects actively developed by Nvidia which provide GPU implementations of most Apache Spark operations and data transformations. Thus, Apache Spark applications, including SQL queries, can be executed on GPUs with no change to the application code. All of the TPC-H benchmark queries execute on a common database, however the SQL operations across queries is diverse. Therefore the characteristics and performance profile of each query can be wildly different. Two versions of the TPC-H benchmark were evaluated in this study, compressed and uncompressed. In the compressed benchmark the database has been stored in parquet files that have been compressed using Google's snappy compression algorithm [15]. The uncompressed benchmark uses database information stored in raw parquet files.

Similar to spark-rapids, cuGraph supports applications expressed in high level languages such as Python and executes on top of optimized Nvidia Rapids C++ libraries. All benchmarks were run with input graphs containing 262,144 vertices with randomly generated edges and weights.

Additional benchmarks come from Rodinia [22], a benchmark suite with a broad range of parallel communication patterns, Cutlass [12], a high performance GPU matrix multiplication suite, Parboil [55], a suite of throughput computing applications, Polybench [30], a suite of benchmarks with static control flow, and Deepbench [13] a CNN/RNN deep learning suite.

Abbreviation	Suite	Name
tpcU-q8	TPC-H Uncompressed [10]	Query 8
tpcC-q9	TPC-H Compressed [10]	Query 9
pb-mriq	Parboil [55]	Magnetic Resonance Imaging Q
pb-mrig		Magnetic Resonance Imaging Gridding
pb-sad		Sum of Absolute Differences
pb-sgemm		Single Precision Dense Matrix Multiply
pb-cutcp		Distance-Cutoff Coulombic Potential
cutlass-4096	Cutlass [12]	Matrix Convolution
rod-lavaMD	Rodinia [22]	Particle Potential
rod-bp		Back Propagation
rod-srad		Speckle Reducing Anisotropic Diffusion
rod-htsp		Hotspot 3D
cg-lou	cuGraph [16]	Louvain
cg-bfs		Breadth First Search
cg-sssp		Single Shortest Path
cg-pgrnk		Pagerank
cg-wcc		Weakly Connected Components
cg-katz		Katz Centrality
cg-hits		Hits Link Analysis
ply-2dcon	Polybench [30]	2D Convolution
ply-3dcon		3D Convolution
db-conv-tr	Deepbench [13]	CNN Train
db-conv-inf		CNN Inference
db-rnn-tr		RNN Train
db-rnn-inf		RNN Inference

TABLE III: Application Abbreviations

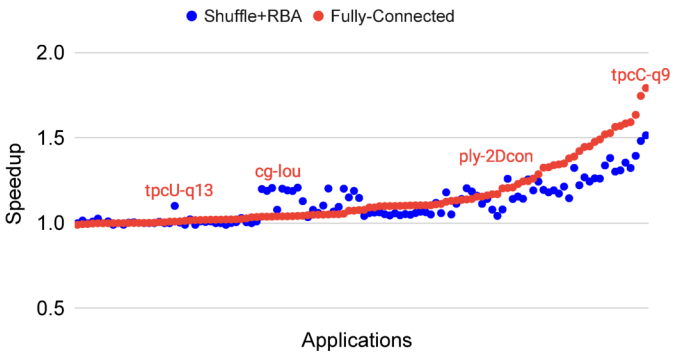


Fig. 9: Design performance on all applications - speedup normalized to GTO warp scheduler & RR sub-core scheduler

VI. Experimental Results

Figure 9 shows the performance of our combined design on all applications from all 8 suites. Across all applications, Shuffle+RBA had an average speedup of 10.6%, only 2.6% less than the 13.2% average speedup of the fully connected SM. Note that RBA outperforms fully connected in some applications (see section VI-A). Figure 10 summarizes the results of our designs and register bank stealing [36] on a subset of applications limited by the read operand stage or sub-core issue imbalance. The RBA warp scheduler achieved an average speedup of 11.1%. RBA performs better than doubling

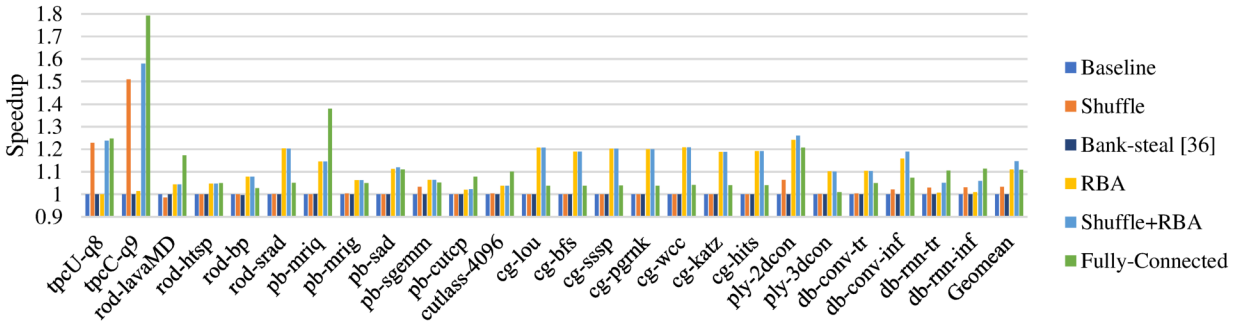


Fig. 10: Summary design performance of applications sensitive to SM subdivision - speedup normalized to GTO warp scheduler & RR sub-core scheduler

the number of CUs per sub-core, which we will show achieves an average of 4.1%, while only increasing the power and area of the operand collector and warp scheduler by 1%.

We compared the effectiveness of RBA against register bank stealing [36], and found that bank stealing provides less than 1% speedup on average due to modern GPUs having fewer register banks and CUs per scheduler than prior generations. The V100 only has 2 CUs and banks per sub-core, whereas the GPU evaluated in [36] had 10. Bank stealing relies on free CUs to allocate a candidate warp ahead of normal issue to fill unused register bank accesses. As there are only 2 CUs per sub-core, opportunities for bank-stealing rarely arise. The aim of bank stealing is to better utilize highly banked register files that are often not being accessed, conversely RBA seeks to make better use of a highly contended register file with a limited number of banks. More details of RBAs effectiveness in reducing bank conflicts is discussed in section VI-B3.

The TPC-H SQL benchmarks exhibit large amounts of sub-core issue imbalance and therefore benefit most from shuffle and SRR. The larger imbalance in the compressed query benchmark is due to the highly warp-specialized snappy [15] decompression kernel, which has an average issue imbalance on the order of 100x. In applications where sub-core issue is not imbalanced the assignment techniques do not improve or degrade performance beyond the baseline RR sub-core scheduler. Further analysis of the sub-core assignment designs is discussed in section VI-C.

A. Why does RBA outperform fully-connected in some apps?

A fully connected SM allows each thread to distribute its operands across eight banks instead of two, but unlike a partitioned SM, must share bank accesses with requests from warps issued from all four schedulers. Applications with many register file accesses can still suffer conflicts with a fully connected SM. Figure 11 shows that RBA improves performance of a fully connected SM from a geomean of 6.1% to 19.6% in applications where RBA outperforms Fully-Connected. Additionally, figure 14(d-f) shows that RBA can achieve a higher average RF bank utilization than Fully-

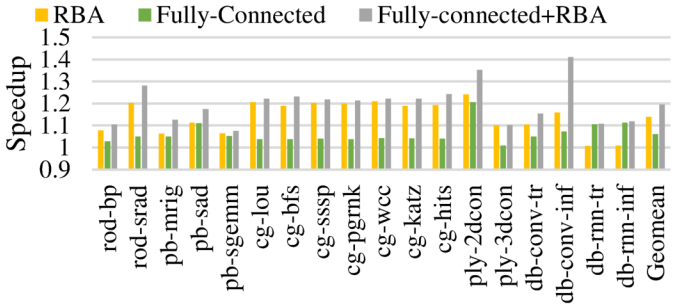


Fig. 11: RBA improves Fully-Connected SM performance in register file sensitive apps

Connected over the entire execution time of the application, resulting in better performance.

B. RBA as an alternative to scaling CUs

1) CU Scaling Performance: Figure 12 shows the performance as the number of CUs per sub-core scales. The values are normalized to a baseline model with 2 CUs per sub-core. Note that these results represent scaling the number of collector units only, the number of register banks per sub-core is held constant at two. The fully connected SM has the same total number of CUs as the baseline design but without sub-core partitioning, therefore each scheduler has access to a shared pool of 8 CUs, and operands can be mapped to any of the 8 banks on the SM. CU scaling improved average performance by 4.1%, 7.1%, and 9.6% for 4, 8, and 16 CUs per sub-core respectively. As shown in figure 10, the RBA scheduler achieved an average speedup of 11.9%. In all of the cugraph applications, the RBA scheduler outperformed the fully connected SM by 15% or more. The cuGraph applications have a large proportion of register intensive instructions, and are therefore sensitive to register file bandwidth. However, the graph applications tend to access a limited number of registers repeatedly and therefore do not benefit from the increased register file banking and full crossbar available in the fully connected SM. In applications other than the cugraph suite, the RBA scheduler achieved an average speedup of 6.7%, which is equivalent to the performance of between 4

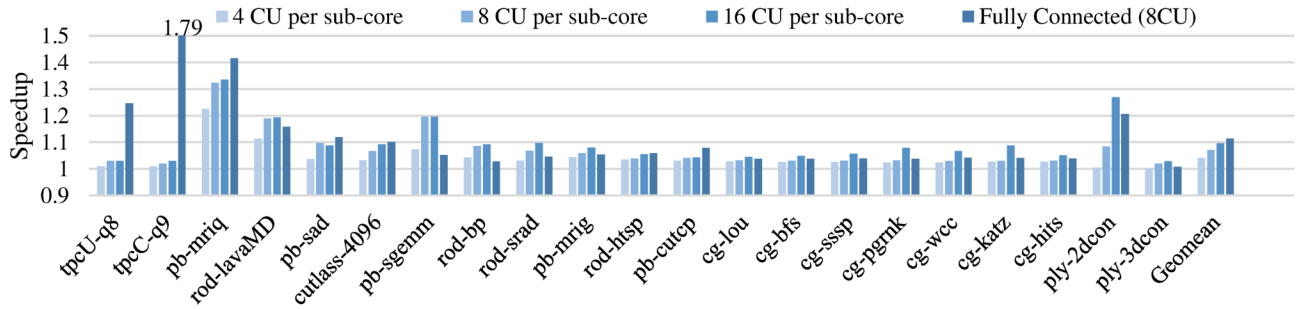


Fig. 12: CU scaling speedup - normalized to 2 CUs per sub-core

and 8 CUs per sub-core. The results in figure 12 indicate diminishing returns for scaling the number of CUs. It is critical to have enough CUs to cover bank conflicts with access requests from other warps, but the performance benefit beyond 8 CUs per sub core is only 2.5%.

In some applications, namely rod-lavaMD, pb-sgemm, rod-bp, rod-srad, pb-mrig, and the cuGraph applications, scaling the number of CUs provides more benefit than having access to a larger number of register banks (i.e. the fully connected SM). Such scenarios can arise when the fully connected SM exhausts its pool of shared CUs causing the warp scheduling stage to stall.

2) Power & Area Cost Comparison: CU scaling vs. RBA : To assess the costs of implementing the RBA design against adding CUs to each sub-core we implemented the Operand Collector and Warp Issue Scheduler (Figure 6) in RTL. The baseline design had 2 CUs per sub-core with a GTO Warp Issue Scheduler. From this baseline we either applied the changes marked in red in Figure 6 to realize the RBA scheduler or adjusted the number of CUs.

We synthesized the designs with a 1 GHz clock in Cadence Genus using a 45 nm process design kit [54]. The register file was built from SRAM modules generated by OpenRAM [32] with the same process design kit, and was configured according to Table II. The area and power results normalized to the baseline design are shown in Figure 13. Increasing the number of CUs to 4 increases the area of the design by 27%, and the power by 60%. Conversely, the RBA design introduces a mere 1% overhead in area and power respectively.

The RBA design is significantly cheaper than scaling the number of CUs because each operand stored in the CU is a vector of 32 values, each requiring a few thousand bits of storage. Moreover, the full crossbar connecting the vector operands is expensive to scale. In stark contrast, the RBA design (1) adds a minimal amount of storage (16 entries x 5-bit RBA score = 80 bits per sub-core) to the existing warp PC table, (2) requires minimal changes to the warp selection logic (the comparator network needs to be widened to compare 5 more bits of input) and (3) calculates the RBA score quickly (with 2 CUs and 3 operands per CU, max queue length is 6).

3) Register Bank Aware Scheduler Results: The RBA scheduler provides the performance benefits of scaling CUs without the significant area and power overheads. Figure 14

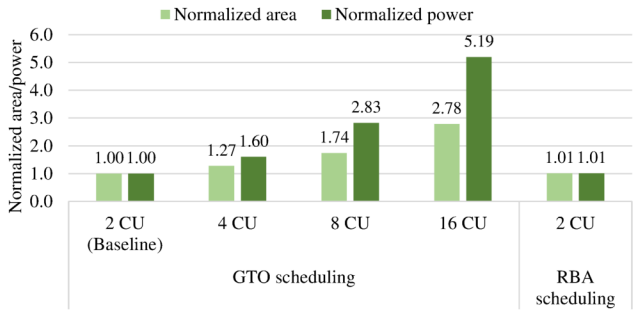
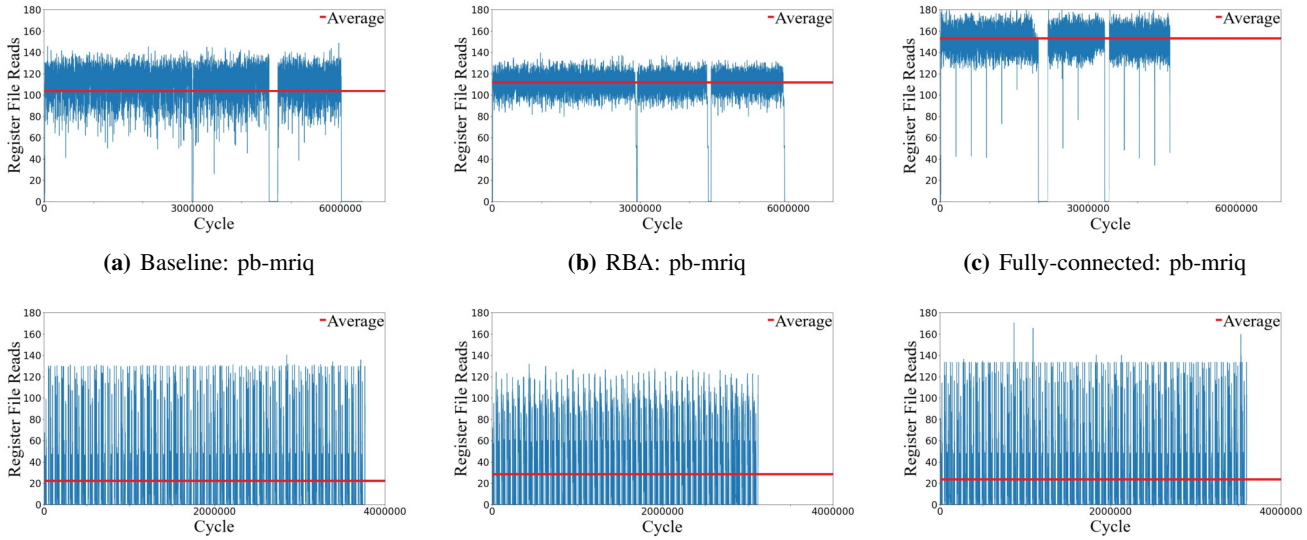


Fig. 13: Area and power cost estimates of scaling the number of CUs per sub-core, compared to a RBA-based design. All designs include the warp issue scheduler, operand collector and two register file banks.

shows the number of register reads aggregated across a single SM plotted for each cycle of execution for the pb-mriq and rod-srad benchmarks. The average number of reads/cycle across the entire application is shown in red. The maximum bandwidth is 256 reads/cycle which would occur if all 8 banks were accessed in a cycle by 8 different warps. The figure indicates RBA improves utilization of pb-mriq by increasing the average number of reads per cycle and reducing the number of cycles with 85 reads or less. The fully connected SM has a much higher average read/cycle rate. 160 reads corresponds to 5 register file banks being accessed in a cycle. The RBA scheduler is constrained to using 2 CUs and therefore has difficulty smoothing bank conflicts during large bursts of register file activity. In contrast, if one of the schedulers in the fully connected SM reaches section of code with a burst of register file activity, it can make use of more than two collector units at a time to help hide any bank conflict stalls.

The RBA scheduler improves average register file throughput beyond that of a fully connected SM in rod-srad (see fig. 14). The average register file reads per cycle across a single SM were 22.2, 27.1, and 23.4 for baseline, RBA, and fully-connected respectively. RBA outperformed the fully connected SM in rod-srad by improving the average utilization of the register file, even though the peak throughput was not improved.



(a) Baseline: pb-mriq **(b)** RBA: pb-mriq **(c)** Fully-connected: pb-mriq

(d) Baseline: rod-srad **(e)** RBA: rod-srad **(f)** Fully-connected: rod-srad

Fig. 14: Number of 4-byte register file read accesses per cycle in two applications: pb-mriq and rod-srad. RBA improves performance beyond fully-connected in rod-srad by increasing average register file read utilization across the entire application (shown in red).

4) RBA Score Latency Sensitivity Study: RBA uses score values from register bank contention to make scheduling decisions. Although computing the scores is a relatively simple process, it is possible that the distance between the register file and the instruction scheduler may cause score updates to influence the critical path. In such a scenario, an RBA implementation may need to latch or pipeline the score update to avoid changing the cycle time. To quantify the effect a delay in score updates has on RBA, we evaluate RBA with score update latencies swept from 0 to 20 cycles. Across the top 15 applications where RBA shows a benefit, on average the performance decreased by $< 0.1\%$ as the latency was increased. We find that the majority of the applications tend to have stable periods of register file access, in which a slightly stale score value still provides a reasonable estimate for making scheduling decisions. Of the 15 applications in the latency study, only one, ply-2Dconv, had a performance impact of $>1\%$. In this application the speedup of RBA decreased from 24.2% to 19.2% with a latency of 20 cycles.

5) RBA Bank Scaling Sensitivity Study: We found that the effectiveness of RBA is reduced (from 19.3% to 15.4% on average) when the number of banks per sub-core is doubled from 2 to 4. As the number of banks increases the read-operand stage becomes less of a bottleneck and there are fewer opportunities for RBA to improve register file utilization.

C. Scheduler Issue Balancing Results

TPC-H is not limited by the read operand stage and therefore benefits only a few percent from the RBA warp scheduler design. Both sub-core assignment hash functions maintain performance across balanced applications. This can be seen from the performance of Shuffle on the Parboil, Cutlass, Cugraph, and Rodinia applications in figure 10.

Figures 15 and 16 show the speedup of all designs for each query in the TPC-H benchmark suites with compressed and uncompressed databases. SRR and Shuffle had average speedups of 33.1%, 27.4% respectively on the compressed benchmark, and 17.5%, 13.9% respectively on the uncompressed benchmark. SRR performed the best in all queries because the assignment function was designed to match to the warp issue distribution of the benchmarks, i.e. one long running warp every 4th warp. However, the shuffle hash function is more broadly applicable to other imbalance patterns, and was within 5% of SRR on average.

The total number of instructions issued from each of the four schedulers provides insight into how well each design distributes work among the sub-cores. The coefficient of variation quantifies the amount of variation between the four schedulers in relation to the average number of instructions issued from each warp scheduler. A smaller value indicates better balancing across the four sub-cores in each SM. Figure 17 shows the coefficient of variation for SRR and Shuffle on the uncompressed TPC-H benchmarks and is calculated as $c_v = \frac{\sigma}{\mu}$, where σ is the standard deviation and μ is the mean.

The SRR hashing function reduced the coefficient of variation from .80 to .11 on average across all queries. Query 8, which had the largest baseline coefficient of variation of 1.01, experienced the largest performance improvement due to sub-core balancing with a 30.8% speedup. By improving the balance of instructions issued from each sub-core, both shuffle and SRR improved upon the baseline performance within four percent of each other. A larger reduction in sub-

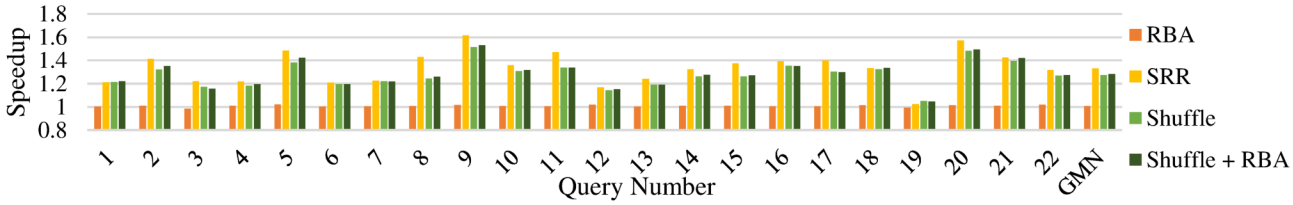


Fig. 15: Compressed TPCH - speedup normalized to GTO warp scheduler & RR sub-core scheduler

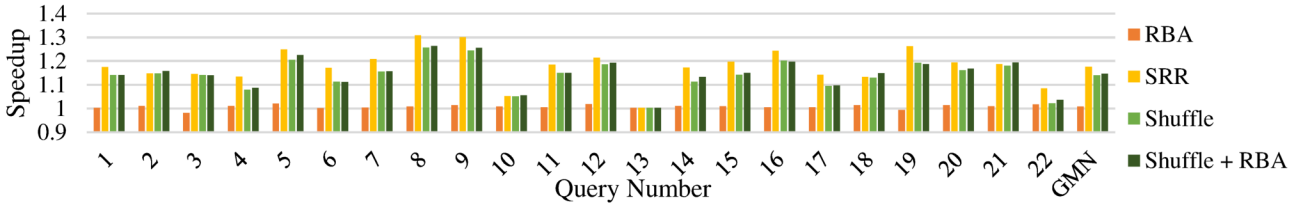


Fig. 16: Uncompressed TPCH - speedup normalized to GTO warp scheduler & RR sub-core scheduler

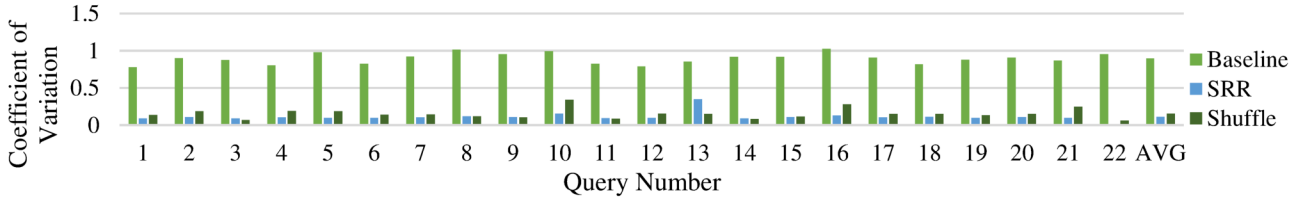


Fig. 17: Uncompressed TPCH - coefficient of variation of total instructions issued from each sub-core

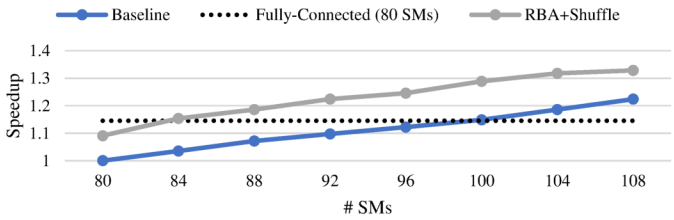


Fig. 18: Geomean performance of compute bound applications over a 4 sub-core SM baseline

core instruction issue imbalance does not always correspond to a larger increase in performance. The number of instructions issued is a imperfect metric for measuring warp completion times because many instructions take a different amount of cycles to execute. For example, memory instructions can take hundreds of cycles more than an arithmetic instruction, especially if misses occur in the cache hierarchy.

D. SM Sensitivity Study

To evaluate the trade off between having fewer fully connected SMs or more partitioned SMs, we perform a sensitivity study that scales the number of partitioned SMs in our baseline. Figure 18 plots the performance of compute-bound applications that benefit from SM scaling and demonstrates that 100 partitioned SMs perform similar to 80 fully connected SMs. Our techniques reduce this value such that only 84 partitioned SMs are needed to match the performance of 80 fully connected SMs.

VII. Related Work

The performance impact of bank conflicts is well known and has been studied in the context of several different micro-architectural structures. In [44] the authors propose a method to avoid DRAM bank conflicts with a novel memory controller that duplicates select lines of data into multiple banks. Similarly in [26], the authors propose duplicating architectural registers into multiple banks to avoid read-operand bank conflicts. RBA differs from both of these designs because it avoids bank conflicts through scheduling rather than the explicit duplication of data across banks.

In [58] the under-utilization caused by allocating and releasing SM resources at a thread block granularity was examined, and a mechanism to allocate and release resources at a warp granularity was proposed. Although the design allowed more warps to be assigned at a time to an SM, it did not adjust the assignment of those warps across multiple sub-cores. Moreover, their design left thread block management of shared memory intact. Shared memory is a driving force behind the sub-core balancing problem because sub-cores have a common shared memory space. Assigning thread blocks at the sub-core granularity would therefore require partitioning the shared memory and decreasing the maximum size available, or changing the programming model.

The authors of [46] propose coordination between multiple warp-schedulers to minimize idle time spent at barriers from warp-level-divergence. Warp scheduling policies can be used to effectively reduce the amount of inter-warp-divergence,

however they cannot address fundamental programmatic imbalances between warps and sub-cores. A work stealing hardware structure seems like an obvious solution to the sub-core issue balancing problem [21]. It is possible to design a system to transfer a warp's context from one sub-core to another dynamically to maintain balanced execution. However, any work stealing based design would be forced to transfer the the register file state of all of the threads within the migrating warp. Since the register file is partitioned across the sub-cores and each thread maintains its own private register file state, such a transfer would be prohibitively expensive. A work stealing design would have a much higher implementation overhead than the hashed sub-core assignment discussed in this work. Random interleaving has been studied as a method to improve memory bandwidth [51] but has not been used for sub-core work assignment.

VIII. Conclusion

This work identified sub-core partitioning as a source of SM underutilization in several generations of modern GPU architectures. Issue imbalance stems from the distribution of warps within a thread block to discrete scheduler and execution blocks, and therefore cannot be resolved through alternative warp or thread block scheduling algorithms. Applications with high levels of warp-level-divergence are particularly susceptible to underutilization due to sub-core imbalance, such as those used in database query processing. A hardware mechanism to reduce the likelihood of pathological imbalances with hashed sub-core scheduling was proposed.

Additionally, the partitioning of register file and operand collector structures has created bottlenecks in the read operand stage of several common GPU applications. A cost benefit analysis of scaling collector units to reduce bank conflicts was presented. A register file bank aware warp scheduling design was outlined to more efficiently utilize operand collector units with a constrained amount of resources, such as those found in partitioned SMs.

Acknowledgment

We thank our anonymous reviewers of MICRO 2022 and HPCA 2023 for their feedback and valuable comments. This work was supported, in part, by NSF CCF #1910924 and CCF #1943379 (CAREER), and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA.

References

[1] "AMD Graphics Cores Next (GCN) Architecture Whitepaper," https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[2] "Introducing RDNA Architecture," https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[3] "NVIDIA A100 Tensor Core GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[4] "NVIDIA GeForce GTX 1080 Whitepaper," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[5] "NVIDIA GeForce GTX 750 Ti Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.

[6] "NVIDIA H100 Tensor Core GPU Architecture Overview," https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[7] "NVIDIA Tesla V100 GPU Architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[8] "NVIDIA Turing GPU Architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[9] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[10] "TPC Benchmark Specification H," http://tpc.org/TPC_Documents_Current_Versions/pdf/tpc-h_v3.0.0.pdf.

[11] "cuDF - GPU DataFrames," https://github.com/rapidsai/cudf, 2021.

[12] "CUTLASS 2.8," https://github.com/NVIDIA/cutlass, 2021.

[13] "DeepBench," https://github.com/baidu-research/DeepBench, 2021.

[14] "RAPIDS Accelerator plugin For Apache Spark," https://github.com/NVIDIA/spark-rapids, 2021.

[15] "Snappy Compression Library," https://github.com/google/snappy, 2021.

[16] "cuGraph - GPU Graph Analytics," https://github.com/rapidsai/cugraph, 2022.

[17] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot Register File: Energy Efficient Partitioned Register File for GPUs," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 589–600.

[18] A. Ariel, W. W. L. Fung, and T. M. Aamodt, "Visualizing Complex Dynamics in Many-Core Accelerator Architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010, pp. 164–174.

[19] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU memory bandwidth via warp specialization," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[20] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging Warp Specialization for High Performance on GPUs," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 119–130.

[21] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, Sep. 1999.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[23] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and Programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[24] J. H. Choquette, M. O. Gautho, and J. E. Lindholm, "Methods and apparatus for source operand collector caching ," U.S. Patent 20130159628A1 Jun 20, 2013.

[25] B. W. Coon, J. R. Nickolls, J. E. Lindholm, R. J. Stoll, N. Wang, J. H. Choquette, and K. E. Nickolls, "Thread group scheduler for computing on a parallel thread processor ," Patent, May, 2012.

[26] N. Duong and R. Kumar, "Register Multimapping: A technique for reducing register bank conflicts in processors with large register files," in *2009 IEEE 7th Symposium on Application Specific Processors*, 2009, pp. 50–53.

[27] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 407–420.

[28] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 235–246.

[29] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 465–476.

[30] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[31] T. Graves and A. Bellina, "Cost Effective Data Processing with Apache Spark and GPU," *GTC*, 2022.

[32] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–6.

[33] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU Register File Virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 420–432.

[34] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," https://arxiv.org/abs/1804.06826, 2018.

[35] Z. Jia and P. V. Sandt, "Dissecting the Ampere GPU Architecture through Microbenchmarking," *GTC*, 2021.

[36] N. Jing, S. Jiang, S. Chen, J. Zhang, L. Jiang, C. Li, and X. Liang, "Bank Stealing for a Compact and Efficient Register File Architecture in GPGPU," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 520–533, 2017.

[37] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 332–343.

[38] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[39] M. Lee, G. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu, "iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 370–381.

[40] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 260–271.

[41] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-Compression: Enabling Power Efficient GPUs through Register Compression," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 502–514.

[42] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads," vol. 43, no. 3S, p. 515–527, Jun. 2015.

[43] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 175–186.

[44] B. Lin, M. B. Healy, R. Miftakhutdinov, P. G. Emma, and Y. Patt, "Duplicon Cache: Mitigating Off-Chip Memory Bank and Bank Group Conflicts Via Data Duplication," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 285–297.

[45] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[46] J. Liu, J. Yang, and R. Melhem, "SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 383–394.

[47] S. Liu, J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "Operand Collector Archtiecture," U.S. Patent 7834881B2 Nov 16, 2010.

[48] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-Aware Warp Scheduling for Throughput Processors," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016.

[49] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 308–317.

[50] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Preemptive Thread Block Scheduling with Online Structural Runtime Prediction for Concurrent GPGPU Kernels," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 483–484.

[51] B. R. Rau, "Pseudo-Randomly Interleaved Memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 74–83.

[52] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 72–83.

[53] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 99–110.

[54] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: An Open-Source Variation-Aware Design Kit," in *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, 2007, pp. 173–174.

[55] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W. mei W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," 2012.

[56] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 528–540.

[57] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 583–595.

[58] P. Xiang, Y. Yang, and H. Zhou, "Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation," in *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[59] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 230–242.

[60] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 609–621.