Evaluating and Mitigating Bandwidth Bottlenecks Across the Memory Hierarchy in GPUs

Saumay Dublish, Vijay Nagarajan, Nigel Topham
University of Edinburgh
{saumay.dublish, vijay.nagarajan, nigel.topham}@ed.ac.uk

***Abstract*—GPUs are often limited by off-chip memory bandwidth. With the advent of general-purpose computing on GPUs, a cache hierarchy has been introduced to filter the bandwidth demand to the off-chip memory. However, the cache hierarchy presents its own bandwidth limitations in sustaining such high levels of memory traffic.**

In this paper, we characterize the bandwidth bottlenecks present across the memory hierarchy in GPUs for general-purpose applications. We quantify the stalls throughout the memory hierarchy and identify the architectural parameters that play a pivotal role in leading to a congested memory system. We explore the architectural design space to mitigate the bandwidth bottlenecks and show that performance improvement achieved by mitigating the bandwidth bottleneck in the cache hierarchy can exceed the speedup obtained by a memory system with a baseline cache hierarchy and High Bandwidth Memory (HBM) DRAM. We also show that addressing the bandwidth bottleneck in isolation at specific levels can be sub-optimal and can even be counter-productive. Therefore, we show that it is imperative to resolve the bandwidth bottlenecks synergistically across different levels of the memory hierarchy. With the insights developed in this paper, we perform a cost-benefit analysis and identify cost-effective configurations of the memory hierarchy that effectively mitigate the bandwidth bottlenecks. We show that our final configuration achieves a performance improvement of 29% on average with a minimal area overhead of 1.6%.

I. Introduction

With the advancement of parallel computing in the domain of general-purpose applications, GPUs are increasingly used to address the computational demands of such workloads. Due to high levels of multithreading, such workloads present a high demand on the off-chip memory bandwidth. This has led to the introduction of deeper memory hierarchies to filter the bandwidth demand to the off-chip memory. However, due to high cache miss rates and cache thrashing [1], the off-chip bandwidth bottleneck is only partly mitigated. More importantly, the cache hierarchy exposes its own bandwidth limitations in sustaining such high levels of memory traffic [2].

Traditional approaches in the realm of CPU and GPU view the bandwidth bottleneck largely in the off-chip memory and address it by employing schemes such as efficient scheduling policies [3], incorporating fairness [4], and scaling the memory technology [5]. As a result, the off-chip bandwidth bottleneck is well understood and researched. However, as the bandwidth bottleneck in modern GPUs is distributed across the entire memory hierarchy, addressing the problem only in the off-chip memory no longer serves as a panacea for the entire memory system. Therefore, it motivates us to rethink and evaluate the bandwidth implications of other levels of the memory hierarchy.

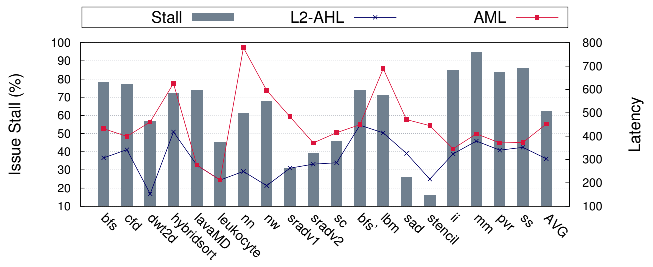

Fig. 1. Issue-stall cycles, Average Hit Latencies to L2 (L2-AHL) and Average Memory Latencies (AML) for memory-intensive applications.

To this end, we aim to characterize the severity of the bandwidth problem posed by the three levels of the memory hierarchy, *viz.*, private L1s, shared L2 and off-chip memory, and also characterize the role of peripheral network elements such as interconnects and buffers. In this paper we show that, due to bandwidth limitations, there is severe congestion between the L1 and L2 as well as between the L2 and off-chip memory. Such high levels of congestion lead to increased memory latencies [6], and this has three major implications. ❶ In memory-intensive applications, due to insufficient computation to mask high memory latencies, such latencies appear in the critical path of system performance. ❷ High latencies of outstanding miss requests lead to prolonged contention for cache resources such as Miss Status Holding Registers (MSHRs) and replaceable cache lines. This effect increases memory latencies even further, as succeeding requests get serialized and have to wait for outstanding misses to complete and relinquish the resources. ❸ Back pressure from a congested lower level further throttles the cache pipeline and prevents it from operating at peak throughput, exacerbating the bandwidth limitation in the cache hierarchy. A combination of the above factors force the cores to stall, leading to performance degradation. In Fig. 1 we show that memory-intensive applications, when run on a simulated GTX 480 GPU, exhibit high average memory latencies (AML; 452 cycles on average) and spend a considerable fraction of application runtime in a stalled state (62% on average) waiting for memory operations to complete. Additionally, high average L2 hit latencies (L2-AHL; 303 cycles on average) indicate that there is considerable congestion between the L1 and the shared L2 cache and therefore suggests that the high average memory latencies are due to bandwidth limitations in both the cache hierarchy and the off-chip memory.

In order to reduce congestion in the memory system, we explore several design choices at each level of the memory hierarchy and evaluate their efficacy in alleviating the bandwidth bottleneck. We conduct a design space exploration and show that solving the problem in isolation in specific levels of the memory hierarchy can give sub-optimal results and can even be counter-productive, only creating even more congestion elsewhere in the memory system. For instance, we observe that to prevent throttling of L1 cache, increasing the L1 bandwidth by increasing the MSHRs to handle more outstanding misses can lead to performance degradation due to even higher congestion between L1 and L2. We verify this observation on a real GTX 480 GPU chip by increasing the core frequency, effectively increasing the L1 request rate, and observe a performance degradation (detailed discussion in Section VI). On the other hand, matching the increased bandwidth demand of L1 at the L2 cache significantly improves performance which even exceeds the performance achieved by a memory system with baseline cache hierarchy and High Bandwidth Memory (HBM) DRAM. Therefore, in order to efficiently solve the bandwidth bottleneck, we show that it is imperative to address the bandwidth demand of different memory levels in tandem and provide a synergistic solution. Additionally, we use the insights developed in this paper to perform a cost-benefit analysis and propose efficient ways to mitigate the bandwidth bottlenecks at different levels of the memory hierarchy. Overall, this paper expands the understanding of the bottlenecks across the GPU memory hierarchy and serves as a guide for architects and programmers to optimally scale bandwidth of the memory hierarchy and write bandwidth-sensitive programs respectively.

In summary, we make the following contributions:

- We quantify the congestion levels across the GPU memory hierarchy and investigate the causes.
- We identify various design choices in the memory system and their efficacy in mitigating the bandwidth bottlenecks.
- We conduct a design space exploration and show that synergistic scaling of L1 and L2 resources can reasonably match or even exceed the benefits of an HBM DRAM.
- We identify cost-effective configurations of the memory hierarchy and observe a performance improvement of up to 29% on average with a minimal area overhead of 1.6%.

II. Background

A. Baseline Architecture

In this study, we consider a baseline similar to NVIDIA's Fermi architecture [7], [8]. However, as the organization of the memory hierarchy is fairly consistent across different architectures, we expect our observations to be applicable to Kepler and Maxwell as well. As shown in Fig. 2, a typical GPU consists of several execution units organized into a set of highly multithreaded SIMT Cores (or simply Cores). A set of register files occupy the highest level in the memory hierarchy and are used to maintain the state of several concurrent threads. The next level is formed by the private caches of a core, *viz*. L1 data cache, shared memory (scratchpad) and read-only texture and constant caches. Private caches are backed by a shared L2 cache that has an access latency of around 120 cycles for non-texture accesses in an uncongested memory hierarchy. The L2 cache is organized into multiple banks and each L2 bank communicate with the cores through a router in the *crossbar*, which transfers data packets at the granularity of *flits*. The shared L2 is further backed by an off-chip memory that has an additional access latency of around 100 cycles, excluding arbitration delays within the DRAM, which are governed by DRAM bandwidth. For general-purpose applications, prior works have shown low utilization and contention for register files [9], [10]. Therefore, we focus only on private L1, shared L2 and off-chip memory for bottleneck analysis as register files are seldom a bottleneck in such applications.

TABLE I
BASELINE ARCHITECTURE PARAMETERS FOR GPGPU-SIM

Parameter	Value
Core	15 SMs, Greedy-then-oldest (GTO) scheduler
Clock frequency	Core @ 1.4 GHz; Crossbar/L2 @ 700 MHz
Threads per SM	1536
Registers per SM	32768
Shared Memory	48 KB
L1 Data Cache	16KB, 128B line, 4-way, LRU, write-evict, 32 MSHR entries, 8-entry miss queue
Interconnect	Crossbar, Fly-topology, 32B flit size
L2 Cache	768 KB, 128B line, 8-way, LRU, write-back, 12 banks, 32 MSHRs, 8-entry miss queue, 32B data port width, 8-entry request queue
DRAM	GDDR5 DRAM, Command clock 924 MHz, FR-FRCFS 384-bits net buswidth, 6 Memory Partitions, 2 DRAM chips/partition, 32-bits buswidth/chip, 8 bytes burst length, 16 DRAM banks/chip
DRAM Timing Constraints	CCD = 2, RRD=6, RCD=12, RAS=28, RP=12, RC=40, CL=12, WL=4, CDLR=5, WR=12

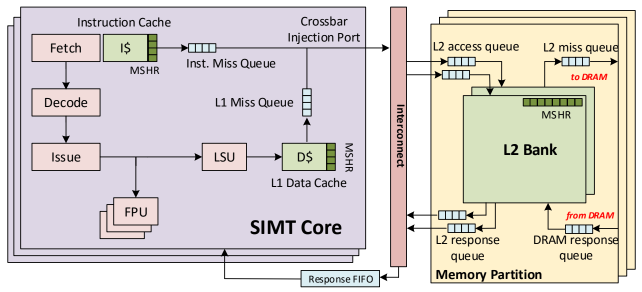

Fig. 2. Baseline GPU Architecture

B. Simulation Framework

We model a GTX 480 GPU on a cycle-accurate simulator GPGPU-Sim (v3.2.2) [11] with the baseline architecture parameters listed in Table I. We use GPUWattch [12] to compute the area and power in our experiments.

C. Workloads

For the purpose of this study, we use applications from three major general-purpose benchmark suites, *viz*., Rodinia (v3.0) [13], MapReduce [14] and Parboil [15]. In Table II, we list the memory-intensive benchmarks sorted by the speedup shown on an infinite bandwidth memory system (P_{∞}). We also show the performance improvement observed on a memory system with baseline cache hierarchy and an infinite bandwidth DRAM (P_{DRAM}). We discuss the observations in Section III-B.

TABLE II
LIST OF WORKLOADS: P_∞ – SPEEDUP WITH INFINITE BANDWIDTH MEMORY SYSTEM; P_{DRAM} – SPEEDUP WITH A BASELINE CACHE HIERARCHY AND INFINITE BANDWIDTH DRAM.

#	Suite	Benchmark	Abbrv.	P_∞	P_{DRAM}
1	Map.	Matrix Multiplication	mm	4.90	1.01
2	Par.	Lattice-Boltzman Method	lbm	3.40	1.87
3	Map.	Similarity Score	ss	3.23	1.00
4	Rod.	Nearest Neighbour	nn	3.11	1.84
5	Rod.	Hybrid Sort	hybridsort	3.10	1.24
6	Rod.	Computational Fluid	cfd	3.08	1.06
7	Map.	Page View Rank	pvr	2.89	1.01
8	Rod.	Breadth-First Search	bfs	2.84	1.00
9	Rod.	Particle Potential	lavaMD	2.70	1.00
10	Rod.	Stream Cluster	sc	2.70	1.13
11	Par.	Breadth-First Search	bfs'	2.10	1.00
12	Map.	Inverted Index	ii	1.98	1.00
13	Rod.	Speckle Reduction	sradv1	1.51	1.19
14	Rod.	Speckle Reduction	sradv2	1.49	1.08
15	Rod.	Needleman-Wunsch	nw	1.43	1.09
16	Par.	PDE Solver	stencil	1.23	1.20
17	Rod.	Wavelet Transform	dwt2d	1.20	1.14
18	Par.	Sum of Absolute Differences	sad	1.16	1.09
19	Rod.	Tracking Microscopy	leukocyte	1.08	1.00
			Average	2.37	1.15

III. MOTIVATION

In this section, we motivate the need to mitigate the bandwidth bottleneck in GPUs and discuss the potential benefits.

A. Latency tolerance

Multithreaded processors employ massive thread-level parallelism (TLP) to hide memory latencies. Upon encountering an instruction that is waiting on a long latency memory operation, the corresponding warp is de-scheduled and an active warp (if any) is scheduled, thereby overlapping the latency of memory operation. Therefore, GPUs are usually tolerant to memory latencies. However, memory-intensive applications often run into memory misses causing all of the warps to stall due to pending memory instructions. In such a case, miss latencies get exposed due to lack of sufficient overlapping computation and therefore lie in the critical path, directly impacting performance.

Fig. 3 shows the impact of memory latencies on performance, using a representative set of benchmarks from Table II. In this study, we modify the memory hierarchy of the baseline architecture so that all the L1 miss responses are returned with a fixed and pre-determined latency that is varied in the simulator and is represented on the x-axis. The resultant performance is plotted on the y-axis which is normalized to the performance of the baseline architecture.

We observe that for most benchmarks such as *nn*, *sc* and *lbm*, the performance remains fairly tolerant to modest L1 miss latencies. This is because the cores are able to effectively overlap such latencies with computation, in line with the philosophy of multithreaded architectures. However, when the memory latencies are higher, there is a direct impact on performance, indicating that such high latencies lie in the critical path. For instance, IPC for *nn* reduces modestly from 3.3× to 3.03× (normalized to baseline IPC) on varying the miss latencies from 0 to 250 cycles. However, further increasing the L1 miss latencies rapidly degrades performance, reducing the IPC to 1.9× in the next 250 cycles. Furthermore, other benchmarks such as *leukocyte* and *dwt2d* are sensitive to even small latencies, indicating a lack of sufficient thread-level parallelism.

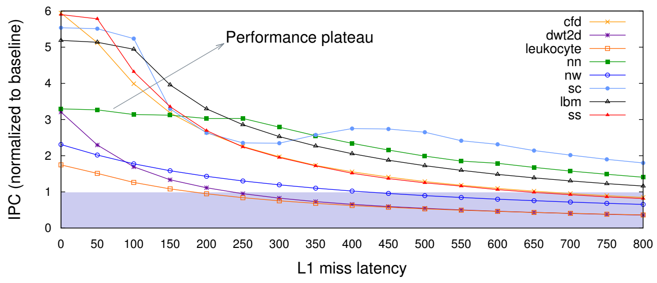

Fig. 3. Performance variation with increasing L1 miss latency.

In addition, we make two major observations about the *baseline memory latencies*, *i.e.*, the point on the x-axis where the performance curve intercepts the baseline IPC of 1× (shaded region), and therefore, matches the average memory latency of the baseline architecture. ❶ For most benchmarks, the baseline memory latencies are significantly higher than the latencies of *performance plateau* (or peak performance). Therefore, the baseline performance is well beyond the effective operating range of latency tolerance. ❷ The baseline memory latencies are also critically higher than the ideal access latencies of L2 (120 cycles) and DRAM (additional 100 cycles via L2). This suggests that there is considerable congestion in the memory system since traversing the memory system takes significantly higher latencies than the minimum memory access latencies of L2 and DRAM. In summary, the above results indicate that there lies a significant opportunity to improve performance by reducing the latencies incurred due to congestion in the memory hierarchy.

B. Performance impact of reducing congestion

In Table II, we have shown the speedup obtained with an infinite bandwidth memory system (P_∞) and observe an average performance improvement of 2.37×. In such a case, L1 miss requests do not suffer any congestion-related slowdown in the memory system and only incur the minimum memory access latencies of 120 cycles to L2 (for non-texture accesses) and another 100 cycles to off-chip memory for L2 miss requests. Therefore, the speedup can be mapped to Fig. 3 between the latency range of 120 to 220 cycles, with the average memory latency depending on the L2 miss rate. We also show the performance improvement with an infinite bandwidth DRAM appended to a baseline cache hierarchy (P_{DRAM}). In such a case, L1 miss requests suffer congestion-related slowdown only in the cache hierarchy and access the off-chip memory with a constant 100 cycle latency without incurring any congestion or timing limitations in the DRAM. We notice an average performance improvement of only 1.15× which is considerably less than the average P_∞, which includes an infinite bandwidth to both caches as well as DRAM. A comparatively lower performance improvement of P_{DRAM} suggests that the existing bandwidth bottleneck in the cache hierarchy plays a crucial role in increasing the miss latencies and thereby slowing down memory-intensive applications. In the subsequent sections, we investigate the cause of such high congestion in the memory system, focusing not only on the off-chip memory but also on the cache hierarchy since it is critical to performance. We also analyze the finer implications of congestion that cause performance degradation. Using these insights, we explore the opportunities to reduce the congestion-related latencies and show how it translates to performance improvements.

IV. Dissecting the Bandwidth Bottleneck

In a typical memory hierarchy, the bandwidth demand tapers down the memory system [16]. In principle, this is because each level filters the bandwidth demand to the lower level and therefore, the lower levels require only a fraction of bandwidth of the higher levels. However, if the bandwidth provided by the lower level is insufficient to service the bandwidth demand of the higher level, requests queue up in the memory system due to the *bandwidth skew* between the adjacent levels of the memory. This can lead to congestion in the network between the two levels and as a consequence, requests in the higher level will have to wait for longer durations to get serviced.

In Fig. 4 and Fig 5, we quantify the congestion between adjacent memory levels through an occupancy histogram of access queues to L2 and DRAM respectively. The stacked bars for each benchmark indicate the occupancy levels in the access queue, aggregated throughout the *usage lifetime* of the queue. We define usage lifetime as the time when the queues are occupied by at least one request. The occupancy histogram of the buffers between the adjacent memory levels, therefore, serves as a measure of the bandwidth skew, and thereby indicates the degree of congestion between the two levels. In Fig. 4 we note that on average, the access queues to L2 are full (indicated by the 100% occupancy bar in black) for 46% of their usage lifetime. Such high congestion aligns with the observation of high L2 access latencies. Similar to the congestion between L1 and L2, high bandwidth demand of L2 misses and low DRAM service rate causes the DRAM access queues to get full leading to congestion between the two levels. In Fig. 5 we note that on average, DRAM access queues are full for 39% of their usage lifetime.

A. Implications of congestion

Limited bandwidth to traverse the memory system, and queuing delays due to congestion, lead to high memory latencies. Such high latencies are critical to system performance and cause performance degradation (as shown in the post-plateau region in Fig. 3). In this subsection, we delve further into the finer implications of high latencies (and congestion) and show how it leads to performance degradation. We summarize the results in Fig. 7.

1) Data and Fetch Hazards: When a warp encounters an instruction that is waiting on a pending memory (or compute) operation due to a data dependency, it is de-scheduled and no longer participates in thread level parallelism. This condition

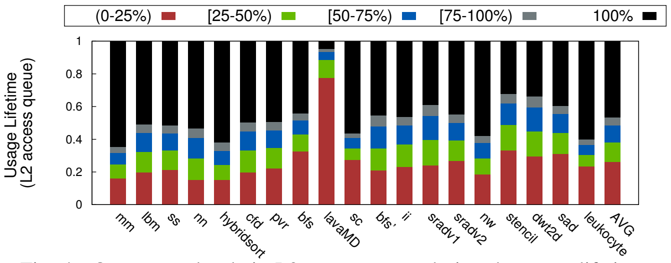

Fig. 4. Occupancy levels in L2 access queue during the usage lifetime

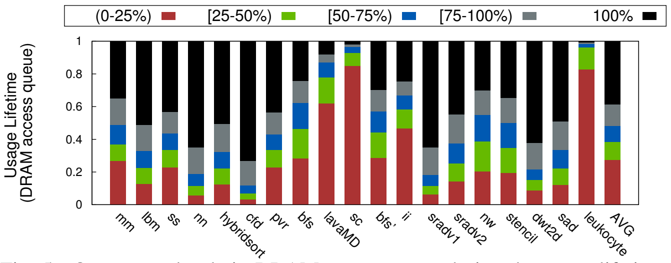

Fig. 5. Occupancy levels in DRAM access queue during the usage lifetime

is known as a *data hazard*. Once the pending memory (or compute) operation completes, the data dependency is resolved and the warp is allowed to resume execution. Since floating-point operation latencies are fairly small, the majority of data hazards are caused by pending loads [17]. When all warps are de-scheduled due to data hazards, which is often the case in memory-intensive applications, the core is forced to stall. In such a scenario, memory latencies contribute directly to stall cycles and govern how soon a warp can be released from a data hazard to continue execution.

Since instruction cache misses share the congested memory system with irregular data misses, high response latencies drain the instruction buffers, thereby de-scheduling the warp at instruction fetch. This is known as a *fetch hazard*. High instruction cache misses can cause the fetch buffer to drain for all warps. This causes the core to stall until the instruction misses complete and the warp resumes decoding.

2) Structural Hazards: High miss latencies can lead to prolonged contention of limited cache resources that are used to maintain the context of outstanding miss requests. This prevents the cache from sending new miss requests to the lower level in the memory system. This condition is known as a *structural hazard*. This further adds to the miss latency since the new misses get serialized, as they have to wait for the pending requests to complete and relinquish the resources. A structural hazard can occur due to a lack of free MSHR entries in a cache to hold the context of a new miss request. Since Fermi employs an allocate-on-miss policy for reserving new cache lines, a structural hazard can also be caused due to a lack of replaceable cache lines in a set as all cache lines might be reserved by pending miss requests.

3) Memory back pressure: In a congested memory system, due to the inability of network queues to accept new requests, preceding queues get full. This cascading effect of congestion percolates up to the higher levels of the memory hierarchy and is known as *memory back pressure*. When memory back pressure reaches the higher level cache, it manifests as a

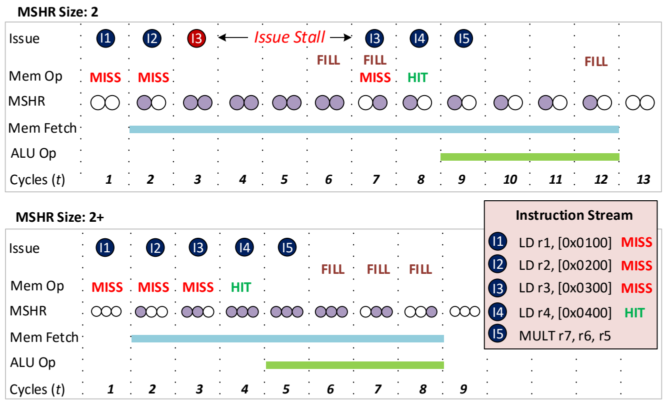

Fig. 6. Illustrating the effects of structural hazards

structural hazard due to lack of free entries in cache miss queues and therefore, prevents the cache from issuing a new miss request. For instance, back pressure from slow off-chip memory fills up the DRAM scheduler queue, in turn causing the L2 miss queues to get full. This leads to a structural hazard in the L2 cache as it cannot issue a new miss, thereby stalling the entire L2 cache pipeline. The back pressure eventually percolates up to the L1 cache and throttles core performance.

4) Discussion: Apart from further increasing the miss latencies, structural hazards (due to lack of cache resources or back pressure) have the following two major effects:

- *Increased hit latencies.* As structural hazards stall the cache pipeline, they prevent the succeeding requests from accessing the cache even if such requests are cache hits. This results in higher latencies for cache hits.
- *Restricted parallelism on cores.* A structural hazard in the load-store unit can cause all warps to stall when they attempt a memory instruction. This prevents the independent compute instructions in the instruction stream from getting issued, as the preceding memory instructions are waiting for the structural hazard to resolve. This serialization of memory and compute instructions prevents the core from hiding any further memory latencies, and thus performance suffers.

In Fig. 6, we illustrate the above two scenarios with the help of an example. In the first case, we assume an MSHR with two entries thereby allowing only two outstanding misses. Whereas in the second case, we assume a higher number of MSHRs that do not pose a structural limitation. For the sake of simplicity, we assume 6 cycles memory latency for an L1 load miss and 4 cycles for an ALU operation. In the first case, upon encountering the first two load misses, *i.e.*, I1 and I2, the MSHR gets full and can no longer accept any more misses. Since I3 is also a miss, it encounters a structural hazard and therefore stalls the L1 cache pipeline, in turn stalling the load-store unit (LSU). A succeeding cache hit in I4 needs to wait to access the L1 cache as there is a blocking I3 waiting for prior misses to relinquish the MSHR resources. Therefore, I4 gets serialized with the outstanding misses leading to a higher hit latency of I4. Additionally, a successive multiplication instruction, I5, needs to wait in the instruction queue as the previous instruction from the same warp is

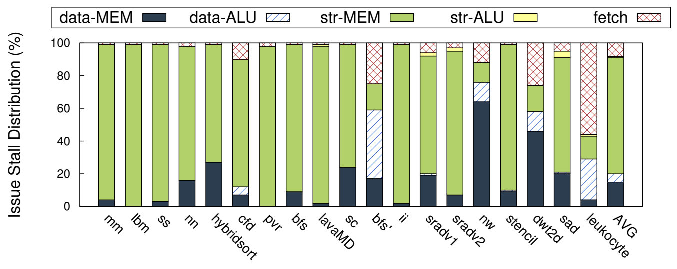

Fig. 7. Issue-stall cycle distribution depicting stalls due to data hazards (data-MEM and data-ALU), structural hazards (str-MEM and str-ALU) and fetch hazard (fetch).

pending at the issue stage. This structural dependency forces the execution units to remain stalled despite an independent multiplication instruction in the instruction stream. Therefore, I4 proceeds with the hit only at $t = 8$, after the response for the first load relinquishes an MSHR entry and unblocks the LSU. Thereafter, multiplication begins at $t = 9$ completing the execution at $t = 12$. In contrast, in the second scenario with no structural hazards, all independent instructions are issued successively. I4 results in a hit at $t = 4$ and ALUs begin computation at $t = 5$, completing the execution at $t = 8$. Note that in real systems, the miss latencies exceed hundreds of cycles thereby magnifying the effect of such structural hazards.

5) Summary: In Fig. 7, we demonstrate the distribution of the core's issue-stall cycles and attribute the cause of stall to one of the following reasons: data hazard due to a pending memory (*data-MEM*) or compute (*data-ALU*) operation; structural hazard due to resource contention in memory unit (*str-MEM*) or compute unit (*str-ALU*); and fetch hazard due to lack of instructions in the fetch buffer (*fetch*). As different warps can encounter different hazards in the same cycle, we consider a stall cycle as a data hazard when no warp can be issued due to existing data dependencies and the corresponding functional units do not pose a structural limitation for at least one warp. Similarly, a stall cycle is considered as a structural hazard when at least one warp, without any data dependencies, can be issued but is forced to stall due to resource contention in the corresponding functional units. We note that structural hazards from the memory stage form a major portion of the stalls with an average of 71% of issue-stall cycles. Data hazards due to pending memory instructions and fetch hazards contribute to 15% and 8% of issue-stall cycles on average, respectively. On average, data and structural hazards due to arithmetic units form very small portions of the issue-stall cycles, *i.e.*, 5.5% and 0.5% respectively.

B. Causes of congestion

In the previous sections, we observed that there is high congestion across the memory hierarchy due to distributed bandwidth bottleneck that leads to performance degradation. In order to construct the design space for mitigating congestion, we now explore finer causes of congestion by analyzing each memory level in detail.

1) Off-chip memory: Off-chip memory has been studied widely in context of bandwidth utilization [3], [4]. DRAM timing constraints, such as *activate* and *precharge* delays,

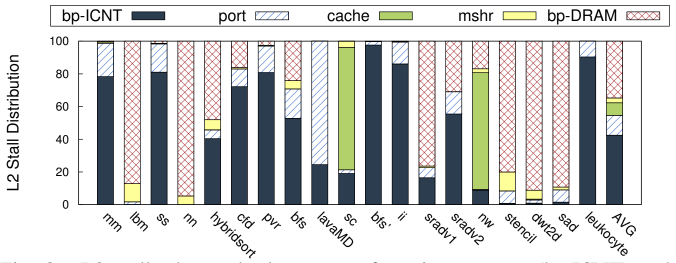

Fig. 8. L2 stalls due to back pressure from interconnect (bp-ICNT) and DRAM (bp-DRAM) and contention on L2 data port, cache lines and MSHRs.

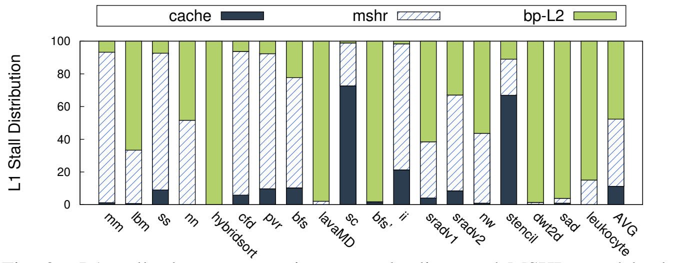

Fig. 9. L1 stalls due to contention on cache lines and MSHRs, and back pressure from L2 cache (bp-L2).

prevent DRAM from operating at peak throughput. Such constraints lead to low *bandwidth efficiency* in the DRAM, *i.e.*, the ratio of time when DRAM is transferring data on the memory bus to the time when there is at least one pending request in the DRAM scheduler queue. Therefore, a bandwidth efficiency of 100% would mean that the DRAM is always operating at peak throughput. In our simulations, we observe a low average bandwidth efficiency of 41% and a maximum of 65% for *stencil*.

2) L2 cache: Since L2 cache interacts with both DRAM at the lower level and L1 cache at the higher level, a myriad of factors can clog the L2. First, structural hazards due to a lack of MSHRs or non-replaceable cache lines can block the L2 pipeline. Second, memory back pressure due to congestion in the DRAM access queues can stall the L2 miss queue, creating another structural hazard at the L2 cache. Third, a busy L2 data port due to an ongoing cache line fill from DRAM or an ongoing read of an L2 cache line can cause port contention, forcing the subsequent L2 hits to wait before another cache line can be read. And finally, as L2 responses are injected into the crossbar at the granularity of *flits* (or network packets), it can take several cycles to inject an entire cache line. This forces the L2 responses to wait for long durations in the L2 response queue, eventually asserting back pressure on the L2.

In Fig. 8 we quantify the L2 cache stalls due to the above factors. We note that on average, structural hazards due to lack of MSHRs and replaceable cache lines contribute to 3% and 8% of L2 cache stalls. Memory back pressure from DRAM contribute to 35% of total stalls whereas L2 data port contention leads to 12% of stall cycles on average. Back pressure from L2 response queues due to slow crossbar injection rate leads to 42% of L2 stalls on average, appearing as the main cause of congestion at the L2 cache.

3) L1 cache: We perform a similar analysis for L1 cache to determine the prime factors that stall the L1 cache pipeline. L1 cache can stall due to structural hazards from MSHR as well as due to non-replaceable cache lines, similar to L2 cache. Also structural hazard due to back pressure from L2 can stall the L1 cache pipeline. In Fig. 9 we quantify the impact of such parameters. We note that on average, MSHR and cache line contention contribute to 41% and 11% of total L1 stalls and L2 back pressure is responsible for 48% of L1 stalls. Therefore, back pressure from L2 appears as the major cause in throttling the L1 cache, followed by MSHR contention and cache contention.

4) Summary: The above discussion provides insight into the reasons behind the stalls in the memory system and therefore, serves as a guiding tool in pruning the design space to best mitigate the high congestion between different levels of the memory hierarchy. We also observe the relative importance of parameters across caches. For instance, we note that the scarcity of MSHRs in L1 caches has a huge impact as they contribute to 41% of L1 stalls on average. On the other hand, MSHRs in L2 do not block the L2 cache as they contribute to only 3% of L2 stalls. We also note that back pressure contributes to significant proportion of stall cycles at both L1 and L2 caches.

V. Consolidating the Design Space

In this section, we use insights from the above analysis to consolidate the design parameters that can be effective in mitigating congestion in the memory hierarchy. In the previous section, we observed that stalls at different levels of the memory hierarchy prevent caches (and cores) from operating at peak throughput. However, removing all such stalls and operating at peak throughput may not always alleviate congestion *as the peak throughput itself can be a limiting factor*. Therefore, we classify the micro-architectural parameters into the following two categories:

❶ Type '=': Parameters that minimize stalls, allowing for the caches and cores to operate at peak throughput.

❷ Type '+': Parameters that increase the peak throughput.

In the following subsections, we identify such parameters for the above categories and summarize our architectural design space in Table III.

A. Off-chip memory

The baseline architecture employs *First-Ready First-Come-First-Serve* (FR-FCFS) scheduling policy that prioritizes accesses to an already opened DRAM row from a pool of pending requests in the scheduler queue to achieve higher row-buffer hits. To maximize the benefit of FR-FCFS scheduling, we increase the scheduler queue size and allow the DRAM to search in a larger pool of pending requests and schedule more row-buffer hits. Maximizing row-buffer hits allows the DRAM to operate closer to the peak DRAM throughput, increasing the bandwidth efficiency. In order to maximize bank-level parallelism, we increase the number of banks per DRAM chip while keeping the size of the DRAM constant. This reduces the number of rows per bank and therefore, spreads the accesses to different banks thereby increasing concurrency. Finally, to

TABLE III
CONSOLIDATED DESIGN SPACE SHOWING BASELINE, SCALED (4×) AND COST-EFFECTIVE CONFIGURATIONS.

Design Parameter	Type	Baseline value	Scaled value (4×)	Cost-effective
(a) DRAM				
Scheduler queue	=	16 entries	64 entries	16 entries
DRAM Banks	=	16 banks/chip	64 banks/chip	16 banks/chip
Bus width	+	384-bits	1536-bits	384-bits
(b) L2 Cache				
L2 miss queue	=	8 entries	32 entries	32 entries
L2 response queue	=	8 entries	32 entries	32 entries
MSHR	=	32 entries	128 entries	32 entries
L2 access queue	=	8 entries	32 entries	32 entries
L2 data port	+	32 bytes	128 bytes	32 bytes
Flit size (crossbar)	+	32+32 bytes	128+128 bytes	16+48 bytes
L2 banks	+	12 banks	48 banks	12 banks
(c) L1 Cache				
L1 miss queue	=	8 entries	32 entries	32 entries
MSHR (L1D)	=	32 entries	128 entries	48 entries
Memory pipeline width	=	10	40	40

increase the peak throughput of DRAM, we increase the bus width of each DRAM chip.

B. L2 cache

To prevent throttling of L2 cache due to back pressure from DRAM, we increase the L2 miss queue size to allow more L2 misses to be buffered in the access path to DRAM. Similarly, we increase the size of the L2 response queue to mitigate the back pressure from the response network. To reduce structural hazards due to cache resources, we increase the MSHRs. Stalls due to lack of non-replaceable cache lines can be resolved by increasing the capacity or associativity of L2 cache. However, such parameters reduce the miss traffic to the lower level thereby altering the bandwidth demand. Since we focus on performance of the memory system given a fixed bandwidth demand, we do not alter these parameters as it leads to an unfair comparison in the context of bandwidth bottlenecks. Instead, we increase the L2 access queue size to allow more requests to be buffered at a stalled L2, avoiding back pressure to L1 cache. Therefore, all the above parameters allow L2 (and higher levels) to operate closer to the peak throughput. Finally, to increase the peak throughput of L2, we increase the L2 data port width, crossbar flit size and L2 banks. We also note that other design parameters such as L2 and crossbar frequencies also achieve the goal of mitigating congestion. However, we restrict ourselves to representative parameters that demonstrate the effect of increasing the L2 bandwidth.

C. L1 cache

We reduce the impact of back pressure from L2 cache by increasing the L1 miss queue size. We also increase the MSHRs to reduce the structural hazards. Similar to L2, we do not increase the capacity and associativity of the L1 cache to mitigate cache line contention. Instead, we increase the width of the memory pipeline on the core to allow the load-store unit to buffer more pending cache requests. The above parameters prevent the core from throttling, thereby allowing it to operate closer to the peak throughput.

VI. DESIGN SPACE EXPLORATION

In this section, we evaluate the design space by scaling the bandwidth of different levels of the memory hierarchy through the architectural knobs listed in Table III. As a typical HBM [5] provides up to 4× bandwidth compared to GDDR5 DRAM, we evaluate similar factor of scaling in other levels of the memory.

A. Results

In Fig. 10, we demonstrate the results obtained by scaling the design parameters by a factor of 4×. We begin by discussing the performance improvement by increasing the bandwidth in independent levels of the memory hierarchy. Later, we discuss the combined effects of increasing the bandwidth across adjacent memory levels, followed by scaling the bandwidth across the entire memory hierarchy.

1) L1 cache: On increasing the L1 resources, we see an average performance improvement of 4%. We observe the maximum speedup of 240% for *sc* followed by a speedup of 16% for cfd. The reason for the observed speedup lies in the fact that increased resources reduce the structural hazards on L1 cache. This results in better overlap of memory operations with computation and lower latencies of cache hits, as we have illustrated in Fig. 6.

On the other hand, we notice that for some other benchmarks, the performance drops on increasing the L1 resources. For instance, *mm* and *ii* suffer a slowdown of 33% and 25% respectively. This is because on one hand, increasing the L1 resources allows the L1 cache to operate at peak throughput, at the same time it also leads to higher congestion between L1 and L2, as the increased bandwidth demand of L1 is not matched by the bandwidth provided by L2. Since higher congestion causes greater interleaving of requests from different cores, requests from the same core (and therefore same warps) get more sparse in the memory system thereby delaying the *tail request* of a warp. Since a core can resume execution only on receiving all the memory requests generated by a warp, it causes significantly higher stalls as none of the cores can resume execution any earlier than baseline. Additionally, we also notice a significantly higher L2 miss rate for applications showing slowdown. For instance, the L2 miss rate increases from 16% to 58% for *mm* and from 15% to 62% for *ii*. This is also due to higher interleaving of request streams from different cores that exhibit low inter-core locality thereby causing cache thrashing and destroying the intra-core locality in the L2 cache.

We verify the above behavior on a real GTX 480 GPU by increasing the core frequency for representative benchmarks and note a performance degradation of up to 10%, as shown in Fig. 11. Increasing the core frequency is analogous to increasing the L1 cache resources as it increases the request rate (or bandwidth demand) from L1 to L2. Interestingly, performance improves on reducing the core frequency as the reduced bandwidth demand by L1 resonates well with the bandwidth offered by L2.

2) L2 cache: By scaling the L2 cache resources, we observe an average performance improvement of 59%. We observe the maximum speedup of 266% for *mm*, which is also the most bandwidth-sensitive application. A significant performance im-

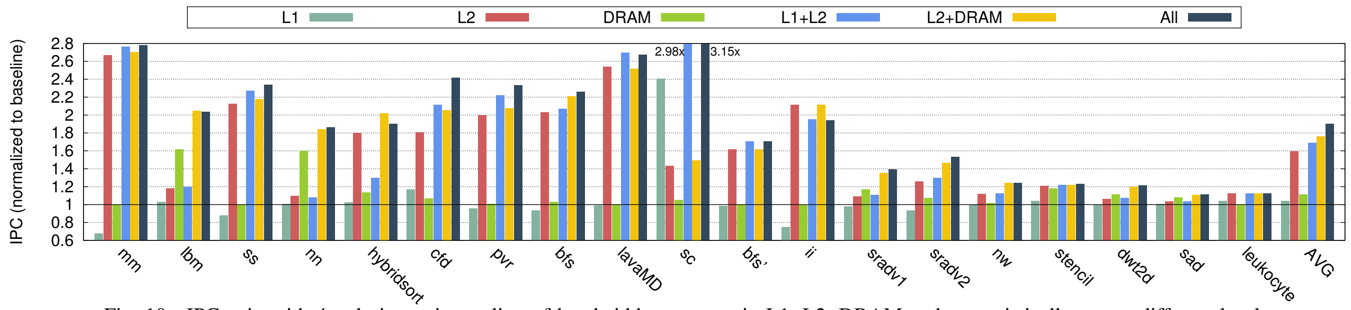

Fig. 10. IPC gain with 4× design-point scaling of bandwidth resources in L1, L2, DRAM and synergistically across different levels.

provement by scaling the L2 parameters signifies the criticality of the L2 bandwidth to the overall system performance.

3) Off-chip memory: Upon increasing the DRAM bandwidth, we observe an average performance improvement of 11%. We observe the maximum speedup of 61% for *lbm* followed by a speedup of 60% for *nn*. Note that the improved DRAM bandwidth matches the bandwidth offered by High Bandwidth Memory (HBM) and is representative of HBM performance. We note that the average improvement is in close proximity to the performance improvement of 15% obtained on a memory system with baseline cache hierarchy and an infinite bandwidth DRAM (average P_{DRAM}). However, it is considerably less than the performance improvement achieved on increasing the L2 cache bandwidth.

4) L1 and L2 cache: Upon synergistically increasing the bandwidth of the cache hierarchy, we observe an average performance improvement of 69%. We note that it is higher than the sum of gains obtained by improving the bandwidth in both cache levels independently, *i.e.*, 4% from L1 and 59% from L2. We observe in *mm* that even though increasing the L1 bandwidth alone resulted in performance degradation of 33%, increasing the L1 bandwidth with L2 results in a performance improvement of 276% which is even higher than 266% obtained by increasing the L2 bandwidth alone. A similar effect is seen in *ss*. We can therefore conclude that despite a slowdown on increasing the structural resources at L1, synergistic scaling of L1 and L2 results in a much higher performance improvement, which is greater than the standalone improvement of L2. We also observe that the average speedup by mitigating the bandwidth bottleneck in the cache hierarchy (69%) is significantly better than the speedup obtained by a memory system with baseline cache hierarchy and an HBM DRAM (11%).

We note an exception for *ii*, where combined scaling of L1 and L2 led to a lower speedup when compared to standalone scaling of L2 cache. However, we verify in our experiments that on further increasing the L2 bandwidth, synergistic scaling starts giving better results. This indicates that for *ii*, the increased L2 bandwidth in Fig. 10 is not yet sufficient for the increased bandwidth demand of L1.

5) L2 and off-chip memory: We observe an average performance improvement of 76% upon increasing the bandwidth at both L2 and DRAM. It is worth noting that it is in close proximity to the average speedup obtained by synergistically scaling the L1 and L2 bandwidth (69%).

6) All memory levels: We observe an average performance improvement of 90% on increasing the bandwidth of the cache hierarchy as well as the off-chip memory.

7) Summary: We conducted a limited design space exploration on architectural parameters relevant to the memory bandwidth in GPUs. We observed an average speedup of 4%, 59% and 11% on increasing the bandwidth of L1, L2 and DRAM alone. We further observed an average speedup of 69% and 76% on increasing the combined bandwidth of L1-L2 and L2-DRAM. Finally, we observed an average speedup of 90% on increasing the bandwidth of the entire memory system. Therefore, we demonstrate the criticality of cache hierarchy in mitigating congestion. We also demonstrate that synergistic scaling yields better results than increasing the bandwidth of the memory levels independently. And finally, we show that mitigating congestion in the cache hierarchy exceeds the benefit obtained by a memory system with HBM DRAM.

VII. Cost-Benefit Analysis

In Section V, we classified the architectural design space into two categories: Type '=' and Type '+'. Later, we evaluated the effect of scaling these parameters by a factor of 4×. However, such a scaling across all parameters is typically not practical due to cost overheads. Therefore, we qualitatively analyze the cost versus benefit associated with the parameters in the design space and arrive at a cost-effective configuration to scale the bandwidth across the memory hierarchy. We summarize the cost-effective configuration parameters in Table III.

A. Cost-effective design space

Type '=' parameters listed in Table III typically include buffers and MSHRs, and enable the memory levels to operate closer to the peak throughput. Buffers are simple structures and present minimal overhead in scaling. However, MSHRs are fully associative arrays and indexing high number of requests can be expensive. Since we have already observed in Fig. 8 that L2 seldom stalls due to MSHR contention, we only consider increasing MSHRs in the L1 cache.

Type '+' parameters in the cache hierarchy such as crossbar flit size, L2 data port width and L2 banks are more complex than simple buffers and MSHRs, and therefore incur considerable cost in scaling. As shown in Fig. 8, L2 data port only contributes to 12% of total L2 stalls on average. Due to its low contribution to the overall L2 stalls, we do not consider it for scaling. On the other hand, back pressure from interconnection network contributes to 42% of L2 stalls on average. While

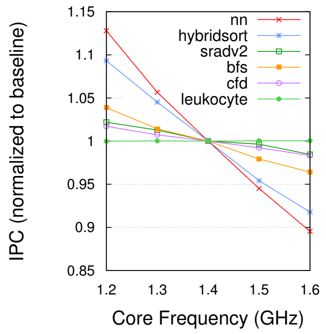

Fig. 11. Core frequency variation on real GTX 480 GPU.

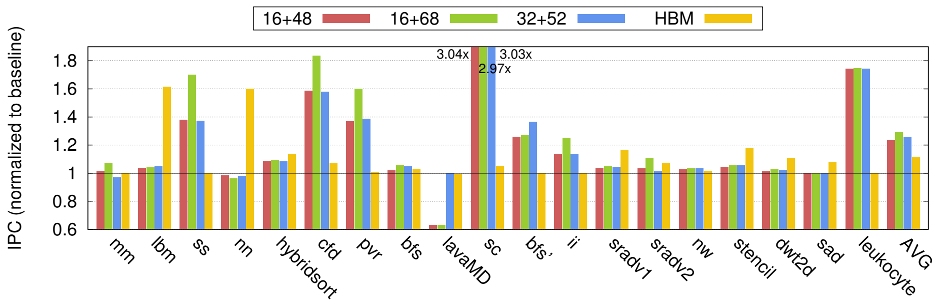

Fig. 12. Performance gain with cost-effective configurations in order of increasing or equal cost overheads, normalized to the baseline architecture.

both L2 banks and flit size improve interconnect bandwidth and resolve such stalls, we do not consider increasing the L2 banks. This is because each L2 bank has an independent port to the crossbar and therefore, increasing the L2 banks would lead to higher number of routers in the crossbar, in turn increasing the router area. In addition, router at the cores would now need to arbitrate over higher number of destinations, increasing the energy demands of the crossbar. Instead, we only increase the flit size of the crossbar as it increases the point-to-point bandwidth without significantly increasing the router area or arbitration energy.

B. Asymmetric Crossbar

The baseline crossbar offers a uniform flit size of 32 bytes for all nodes between core-to-L2 as well as L2-to-core. However, the bandwidth demand of the reply network (L2-to-core) is higher than that of the request network (core-to-L2). This is because the majority of request packets are load requests that amount to only 8 byte packets, whereas the majority of reply packets are load responses that amount to 128 byte cache lines. Although write requests in the request network present a higher bandwidth demand, such requests are relatively infrequent and the latency for such requests is not in the critical path of system performance. Therefore, we consider an asymmetric crossbar with lower request bandwidth (16 bytes) and higher reply bandwidth (48 bytes), henceforth referred to as the *16+48* crossbar configuration. Note that we do not increase the net area of the crossbar as the total number of point-to-point wires in the *16+48* crossbar are same as the baseline *32+32* crossbar. We also discuss other crossbar configurations such as *16+68* and *32+52* with minor cost overheads over the baseline architecture.

C. Results with cost-effective configuration

As shown in Fig. 12, for the *16+48* cost-effective configuration summarized in Table III, we observe an average performance improvement of 23.4%. It exceeds the average performance improvement of 11% with HBM. We note an exception for *lavaMD* which shows a performance drop of 37%. This is because *lavaMD* is limited at L1 by the L2 back pressure (Fig. 9) which gets aggravated due to reduced flit size in the request network. Even increasing the interconnect reply bandwidth does not cause much benefit as it is limited at L2 by the data port width (Fig. 8). Additionally, in our experiments we note that a standalone asymmetric crossbar without scaling other L1 and L2 parameters result in a lower speedup of 15.5%, thus emphasizing the importance of synergistic scaling.

We also evaluate *16+68* and *32+52* cost-effective configurations and observe a performance improvement of 29% and 25.7% respectively. While both the above crossbar configurations have equal point-to-point connections in total, we notice higher reward in investing more bandwidth in the reply network due to its higher bandwidth demand.

Overhead: We use GPUWattch [12] to estimate the area of our proposed architecture. We first compute the additional storage required in the cost-effective configuration for buffers and MSHRs. We assume each buffer entry to be 128 byte wide, while each miss queue and MSHR entry to be 8 byte wide. This results in a net storage overhead of 94 KB and amounts to an area overhead of 7.48 mm^2 at 40 nm technology, computed using existing values in GPUWattch. This amounts to an overall increase in the die area by around 1.1% with respect to baseline processor architecture area of 700 mm^2. We do not report power overhead as it is minimal and within the error margin of the simulator.

The baseline *32+32* interconnection network occupies a total area of 27 mm^2, while the wires contribute to 11.6 mm^2. Therefore, on increasing the point-to-point connections by 20 bytes in *16+68* and *32+52* crossbar, we incur an additional overhead of 3.62 mm^2. Therefore, along with overhead of buffers and MSHRs, the above two configurations result in a net area overhead of around 1.6%.

VIII. Related Work

Several prior schemes have been proposed for GPUs to reduce the performance impact of bandwidth bottlenecks. Kim *et al.* [17] proposed pre-execution of independent instructions in a warp to minimize the impact of data and structural dependencies. Similarly, Sethia *et al.* [2] proposed a re-execution queue to reduce the L1 cache hit latencies in presence of structural hazards. These schemes are orthogonal to our work as we aim to reduce the hazards itself, instead of circumventing them, and focus on the bandwidth bottlenecks across the entire memory hierarchy. Other proposals aim to maximize cache utilization to reduce the off-chip bandwidth demand by maximizing locality and using efficient scheduling policies [1].

In contrast, we analyze the bandwidth bottlenecks given a fixed bandwidth demand. Zhao *et al.* [18] propose a ring-based on-chip network to improve response bandwidth. Ziabari *et al.* [19] propose an asymmetric NoC architecture to improve energy efficiency by scaling down bandwidth of the request network. On the other hand, we aim to mitigate congestion by scaling up the bandwidth of the response network, *in tandem with other cache resources* such as MSHRs and buffers across different memory levels.

Analytical models have also been proposed to estimate the optimum cache levels [16] and DRAM parameters [20]. In contrast, we investigate the finer parameters such as MSHRs, cache banks, *etc.*, that lead to congestion in a given memory hierarchy and propose several knobs to improve performance while keeping the cache capacity and levels constant. O'Neil *et al.* [10] perform sensitivity studies to interconnect and DRAM bandwidths in GPU, whereas we perform an exhaustive analysis of the bottlenecks across the memory hierarchy and motivate the cost-effective design space based on the insights about the stalls in the memory hierarchy. Alsop *et al.* [21] propose a GPU stall inspector to identify source of stalls in closely coupled heterogeneous CPU-GPU architectures whereas we focus on the bandwidth bottlenecks in discrete GPUs.

IX. Conclusion

In this work, we demonstrate the bandwidth limitations posed by the memory hierarchy in GPUs. We observe that the bandwidth bottleneck is distributed across the memory hierarchy and is not limited to the off-chip memory. The bandwidth bottlenecks lead to high congestion in the memory hierarchy, in turn leading to high latencies that appear in the critical path. We characterize the stalls across the memory hierarchy and isolate the causes of congestion at each memory level. As a result, we identify the parameters to enable different levels of the memory hierarchy to operate at peak throughput or increase the peak throughput between adjacent memory levels. Using these architectural knobs, we conduct a design space exploration and show that increasing the bandwidth in isolation at specific levels of the memory hierarchy can be sub-optimal, and can even lead to performance degradation. We also show that the performance improvement obtained by synergistically improving the bandwidth of the cache hierarchy surpasses the speedup achieved by a memory system with baseline cache hierarchy and HBM DRAM. Using the insights developed in this paper, we perform a cost-benefit analysis and identify cost-effective configurations of the memory hierarchy to best mitigate the bandwidth bottlenecks. Our cost-effective configuration comprises of an asymmetric crossbar alongside architectural optimizations to allow L1 and L2 to operate closer to the peak throughput. We show that our final configuration achieves a performance improvement of 29% on average with a minimal area overhead of 1.6%.

References

[1] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 72–83, IEEE Computer Society, 2012.

[2] A. Sethia, D. Jamshidi, and S. Mahlke, "Mascar: Speeding up GPU warps by reducing memory pitstops," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 174–185, Feb 2015.

[3] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *IEEE 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–12, Jan 2010.

[4] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2008.

[5] "HYNIX HBM." https://www.skhynix.com/eng/product/dramHBM.jsp.

[6] S. Dublish, V. Nagarajan, and N. Topham, "Cooperative Caching for GPUs," *ACM Trans. Archit. Code Optim.*, vol. 13, Dec. 2016.

[7] Nvidia Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," tech. rep., Nvidia Corporation, 2009.

[8] "GPGPU-Sim Manual." http://gpgpu-sim.org/manual.

[9] M. Abdel-Majeed and M. Annavaram, "Warped Register File: A Power Efficient Register File for GPGPUs," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, (Washington, DC, USA), pp. 412–423, IEEE Computer Society, 2013.

[10] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," in *IISWC*, pp. 130–139, Oct 2014.

[11] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator.," in *ISPASS*, pp. 163–174, IEEE, 2009.

[12] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 487–498, ACM, 2013.

[13] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, (Washington, DC, USA), pp. 44–54, IEEE Computer Society, 2009.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 260–269, ACM, 2008.

[15] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[16] G. Sun, C. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y. K. Chen, "Moguls: A model to explore the memory hierarchy for bandwidth improvements," in *38th Annual International Symposium on Computer Architecture (ISCA), 2011*, pp. 377–388, June 2011.

[17] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 163–175, March 2016.

[18] X. Zhao, S. Ma, C. Li, L. Eeckhout, and Z. Wang, "A heterogeneous low-cost and low-latency ring-chain network for gpgpus," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 472–479, Oct 2016.

[19] A. K. Ziabari, J. L. Abellán, Y. Ma, A. Joshi, and D. Kaeli, "Asymmetric NoC Architectures for GPU Systems," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, NOCS '15, (New York, NY, USA), pp. 25:1–25:8, ACM, 2015.

[20] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "ANATOMY: An Analytical Model of Memory System Performance," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, (New York, NY, USA), pp. 505–517, ACM, 2014.

[21] J. Alsop, M. D. Sinclair, R. Komuravelli, and S. V. Adve, "GSI: A GPU Stall Inspector to Characterize the Source of Memory Stalls for Tightly Coupled GPUs," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016*.