Managing GPU Concurrency in Heterogeneous Architectures

Onur Kayıran* Nachiappan Chidambaram Nachiappan* Adwait Jog* Rachata Ausavarungnirun†
Mahmut T. Kandemir* Gabriel H. Loh‡ Onur Mutlu† Chita R. Das*
* *The Pennsylvania State University* † *Carnegie Mellon University* ‡ *Advanced Micro Devices, Inc.*
*Email: *{onur, nachi, adwait, kandemir, das}@cse.psu.edu, †{rachata, onur}@cmu.edu, ‡gabriel.loh@amd.com*

***Abstract*—Heterogeneous architectures consisting of general-purpose CPUs and throughput-optimized GPUs are projected to be the dominant computing platforms for many classes of applications. The design of such systems is more complex than that of homogeneous architectures because maximizing resource utilization while minimizing shared resource interference between CPU and GPU applications is difficult. We show that GPU applications tend to monopolize the shared hardware resources, such as memory and network, because of their high thread-level parallelism (TLP), and discuss the limitations of existing GPU-based concurrency management techniques when employed in heterogeneous systems. To solve this problem, we propose an integrated concurrency management strategy that modulates the TLP in GPUs to control the performance of both CPU and GPU applications. This mechanism considers both GPU core state and system-wide memory and network congestion information to dynamically decide on the level of GPU concurrency to maximize system performance. We propose and evaluate two schemes: one (CM-CPU) for boosting CPU performance in the presence of GPU interference, the other (CM-BAL) for improving both CPU and GPU performance in a balanced manner and thus overall system performance. Our evaluations show that the first scheme improves average CPU performance by 24%, while reducing average GPU performance by 11%. The second scheme provides 7% average performance improvement for *both* CPU and GPU applications. We also show that our solution allows the user to control performance trade-offs between CPUs and GPUs.**

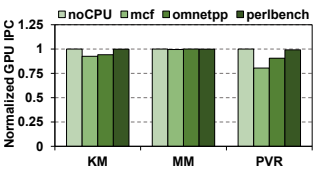
1 Introduction

GPUs have made headway as the computational workhorses for many throughput-oriented applications compared to general purpose CPUs [32]. Traditionally, a GPU operates in conjunction with a CPU in a master-slave mode, where the CPU offloads parallel parts of a computation to the GPU. Each kernel can spawn thousands of threads to utilize the full potential of the thread-level parallelism (TLP) available in a GPU. Current programming models like OpenCL [25] and CUDA [37] facilitate such programming on GPUs. While this master-slave mode of computation has become more powerful with innovations in both GPU hardware and software, it has some limitations that may impede pushing its performance envelope further. These limitations include the high overheads due to separate address spaces that require explicit data movement between the CPU and the GPU, and the lack of tight coordination between the CPU and the GPU for dynamically sharing the computation. Although concepts such as dynamic parallelism [39], where a GPU can generate new work for itself, Hyper-Q [39], where multiple CPUs can launch computations on a GPU, and HSA (Heterogeneous System Architecture) [2], where there is a unified architecture and programming model for CPUs and GPUs, are being explored, these concepts are still evolving and require effective resource management frameworks to make GPU-based computing more efficient.

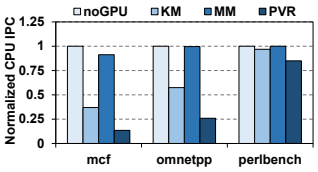
A recent trend in GPU-based computing has been the design of heterogeneous multicore architectures consisting of CPUs and GPUs on the same die. AMD's accelerated processing units (APUs) [3], NVIDIA's Echelon project [11], and Intel's Sandybridge/Ivybridge [18] architectures indicate the commercial interest and future directions in designing such heterogeneous multicores. However, the design space for such a cohesive but heterogeneous platform is much more complex than that of a homogeneous multicore system. For example, the number and placement of the CPU and GPU cores on a chip, the design of the underlying on-chip network fabric, the design of the cache/memory hierarchy, including resource management policies, and concurrency[1] management strategies for effectively utilizing the on-chip shared resources are some of the open issues that have not been sufficiently explored. In this paper, we focus on understanding and alleviating the shared-resource contention in a heterogeneous architecture because the contention problem is likely to be more aggravated in a heterogeneous system due to resource sharing between CPUs and GPUs, which have strikingly different resource demands.

CPU applications tend to be latency sensitive and have lower TLP than GPU applications, while GPU applications are more bandwidth sensitive. These disparities in TLP and sensitivity to latency/bandwidth may lead to low and unpredictable performance when CPU and GPU applications share the on-chip network, last-level cache (LLC), and memory controllers (MCs). To illustrate this, we ran nine different mixes of CPU and GPU applications on an integrated 14-core CPU and 28-core GPU platform (described in Section 5). We also ran these applications in isolation to estimate the performance impact of resource sharing on applications. Figure 1a shows GPU performance when running each GPU application (`KM`, `MM` and `PVR`) with and without one of the three different CPU applications (from the most memory-intensive to the least: `mcf`, `omnetpp`, `perlbench`). Figure 1b shows CPU performance when running each CPU application with and without one of the three GPU applications. We observe, from these figures, that the interference between CPU and GPU applications

[1]We use "concurrency", "TLP", and "parallelism" interchangeably.



(a) Effect of CPU Applications on GPU performance.



(b) Effect of GPU Applications on CPU performance.

Figure 1: Effects of heterogeneous execution on performance.

causes performance losses for both classes. CPU applications, however, are affected much more compared to GPU applications. For example, when `mcf` (a CPU application) is run with `PVR` (a GPU application), `mcf`'s performance drops by 84%, whereas `PVR`'s performance drops by only 20%. The performance losses observed on both classes of applications are primarily due to contention in shared hardware resources. In fact, the high TLP of GPU applications causes GPU packets to become the dominant consumers of shared resources, which in turn degrades CPU performance. Hence, concurrency management for GPUs is essential for controlling CPU, GPU, and overall system performance in a heterogeneous CPU-GPU system.

Previous works have explored concurrency control mechanisms in the context of GPUs. These works either target performance improvements only for cache-sensitive applications (CCWS [43]) or propose solutions based on the latency tolerance of GPU cores (DYNCTA [24]). These mechanisms are oblivious to the CPU cores and do not take into account system-wide information (such as memory and network congestion). Therefore, they lead to suboptimal CPU performance when applied to heterogeneous CPU-GPU systems due to the substantial difference between the latency tolerance of GPU and CPU cores.

In this paper, we introduce a new GPU TLP management mechanism that improves performance of *both* CPU and GPU applications in a heterogeneous system. The key idea is to regulate the number of active GPU warps (i.e., GPU TLP) to control CPU *as well as* GPU performance based on information obtained by monitoring system-wide congestion and latency tolerance of GPU cores. We propose two schemes targeted for different scenarios. Our first scheme, *CPU-Centric Concurrency Management* (*CM-CPU*), aims to improve the performance of CPU applications because they are observed to be the worst sufferers in an integrated platform. CM-CPU limits the number of active warps in GPUs by monitoring two global metrics that reflect *memory* and *network* congestion. While this scheme is very effective in boosting CPU performance, it causes most of the GPU applications to experience performance degradation. To recover the performance loss of such applications, we propose our second scheme, *Balanced Concurrency Management* (*CM-BAL*), which improves overall system performance based on the user's[2] preference for higher CPU or GPU performance. To achieve this, CM-BAL observes system-wide congestion metrics as well as the latency tolerance of the GPU cores to modulate the GPU TLP.

While one may argue that the alternative approach of partitioning network and memory resources between the CPU and the GPU might solve the contention problem, we show that such resource isolation actually leads to severe underutilization of resources, which in turn hurts either CPU or GPU performance, or both, significantly (Section 2.2). For this reason, we use a system with shared resources as our baseline in most of our evaluations, but also show (in Section 6.4) that our proposed schemes work effectively in a system with partitioned resources as well.

Our main contributions are as follows:

- We show that existing GPU concurrency management solutions [24, 43] are suboptimal for maximizing overall system performance in a heterogeneous CPU-GPU system due to the large differences in latency/bandwidth requirements of CPU and GPU applications.
- We introduce two GPU concurrency management schemes, CM-CPU and CM-BAL, which can be implemented with any warp scheduling technique by monitoring memory system congestion of the heterogeneous architecture and latency tolerance of GPU cores. The key idea is to dynamically adjust the number of active warps on the GPU to either unilaterally boost the CPU performance or enhance the entire system performance based on the importance of CPU or GPU performance as determined by the user.
- We show that both of our schemes reduce the monopolization of the shared resources by GPU traffic. CM-BAL also allows the user to control the performance trade-off between CPU and GPU applications.
- We extensively evaluate our proposals with 36 workloads on an integrated 28-core GPU and 14-core CPU simulation platform and show that CM-CPU improves CPU performance by 24%, but reduces GPU performance by 11%. CM-BAL provides 7% performance improvement for *both* CPU and GPU applications, without hurting GPU applications in any workload. We analyze the performance benefits and provide sensitivity studies, showing that improvements provided by the proposed techniques are robust.

To our knowledge, this is the first work that introduces new GPU concurrency management mechanisms to improve *both* CPU and GPU performance in heterogeneous systems.

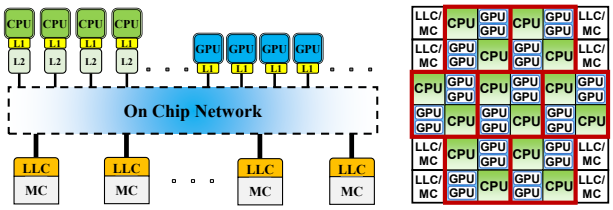
2 Analysis and Background

Before discussing the necessity of TLP management in a heterogeneous platform and proposing our solution, we describe our baseline architecture consisting of cores, a network-on-chip (NoC), and memory controllers (MCs).

2.1 Baseline Configuration

Our baseline architecture configuration places throughput-optimized GPU cores and latency-optimized CPU cores on the same chip, and connects these cores to the shared LLC and MCs via an interconnect, as shown in Figure 2a. In order to provide a scalable design for heterogeneous architectures, we use a "tile-based" architecture, as shown in Figure 2b.

[2]"User" can be an actual user, an application, or the system software.



(a) Baseline system configuration *(b)* 2D mesh layout

Figure 2: Baseline architecture.

Our baseline configuration consists of 7 processing tiles, where each tile has 4 GPU and 2 CPU cores. We choose a GPU to CPU core ratio of 2:1 because a single GPU core (i.e., streaming multiprocessor, SM) in Fermi GF110 (45nm technology) occupies roughly half the area of one Nehalem CPU core (45nm technology). We form a tile consisting of 4 GPU and 2 CPU cores, and replicate these tiles to provide scalability. Section 6.4 analyzes sensitivity to the number of GPU cores per tile. A total of 14 CPU and 28 GPU cores are connected to 8 LLC slices through a 6x6 2D mesh NoC.[3] The LLC, shared by both CPUs and GPUs, is distributed and each slice is directly attached to an MC. Cache blocks are mapped to LLC tiles and MCs in chunks of 256B.

We use two warp schedulers [38] per GPU core, with the Greedy-then-oldest (GTO) [43] policy. We use GTO because we found that it performs better than the round-robin and two-level schedulers [36] (also confirmed by Rogers et al. [43]). GTO already has a notion of TLP management and slightly reduces L1 contention among warps. We use the Static Wavefront Limiting (SWL) warp scheduler [43] to control the number of warps that can be scheduled to the pipeline. This scheduler also uses the GTO policy to choose between the available warps that can be issued to the pipeline. Due to the available hardware resources, the maximum number of warps in a core is 48.

2.2 Network and Memory Controller Configuration

Design of a scalable network and memory system for heterogeneous architectures is more challenging than for CMPs or GPUs due to the complex interactions and resource sharing between the two classes of applications. Previous work on NoCs for heterogeneous architectures either investigated the effects of a ring-based design on the system with a low number of cores [31] or partitioning of the virtual channels (VCs) [30]. Similarly, previous work on CPU memory channel/bank partitioning shows that partitioning the available memory channels/banks across applications can reduce interference between them [13, 20, 35, 49], while Ausavarungnirun et al. assume a shared memory controller between the CPU and GPU to maximize bandwidth utilization [4]. However, coordinated design of the NoC and the memory controllers for heterogeneous architectures is not sufficiently explored.

To choose a good baseline for studying the impact of concurrency management in detail, we start with the 2D mesh based design in Section 2.1 and examine four possible scenarios: *1) shared NoC and shared memory channels for both CPUs and GPUs; 2) partitioned NoC (one for CPUs and one for GPUs) and shared memory channels; 3) shared NoC and dedicated memory channels; and 4) partitioned NoC and dedicated memory channels.* We conduct an evaluation for these scenarios, keeping the network bisection bandwidth and the amount of resources constant.

Shared Network and Shared MCs. This all-shared configuration, although resulting in interference of CPU and GPU requests, maximizes resource utilization and provides the best baseline in terms of performance. Thus, we use it as our *baseline* and evaluate other configurations with respect to this. We run representative GPU applications with `omnetpp`, a high-MPKI CPU application that is sensitive to GPU traffic.[4]

Partitioned Network and Shared MCs. This design has similarities to some AMD APU implementations [3]. Two separate networks for the CPU and GPU traffic reduce CPU-GPU interference and probably helps latency-sensitive CPU applications, but would lead to resource under-utilization.

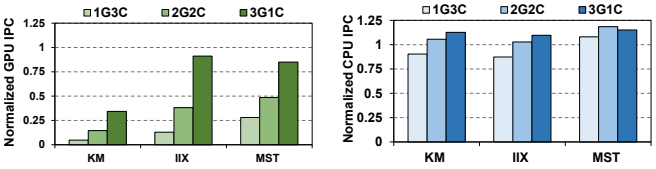
The shared network uses 4 VCs for each router output port and has 32B links. The partitioned network uses the same amount of VC and link resources in total, but divides these resources among the CPU and GPU traffic. For example, 3G1C configuration allocates 3 VCs and 24B links to the GPU network, and 1 VC and 8B links to the CPU network. For all configurations, we employ separate networks for request and reply traffic to prevent protocol deadlocks [12].

Figure 3a shows that the shared network is more suitable when GPU performance is preferred over CPU performance, because GPU traffic requires more network resources. Also, in the partitioned network, allocating more resources to the GPU traffic results in better GPU performance. On the other hand, Figure 3b demonstrates that the partitioned network is more suitable when CPU performance is preferred over GPU performance, as CPU applications are usually latency sensitive, and a dedicated network eliminates the interference caused by GPU packets. A counter-intuitive observation here is that increasing CPU network resources sometimes hurts CPU performance, because most CPU packets stall at the MCs, blocked by GPU packets that are waiting in MCs to be injected into the reply network (also observed by others [6]). In this scenario, increasing CPU network resources causes the system bottleneck to shift from memory to the GPU reply network.

Shared Network and Dedicated MCs. In this design, both CPUs and GPUs share the NoC, but each MC is dedicated to serve either CPU or GPU traffic. Figure 4 shows the effects of partitioning MCs. 1G7C denotes that 1 and 7 MCs are allocated for GPU and CPU traffic, respectively. We observe that GPU performance tends to degrade with fewer MCs, whereas CPUs always prefer having 4 dedicated MCs. Allocating more than 4 MCs to CPUs causes GPU

[3]Each node in the 6x6 mesh contains a router.

[4]Using a low-MPKI application (e.g., `perlbench`) leads to a lower impact on CPU performance, but our conclusions do not change.

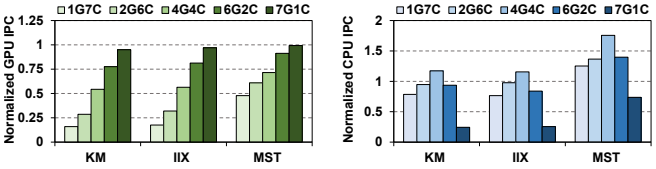


(a) Effect of network design on GPU performance.

(b) Effect of network design on CPU performance.

Figure 3: Effect of network design on GPU and CPU performance. Results are normalized to shared network/shared MCs configuration. All CPU cores run `omnetpp`.

packets to congest their own dedicated MCs, eventually congesting the shared request network, and thus leading to lower CPU performance. Sharing MCs does not greatly hurt CPU performance compared to equal partitioning of MCs (except when the GPU runs `MST`) and always provides better performance for the GPU. As expected, overall memory bandwidth utilization drops with dedicated MCs.

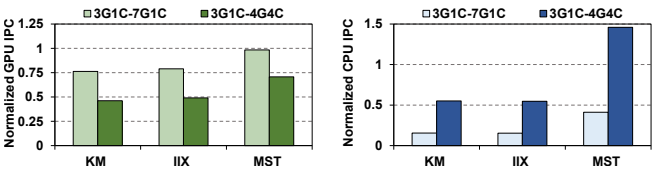


(a) Effect of MC design on GPU performance.

(b) Effect of MC design on CPU performance.

Figure 4: Effect of partitioned MC design on GPU and CPU performance. Results are normalized to shared network/shared MCs configuration. All CPU cores run `omnetpp`.

Partitioned Network and Dedicated MCs. In this analysis, based on the above discussion, we use the 3G1C configuration for the network, as it is the best-performing network partitioning; and we evaluate 4G4C and 7G1C configurations for MCs as they are the best for CPUs and GPUs, respectively. Figure 5 shows that partitioning both the network and the MCs is *not* preferable compared to sharing both the network and the MCs (except for `MST` when memory partitioning is done equally) as such partitioning greatly degrades both CPU and GPU performance.



(a) Effect of resource partitioning on GPU performance.

(b) Effect of resource partitioning on CPU performance.

Figure 5: Effect of resource partitioning on GPU and CPU performance. Results are normalized to shared network/shared MCs configuration. All CPU cores run `omnetpp`.

Based on the above analyses, we conclude that partitioning shared resources lead to underutilization of these resources, and thus we use a baseline with a shared network and shared MCs as it performs the best. Section 6.4 revisits the use of a partitioned network.

3 Motivation

We make a case for a better TLP control technique by discussing the limitations of existing techniques and the impact of TLP in a CPU-GPU environment.

3.1 Limitations of Existing Techniques

We first discuss the limitations of two existing concurrency management techniques proposed for GPUs when applied to heterogeneous CPU-GPU platforms.

Cache-Conscious Wavefront Scheduling (CCWS). Rogers et al. [43] proposed a throttling technique for GPUs that prevents warps from issuing loads if high TLP causes thrashing in the L1 data caches. This technique detects L1D cache thrashing by employing a victim cache. Each hit in the victim cache increases a score signifying the level of thrashing. A hit in a victim cache only happens when an evicted cache line is accessed again. Thus, this mechanism works mainly for cache-sensitive applications that benefit significantly from higher cache capacity, and is agnostic of the memory system beyond the L1D cache. This means that CCWS would likely not provide improvements for cache-insensitive applications that could still benefit from throttling. In fact, Kayiran et al. [24] demonstrate that there are applications that do not benefit from an L1D cache, but benefit from lower concurrency. Importantly, CCWS is not aware of the interactions of CPU and GPU in the memory system, and does not target managing concurrency to improve CPU and overall system performance.

DYNCTA. Kayiran et al. [24] proposed a CTA scheduler that limits the number of CTAs executing on a GPU core. This mechanism modulates GPU TLP based on *only* the *latency tolerance of GPU cores*. Because the latency tolerance of GPU and CPU cores are different, the decision based on GPU latency tolerance might *not* be optimal for CPUs. Figure 6 shows an example demonstrating this problem. In this example, GPU performance is mostly insensitive to the number of concurrently executing warps, except for the 1 warp-per-core case. Because the latency tolerance of GPUs does not change drastically with limited concurrency (except when TLP is greatly reduced to 1 warp), and because DYNCTA takes into account only the latency tolerance of the GPUs to modulate GPU concurrency, DYNCTA rarely reduces the concurrency level below 8 warps. Changing the concurrency level between 4 and 48 warps greatly affects memory congestion caused by the GPUs, but has little impact on GPU performance due to the latency tolerance provided by ample TLP. However, this memory congestion causes significant performance losses for CPU applications, which is not taken into account by DYNCTA. Thus, since DYNCTA modulates GPU TLP based solely on GPU latency tolerance without taking into account any effect GPU TLP has on the CPUs, CPU applications perform poorly with DYNCTA. In this example, executing only 4 warps per GPU core would improve CPU performance by more than 20%

without affecting GPU performance significantly, compared to DYNCTA. This demonstrates that, although DYNCTA is successful in managing GPU TLP to optimize GPU performance, it fails to do so for CPU or overall system performance, motivating the necessity for an integrated concurrency management solution for heterogeneous systems.

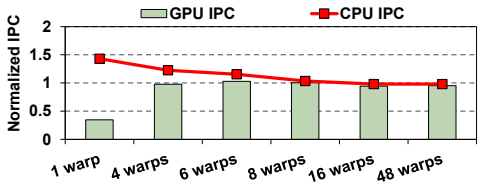


Figure 6: GPU and CPU IPC with different GPU concurrency levels, normalized to DYNCTA. GPU cores run BFS, CPU cores run a High-MPKI application mix (H2 in Table IV).

3.2 CPU-GPU Interaction in Heterogeneous Execution

We analyze the impact of the TLP of GPU applications on system performance in a CPU-GPU system, to motivate the necessity of TLP control in GPUs.

3.2.1 Effects of GPU Concurrency on GPU Performance: The effects of GPU TLP on GPU performance have been studied by prior research [24, 43]. Figure 7 shows the effect of six different concurrency levels (between 4 to 48 warps) on GPU performance for representative applications.[5] GPU IPC is normalized to the baseline scheduler, where there can be at most 48 warps running concurrently. The results demonstrate that while some applications, such as BP and MM, benefit from high concurrency (due to improved parallelism and latency tolerance), other applications such as MUM, suffer from it. There are two primary reasons for reduced GPU performance at high concurrency levels. First, cache-sensitive applications might thrash the caches by issuing many memory requests [43] at higher concurrency levels, resulting in reduced cache effectiveness, and in turn lower performance. Second, the high number of requests generated by more threads causes congestion in the memory subsystem [24], degrading overall GPU performance.

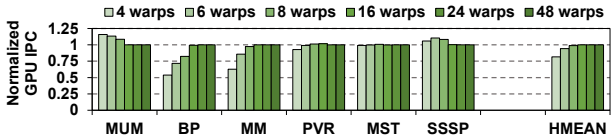


Figure 7: Effect of GPU TLP on GPU performance for representative GPU applications. All CPU cores run omnetpp.

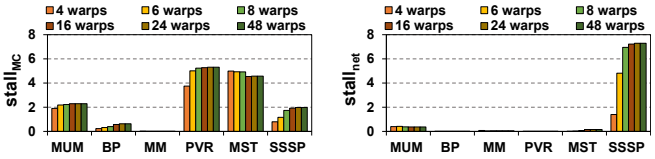
Memory subsystem congestion is observed at two points: 1) the MC cannot accept new packets coming from the request network, as the memory queue is full, 2) the MC cannot inject packets into the reply network because the reply network is congested. To analyze the sources of congestion, we examine two metrics:

[5]This figure shows the effect of GPU TLP on GPU performance. The chosen CPU workload has little impact on our observations.

1) $stall_{MC}$ = The number of stalled requests (per cycle) due to an MC being full.
2) $stall_{net}$ = The number of stalled requests (per cycle) due to reply network being full.

Both metrics can take values between 0 and the number of MCs. Figure 8 shows the value of these congestion metrics when concurrency is varied between 4 and 48 warps. We run representative GPU applications alongside omnetpp on the CPU side,[6] and normalize the results to the 48-warp concurrency level. In MUM, lower concurrency reduces memory congestion, which improves GPU performance by almost 20%. In SSSP, lower concurrency reduces both memory and interconnect congestion, improving performance. However, reducing the number of warps from 6 to 4 leads to insufficient latency tolerance and thus a drop in performance. In PVR, very low concurrency reduces congestion as well as latency tolerance of the application, causing a drop in performance. In MST, memory pressure does not reduce with reduced concurrency, thus not affecting its performance. Performance of MM and BP drops significantly at low concurrency levels because these two applications require high TLP to effectively hide the memory access latencies. Note that these metrics also reflect the effects of heterogeneous execution on the shared LLC.

Figure 8 shows that the main performance bottleneck in the system is likely memory bandwidth as many workloads have high values for $stall_{MC}$. The reply network is the bottleneck for SSSP. Applications that do not suffer greatly from MC stalls (BP and MM) strongly prefer higher concurrency levels for effective latency tolerance (see Figure 7).



(a) Effect of TLP on $stall_{MC}$. *(b)* Effect of TLP on $stall_{net}$.

Figure 8: Effect of GPU TLP on memory subsystem for representative GPU applications. All CPU cores run omnetpp.

3.2.2 Effects of GPU Concurrency on CPU Performance: As high GPU TLP leads to congestion in the memory subsystem, it causes longer latencies for memory requests. However, because CPUs cannot provide the level of thread- and memory-level parallelism that GPUs can, to tolerate long latencies, the increase in memory latencies likely hurt CPU performance. We observe that GPU applications pressurize the memory more than CPU applications. Thus, CPUs are highly affected by the GPU applications, whose memory pressure usually depends on their TLP.

Figure 9 shows the effect of GPU TLP on CPU performance when the CPU cores are running omnetpp (a high-MPKI application) alongside representative GPU applications. CPUs benefit from reduced congestion (see Figure 8) when run with GPU applications MUM, PVR, and SSSP.

[6]We observe the same trends when all CPU cores run perlbench.

MM does not put any pressure on the memory subsystem, thus does not affect CPU execution. Although reducing TLP does not affect the congestion metrics for BP and MST (see Figure 8), the significant reduction in LLC miss rates with reduced TLP leads to CPU performance improvement. Therefore, the scope for CPU performance improvement by reducing GPU TLP depends on the GPU application's memory intensity.

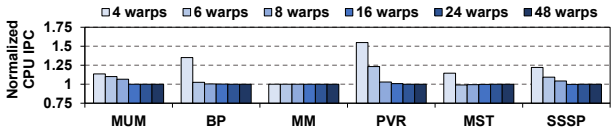


Figure 9: Effect of GPU TLP on CPU performance when all CPUs run omnetpp with each GPU application.

Figure 10 shows the results for the same experiment, except now the CPU cores run a very-low-MPKI application (perlbench). perlbench generates very few memory requests due to its very low MPKI, and, therefore, its performance is not bottlenecked by the memory service it receives.[7] Thus, even when it is run with GPU applications that saturate memory bandwidth, its performance does not degrade greatly. We conclude that very low-MPKI CPU applications have a limited scope for performance improvement by lowering GPU concurrency.

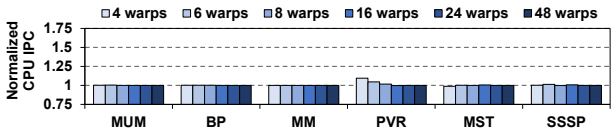


Figure 10: Effect of GPU TLP on CPU performance when all CPUs run perlbench with each GPU application.

Summary. Reducing GPU TLP might (positively or negatively) or might not affect GPU performance, but does not hurt CPU performance and can potentially greatly improve it. Thus, a prudent TLP management scheme can improve performance in a heterogeneous CPU-GPU system.

4 Managing GPU Concurrency

We describe our new approach for managing concurrency in heterogeneous systems.

Overview. Our proposal aims to 1) reduce the negative impact of GPU applications on CPUs, and 2) control the performance trade-off between CPU and GPU applications based on the user's preference. To achieve these goals, we propose two schemes. The first scheme, CM-CPU, achieves the first goal by reducing GPU concurrency to unilaterally boost CPU performance (Section 4.1). The second scheme, CM-BAL, achieves the second goal by giving the user multiple options to control the level of GPU concurrency (Section 4.2). This flexibility allows the user to control the performance of both classes of applications, and also to achieve balanced improvements for both CPUs and GPUs.

Before describing each scheme in detail, we first discuss why we employ concurrency management, why we do it specifically in GPU cores, and how we manage concurrency.

Employing Concurrency Management. Memory bandwidth is the critical bottleneck in most workloads, as shown in Section 3. To mitigate the ill effects of bandwidth saturation, solutions at different layers are possible. Lee and Kim [29] propose a new shared cache partitioning mechanism, and Ausavarungnirun et al. [4] propose new memory scheduling techniques. Both approaches aim to solve this problem at the sink, where the problem is observed. However, the source of the problem is the high number of requests generated by the GPU, leading to severe contention at the shared resources. Thus, we propose concurrency management strategies that attack the problem at the source, instead of the sink. Such an approach, by controlling the rate at which memory requests are issued, can reduce contention at the shared cache, network and memory.

Employing Only GPU-based Concurrency Management. We employ TLP management mechanisms only on GPUs (as opposed to also throttling the CPUs as done in [15, 47, 50]) due to two reasons. First, throttling CPUs does not have a significant effect on GPU performance but throttling GPUs can have a great effect on CPU performance, since GPUs tend to have many more memory requests than the CPUs [4]. Our experimental results show that reducing the number of CPU cores in our baseline from 14 to 8 has only 2-3% performance impact on GPU applications when both CPU and GPU cores are running memory-intensive applications. We showed in Section 1 that the effect of the GPUs on CPU performance is much higher than the effect of the CPUs on GPU performance. Second, unlike in GPUs, throttling CPUs to reduce memory contention [15, 47, 50] is likely to result in substantial performance loss for the throttled CPUs as a CPU is inherently much less latency tolerant and throttling reduces its latency tolerance even more [15]. Therefore, our strategy is to tune the impact of GPU execution on CPU performance by controlling the TLP of GPUs.

Reducing GPU TLP. GPU TLP can be reduced by modulating the number of executing warps [43] (Section 2.1). Reducing the number of concurrently executing warps would result in fewer memory requests sent by the GPU cores to the memory system. As long as there is enough parallelism in the cores, GPU performance should not drop.

4.1 CPU-Centric Concurrency Management (CM-CPU)

The main goal of CM-CPU is to *reduce GPU concurrency to boost CPU performance.* Recall from Section 3.2.2 that, if lowering GPU TLP reduces $stall_{MC}$ or $stall_{net}$, CPU applications have good scope for performance improvement. The key idea of CM-CPU is to 1) dynamically monitor the two congestion metrics, $stall_{MC}$ and $stall_{net}$, and 2) modulate GPU TLP based on these metrics. If at least one of these metrics is high, the memory subsystem is assumed

[7] As previous works have shown [13, 14, 26–28, 34, 35], last-level cache MPKI (or miss rate) is highly correlated with an application's sensitivity to memory latency or bandwidth.

to be congested either at the MCs or the reply network. Then, we reduce the level of concurrency by reducing *the number of active warps running on GPUs* (called, the *active warp limit*). If both metrics are low, we increase TLP, which might improve GPU performance. Otherwise, we keep the active warp limit unchanged. We use two thresholds, t_H and t_L to determine if these metrics are *low*, *medium*, or *high*. We use a medium range to avoid ping-ponging between different concurrency levels. If the active warp limit is between 8 and 48, increasing or decreasing concurrency is done in steps of 2 warps. If the active warp limit is between 1 and 8, the modulation step is 1, because the effect of changing concurrency is more significant in that range.

Figure 11 shows the architectural diagram of our proposal. To calculate $stall_{MC}$ (❶) and $stall_{net}$ (❷), we collect the congestion metrics (❸, ❹) at each memory partition separately. A centralized unit, called the CM-CPU logic (❺), periodically (with an *interval* of 1024 cycles)[8] aggregates these congestion metrics to calculate $stall_{MC}$ and $stall_{net}$, which it compares against t_H and t_L. Based on the result, it sends the modulation decision to the GPU cores. The CM-CPU logic is placed next to the global CTA scheduler. The scheduler has a dedicated connection to all the GPU cores [38, 39], and communicates with them frequently. The CM-CPU logic also needs to communicate with the memory partitions, where ❸ and ❹ are calculated, requiring dedicated connections with the memory partitions.

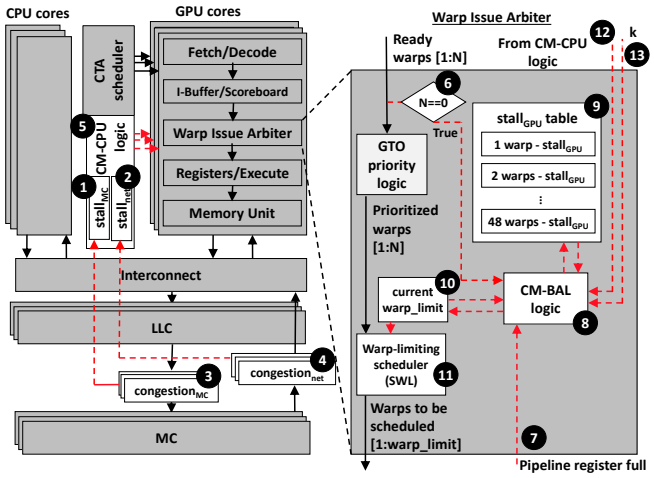


Figure 11: Proposed hardware organization. Additional hardware shown as white components and dashed arrows.

The downside of CM-CPU is that throttling the GPUs solely based on memory/network congestion metrics might hurt GPU applications that require high TLP for sufficient latency tolerance. For such applications, CM-CPU can fail to provide enough latency tolerance due to aggressive throttling. Because the memory congestion metrics are not correlated well with the latency tolerance of the GPU cores, modifying t_H and t_L to provide improvements for CPUs as well as latency tolerance for GPUs is not trivial.

4.2 Balanced Concurrency Management (CM-BAL)

CM-BAL tries to *recover the performance of GPU applications that suffer due to insufficient latency tolerance imposed by CM-CPU, up to the extent specified by the user.* The key idea of CM-BAL is to: 1) detect whether or not GPU cores have enough latency tolerance, and 2) maintain or increase the number of active warps if GPU cores are not able to hide memory latencies due to insufficient concurrency. To quantify the latency tolerance of GPU cores, for each core and each monitoring *interval*, we calculate the number of cycles during which the core is not able to issue a warp: $stall_{GPU}$. We use two warp schedulers per core in our baseline, so $stall_{GPU}$ for a core can be between 0 and twice the number of cycles in an interval. $stall_{GPU}$ is incremented if there are no ready warps that passed the scoreboard check (❻) and the pipeline register for the next stage is full[9] (❼). CM-BAL logic (❽) calculates the moving average of $stall_{GPU}$ for each concurrency level in each interval. These moving averages are stored in a table (❾). Depending on the difference between moving averages of $stall_{GPU}$ of a particular concurrency level and its neighboring levels, CM-BAL logic either increases or maintains the concurrency level (❿). Based on this level, the SWL scheduler (⓫) limits the number of warps that can be issued to the pipeline.

We observe that the performance difference between higher TLP levels (e.g., between 48 warps and 24 warps, which is negligible in most workloads) is much smaller than the performance difference between lower TLP levels. To take this into account, our algorithm changes the TLP level not with fixed steps, but using predetermined levels (❾): 1, 2, 3, 4, 6, 8, 16, 24, and 48 warps. We maintain a moving average of $stall_{GPU}$ for each of these levels. At the end of an interval, we update the one that has been used during that interval. Because during different phases of execution $stall_{GPU}$ for a particular level might have significantly different values, we opted to use moving averages in order not to use information coming from only a single interval and move to a suboptimal level. All moving average entries are invalidated at the beginning and end of a kernel. A new entry is created for a level with the value of $stall_{GPU}$ of the interval in which the concurrency level was used for the first time. In subsequent intervals during which the same level is used, CM-BAL logic updates the moving average using $mov\_avg_{new} = mov\_avg_{old} \times 0.25 + stall_{GPU} \times 0.75$.

Figure 12 shows how CM-CPU and CM-BAL work. The x-axis shows the level of GPU concurrency (TLP), and the y-axis denotes $stall_{GPU}$. This figure illustrates a typical scenario[10] where $stall_{GPU}$ increases with low concurrency,

[8] We chose an interval of 1024 cycles empirically (but we did not optimize the value). Such a small interval enables quick adaptation to fast fluctuations in memory and network congestion.

[9] This might happen due to instruction latencies and write-back contention for ALU instructions, and shared memory bank-conflicts and coalescing stalls for memory instructions.

[10] This curve does not necessarily have to be parabolic. Its exact shape depends on the application's reaction to the variation in TLP.

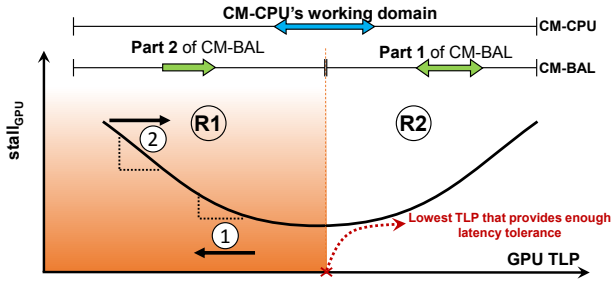


Figure 12: Operation of our schemes. CM-CPU increases/decreases TLP for all TLP ranges. CM-BAL might use Part 2 to improve latency tolerance by increasing TLP if TLP is low.

mainly due to low latency tolerance, and increases with high concurrency, mainly due to high memory contention. **R1** is the region where a GPU application does not have enough latency tolerance. **R2** is the region where the application has enough latency tolerance, but it *might* be suffering from memory contention. CM-CPU works on the entire TLP spectrum and increases/decreases concurrency based on two congestion metrics, but it does not consider $stall_{GPU}$. CM-BAL, on the other hand, operates in two parts. Part-1 (⓬ in Figure 11) operates the same way as CM-CPU (with the exception of using concurrency levels, instead of fixed steps), and works in **R2**, as shown in Figure 12. If the application is in **R1**, Part-2 of CM-BAL overrides any decision made by CM-CPU, and *might* increase TLP. There are multiple reasons as to why an application moves into **R1**. First, due to a change in the application's behavior (the curve shifts on the x-axis), the current TLP point might move into **R1**. Second, based on memory contention, CM-CPU might push the current TLP point of the application into **R1**.

We explain how Part-2 of CM-BAL works in **R1**. This part makes three decisions. First, as shown in ① in Figure 12, if decreasing the number of active warps, i.e., moving to the left on the x-axis, would *potentially* increase $stall_{GPU}$ by more than a parameter k (⓭ in Figure 11), i.e., reducing concurrency might increase $stall_{GPU}$ by more than k, CM-BAL does not allow concurrency to be reduced. Second, as shown in ② in Figure 12, if increasing the number of active warps, i.e., moving to the right on the x-axis, would *potentially* reduce $stall_{GPU}$ by more than k, i.e., increasing concurrency might decrease $stall_{GPU}$ by more than k, CM-BAL increases concurrency to the next level. In order to predict if changing concurrency would increase/decrease the value of $stall_{GPU}$ by more than k in either case, CM-BAL logic compares the moving average of $stall_{GPU}$ of the current level with that of the lower/upper level.[11] Third, if the algorithm stays in the same concurrency level for 4 consecutive intervals, we increase or decrease TLP if the current concurrency level is less than 6 or greater than or equal to 6, respectively (all thresholds empirically determined). We do this to solely update the moving average of $stall_{GPU}$ for neighboring levels as those values might become unrepresentative of the current phase of the application if not updated.

The higher the k value is, the more difficult it is to improve the latency tolerance of GPU cores. Thus, we use this parameter as a user-controlled knob (⓭) to specify the GPU priority (or, the importance of the GPU relative to the CPU). We provide the user with four different levels of control for this knob: CM-BAL$_1$, CM-BAL$_2$, CM-BAL$_3$, and CM-BAL$_4$. CM-BAL$_1$ and CM-BAL$_4$ provide the highest and the lowest GPU performance, respectively. The corresponding k for each of these four levels is linearly proportional to the number of core cycles in an interval. k can take any value between 0 and $stall_{GPU}$. When k is set to $stall_{GPU}$, CM-BAL converges to CM-CPU.

5 Experimental Methodology

Simulated System. Table I provides the configuration details of GPU and CPU cores. A GPU core contains 32-wide SIMD lanes and is equipped with an instruction cache, private L1 data cache, constant and texture caches, and shared memory. Each CPU core is a trace-driven, cycle-accurate, 3-issue x86 core; and has a private write-back L1 instruction/data and L2 caches. We use a detailed GDDR5 DRAM timing model [38].

Table I: Baseline configuration.

GPU Core Config.	28 Shader Cores, 1400MHz, SIMT Width = 16 × 2
GPU Resources / Core	Max.1536 Threads (48 warps, 32 threads/warp), 48KB Shared Memory, 32684 Registers
GPU Caches / Core	16KB 4-way L1 Data Cache, 12KB 24-way Texture, 8KB 2-way Constant Cache, 2KB 4-way I-cache, 128B Line Size
CPU Core Config.	14 x86 Cores, 2000MHz, 128-entry instruction window, OoO Fetch and Execute 3 instructions/cycle, max. 1 memory instruction/cycle
CPU L1 Cache / Core	32KB 4-way, 2-cycle lookup, 128B Line Size
CPU L2 Cache / Core	256KB 8-way, 8-cycle lookup, 128B Line Size
Shared LLC Cache	1 MB/Memory Partition, 128B Line, 16-way, 700MHz
Default Warp Scheduler	Greedy-then-oldest [43]
Features	Memory Coalescing, Inter-warp Merging, Post Dominator
Interconnect	6 × 6 Shared 2D Mesh, 1400MHz, XY Routing, 2 GPU cores per node, 1 CPU core per node, 32B Channel Width, 4VCs, Buffers/VC = 4
Memory Model	8 Shared GDDR5 MCs, 800 MHz, FR-FCFS [42, 51], 8 DRAM-banks/MC
GDDR5 Timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{CCD}=2, t_{RCD}=12, t_{RRD}=6, t_{CDLR}=5, t_{WR}=12

Evaluation Methodology. To evaluate our proposal, we integrated GPGPU-Sim v3.2.0 [5] with an in-house cycle-level x86 CMP simulator. The simulation starts with CPU execution. After the slowest CPU core warms up with 500K instructions, GPU execution starts. To measure CPU performance, we run until the slowest core reaches 5 million instructions. To measure GPU performance, we run the application until completion or 100 million instructions, whichever comes first.

Application Suites for GPUs and Benchmarks for CPUs. We evaluate a wide range of applications for GPUs and CPUs. Our repertoire of 13 GPU applications, shown in Table II, come from Rodinia [9], Parboil [46], MapReduce [17], the CUDA SDK [5] and LonestarGPU [7]. We classify these applications into three categories based on

[11] If the moving average of $stall_{GPU}$ in the lower or upper level is invalid for this comparison, Part-2 does not override the decision made by Part-1 of CM-BAL.

their optimal TLP. The applications that perform the best with higher TLP values belong to *Type-H*, and they have high latency tolerance. The applications that perform the best with lower TLP values due to lower congestion and better cache performance belong to *Type-L*. The remaining applications that are mostly insensitive to TLP belong to *Type-M*. The CPU applications we use include scientific, commercial, and desktop applications drawn from the SPEC® CPU 2000/2006 INT and FP suites and commercial server workloads. We choose representative execution phases from these applications using PinPoints [33]. We classify the CPU applications into three types based on their LLC misses-per-kilo-instruction (MPKI) rates, as shown in Table III.

Table II: GPU applications: *Type-H*: Applications that benefit from higher TLP. *Type-M*: Applications that are mostly insensitive to TLP. *Type-L*: Applications that benefit from lower TLP. Optimal TLP (Opt. TLP) is given in number of warps.

#	Suite	Application	Abbr.	Type	Opt. TLP
1	Rodinia	Backpropagation	BP	Type-H	48
2	Rodinia	Hotspot	HOT	Type-H	16-48
3	Rodinia	Pathfinder	PATH	Type-H	48
4	Parboil	Matrix Mult.	MM	Type-H	16-48
5	CUDA SDK	BlackScholes	BLK	Type-M	16
6	MapReduce	Page View Count	PVC	Type-M	8
7	MapReduce	Page View Rank	PVR	Type-M	8
8	LonestarGPU	Breadth-First Src.	BFS	Type-M	6
9	LonestarGPU	Min. Span. Tree	MST	Type-M	8
10	CUDA SDK	MUMerGPU	MUM	Type-L	4
11	MapReduce	InvertedIndex	IIX	Type-L	1
12	LonestarGPU	Single-Source Shortest Paths	SP	Type-L	4
13	LonestarGPU	Survey Propagation	SSSP	Type-L	6

Table III: CPU applications: L2 MPKI and classification

#	App.	L2 MPKI	Type	#	App.	L2 MPKI	Type
1	perlbench	0.2	L-MPKI	18	tpcw	4.8	M-MPKI
2	povray	0.2	L-MPKI	19	Gems	5.2	M-MPKI
3	tonto	0.2	L-MPKI	20	hmmer	7	M-MPKI
4	applu	0.3	L-MPKI	21	astar	7.1	M-MPKI
5	calculix	0.3	L-MPKI	22	sjbb	7.5	M-MPKI
6	gcc	0.3	L-MPKI	23	swim	10	M-MPKI
7	namd	0.4	L-MPKI	24	ocean	10.6	M-MPKI
8	barnes	0.7	L-MPKI	25	sphinx3	12	M-MPKI
9	gromacs	0.7	L-MPKI	26	libquantum	12.5	M-MPKI
10	sjeng	0.8	L-MPKI	27	art	13.6	M-MPKI
11	dealII	1	L-MPKI	28	lbm	14	M-MPKI
12	wrf	1.3	L-MPKI	29	leslie3d	14.2	M-MPKI
13	h264ref	1.5	L-MPKI	30	xalan	14.6	M-MPKI
14	cactus	1.6	L-MPKI	31	milc	25	H-MPKI
15	gobmk	1.6	L-MPKI	32	omnetpp	25.8	H-MPKI
16	bzip2	3.1	M-MPKI	33	mcf	49.8	H-MPKI
17	sjas	4	M-MPKI	34	soplex	103.2	H-MPKI

Workloads. From the three CPU application types, we form two CPU mixes per type,[12] listed in Table IV. Each of these mixes are then coupled with a GPU application to form a workload. The GPU application is chosen in such a way that each CPU mix gets paired with two randomly chosen GPU applications from each of the three GPU application types. Thus, we have six workloads for each CPU mix, for a total of 36 workloads. These workloads cover all possible combinations of CPU and GPU application types; 3 CPU and 3 GPU application types form 9 workload types (e.g., Type-H/L-MPKI), and 4 workloads for each workload type.

Table IV: List of CPU workloads (application mixes).

CPU Mix	Applications in the mix
L1	povray, tonto, applu, calculix, gcc, namd, barnes, gromacs, sjeng, dealII, wrf, h264ref, cactusADM, gobmk
L2	sjeng, dealII, wrf, h264ref, cactusADM, gobmk, sjeng, dealII, wrf, h264ref, cactusADM, gobmk, sjeng, dealII
M1	sjas, tpcw, Gems, hmmer, astar, sjbb, swim, ocean, sphinx3, libquantum, art, lbm, leslie3d, xalan
M2	ocean, sphinx3, libquantum, art, lbm, leslie3d, xalan, ocean, sphinx3, libquantum, art, lbm, leslie3d, xalan
H1	milc, soplex, milc, omnetpp, mcf, soplex, milc, omnetpp, mcf, soplex, mcf, soplex, omnetpp, mcf
H2	omnetpp, omnetpp, omnetpp, omnetpp, omnetpp, milc, milc, milc, milc, milc, soplex, soplex, mcf, mcf

Performance Metrics.. To capture GPU performance, we use GPU speedup (SU_{GPU}), which is the ratio of its instructions-per-cycle (IPC) when it runs along with CPU to its IPC when it runs alone. We use weighted speedup to capture CPU performance (WS_{CPU}) [44]. WS_{CPU} is defined as $\sum_{i=1}^{n}(IPC_{i,multiprogram}/IPC_{i,alone})$, where *n* is the number of CPU applications in the workload. All average speedup results in the paper use *harmonic mean*. Based on user preferences, one might want to change the importance given to CPU or GPU when calculating speedup [4]. For this reason, we use the *overall system speedup* (OSS) metric [4]: $OSS = (1-\alpha) \times WS_{CPU} + \alpha \times SU_{GPU}$, where α is a number between 0 and 1. A higher α indicates higher GPU importance. Note that α is only an *evaluation metric parameter*, not an input to our schemes.

Mechanism Parameters. k is a user-dependent parameter to specify the relative importance of GPU vs. CPU. A high value of k places less importance on the GPU. In our evaluations, we use four values for k: 32 (CM-BAL$_1$), 64 (CM-BAL$_2$), 96 (CM-BAL$_3$), 128 (CM-BAL$_4$). To achieve balanced improvements for both the CPU and the GPU, the user can set k to 32 (called the CM-BAL$_1$ configuration).

Mechanism Thresholds. We use 1 and 0.25 for t_H and t_L, respectively. These thresholds are determined empirically, and they correlate well with their effect on CPU applications' performance, as given in Figure 8. These thresholds also correlate well with the memory bandwidth saturation and reply network saturation, as memory access latencies start to increase non-linearly beyond the high threshold congestion range identified by these thresholds. Monitoring interval is set to 1024 cycles. The thresholds require calibration for different system designs.

Hardware Overhead. As shown in Figure 11, in each core, the warp scheduler requires a table (❾ in Figure 11) with 99-bits[13], a 6-bit register (❿), and a comparator (❻). CM-BAL logic (❽) requires two 10-bit adders and two shift registers. The CTA scheduler (❺) requires two 13-bit counters and two adders to calculate/store the congestion metrics, two 13-bit

[12]We limited the number of CPU mixes because we found that the behavior of the mixes that belong to the same category are similar.

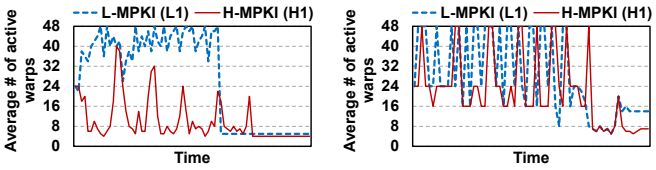
[13]There are nine entries in the table. Each entry requires 11 bits because it is incremented by two warp schedulers during a 1024-cycle interval.

registers to hold the congestion thresholds, and four comparators for the CM-CPU decision. Each memory partition (❸, ❹) requires two 10-bit counters to sample the congestion metrics. As such, the total storage cost of our proposal is 16 bytes per GPU core, 3 bytes per memory partition, and 8 bytes for the CTA scheduler.

6 Experimental Results

6.1 Dynamism of Concurrency

We first analyze the effect of our techniques on GPU concurrency with two studies. First, Figure 13 shows the average number of active warps on GPU cores during the course of execution. Figure 13a shows the concurrency behavior of a Type-H GPU application (PATH) when it is executed alongside two different CPU mixes (L1 and H1), with CM-CPU. Because higher MPKI CPU applications exert more pressure on the memory system than lower MPKI applications, and because CM-CPU decisions are based solely on congestion metrics, the number of active warps is lower when a GPU application is executed alongside higher MPKI applications. This causes CM-CPU to hurt the performance of Type-H GPU applications when boosting CPU performance. Figure 13b shows that the TLP in CM-BAL$_1$ is higher than in CM-CPU for the same workloads (except for sudden drops from 48 to 24 warps with the L-MPKI workload because there are no concurrency levels between 24 and 48 warps). As expected, L-MPKI CPU workloads lead to higher TLP compared to H-MPKI workloads. We observed similar behavior for Type-M and Type-L GPU applications when run with L- and H-MPKI CPU mixes.



(a) PATH with CM-CPU. *(b)* PATH with CM-BAL$_1$.

Figure 13: Average number of active warps over time.

Second, Figure 14 provides insight into the working of CM-BAL by showing the relationship between the metrics monitored by CM-BAL and the number of active warps on a core, over a short period of time, in one of our workloads (Type-L/H-MPKI – MUM/H1). In this plot, $diff_{low}$ = ($stall_{GPU}$ of the lower concurrency level − $stall_{GPU}$ of the current concurrency level); and $diff_{up}$ = ($stall_{GPU}$ of the current concurrency level − $stall_{GPU}$ of the upper concurrency level). If $diff_{low} > k$, CM-BAL does not reduce the GPU TLP. If $diff_{up} > k$, CM-BAL increases the GPU TLP (see Section 4.2). At Ⓐ, since $diff_{up} > k$, the number of warps is increased for the next interval. This workload stresses the memory, leading to high $stall_{MC}$. As a result, at Ⓑ, Part-1 of CM-BAL wants to reduce TLP. However, because $diff_{low} > k$ (i.e., reducing TLP would likely lead to increased GPU stall time), Part-2 of CM-BAL kicks in, and TLP is not reduced. In contrast, a similar scenario at Ⓒ leads to lower TLP in the next interval, because the concurrency level has remained unchanged for the past 4 consecutive intervals. At Ⓓ, Part-1 of CM-BAL reduces TLP because $diff_{up} < k$, and $diff_{low} < k$. This example shows that CM-BAL effectively adapts concurrency to dynamic workload behavior at fine granularity.

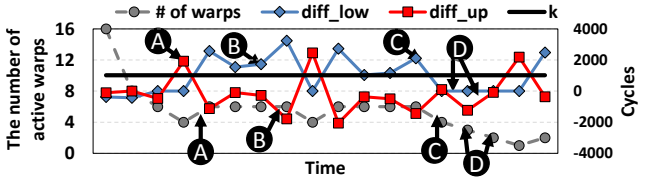


Figure 14: Number of active warps over time with CM-BAL. $diff_{low}$, $diff_{up}$, and k are shown in the secondary y-axis.

Figure 15 shows the average number of warps in a GPU core across nine workload types, with CM-CPU and CM-BAL$_1$. As expected, CM-CPU reduces concurrency very aggressively. CM-BAL$_1$ maintains high TLP for Type-H applications, low TLP for Type-L applications, and reduces TLP of Type-M applications as much as possible without hurting them. Hence, our techniques effectively modulate GPU concurrency in an application-aware manner.

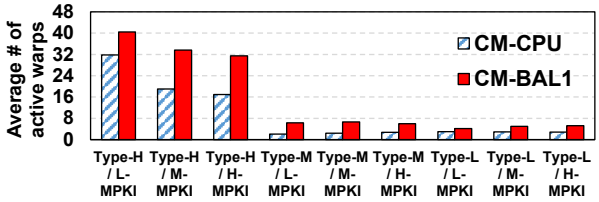


Figure 15: Average number of warps with CM-BAL$_1$.

6.2 Application Performance Results

Figure 16 shows the average performance of GPU applications obtained by our schemes for all workload types. The results are normalized to the performance of the same workloads when they are executed using the baseline warp scheduler, GTO. We also evaluate DYNCTA after tuning its parameters. We make several observations. First, DYNCTA works only for Type-L applications, and improves their performance by 7%, while not hurting any workload by more than 2%. Second, CM-CPU causes significant performance losses in Type-H and Type-M applications, as those applications require high latency tolerance and CM-CPU is aware only of memory congestion, not the latency tolerance of GPU cores. However, CM-CPU improves the performance of Type-L applications due to successful congestion management. We observe high benefits for Type-L/L-MPKI workloads due to two workloads that run IIX. IIX prefers 1-2 active warps, but its CTAs have more warps. Because DYNCTA manages the TLP at the CTA level, it cannot reduce concurrency as much CM-CPU can. Third, CM-BAL$_1$, recovers the losses caused by CM-CPU for Type-H and Type-M applications by providing more latency tolerance. As a result, the worst performing GPU application

suffers only 4% performance loss. No workload has higher GPU performance with CM-CPU than with CM-BAL$_1$. As k increases, GPU benefits of CM-BAL reduce. Overall, CM-CPU causes 11% performance loss, and CM-BAL$_1$ provides 7% performance improvement for GPU applications.

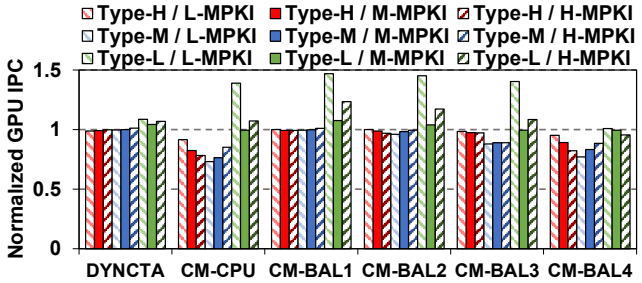


Figure 16: GPU speedup over baseline with 48 warps.

Figure 17 shows the performance of CPU workloads with our schemes, normalized to that of our baseline. First, as discussed in Section 3.1, DYNCTA is not suitable for improving CPU performance, and we observe that it provides only 2% performance improvement for CPUs. Second, CM-CPU reduces the number of active warps very aggressively, providing high benefits for CPUs. Because the GPU performance loss of Type-M applications in CM-CPU is substantial due to aggressive throttling, the corresponding CPU benefits are high. Third, CM-BAL$_1$ throttles Type-H applications conservatively. Thus, CPU benefits for those applications are low. As k increases, CPU benefits of CM-BAL increase, and for the largest possible k (described in Section 4.2), these benefits converge to those of CM-CPU. Fourth, CM-BAL provides a knob for controlling trade-offs between CPU and GPU performance, with CM-BAL$_1$ specifically helping GPU applications and CM-BAL$_4$ helping CPU applications. Overall, CM-CPU and CM-BAL$_1$ provide 24% and 7% average CPU performance improvement, respectively, across all workloads.

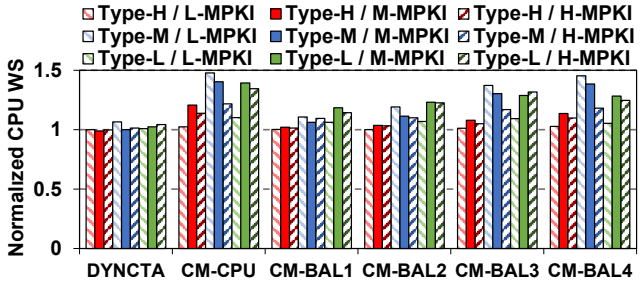
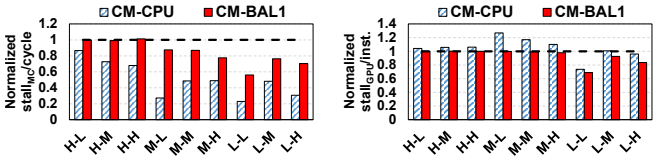


Figure 17: CPU weighted speedup over baseline with 48 warps.

Figure 18 shows the effects of our schemes on the metrics used by them, normalized to their values in our baseline. We do not show $stall_{net}$ because it is negligible in most workloads. $stall_{MC}$ per cycle goes down significantly with CM-CPU, due to aggressive throttling. However, this leads to an increase in $stall_{GPU}$ per executed GPU instruction in most workloads, i.e., lower latency tolerance for GPU cores. CM-BAL$_1$ reduces memory congestion, although not as much as CM-CPU. Yet, it also does not cause lower latency tolerance (i.e., increased stalls) for GPU cores; in fact, it reduces GPU stalls in most Type-L applications.



(a) $stall_{MC}$ per cycle. *(b)* $stall_{GPU}$/GPU instruction.

Figure 18: $stall_{MC}$ and $stall_{GPU}$ normalized to 48 warps, for each workload type. H-L denotes Type-H/L-MPKI workload.

Overall System Speedup. Figure 19 compares the baseline, DYNCTA, and our schemes in terms of Overall System Speedup (OSS) [4], defined in Section 5. Recall that a higher α value indicates that GPU performance is more important for overall system performance. Several observations are in order. First, DYNCTA is slightly better than the baseline for all α values, indicating balanced improvements for both CPUs and GPUs. Second, CM-BAL$_1$ also provides balanced improvement, but is significantly better than DYNCTA. Third, for small α, where preference is given to CPU performance, CM-CPU is the best scheme. As α approaches 0.5, where the OSS metric equally weights CPU and GPU performance, both schemes provide almost equal performance. Because CM-CPU deteriorates GPU performance, its OSS drops below that of the baseline for large α. Fourth, with large k, CM-BAL becomes more preferable for small α, but performs worse than the baseline for large α, showing the trade-off between CPU and GPU performance in CM-BAL. These results show that the user can pick a different CM-BAL level (based on preference of how important the GPU is relative to the CPU) that would lead to the highest OSS. Hence, CM-BAL can be configured to consistently provide better overall system performance than state-of-the-art designs regardless of how important the CPU or the GPU is to system performance.

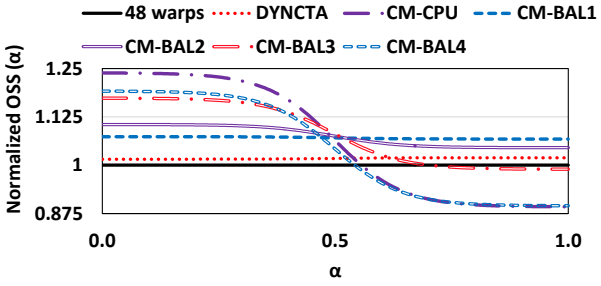
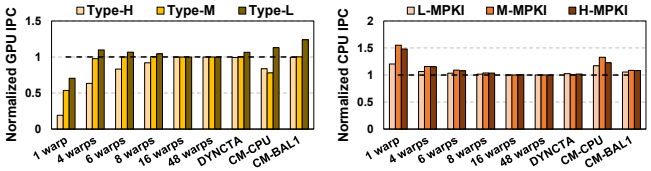


Figure 19: Normalized Overall System Speedup of our schemes over baseline with 48 warps for all α.

6.3 Comparison to Static Warp Limiting

Figure 20a compares the GPU performance of our schemes with static limiting (SWL) schemes, where the GPU cores execute a *constant* number of warps concurrently throughout the whole execution. Throttling down from 48



(a) GPU performance. *(b)* CPU performance.

Figure 20: Our schemes vs. static warp limiting. Performance comparisons are clustered based on GPU/CPU application types.

warps to 4 causes performance loss for Type-H applications, improves Type-L applications, and does not affect Type-M applications. Further throttling hurts GPU applications significantly. CM-BAL$_1$ provides better performance for GPUs compared to all static warp limits. Figure 20b shows the effect of static warp throttling on CPU performance. Although limiting the number of warps to 4 or 6 benefits CPU performance, this comes at the expense of significant GPU performance degradation.[14] We conclude that static warp throttling cannot enable flexible performance trade-off between the CPU and the GPU and optimization of overall system performance as CM-BAL does (Section 6.2).

6.4 Sensitivity Experiments

Scalability. Our scalability analysis shows that varying the number of GPU cores in a tile between 2, 4 (default), and 6 shows minimal change in performance benefits of our techniques. We also evaluated a smaller and less power-hungry system with 4 GPU cores, 3 CPU cores, and 2 MCs connected via a 3×3 mesh. This configuration is closer to currently available systems in terms of the number of cores. CM-BAL$_1$ provides 13% and 2% average CPU and GPU performance improvement, respectively, on this system, even with unoptimized thresholds.

Partitioned vs. Shared Resources. We evaluated CM-BAL$_1$ using a partitioned network and shared MCs. On this baseline, which performs significantly worse than the shared network-shared MC baseline (Section 2.2), CM-BAL$_1$ provides 15% GPU and 8% CPU performance improvement.

6.5 Other Analyses and Discussion

LLC Contention. Our mechanisms reduce LLC contention mainly due to two reasons. First, TLP management reduces the rate at which memory requests are issued from GPU cores. Second, TLP management reduces L1 miss rates, leading to fewer LLC accesses. Overall, we observe that CM-BAL$_1$ reduces the number of LLC accesses by 13%.

Power Consumption. Recent works [16, 45] show that concurrency management in GPUs not only improves performance but also increases energy-efficiency for memory-intensive applications. Thus, we expect our schemes to improve energy-efficiency for memory-intensive workloads, and not reduce it for compute-intensive workloads.

[14] At the extreme case, completely disabling GPU execution leads to 69% higher CPU performance over the baseline.

Effect of GPU Aggressiveness. Although we evaluate our proposal with a Fermi-like GPU model (Section 5), we also expect to observe benefits in newer architectures, due to increased TLP and increased contention for shared memory resources. For example, in the newer NVIDIA Kepler [39]: 1) GPU cores have more processing elements, leading to more GPU TLP, 2) L1 cache capacity is the same as Fermi [38], leading to higher cache contention, 3) GDDR5 memory bandwidth [1] has not increased proportionally with core count, leading to higher memory contention. We envision that these three factors, which indicate trends in GPU platform design, will make concurrency management even more important in newer and future architectures.

7 Related Work

When CPU and GPU applications are co-scheduled on the same hardware, they interfere with each other in the shared resources. In this context, we discuss the prior works that address such interference at various points in the hierarchy.

Managing Interference at NoC, Caches, and Memory. Lee et al. characterize network contention [31] and propose virtual channel partitioning [30] to mitigate it. Ausavarungnirun et al. [4] propose a memory scheduling design that reduces the interference of CPU and GPU applications. Jeong et al. [19] provide a mechanism that provides QoS for GPU applications in a CPU-GPU system by aptly prioritizing requests at the MCs. Lee and Kim [29] observe interference between CPU and GPU applications at the LLC, and propose cache management techniques that also consider TLP of GPU applications. Jog et al. [21] propose a simple round-robin memory scheduler for reducing contention caused by multiple GPU applications. The focus of all these works is to handle the application interference at only one of the levels of the memory hierarchy. In contrast, we show that GPUs create interference to CPUs at all parts of the shared memory hierarchy. To address this interference, we modulate the TLP of GPU applications by considering system-wide congestion metrics and GPU latency tolerance.

Managing Interference from the Core. Kayiran et al. [24] and Rogers et al. [43] modulate TLP in GPUs to reduce contention in the memory system and L1 caches, respectively. Jog et al. [22, 23] propose warp scheduling techniques to address contention in GPU caches and memory. Many source throttling schemes [8, 10, 15, 40, 41, 48] are proposed in the context of CPUs to address congestion in NoCs and memory. None of these works are designed or evaluated for CPU-GPU architectures. To our knowledge, this is the first work that uses source throttling to 1) to control interference between *both CPU and GPU* applications, and 2) to reduce interference *both at the NoCs and main memory in a heterogeneous CPU-GPU system.*

8 Conclusions

TLP management in GPUs is essential in hybrid CPU-GPU environments with shared resources to maximize overall system throughput. In this paper, we introduce a new GPU concurrency throttling strategy that tunes the impact

of GPU applications on both CPU applications and system performance. Our strategy takes into account both system-wide memory/network congestion and GPU latency tolerance to dynamically decide the level of GPU concurrency. We propose two different implementations of our strategy: 1) to boost the performance of CPUs, and 2) to flexibly improve overall system performance based on the user's preference for higher CPU or GPU performance. Experimental evaluations show that the first scheme significantly improves average CPU performance, but causes some GPU performance loss. The tunable second scheme, when configured to improve CPU and GPU performance in a balanced manner, provides 7% performance benefit for *both* CPU and GPU applications, without significantly hurting any workload's performance. We conclude that our GPU TLP management framework provides a flexible and efficient substrate to maximize system performance and control CPU-GPU performance trade-offs in modern heterogeneous architectures.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research is supported in part by NSF grants #1212962, #1205618, #1213052, #1302225, #1302557, #1317560, #1320478, #1409095, #1439021, #1439057, #1409723, the Intel Science and Technology Center on Cloud Computing, SRC and Intel.

REFERENCES

[1] G. Abbas. (2012) NVIDIA CUDA: Kepler vs. Fermi Architecture. Available: http://blog.cuvilib.com/2012/03/28/nvidia-cuda-kepler-vs-fermi-architecture/
[2] Advanced Micro Devices, Inc. (2013) What is Heterogeneous System Architecture (HSA)? Available: http://www.amd.com/en-gb/innovations/software-technologies/hsa
[3] AMD. (2014) Compute Cores. Available: http://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf
[4] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[5] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.
[6] A. Bakhoda *et al.*, "Throughput-effective On-chip Networks for Manycore Accelerators," in *MICRO*, 2010.
[7] M. Burtscher *et al.*, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, 2012.
[8] K. K.-W. Chang *et al.*, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks," in *SBAC-PAD*, 2012.
[9] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.
[10] H. Cheng *et al.*, "Memory Latency Reduction via Thread Throttling," in *MICRO*, 2010.
[11] W. Dally, "GPU Computing to Exascale and Beyond," in *SC*, 2010.
[12] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.
[13] R. Das *et al.*, "Application-to-core Mapping Policies to Reduce Memory System Interference in Multi-core Systems," in *HPCA*, 2013.
[14] R. Das *et al.*, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.
[15] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *ASPLOS*, 2010.
[16] M. Gebhart *et al.*, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *ISCA*, 2011.
[17] B. He *et al.*, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.
[18] Intel. (2012) Products (Formerly Ivy Bridge). Available: http://ark.intel.com/products/codename/29902/
[19] M. K. Jeong *et al.*, "A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.
[20] M. K. Jeong *et al.*, "Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems," in *HPCA*, 2012.
[21] A. Jog *et al.*, "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *GPGPU*, 2014.
[22] A. Jog *et al.*, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.
[23] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.
[24] O. Kayiran *et al.*, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.
[25] Khronos OpenCL Working Group, "The OpenCL Specification," 2008.
[26] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[27] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[28] C. J. Lee *et al.*, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching" ," in *MICRO*, 2009.
[29] J. Lee and H. Kim, "TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture," in *HPCA*, 2012.
[30] J. Lee *et al.*, "Adaptive Virtual Channel Partitioning for Network-on-chip in Heterogeneous Architectures," *ACM TODAES*, 2013.
[31] J. Lee *et al.*, "Design Space Exploration of On-chip Ring Interconnection for a CPU-GPU Heterogeneous Architecture," *JPDC*, 2013.
[32] E. Lindholm *et al.*, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, no. 2, 2008.
[33] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.
[34] A. K. Mishra *et al.*, "A Heterogeneous Multiple Network-on-chip Design: An Application-aware Approach," in *DAC*, 2013.
[35] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning," in *MICRO*, 2011.
[36] V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*, 2011.
[37] NVIDIA. (2011) CUDA C/C++ SDK Code Samples. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples
[38] NVIDIA. (2011) Fermi: NVIDIA's Next Generation CUDA Compute Architecture. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[39] NVIDIA. (2012) NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
[40] G. Nychis *et al.*, "Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?" in *Hotnets*, 2010.
[41] G. Nychis *et al.*, "On-chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects," in *SIGCOMM*, 2012.
[42] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.
[43] T. G. Rogers *et al.*, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.
[44] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.
[45] S. Song *et al.*, "Energy-efficient Scheduling for Memory-intensive GPGPU Workloads," in *DATE*, 2014.
[46] J. A. Stratton *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Univ. of Illinois, Tech. Rep. IMPACT-12-01, March 2012.
[47] M. A. Suleman *et al.*, "Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs," in *ASPLOS*, 2008.
[48] M. Thottethodi *et al.*, "Self-Tuned Congestion Control for Multiprocessor Networks," in *HPCA*, 2001.
[49] M. Xie *et al.*, "Improving System Throughput and Fairness Simultaneously in Shared Memory CMP Systems via Dynamic Bank Partitioning," in *HPCA*, 2014.
[50] X. Zhang *et al.*, "Hardware Execution Throttling for Multi-core Resource Management," in *USENIX*, 2009.
[51] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order," Patent U.S. 5,630,096, 1997.