This is a 2-page summary of the original paper – Managing GPU Concurrency in Heterogeneous Architectures, MICRO-2014

Concurrency Management 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.*

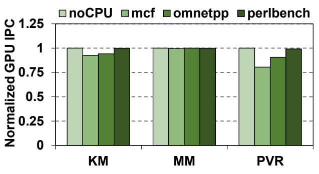
Abstract—**Heterogeneous architectures consisting of general-purpose CPUs and throughput-optimized GPUs are projected to be a dominant computing platform for many classes of applications. The design of such systems is more complex than for a homogeneous architecture because maximizing resource utilization while minimizing the interference between CPU and GPU applications is difficult. We show that GPU applications tend to monopolize the shared resources such as memory and network because of their high thread-level parallelism (TLP). 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. It 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 two schemes, one targeted specifically for unilaterally boosting CPU performance (CM-CPU), and the other (CM-BAL) for a balanced improvement of both CPU and GPU applications. 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. 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.**[1]

I. Summary

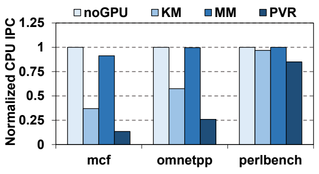
A. Problem and Solution

Problem: Application Interference. 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). The interference between CPU and GPU applications causes performance losses for both classes. CPU applications, however, are affected much more compared to GPU applications. 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.

Illustration: Application Interference. We ran different mixes of CPU and GPU applications concurrently, and also in isolation to estimate the performance impact of resource sharing on applications. Figure 1a shows GPU performance when running each GPU application with and without one of the three different CPU applications. Figure 1b shows CPU performance when running each CPU application with and without one of the three GPU applications. These results confirm that CPU applications are hurt more than GPU applications when they are co-scheduled.



(a) Effect of CPU Applications on GPU performance.



(b) Effect of GPU Applications on CPU performance.

Figure 1: Effects of heterogeneous execution on performance.

Naive Solution: Resource Partitioning. A naive approach to reduce the interference of CPU and GPU applications is to isolate these applications from each other by partitioning the shared resources. Our experimental evaluations show that partitioning the network (keeping the network bisection bandwidth and the amount of resources constant) helps latency-sensitive CPU applications, but causes resource underutilization, leading to a drop in GPU performance. Similarly, dedicating some MCs to serve GPU requests and the others to serve CPU requests results in lower memory bandwidth utilization, and reduces overall system performance. We also observe that partitioning both the network and the MCs is not preferable except for a few cases, as such partitioning degrades both CPU and GPU performance. Thus, partitioning resources is not a desirable option from the performance standpoint.

Idea: Employing Concurrency Management. The source of the problem is the high number of requests generated by the GPU, leading to severe contention for the shared resources. Rather than solving the problem where they are observed, such as in the network or at the MCs, we propose TLP management strategies that attack the problem at the source, instead of at the sink. By controlling the number of concurrently executing GPU warps, one can also control the rate at which memory requests are issued, and can reduce contention at the shared cache, network and memory.

Key Observation: Difference in Latency Tolerance be-

[1]**Original paper:** Managing GPU Concurrency in Heterogeneous Architectures [1]

tween CPUs and GPUs. Because the latency tolerance of GPU and CPU cores are different, TLP management solely based on GPU latency tolerance (as done by DYNCTA [2]) might *not* be optimal for CPUs. Figure 2 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. 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 loss for the CPU applications. An approach that solely considers the latency tolerance of GPU cores to modulate GPU TLP would not reduce the concurrency level below 8 warps, and CPU applications would perform poorly compared to an approach that considers the latency tolerance of both classes of applications.

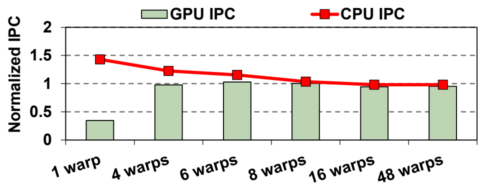


Figure 2: GPU and CPU IPC with different GPU concurrency levels, normalized to DYNCTA.

Solution 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. The second scheme, CM-BAL, achieves the second goal by giving the user multiple options to control the level of GPU concurrency. 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.

Figure 3 shows the overview of our schemes. The y-axis ($stall_{GPU}$) denotes the GPU stall cycles, and the x-axis denotes the GPU TLP. GPU stall cycles increase with low concurrency, mainly due to low latency tolerance, and increase with high concurrency, mainly due to high memory/cache contention [2], [3], [4]. CM-CPU dynamically monitors memory system congestion. It reduces GPU TLP if congestion is high, and increases it if congestion is low. This decision only considers memory system congestion, and does not take GPU latency tolerance into account. The downside of CM-CPU is that throttling the GPUs solely based on memory system congestion might hurt GPU applications that require high TLP for sufficient latency tolerance. To recover the poor GPU latency tolerance due to CM-CPU, CM-BAL detects whether or not GPU cores have enough latency tolerance (whether the application suffers high stall cycles due to low TLP, as shown in R1 in Figure 3), and maintains or increases the number of active warps if GPU cores are not able to hide memory latencies due to insufficient concurrency. We provide the user with four different levels of control for increasing the GPU TLP: 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.

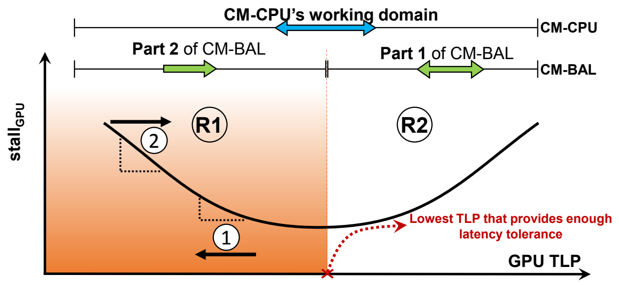


Figure 3: Operation of our schemes. CM-CPU increases/decreases TLP for all TLP ranges. CM-BAL might also improve GPU latency tolerance by increasing TLP if TLP is low.

B. *Evaluation*

Key Results. We evaluate our proposal on an integrated 28-core GPU and 14-core CPU simulation platform with 36 diverse workloads. CM-CPU, on average, provides 24% performance improvement for CPU applications. However, due to aggressive reduction in GPU TLP, average GPU performance drops by 11%. This performance loss is more prominent in workloads that consist of GPU applications that prefer high TLP to hide memory access latencies. CM-BAL, when configured to improve CPU and GPU performance in a balanced manner, recovers the GPU performance losses experienced by CM-CPU, and provides 7% performance benefit for both CPU and GPU applications, without significantly hurting any workloads 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.

Overhead. The total storage cost of our proposal is 16 bytes per GPU core, 3 bytes per memory partition, and 8 bytes for the global CTA scheduler.

References

[1] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.

[2] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.

[3] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[4] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.