Cooperative Caching for GPUs

SAUMAY DUBLISH, VIJAY NAGARAJAN, and NIGEL TOPHAM, University of Edinburgh

The rise of general-purpose computing on GPUs has influenced architectural innovation on them. The introduction of an on-chip cache hierarchy is one such innovation. High L1 miss rates on GPUs, however, indicate inefficient cache usage due to myriad factors, such as cache thrashing and extensive multithreading. Such high L1 miss rates in turn place high demands on the shared L2 bandwidth. Extensive congestion in the L2 access path therefore results in high memory access latencies. In memory-intensive applications, these latencies get exposed due to a lack of active compute threads to mask such high latencies.

In this article, we aim to reduce the pressure on the shared L2 bandwidth, thereby reducing the memory access latencies that lie in the critical path. We identify significant replication of data among private L1 caches, presenting an opportunity to reuse data among L1s. We further show how this reuse can be exploited via an L1 Cooperative Caching Network (CCN), thereby reducing the bandwidth demand on L2. In the proposed architecture, we connect the L1 caches with a lightweight ring network to facilitate intercore communication of shared data. We show that this technique reduces traffic to the L2 cache by an average of 29%, freeing up the bandwidth for other accesses. We also show that the CCN reduces the average memory latency by 24%, thereby reducing core stall cycles by 26% on average. This translates into an overall performance improvement of 14.7% on average (and up to 49%) for applications that exhibit reuse across L1 caches. In doing so, the CCN incurs a nominal area and energy overhead of 1.3% and 2.5%, respectively. Notably, the performance improvement with our proposed CCN compares favorably to the performance improvement achieved by simply doubling the number of L2 banks by up to 34%.

CCS Concepts: • **Computer Systems Organization → Single Instruction, Multiple Data;**

Additional Key Words and Phrases: GPGPU, bandwidth bottlenecks, intercore reuse

ACM Reference Format:
Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2016. Cooperative Caching for GPUs. ACM Trans. Archit. Code Optim. 13, 4, Article 39 (December 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/3001589

1. INTRODUCTION

Current GPUs are no longer perceived as accelerators solely for graphic workloads and now cater to a much broader spectrum of applications. In a short time, GPUs have proven to be of substantive significance in the world of general-purpose computing. The massive compute power of GPUs and recent innovations in their architecture [NVIDIA 2009, 2012] have helped to unleash the latent potential of several non-graphical applications, adding momentum to the rise of general-purpose computing on GPUs (GPGPUs).

Motivated by the pervasive impact of GPUs in the field of general-purpose computing, manufacturers have introduced configurable on-chip cache hierarchies to their recent architectures to cater to the locality needs of non-streaming applications. Despite performance improvement for certain applications, however, the utilization of

Authors' addresses: S. Dublish, V. Nagarajan, and N. Topham, School of Informatics, University of Edinburgh, UK; emails: {saumay.dublish, vijay.nagarajan, nigel.topham}@ed.ac.uk.

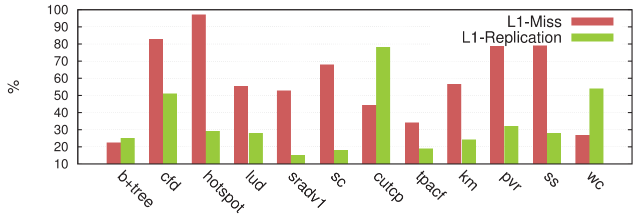


Fig. 1. (a) *L1-Miss*: L1 cache miss rates. (b) *L1-Replication*: Percentage of L1 misses cached in remote L1 caches.

these caches is far from perfect; this is evident from the high cache miss rates seen on many GPUs. As shown in Figure 1(a), on NVIDIA's Fermi GPU, general-purpose applications across a variety of benchmark suites show high L1 miss rates, indicating that the current cache management techniques are unable to utilize these caches effectively. As a consequence of high L1 miss rates, pressure on the L2 bandwidth increases, thereby increasing the memory access latencies due to congestion in the L2 access path. In our experiments (discussed later in Section 5), we observe that due to congestion in the L1-L2 interconnect and L2 access queues, L2 accesses take up to 2 to 3× more cycles compared to the normal access latency of L2. In memory-intensive applications, due to lack of active compute threads to overlap such high memory access latencies, *increased latencies to the lower-level get exposed* and appear in the critical path [Dublish et al. 2016], reducing system performance.

Goal. In this article, our goal is to reduce the memory access latencies that cannot be hidden by multithreading in memory-intensive applications. Since one of the major reasons for such high latencies is the congestion in the L2 access path (due to the high number of requests sent to the lower-level), *we aim to reduce this congestion*.

Observation. In streaming applications, cores work on independent data with little or no overlap in the working dataset. However, in general-purpose applications, we observe a considerable potential for data reuse across different cores. Figure 1(b) shows that a significant percentage of miss requests generated by L1s is for data already present on a *non-local (or remote) L1 cache*. If we can exploit this reuse within L1s, duplicate requests to the shared L2 can be potentially eliminated. This would result in reduced congestion and faster lower-level access for the remaining requests.

Proposal. In this article, we propose a Cooperative Caching Network (CCN) for L1 caches in GPUs to improve the efficiency of the L1 cache hierarchy in filtering the requests to the L2 cache. In our proposed scheme, we connect the private L1 caches in a lightweight ring network to facilitate communication of reusable data among the L1 caches. In doing so, we reduce the average memory access latency due to the following two reasons. First, a fraction of L1 load misses, with reusable data cached on remote L1s, can now completely bypass the high latency access path to L2. They are instead serviced by the CCN with significantly lower latencies (42 cycles on average based on our experiments) as compared to the L2 round-trip access latencies, or simply L2 access latencies (which is approximately 300 cycles due to congestion). Second, cooperatively sharing reusable data within the L1 caches via the CCN reduces the traffic to the L2 cache. This relieves the pressure on the interconnect, as well as on the L2 access queues, thereby reducing the L2 access latencies (by 78 cycles on average). Thus, the CCN provides a faster access to L2 for miss requests that do not find a sharer in the CCN.

In effect, our proposed architecture services a portion of L1 misses collaboratively within the L1 caches with much lower latencies than the L2 access latency. This leads to less congestion in the L2 access path, thereby accelerating memory response for requests that do not find a reusable copy in remote L1 caches. However, in the absence of reuse (e.g., in streaming applications), unsuccessful probes in the CCN add an additional overhead to the L1 load misses. In such cases, due to no reduction in congestion, the CCN overhead does not get compensated and results in an overall performance penalty. Therefore, in our final scheme, we propose Cooperative Caching Network with Request Throttling (CCN-RT). It dynamically adapts to the coarse-grain reuse patterns shown across an entire application, thereby bypassing the CCN when there is little or no reuse.

In summary, we make the following contributions:

—We provide fresh insight into the intercore reuse patterns within GPUs by profiling the communication characteristics over a diverse range of GPGPU applications.
—We propose the CCN, a cooperative caching architecture for GPUs that is cognizant of the intercore reuse.
—By servicing reusable requests via the CCN, we reduce the overall bandwidth demand on the L2 cache, boosting performance for memory-intensive applications that show high levels of sharing across L1s.
—With our final proposal, CCN-RT, we show an average performance gain of 14.7% for applications that exhibit reuse while being benign to applications with no reuse.
—We also reduce the average memory latency (AML) by 24%, L1 to L2 traffic by 29%, and core stall cycles by 26%. Our proposal incurs nominal area and energy overheads of 1.3% and 2.5%, respectively.

The remainder of the article is organized as follows. Section 2 provides an overview of the baseline architecture for our study and characterizes the workloads. Section 3 investigates the reuse patterns for general-purpose applications and assesses the efficacy of cooperative caching in GPUs. Section 4 presents our CCN for L1 caches in GPUs. Section 5 evaluates the architecture and proposed optimizations to our baseline proposal. Section 6 compares the CCN to alternate solutions. Section 7 discusses the related work and positions our findings in the current state of the art, and Section 8 concludes the article by summarizing the findings and contributions of this work.

2. BACKGROUND

2.1. CUDA Programming Model

A typical CUDA program consists of data-parallel structures called *kernels*, which are executed on the GPU. The large number of threads of a kernel are organized into structured blocks of computation known as *thread blocks*. Each thread block consists of several smaller group of threads called *warps*—the smallest granularity of scheduling threads in a GPU core.

2.2. Baseline Architecture

As shown in Figure 2, a typical GPU consists of several execution units organized in a set of highly multithreaded and pipelined cores that are referred to as streaming multiprocessors (SMs[1]). In this study, we consider a baseline similar to NVIDIA's Fermi architecture. Our baseline GPU consists of 15 SMs, each with a 32-lane SIMD unit. Each core consists of a private L1 data cache, shared memory (scratchpad), and read-only texture and constant caches. Private caches of a core are backed by a shared L2 cache that has an access latency of around 120 cycles for non-texture accesses. The L1 data caches are non-coherent and employ *write-through, no-write-allocate* policy. The baseline parameters are summarized later in Table II.

[1]In this article, we use the terms *core* and *SM* interchangeably.

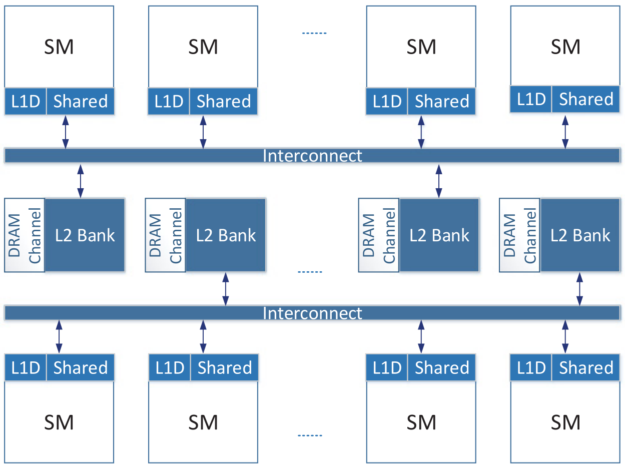


Fig. 2. Baseline GPU architecture.

2.3. Benchmarks

For the purpose of this study, we use CUDA applications from three major general-purpose benchmark suites: Rodinia (v3.0) [Che et al. 2009], MapReduce [He et al. 2008], and Parboil [Stratton et al. 2012]. We categorize the benchmarks according to their sensitivity to the memory hierarchy. Table I lists the benchmarks sorted by the speedup (PerfX) shown on a perfect memory system that has zero access latency to lower-level memories and infinite bandwidth between memory hierarchies on a Fermi-like GPU.

A program is said to be memory-intensive if it constitutes several threads comprising long latency memory operations. The performance of memory-intensive applications is usually bounded by the bandwidth to lower-level memories. This is because a large number of memory requests are kept waiting in each memory partition due to limited bandwidth, thereby delaying memory responses and causing the cores to potentially stall. Therefore, the magnitude of speedup on a perfect memory system essentially indicates the gravity of bandwidth problem in the benchmarks.

3. NEED FOR COOPERATION

Graphics and general-purpose workloads exhibit different memory access patterns. In graphics applications, kernels operate on independent data of streaming nature, and therefore different thread blocks are executed in considerable isolation. On the other hand, general-purpose applications show varying amounts of reuse within the thread blocks and also at the boundaries with neighboring thread blocks. For instance, in scientific application such as computation of Coulombic potential (*cutcp*), atoms are organized in a 3D lattice. A subgroup of atoms constitutes a thread block, and the entire lattice is divided into multiple thread blocks. To compute the potential difference on the atoms at the edges and corners of a sublattice (or thread block), Coulombic potential contributed by atoms from surrounding sublattices needs to be read and hence requires sharing and reuse of data among neighboring thread blocks. When such thread blocks are scheduled on different cores on a GPU, it results in intercore reuse. In current GPUs, reuse across thread blocks on different cores can only be exploited by localizing the data on the L2 cache and not any closer. But in doing so, cores have to incur the congestion delays in the L1-L2 interconnect, as well as the delays in the L2 access

Table I. Benchmark Characterization

S. No.	Suite	Benchmark	ABV.	Dataset	PerfX	μRC
1	MapReduce	Matrix Multiplication	mm	768 × 768 data points	9.86	4%
2	MapReduce	Similarity Score	ss	1024 × 256 data points	6.18	28%
3	Rodinia	Computational Fluid	cfd	200000 elements	6.17	51%
4	MapReduce	Page View Rank	pvr	21MB	5.93	32%
5	Rodinia	Stream Cluster	sc	16384 points; 256 dimension	5.49	18%
6	Rodinia	Breadth-First Search	bfs′	1000000 nodes	5.18	3%
7	Rodinia	Wavelet Transform	dwt2d	1024 × 1024	4.96	7%
8	Parboil	Lattice-Boltzmann Method	lbm	120 × 120 × 150 data points	4.49	0%
9	MapReduce	*K*-Means	km	10000 × 3 data points; 24 clusters	3.85	24%
10	Rodinia	Hybrid Sort	sort	4194304 floating points	3.68	1%
11	Parboil	Breadth-First Search	bfs	8500000 nodes	3.57	6%
12	Rodinia	Particle Potential	lavaMD	7 × 7 × 7 boxes	2.81	1%
13	Parboil	2D Histogram	histo	10000 × 4 dimension	2.63	1%
14	MapReduce	String Match	sm	4MB	2.52	3%
15	Rodinia	Cardiac Myocyte	myocyte	100 instances	2.38	1%
16	Rodinia	Needleman-Wunsch	nw	2048 × 2048 data points	2.31	8%
17	Rodinia	Graph Traversal	b+tree	10000 nodes	2.21	25%
18	MapReduce	Inverted Index	ii	28MB	2.19	2%
19	Rodinia	Particle Filter	pfloat	128 × 128 × 10	2.15	8%
20	Rodinia	Tracking Microscopy	leukocyte	176MB	1.88	1%
21	MapReduce	Word Count	wc	96KB	1.86	54%
22	Parboil	Sum of Absolute Diff.	sad	52KB vs. 52KB frame	1.76	3%
23	Rodinia	Speckle Reduction	sradv1	512 × 512 data points	1.74	15%
24	Rodinia	Speckle Reduction	sradv2	2048 × 2048 data points	1.70	16%
25	Parboil	Cartesian Gridding	mri-g	61MB	1.49	2%
26	Rodinia	*K*-Means	kmeans	204800 data points; 34 features	1.47	0%
27	Rodinia	Matrix Decomposition	lud	2048 × 2048 data points	1.27	28%
28	Parboil	PDE Solver	stencil	512 × 512 × 64 input	1.23	6%
29	Rodinia	Heart Wall Tracking	heartwall	49MB	1.19	0%
30	Rodinia	Back Propagation	backprop	65536 input nodes	1.10	3%
31	Rodinia	Thermal Modeling	hotspot	512 × 512 data points	1.07	29%
32	Parboil	Coulombic Potential	cutcp	96604 atoms	1.00	78%
33	Parboil	MRI Reconstruction	mri-q	64 × 64 × 64 data points	1.00	0%
34	Parboil	Angular Correlation	tpacf	10391 data points	1.00	19%

Note: PerfX, speedup with perfect memory; μRC, percentage of total L1 load misses that have reusable data on a remote L1.

queues. Thus, for those applications that are bounded by the bandwidth to the lower level, it degrades the overall performance by clogging the access path to L2.

3.1. Intercore Reuse

To quantify the degree of temporal and spatial reuse of global data between thread blocks, we analyze the L1 miss traffic of each core. In Table I, we show the reuse coefficient (μRC), which is the percentage of miss requests received by the L2 cache from private L1 caches for addresses that reside remotely on at least one L1 cache. We see a maximum μRC of up to 78% with an average of 14% across all benchmarks. A high μRC for some benchmarks indicates that reuse requests from L1 caches form a large portion of traffic to L2. It is worth noting that we only consider it as reuse if the load miss address is cached on a remote L1 *at the time of the miss*.

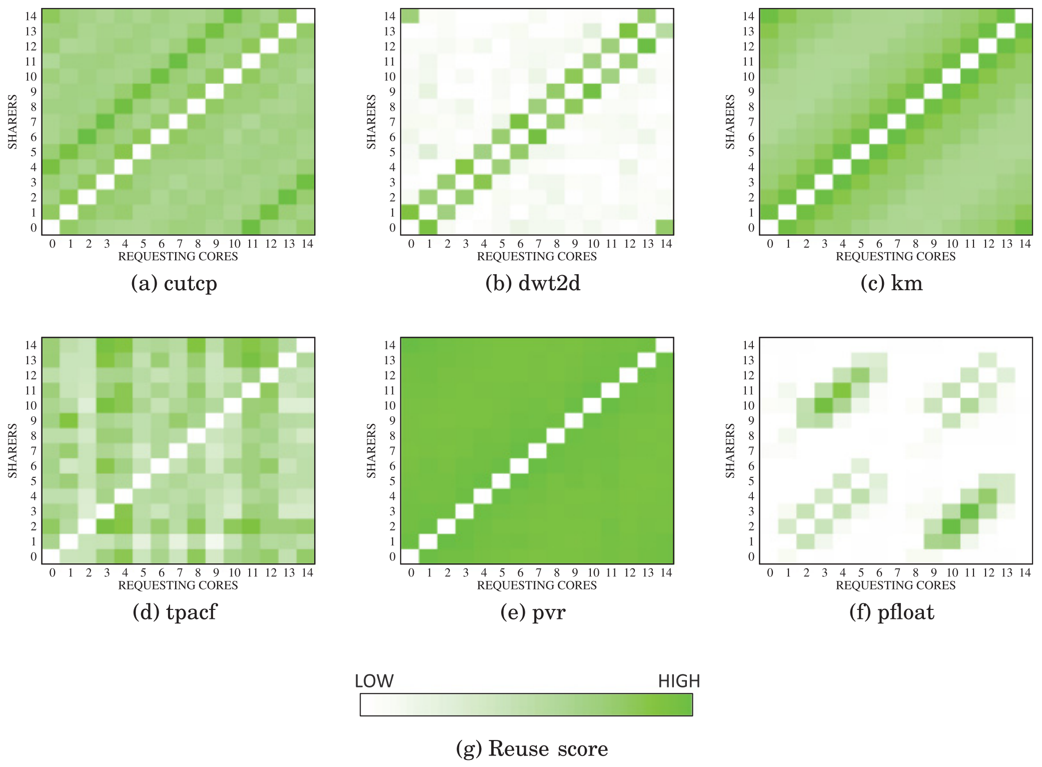


Fig. 3. Heatmaps indicating intercore reuse by cores on the x-axis for data cached on the cores on the y-axis. Dark spots in the heatmaps indicate high reuse between the corresponding cores at their x and y coordinates.

In Figure 3, we further characterize the intercore reuse patterns at the granularity of each core with every other core, providing deeper insight into the reuse dynamics. For brevity, we show the set of distinct observed patterns and omit those that replicate the patterns shown here. The x-axis indicates the cores that incur an L1 load miss, and the y-axis indicates the sharers for that miss. A dense area in the heatmap at an (x, y) coordinate indicates that a high proportion of load miss requests by core-x are cached by L1 at core-y. For instance, *cutcp* shows a prominent reuse of data cached at a distance of four cores from the location of the miss, *dwt2d* shows a strong reuse between neighbors, *km* shows a gradual decline in reuse as we go further from the core, and *tpacf* shows considerable levels of reuse across all cores.

3.2. Efficacy of Cooperation

In the previous section, we showed that for general-purpose applications there is considerable reuse across L1 caches. We refer to those load requests as *reuse requests* that miss in the local L1 but hit in a remote L1. By removing such reuse requests (also quantified as μRC) from the pool of total misses going to the L2 cache, we can reduce the pressure on L2 bandwidth. To assess the efficacy of reducing the bandwidth demand on the overall performance, we begin by examining the performance improvement when the reuse requests do not congest the access path to L2. In these cases, reuse requests are instead serviced cooperatively within L1s with varying remote L1 access latencies, or *reuse latencies*. Since applications with a low μRC are not expected to show any change, we focus on benchmarks with a high μRC. Later, we demonstrate the effect of our final proposal on applications with a low or zero μRC as well.

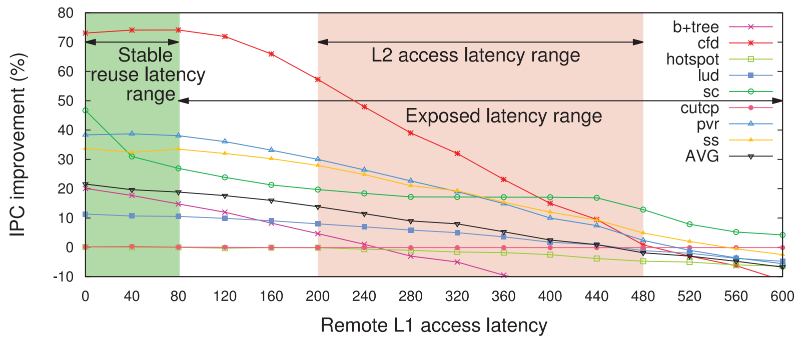


Fig. 4. Speedup of cooperation with varying remote L1 access latencies.

Figure 4 shows the speedup due to cooperation and demonstrates a noticeable improvement in performance, specifically for memory-intensive applications with a high μRC. For instance, *cfd* and *pvr* show performance improvements of up to 73% and 38%, respectively. Both of these applications are severely bounded by the memory bandwidth and at the same time exhibit high reuse. On the other hand, despite high reuse in *cutcp* and *hotspot*, there is no significant gain in IPC since bandwidth is not critical for these benchmarks.

Another *key observation* in this study pertains to the variation of performance as a function of remote L1 access latency. We observe that the performance improvement in the region between 0 and 80 cycles is fairly stable, with the average IPC gain only changing from 21.5% to 18.8%. This is because in this region, latencies to remote L1s can be effectively hidden by the multithreading on the cores. Moreover, due to reduced congestion in the L2 access path and due to faster response to reuse requests (compared to L2 accesses), the average number of active compute threads on a core increases. This boosts the ability of the cores to further mask the memory access latencies. Due to these effects, reuse latencies up to 80 cycles are effectively hidden by multithreading and do not determine the execution time. However, on further increasing the reuse latencies, performance improvement starts to degrade more rapidly. In fact, the IPC gain returns to nearly 0% when the reuse latencies are varied in the range of L2 access latencies (around 300 cycles). This is because latencies for reuse requests become increasingly exposed and can no longer be hidden by multithreading, despite reduced congestion.

In summary, these initial results indicate that for memory-bound applications, when there is considerable reuse of data across L1 caches, cooperation among the private L1 caches can result in a considerable speedup (up to 21.5% on average). Notably, the observed performance improvement is fairly stable in the reuse latency range of 0 to 80 cycles.

4. COOPERATIVE CACHING

In the previous sections, we observed a potential for cooperative caching on GPUs and assessed its efficacy. We now propose a cooperative caching framework to use the private L1 data caches in an aggregate manner. We begin by formalizing the preceding discussion and analyzing the parameters that contribute to the L2 access latencies for L1 miss requests. Later, we propose a cooperative caching scheme and discuss the architectural details.

4.1. Analytical Model

Here we present a simple analytical model to explain the conditions under which reuse delivers a performance gain. First, in the absence of cooperation between L1s, let l_O

be the AML to access the shared L2 cache. Second, with cooperation between L1s, let h_{reuse} be the fraction of L1 misses that hit in a remote L1 cache. Furthermore, let l_{reuse} be the average hit latency for accesses to remote L1s. As a consequence of reduced congestion in the L2 access path due to remote L1 hits, let δ_{cong} be the reduction in L2 access latency. And finally, let $\delta_{overhead}$ be the cooperation overhead borne by those requests that do not have a shared copy. Therefore, the new AML to L2 upon enabling cooperation, l_C, can be obtained via Equation (1).

$$l_C = (l_O - \delta_{cong} + \delta_{overhead}).(1 - h_{reuse}) + l_{reuse}.h_{reuse} \quad (1)$$

$$\left.\begin{array}{l} l_{reuse} < l_O \\ \delta_{overhead} < \delta_{cong} \end{array}\right\} \text{Criteria for useful cooperation} \quad (2)$$

To derive gain from cooperative caching, l_C must be minimized. Therefore, remote L1 accesses for reuse requests must take less time than a normal L2 access (i.e., $l_{reuse} < l_O$). Additionally, we have already seen in Figure 4 that the maximum gain from cooperation is sustained in the lower reuse latency range (i.e., $l_{reuse} \in (0, 80)$). Finally, for the remaining L2 accesses, the cooperation overhead must be less than the benefit obtained from reducing the congestion in the L2 access path (i.e., $\delta_{overhead} < \delta_{cong}$). A combination of preceding conditions will result in a lower average L2 access latency (i.e., $l_C < l_O$).

How should we go about implementing the cooperative caching framework? Following the approach of traditional multicores, a central directory in the L2 cache [Lebeck and Wood 1995; Acacio et al. 2002; Kaxiras and Keramidas 2010] can be used to store information about the sharers. However, maintaining a directory *as part of the L2* will not mitigate the existing bandwidth problem in *accessing the L2* and instead will only worsen it. This is because the additional control and update traffic to the central directory will further increase the bandwidth demand to the L2 cache. Alternatively, an approach along the lines of cooperative caching schemes for CPUs [Chang and Sohi 2006, 2007; Herrero et al. 2008] may be used. Such schemes aim to minimize hop latencies to find a sharer and retrieve data using a highly interconnected network of L1 caches. However, since we have demonstrated that we have a considerable leeway of around 80 cycles to fetch the shared data from a remote L1, such an aggressive scheme to find a sharer is an overkill for GPUs.

Therefore, we propose a lightweight ring-based CCN. A ring topology is the lowest-degree network and requires the fewest number of intercore connections. It is also lowest in terms of logical complexity and power consumption, as all core-to-core connections will be near-neighbor, and therefore the wires will be short. In addition, all routers in a ring are simple multiplexers, which are more energy efficient than complex crossbar routers. As we have shown that GPUs can tolerate reuse latencies gracefully up to 80 cycles, a ring topology appears to be a cost-effective solution, as it allows us to trade off higher latencies for simplicity and short wires (i.e., lower power consumption and die-area cost).

4.2. Architecture

In our proposed scheme, we facilitate the communication between neighbors by connecting the private L1 caches in a ring via our CCN. The CCN is comprised of two different channels: the request channel and response channel. The request channel comprises a network of request queues (ReqQs), whereas the response channel comprises a network of response queues (RespQs). As shown in Figure 5, each L1 has an independent pair of the aforementioned queues to allow the cache to participate in cooperative caching. The L1 caches interact with their home queues via CCN buffers (CBs), which hold the tag and Core-ID for the load misses, until the CCN is ready to

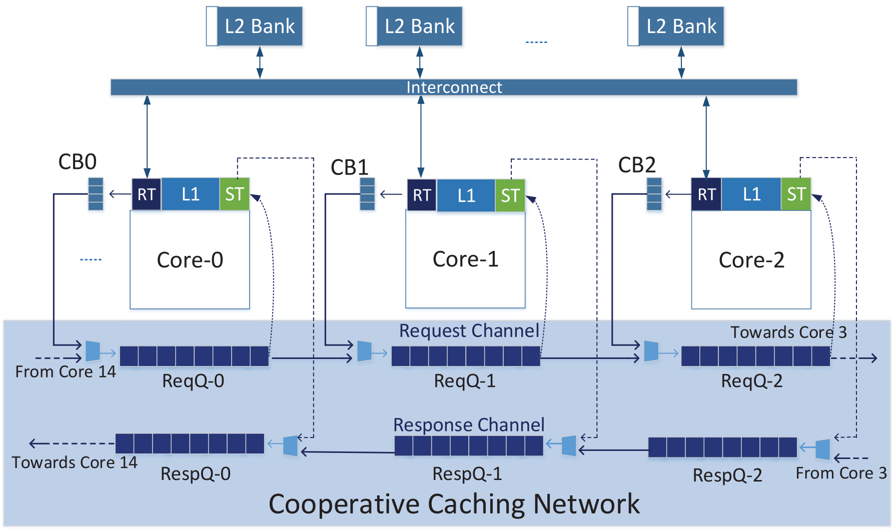


Fig. 5. Cooperative caching network.

accept a request. A new miss request, upon entering the local ReqQ, travels around the request channel by hopping on other ReqQs and probing the different L1 caches on its way. If a remote copy is found on one of the nodes, the response from the hit node is sent back to the requesting core in a similar way by hopping in the reverse direction via the RespQs at each core. Note that a remote L1 copy is considered for sharing only if it is not pending on a cache fill for the requested data at the time of lookup. In other words, pending hits due to outstanding miss requests are not considered for sharing in the CCN.

Specifically, upon incurring a load miss for global data, instead of sending the miss directly to L2, each core pushes the miss tag information into its CB along with the Core-ID, where the request waits until the corresponding ReqQ is ready to accept a new request. At every cycle, valid entries at the head of the ReqQ look up the corresponding L1 cache (if it is not the home core of that request) before hopping onto the next ReqQ. If the request travels back to the requesting core without a reuse copy, it is finally sent to L2. However, if a sharer is found, the sharing core enqueues the response to its RespQ. The response travels back to the requesting core, thereby avoiding an L2 access. If the ReqQs become full due to congestion, the CB eventually stops accepting new miss requests. In such a scenario, the L1 load misses go *directly to L2* until the CCN can start accepting new requests again.

4.2.1. Prioritization Policy for Queues. Each queue in the CCN has a corresponding input multiplexer to select one of the entries out of the two possible input sources. In the request channel, an ReqQ can either accept a new miss request from the home core via the CB or a forwarded request from a preceding ReqQ. In our proposal, we prioritize an older request (from the ReqQ) over a new one (from the CB). This helps in preventing oversubscription of the CCN to new L1 misses by allowing the previously accepted requests to pass through and therefore minimize the round-trip overhead ($\delta_{overhead}$) in the CCN for subscribed requests. Repeated unsuccessful attempts to inject a new request in the CCN due to the preceding prioritization thus causes the CB to become full and hence deflects the L1 misses directly to L2, allowing the CCN to recover from congestion.

In RespQs, however, we prioritize a new cache response (from the core) over an older response (from the RespQ). This is because RespQ latencies do not contribute to $\delta_{overhead}$ but contribute to the reuse latencies l_{reuse}, which has comparatively more relaxed requirements (shown in Figure 4). More importantly, if the response of a new remote hit is not accepted by the RespQ, the tag entry at the head of the corresponding ReqQ that caused the hit is not popped, potentially stalling the entire request network and increasing the $\delta_{overhead}$ in the request channel.

4.2.2. CCN Memory Consistency. The CCN mechanism conforms to the existing memory consistency model supported by Fermi. CUDA provides two types of load instructions [NVIDIA 2016]: a normal load cached at L1 (*ld.ca*) and a direct load to L2 (*ld.cg*), bypassing L1. Due to the *write-through, no-write-allocate* policy of the L1 cache, a write causes the matching cache line in L1 to be invalidated, thereby causing the most recent value to reside in L2. However, due to a weak memory model [Alglave et al. 2015; NVIDIA 2014] and absence of coherence in GPUs, an *ld.ca* accessing L1 on a different core can return a stale value. Litmus tests in Alglave et al. [2015] have also shown that due to weak consistency, an *ld.ca* load may return a stale value on the same core as well, even if preceded by an *ld.cg* to the same address (CoRR). The CCN adopts similar weak memory ordering semantics for *ld.ca* loads; indeed, an L1 miss can return a stale value by snooping other cores via the CCN instead of reading the L2 that may have the latest value. However, since a baseline GPU guarantees reading the most recent value for *ld.cg* loads, the *CCN does not intercept such loads and hence does not further weaken the memory model*. In other words, when a programmer uses *ld.cg* loads to bypass L1, the current memory model ensures that the most recently written value is returned—a correctness guarantee also provided by the CCN.

4.3. Shadow Tags

Since each L1 now services additional tag lookups for CCN requests, such remote lookups could affect the performance of local cache accesses. To eliminate the interference of remote lookups on local requests, we duplicate the tags of the L1 data cache in a separate set of shadow tags adjacent to each L1. The shadow tags always contain an identical copy of the L1 tags, which is achieved by always writing tag updates to both sets of tags simultaneously. As a result, concurrent reads at independent addresses can then take place to L1 tags and shadow tags from the local core and remote lookups, respectively. Therefore, the shadow tags dissociate the performance of each local cache from interference of CCN traffic. However, if a shadow tag lookup succeeds, then the remote access makes a regular L1 access to retrieve the data it needs. This steals a cycle from the L1 data cache, which is taken into account in our performance model.

Overhead. For the largest L1 data cache configuration of 48KB with 128-byte line size, we require 24 upper address bits per tag, assuming 40-bit physical addresses [NVIDIA 2009], plus one valid bit. Considering that the L1 data cache is four-way set associative, the shadow tags are arranged as 96 sets of four 25-bit tags in 96×100 single-ported tag memory.

Way 0		Way 1		Way 2		Way 3	
V_0	Tag_0[39:16]	V_1	Tag_1[39:16]	V_2	Tag_2[39:16]	V_3	Tag_3[39:16]

Therefore, the net storage overhead of the shadow tags is 1,200 bytes per SM and a total of 17.5KB for a 15-core GPU that we consider in our study. However, each remote access has to be checked in multiple shadow tags; these shadow tag memories are small and can be constructed from low-leakage high-density bit cells without impacting the overall cycle time of the ring interconnect.

4.4. Request Throttler

In order to prevent those cores that do not exhibit any intercore reuse from congesting the CCN, we introduce a request throttler (RT) at each core. The purpose of the RT is to throttle the remote lookup requests directly to the L2 cache when prior routing of misses to the CCN proves to be below a threshold level of effectiveness. To do this, each RT periodically samples the CCN performance parameters and at the end of the sampling period computes the success rate in routing its load misses to the CCN during the sampling interval. The success rate is determined by the ratio of hits in the CCN to the total number of requests injected in the CCN by the corresponding L1 cache. If the success rate is below the threshold, the L1 cache bypasses the CCN until the next sampling interval and performs the load miss by sending the request directly to the L2 cache. However, the shadow tags of the throttled cores still participate in the lookup for other requests in the CCN.

To illustrate the working of the RT further, we define the sampling interval as t_S and the periodicity of sampling as t_P, where $t_S \ll t_P$. Therefore, the entire period of execution is logically divided into multiple epochs of duration t_P. We also define H_{min} as the minimum hit rate required in the CCN to derive utility out of cooperation.

At the beginning of an epoch of interval t_P, each core begins by routing the load misses to the CCN for a fixed sampling duration of t_S. During the t_S interval, the RT collects the statistics about the number of requests injected in the CCN (N_{total}) and the number of hits observed for its requests (N_{hits}). At the end of the sampling duration, the RT computes the hit rate (h_{reuse}) in the CCN (i.e., $h_{reuse} = N_{hits}/N_{total}$). If $h_{reuse} >= H_{min}$, the RT continues to inject requests in the CCN for the remaining duration of $(t_P - t_S)$ in the current epoch. On the other hand, if $h_{reuse} < H_{min}$, the RT disables the routing of requests to the CCN for the remaining duration of the epoch. After the current epoch ends, N_{hits} and N_{total} are reset and the RT repeats the entire process again for the new epoch. Therefore, with the help of the RT, we improve the average success rate of sending a load miss to the CCN by preventing those cores from cooperating that are not working on potentially reusable data during specific epochs of execution.

4.5. Working Example

In this section, we further illustrate the workings of the CCN. Figure 6 shows the flow of requests within the CCN. In this example, Core-0 incurs a load miss for a global data in its private L1 cache. In the baseline architecture, this L1 miss would be directly routed to the L2 cache. However, with our scheme, the miss request can either go to the CCN or to the L2 cache. The RT takes this decision for that particular core on the basis of the statistics collected over the most recent sampling interval, t_S. In this example, we assume that h_{reuse} for Core-0 and Core-1 suggests healthy reuse ($>= H_{min}$), and therefore these cores continue to use the CCN. However, Core-2 observes a low reuse in the recent t_S interval, thereby routing all requests directly to the L2 cache for the current epoch.

Thus, to service the miss at Core-0 via the CCN, the tag and Core-ID of the load request are pushed ❶ onto the corresponding CB, CB0. Based on the input prioritization policy for ReqQ, the new tag waits in CB0 until it acquires the priority and is accepted ❷ by ReqQ-0. Upon reaching the head of ReqQ-0, the miss request does not perform a lookup in the shadow tag of Core-0, as it is the home core of the miss request and therefore is directly passed to ReqQ-1 ❸. Upon reaching the head of ReqQ-1, it performs a lookup ❹ in the shadow tag of Core-1. Assuming that it is a hit in Core-1, the shadow tag receives the cache line from the corresponding L1 cache and enqueues the response ❺ in RespQ-1, given that the RespQ-1 is not full. On the other hand, if the RespQ-1 is full, the response is stalled, thereby preventing the tag at the head

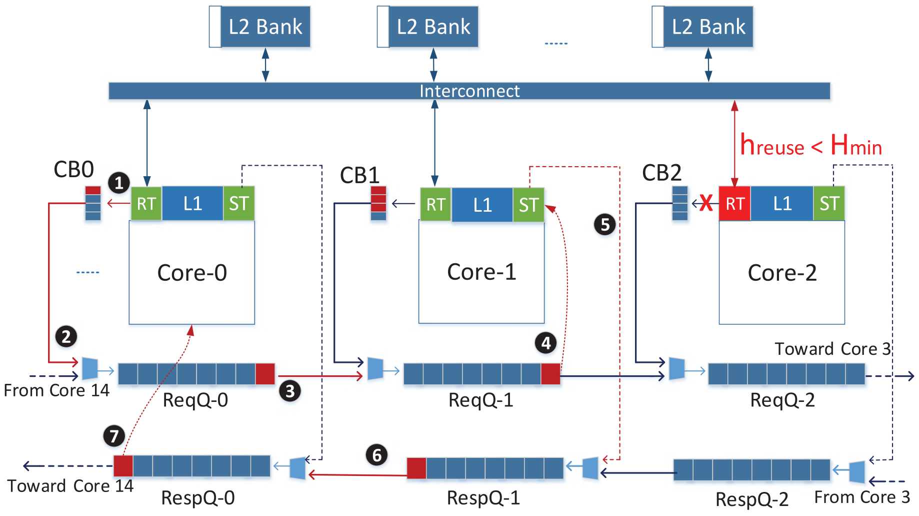


Fig. 6. Workings of the CCN-RT.

of ReqQ-1 from getting popped. Once the response reaches the head of the queue at RespQ-1 and acquires priority to enter the next queue, it is pushed into RespQ-0 ❻. Since Core-0 is the home core of the response, the new entry to RespQ-0 is bypassed to the head of the queue and the response is serviced ❼ to the L1 cache of Core-0, hence completing the request-response cycle.

5. EVALUATION

In this section, we discuss the implementation of our proposed architecture and demonstrate the results.

5.1. Implementation

For the purpose of this study, we implement and evaluate two flavors of our proposed architecture: the CCN-B and CCN-RT. The CCN-B is our baseline CCN architecture that includes a pair of queues and shadow tags at every node of the network, whereas in the CCN-RT we add the request throttling feature to the baseline CCN architecture. Table II(b) summarizes the design parameters for the CCN-B and CCN-RT.

In our implementation, we choose the sampling interval and the periodicity of sampling as 1 million and 10 million instructions, respectively. This is based on the observation that most benchmarks show a single-phase sharing across the entire application. Hence, it allows us to sample for a short duration to get a fairly accurate hint for a large duration that follows the sampling interval. Further, on the basis of our sensitivity studies, we select the threshold hit rate (H_{min}) as 5% (i.e., the minimum number of hits required to derive benefit from cooperative caching). We also observe in our experiments that small eight-entry ReqQs and RespQs provide the most optimal results.

Furthermore, the request and response channels in the CCN are configured to flow in opposite directions. This is because our experiments show that in such a case, servicing reuse requests takes an average of 10 hops compared to a fixed 15 hops when both channels propagate in the same direction.

Table II. Configuration Parameters for GPGPU-Sim and the CCN

Parameter	Value
(a) GPGPU-Sim	
Core	15 SMs, greedy-then-oldest (GTO) scheduler
Clock frequency	Core @ 1.4GHz; Interconnect/L2 @ 700MHz
Threads per SM	1,536
Warp width	32
SIMD lane width	32
Registers per SM	32,768
Shared memory	48KB
L1 data cache	16KB, 128-byte line, 4-way, LRU, write-through, no-write-allocate
L2 cache	768KB, 128-byte line, 8-way, LRU, write-back, 12 banks
DRAM	GDDR5 DRAM, 6 channel, 64-bits per channel, 924MHz
(b) CCN	
CCN buffer	8-entry, 30 bits per entry (26-bit tag + 4-bit Core-ID)
Request queue	8-entry, 30 bits per entry
Response queue	8-entry, ~128 bytes per entry (cache line + Core-ID)
CCN ring	4-byte request channel; 32-byte response channel; 1.4GHz
Shadow tag	1200 byte size (modeled on 48KB L1 data cache)
t_S	1 million instructions
t_P	10 million instructions
H_{min}	0.05 (5% hits)

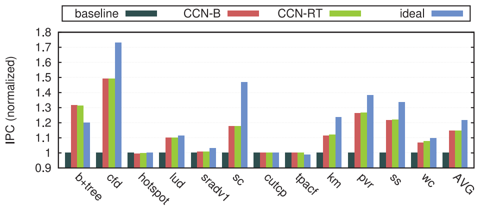
5.2. Experimental Setup

We model the CCN on GPGPU-Sim (version 3.2.2) [Bakhoda et al. 2009] to simulate a Fermi-like GPU with the configuration parameters listed in Table II(a). For energy and area simulations, we use GPUWattch [Leng et al. 2013], a McPAT-based power model integrated in GPGPU-Sim. All CCN transactions have been modeled at *cycle-by-cycle accuracy* in the simulator, which includes queuing delays in the request and response channels, CCN congestion, and L1 cycle stealing by shadow tag accesses. We run all benchmarks either to completion or until they execute 16 billion instructions, whichever comes first.

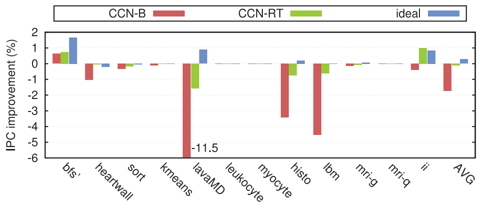
5.3. Results

We begin by evaluating the overall performance improvement with our proposed schemes for benchmarks that exhibit intercore reuse (μRC $>$ 10). We also show the neutrality of our scheme for benchmarks with little or no reuse (μRC $<$ 3). Later we assess the finer parameters for the former set of benchmarks, as applications with intercore reuse are the primary motivation for this study. We do not show the benchmarks between this range, as results of the preceding categories are good indicators of the trend in the rest of the benchmarks. We also compare the results of our proposed schemes (i.e., the CCN-B and CCN-RT) against an *ideal* cooperative caching configuration that services all of the remote hits with zero latency, without incurring any overheads of cooperative caching.

5.3.1. Performance. In Figure 7(a), we show the speedup with the CCN-B and CCN-RT for applications that exhibit reuse. Over the baseline configuration, we observe an average improvement of 14.5% with the CCN-B and 14.7% with the CCN-RT. Memory-bound applications such as *cfd*, *ss*, and *pvr* show higher speedup compared to non–memory-bound applications, as they are more sensitive to bandwidth bottlenecks. *b+tree* shows a higher improvement than the ideal case due to the timing variations in scheduling warps. Such an aberration is also caused due to a higher number of



(a) Speedup for applications with μRC > 10



(b) Percentage improvement in IPC for applications with μRC < 3

Fig. 7. Performance variation with cooperative caching.

coalesced hits on cache lines allocated for ongoing remote L1 accesses, which does not occur in the ideal scenario due to zero cycle latency for remote L1 accesses.

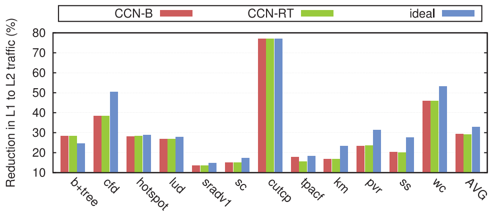
We also assess the impact of cooperative caching on applications that show little or no reuse. For such applications, cooperative caching adds an extra round-trip overhead of going through the CCN, because due to a low μRC, most requests end up going to the L2 cache after an unsuccessful traversal in the CCN. In such cases, the RT helps in preventing the L1 misses from incurring the CCN overhead when there is little or no reuse. In Figure 7(b), we show that with the CCN-B, we see a degradation of up to 11.5% and an average degradation of 1.7% compared to the baseline GPU. However, with the CCN-RT, the maximum degradation reduces to 1.5% with an overall average of 0.1%.

5.3.2. L2 Cache Bandwidth Demand. In Figure 8(a), we demonstrate the effectiveness of our proposed technique in mitigating the L2 cache bandwidth bottleneck. On average, the CCN-RT reduces the traffic to the L2 cache by 29% compared to the baseline GPU. It is in close proximity to the ideal-case average of 33%, indicating that most of the reuse hits on remote L1 caches are captured by the proposed architecture. Virtually no difference between the CCN-B and CCN-RT demonstrates that although throttling diverts most of the non-productive traffic directly to the L2 cache, it does not reduce the number of potential hits in the CCN. If it would divert the useful reuse requests to the L2 cache bypassing the CCN, then we would see a lesser reduction in L2 traffic with the CCN-RT compared to the CCN-B.

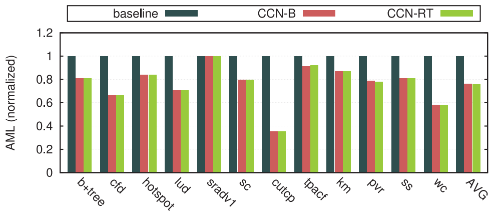
5.3.3. Average Memory Latency. In Figure 8(b), we see an average reduction of 24% in AML with our proposed CCN-RT architecture for applications that show reuse. We observe that *cutcp* shows the maximum reduction of 65% in AML due to a high μRC of 78%. However, it does not translate into performance gain due to its non–memory-bound nature.

5.3.4. Core Stall Cycles. We observed in the preceding results that the performance improved by mitigating the bandwidth problem (indicated by L2 traffic) and by servicing the misses in less time (indicated by AML). This is because cores now spend less time waiting for memory. Therefore, we assess the impact of our proposal on the total number of cycles for which the cores are stalled. In Figure 8(c), we observe a significant reduction in core stall cycles for memory-bound applications such as *cfd* and *sc*, whereas no degradation is seen for non–memory-bound applications like *cutcp* and *tpacf*. On average, we reduce the core stall cycles by 26%, which is in close proximity to the ideal reduction of 28%.

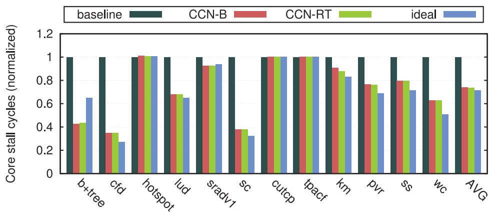
5.3.5. Off-Chip Memory Traffic. To dissociate the effects of L2 and off-chip bandwidths on the overall performance gain, we analyze the change in off-chip memory traffic. As shown in Figure 8(d), we see that for most applications, there is *no visible difference* in



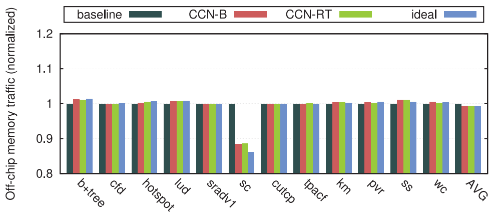
(a) Percentage reduction in L1 to L2 traffic



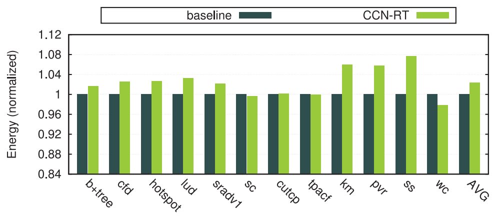
(b) Normalized AML



(c) Normalized core stall cycles



(d) Normalized off-chip memory traffic



(e) Energy dissipation with the CCN

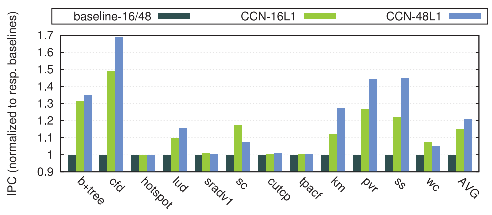
Fig. 8. Experimental results demonstrating the effect of cooperative caching.

the traffic to off-chip memory, indicating that the entire performance improvement can be attributed to the mitigation of the bandwidth bottleneck between private L1s and the shared L2. Therefore, it can be inferred for most benchmarks that in the baseline architecture without the CCN, the reuse requests mostly hit in the L2 cache, thereby only burdening the L2 cache bandwidth with duplicate requests. However, in *sc*, we notice a reduction in DRAM traffic by 12% with the CCN-RT. This indicates that for *sc*, a significant portion of reuse requests to L2 also misses in the L2 cache, adding to the DRAM traffic. As a result, upon removing the reuse requests to the L2 cache with the help of the CCN in *sc*, not only the traffic to the L2 cache is reduced but also the traffic to DRAM is reduced. Therefore, the performance benefit in *sc* with the CCN-RT can be attributed not only to the mitigation of the L2 bandwidth bottleneck but also to the mitigation of the DRAM bandwidth bottleneck.

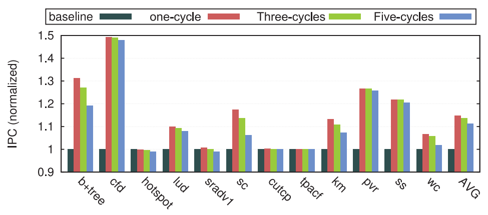
5.3.6. Summary. In the preceding results, we observed that for applications exhibiting reuse, we are able to reduce the traffic to the L2 cache by 29% while also reducing the average memory latency by 24%. As a consequence of the preceding improvements, we reduce the average core stall cycles by 26%, which translates into an average performance improvement of 14.7%.

5.4. Hardware Costs

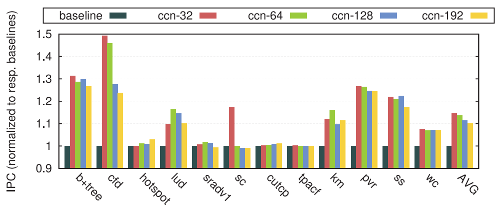
5.4.1. Area. We use GPUWattch [Leng et al. 2013] to estimate the area of our proposed architecture. We use the existing components in GPUWattch to model the CCN



(a) Speedup with varying L1 cache sizes



(b) Speedup with link latencies of one, three, and five cycles



(c) Speedup with varying width of SIMD lanes

Fig. 9. Sensitivity analysis.

components after appropriate scaling wherever necessary. The CCN adds an area overhead of 4.38mm^2 for the ring interconnect and the shadow tags (corresponding to the largest L1 data cache configuration) at 40nm technology. Other storage units, such as CBs and ReqQs and RespQs, add another 4.82mm^2. This amounts to an overall increase in die area by 1.3% with respect to a baseline processor architecture area of 700mm^2.

5.4.2. Energy. With the CCN, cores are stalled for fewer cycles, thereby *reducing the leakage power*. In addition, fewer packets require routing at the energy-inefficient crossbar routers. In addition, lower traffic to L2 leads to lower energy consumption by the NoC. However, high shadow tag lookups for remote cache accesses normalize other energy gains of the CCN, resulting in an average energy overhead of 2.5% (Figure 8(e)).

5.5. Sensitivity Analysis

5.5.1. L1 Cache Size. As Fermi offers configurable L1 cache sizes of 16KB and 48KB, we analyze the sensitivity of our proposal to the L1 cache size. As shown in Figure 9(a), on increasing the L1 cache size to 48KB, we observe an average IPC gain of 20.6% with the CCN compared to 14.7% with the CCN on 16KB L1 (over their respective baselines). This is due to the following reason. Although increasing the L1 cache size reduces the number of capacity/conflict misses, thereby reducing the opportunities to find remote L1 hits in the CCN, we observe that a larger L1 significantly increases the likelihood of finding a remote L1 sharer for a *compulsory miss*. Therefore, due to significant increase in utility of the CCN for compulsory misses on increasing the L1 size (which dominates the decrease in utility of the CCN due to lower conflict/capacity misses), we observe a higher improvement in performance with larger L1s.

5.5.2. Link Latency and Frequency. In this study, we analyze the performance impact of interconnect latencies for every hop on the CCN ring. This is done by varying the core-to-core transfer latency from 1 to 5 cycles (i.e., 15 to 75 cycles for the entire ring). For a 700mm^2 chip, each hop is approximately 3.5mm of on-chip distance, and therefore 1 to 5 cycles at 1.4GHz is a reasonable window to complete the transfer [Beckmann and Wood 2004]. It is worth noting that varying the CCN link latency also captures the effect of running the CCN ring at a fraction of core frequency. Therefore, this study shows the performance variation on using the CCN ring at up to one-fifth of the core frequency (280MHz).

In Figure 9(b), we see that for most applications, the IPC gain is fairly resilient to increasing link latencies (or decreasing ring frequencies). For instance, *cfd* shows a marginal reduction of 1% when the latency increases from one to five cycles. A minority of applications show visible reductions in the gain as link latency increases. For example, the IPC gain of *b+tree* drops from 31% to 19%, although it still maintains a modest overall improvement in performance. On average, we see IPC improvements drop from 14.7% to 13.6% as latency is increased from one to three cycles, settling further at 11.2% when the link latency is increased to five cycles. These results indicate that our proposed scheme is fairly robust to increasing inefficiencies in the ring interconnect (as well as increasing distance between the neighboring cores).

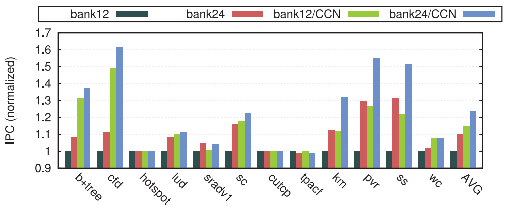
5.5.3. SIMD Lane Width. Each core in NVIDIA's Fermi GPU consists of a 32-lane SIMD unit, with each lane capable of executing one floating point or arithmetic instruction per clock. In this study, we analyze the utility of the CCN on increasing the SIMD lane width. In Figure 9(c), we plot the performance gain with the CCN-RT on baseline configuration with varying SIMD lane widths of 32 (ccn-32), 64 (ccn-64), 128 (ccn-128), and 192 (ccn-192), each normalized to their respective baselines. On average, the performance gain drops modestly from 14.7% to 13.6% on increasing the SIMD lane width from 32 to 64, settling further at 11.4% and 10.2% with SIMD lane widths of 128 and 192, respectively. Although the minor reduction in CCN gain is due to the increased latency tolerance provided by additional SIMD lanes, cooperative caching continues to provide considerable benefits for memory-intensive applications. This is due to the fact that by increasing the SIMD lanes or the compute capability of the cores, only compute-bound applications are expected to show significant speedups and a higher overlap of memory latencies with computation. In contrast, memory-intensive applications lack independent compute instructions and continue to be bottlenecked by memory resources. Therefore, additional compute resources for memory-intensive applications provides only limited additional latency tolerance to the cores due to which cooperative caching continues to be useful in reducing memory latencies that lie in the critical path. However, some benchmarks, such as *lud* and *km*, also show momentary improvement in performance gain with the CCN on increasing the SIMD lane width. We observe that this is because with wider SIMD lanes, a higher number of threads arrive at the memory instructions per cycle, issuing a higher number of requests that may exhibit reuse, thereby amplifying the utility of the CCN in reducing the traffic that could lead to even higher congestion.

5.6. Discussion

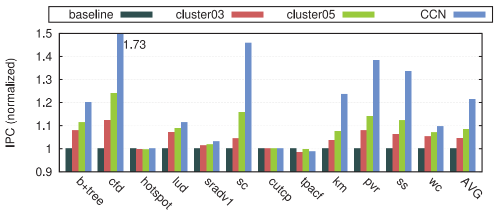
In the future, scalability of the CCN can be addressed by a hierarchical implementation of the proposed ring network [Holliday and Stumm 1994; Ravindran and Stumm 1997]. A sub-CCN ring that contains the requesting core can inquire other sub-CCN rings in *parallel*, thereby decomposing the serial latency of traversing the high number of cores into concurrent transactions to multiple rings. In addition, as coherent caches in GPUs are imminent in future architectures [Martin et al. 2012; Power et al. 2013; Singh et al. 2013], intercore communication via the CCN can also act as a substrate for implementing cache coherence.

6. COMPARATIVE STUDY

In this section, we perform a quantitative and qualitative comparison of the CCN with alternative techniques that address the bandwidth bottleneck in GPUs.



(a) Speedup with 2× L2 banks and the CCN



(b) Ideal speedup with L1 cache clusters

Fig. 10. Comparative study.

6.1. Increasing L2 Banks

An alternative technique to increase the bandwidth to L2 is to increase the number of L2 banks. However, increasing the banks only reduces the congestion in the access path to L2, whereas the CCN, in addition to reducing pressure on L2 bandwidth, provides a significantly faster response for a fraction of miss requests. In our experiments, we observe that the CCN services the reuse requests in 42 cycles (l_{reuse}) on average, for 29% misses (h_{reuse}) that hit in the CCN. For the remaining L2 accesses, the CCN adds a round-trip overhead of 54 cycles ($\delta_{overhead}$). It also reduces the congestion overhead to L2 by 78 cycles (δ_{cong}). Considering that the average L2 access latency without the CCN is 300 cycles (l_O) and substituting the preceding values in Equation (1), the average L2 access latency with the CCN is computed to be 208 cycles (Equation (3)).

$$l_{C(CCN)} = (300 - 78 + 54) \times 0.71 + (42) \times 0.29 = 208 \tag{3}$$

$$l_{C(2\times)} = (300 - 80 + 0) \times 1.0 = 220 \tag{4}$$

$$l_{C(CCN/2\times)} = (300 - 117 + 54) \times 0.71 + (42) \times 0.29 = 180 \tag{5}$$

However, increasing the L2 banks only reduces δ_{cong} (although marginally more than the CCN for some benchmarks) but requires all accesses to go through the L2 access latency, albeit via reduced congestion. Upon substituting corresponding values in Equation (1), the reduced L2 access latency is computed to be 220 cycles (Equation (4)). Therefore, in Figure 10(a), we observe an average IPC improvement of 10.2% upon a 2× increase in L2 banks from 12 to 24. In contrast, the CCN implemented with a 12-bank L2 configuration shows a higher improvement of 14.7% (with *cfd* performing 34% better with the CCN than with 2× L2 banks).

Importantly, the CCN is *partly orthogonal* to increasing the banks at L2. This is because, in addition to reducing the δ_{cong} further, the CCN adds the benefit of faster access to reuse requests. The average L2 access latency in Equation (1) for a CCN architecture on a 24 L2 bank configuration is computed to be 180 cycles (Equation (5)). In Figure 10(a), our experiments show an average performance improvement of 23.5% with both techniques combined.

With respect to the cost, increasing the L2 banks would require a higher number of ports in the crossbar. As the area of a crossbar increases polynomially on increasing the ports, the area overhead will be significant. Energy demands also increase significantly, as each router is more complex and needs to arbitrate on a higher number of nodes. In contrast, the CCN only requires simple multiplexers at each router and scales well with respect to area and energy overheads. Alternatively, increasing the L2 datapath width to provide more L2 bandwidth would also be area intensive, as it entails increasing the area of 15 × 12 core-to-L2 connections in the crossbar, making the crossbar much bulkier. However, the CCN only requires 15 core-to-core connections.

As core-to-L2 connections are typically longer (in addition to being higher) than core-to-core connections in the CCN, there is a higher overhead in scaling the former.

6.2. Sharing Tracker

Tarjan and Skadron [2010] proposed a scheme to exploit reuse within the private caches by using a sharing tracker, a decomposed version of the coherence directory. It aims to reduce the *off-chip memory bandwidth* demand by diverting DRAM accesses to private caches that contain a shared copy.

Although we adopt this intuition to reuse shared copies in private caches, our aim is to reduce the bandwidth demand to the *shared cache* (and not the DRAM as in Tarjan and Skadron [2010]). This is because in the current scenario with recent GPU architectures, exploiting *reuse does not reduce off-chip memory traffic* (as shown in Figure 8(d)), and hence a common directory in the shared cache is not expected to show any benefit since there are not many off-chip memory accesses that it can avoid. In fact, since accessing and maintaining the sharing tracker in the L2 cache adds to the bandwidth demand to L2 without relieving pressure on off-chip bandwidth, it will only exacerbate the problem by increasing the L2 access latencies and thereby worsen the IPC with respect to baseline. For those architectures where off-chip memory traffic is also reduced by exploiting sharing within private caches, the CCN achieves the same but also reduces the traffic to L2 (which we have shown to be critical to performance), therefore providing a significant advantage over a directory approach.

6.3. Clustered Sharing

Keshtegar et al. [2015] proposed an architecture to enable restricted sharing within core clusters. However, in Figure 3, we showed that whereas some benchmarks show higher reuse with neighboring cores, others show a uniform sharing with all cores. In Figure 10(b), we show the ideal performance improvement (with no sharing overheads) obtained by sharing within cache clusters and compare it to an ideal case of the CCN (sharing among all cores). We observe an average IPC gain of 4% and 8% with ideal clusters of three and five L1s, respectively, compared to an average IPC gain of 21% with the ideal CCN. This suggests that for most benchmarks, upon restricting the sharing within cache clusters, SMs lose out on most of the reuse data.

Moreover, the cluster-based proposal by Keshtegar et al. [2015] employs a mesh-type network within a cluster and scales polynomially with the number of cores. Therefore, we expect the area overhead of clusters to exceed the area of ring-based connections in the CCN, which scales linearly with the number of cores. Furthermore, in current GPUs, SMs are placed linearly around the central L2 cache [NVIDIA 2009, 2012], and therefore clusters would require longer wires to connect the far ends of a cluster as compared to only near-neighbor connections in the CCN.

6.4. Summary

In this section, we have shown that the CCN fares well compared to alternative techniques. The CCN performs better than simply increasing the number of L2 banks while also being partly orthogonal to the latter technique. The sharing tracker is expected to show a negative performance gain with the baseline architecture, and restricted sharing within cache clusters significantly reduces intercore reuse.

7. RELATED WORK

Although sharing across L1 caches is a common occurrence in multiprocessors, as emphasized by the prevalent use of a sophisticated coherence infrastructure, we derive significant benefits by exploiting L1 sharing for GPGPU workloads, a property atypical in GPUs. Additionally, in contrast to earlier works [Yazdanbakhsh et al. 2016; Jog et al.

2016] where only the off-chip memory bandwidth is considered critical to performance, we identify the criticality of mitigating congestion in the on-chip cache hierarchy between the L1 and L2 cache. In the following sections, we further discuss several prior works related to the ideas presented in the CCN and cite their key differences.

7.1. Cooperative Caching in CMPs

In the realm of CMPs, Chang and Sohi [2006, 2007] proposed cooperative caching by adapting the coherence infrastructure. Subsequently, Herrero et al. [2008] proposed a scalable distributed cooperative caching scheme by redesigning the coherence engine to provide distributed directories. Both schemes aim to provide aggressive latency and capacity benefits for on-chip caches in CMPs. However, since GPUs are relatively more tolerant to latencies, in this article we address the problem of bandwidth in GPUs. In addition, a directory-based scheme is not directly portable to GPUs due to the lack of coherence infrastructure, and therefore our solution proposes an independent lightweight network.

7.2. Ring Network

Ring topologies have been used extensively in commercial multiprocessors to provide low-cost intercore communication. Larrabee [Seiler et al. 2008] employs a bidirectional ring network to allow on-chip communication between *latency-sensitive* CPU cores, coherent L2 caches, and other blocks, with each link being 64 bytes wide (net width of 128 bytes). The Xeon Phi [Chrysos 2012] also contains bidirectional rings, with each ring composed of three independent rings: a 64-byte data block ring for data transactions, an address/command ring, and an acknowledgement ring for coherence and flow control messages (net width $>$128 bytes). In contrast, the CCN enables bidirectional communication between *latency-tolerant* GPU cores by connecting the incoherent L1 caches in a ring. Due to relaxed latency constraints in the CCN compared to prior ring interconnects in multiprocessors, the bus width for intercore transfers is smaller, with each link being 8 bytes and 32 bytes wide, respectively (net width of 40 bytes). Therefore, our proposal exploits the latency-tolerance property of multithreaded cores to provide low-cost intercore communication through a lightweight ring network.

Furthermore, Campanoni et al. [2014] proposed a ring cache for HELIX-RC that acts as a distributed first-level cache, preceding the private L1 cache. Each ring node has a cache array to cache shared data and satisfies the loads and stores received from its attached core. To avoid coherence complications, memory addresses are permanently mapped to the nodes of the ring cache. In contrast, each node in the CCN ring network includes a shadow tag array, needed only for lookups and not for storage of shared data. Subsequent loads to the shared data via the CCN are performed directly in the corresponding L1 caches, as there is no separate data array for the ring nodes. Therefore, the nodes in the CCN ring network are lighter than nodes in the ring cache proposed in HELIX-RC.

7.3. Shadow Tags

Prior proposals such as Piranha [Barroso et al. 2000] and Niagara [Kongetira et al. 2005] have replicated tag structures of the private L1 caches at the shared L2 cache. Such duplicate L1 tags stored centrally in the L2 cache are typically used to construct partial sharing information, thereby reducing indirections to the coherence engine. Duplicate tag structures are also used to reduce redundant write-back traffic to the L2 cache from multiple L1s that cache the same shared data. However, in the CCN, we replicate the tags adjacent to the corresponding L1 caches and do not complicate the L2 cache control. It is used only to prevent deterioration of L1 cache performance due to remote lookups. Moreover, tag updates to shadow tags incur minimum communication overhead in the CCN due to physical proximity of L1 caches and shadow tags.

7.4. Cache Management

In the field of GPUs, prior proposals such as the sharing tracker [Tarjan and Skadron 2010] and cluster-based schemes [Keshtegar et al. 2015] (discussed previously in Section 6) exploit reuse within GPU cores via central directory and clustered caches, respectively.

Several other schemes have been proposed for GPUs to improve the effective on-chip cache capacity, reduce cache thrashing, and improve locality in L1 and L2 caches. Rhu et al. [2013] proposed a locality-aware memory hierarchy that adaptively adjusts the memory access granularity to prevent overfetching, providing better off-chip bandwidth utilization. Furthermore, with regard to adaptive memory access granularity, Li et al. [2016] proposed a tag-split cache to enable fine storage granularity to improve cache utilization while keeping a coarse access granularity to avoid excessive cache requests. Tarjan et al. [2009] proposed a scheme to tolerate memory miss latencies for SIMD cores by masking out threads in a warp that are waiting on data and allowing other threads to continue execution, hence utilizing the idle execution slots. Rogers et al. [2012, 2013] proposed scheduling techniques that are conscious of the variations in the cache locality, thereby dynamically altering the scheduling policies to maximize inter-warp locality on the L1 data cache. Jia et al. [2012] presented a taxonomy for memory access locality and proposed a compile-time algorithm to selectively utilize the L1 caches. Narasiman et al. [2011] proposed large warp architecture and a two-level warp scheduling technique to make effective use of resources on GPU, whereas Jog et al. [2013] proposed a thread block–aware scheduling policy to improve the cache hit rates of the L1 cache. Choi et al. [2012] employed techniques such as write buffering and read bypassing to reduce DRAM traffic and improve the L2 cache utilization, thereby addressing the bandwidth constraint between shared memory and DRAM. There has also been work on cache management policies for heterogeneous CPU-GPU architectures. Yang et al. [2012] proposed a CPU-assisted prefetching scheme to improve GPU memory latencies by localizing the data in the LLC cache, whereas Lee and Kim [2012] proposed a TLP-aware cache management policy to effectively utilize the LLC for general-purpose workloads.

Broadly, the preceding cache management proposals focus on reducing the miss rate of independent caches by improving cache utilization. However, in the CCN, without reducing miss rate of independent L1s, we reduce the collective bandwidth demand of L1 on L2 by diverting some of the misses to remote L1s. Hence, the mentioned techniques that reduce the miss rate itself are orthogonal to our work. Given the severity of the memory bottleneck in GPUs (as indicated by the magnitude of PerfX in Table I), no technique alone solves the entire problem, and hence such orthogonal techniques can be used in conjunction with the CCN.

7.5. Cache Bypassing

To mitigate the severity of cache thrashing, several cache bypassing techniques have been proposed. In CPUs, Gaur et al. [2011] proposed a bypass policy to selectively fill the exclusive LLC with evicted cache blocks from the higher level. Further, Duong et al. [2012] proposed a policy to protect reusable cache lines from eviction with a dynamically computed protected distance and bypass the miss requests upon lack of unprotected cache lines in a set.

In GPUs, high multithreading and low on-chip cache capacity per thread present additional challenges due to severe cache thrashing. Chen et al. [2014] proposed a dynamic cache management policy that combines L1 cache bypassing and throttling.

In their proposed scheme, warp throttling prevents oversaturation of on-chip cache resources, whereas cache bypassing prevents cache contention, requiring a lower number of warps to be throttled in comparison to stand-alone warp throttling schemes. Li et al. [2015] proposed a locality-driven cache bypassing scheme that uses reuse frequency in a decoupled and extended tag memory to allow allocation in the data memory for only those cache lines that exhibit high reuse.

In summary, the cache bypassing schemes in the GPU improve cache utilization by preserving hot cache lines with high reuse in the the available on-chip caches and bypassing the streaming requests directly to the L2 cache. By preventing eviction of cache lines with high reuse, it helps in eliminating repeated reuse requests from the *same* L1 cache to the L2 cache. However, in our proposed technique, we eliminate the reuse requests from *different* L1 caches to the L2 cache. In other words, cache bypassing techniques reduce *intracore* reuse requests that access the L2 cache, whereas our proposed technique reduces *intercore* reuse requests that access the L2 cache. Therefore, we expect our proposal to be complementary to cache bypassing techniques, as both techniques help in reducing a mutually exclusive set of requests to the L2 cache.

8. CONCLUSION

In this article, we discuss the inefficiencies in the management of L1 caches in GPUs. We show that as a consequence of high L1 miss rates, high traffic to the L2 cache presents a bandwidth bottleneck between L1 and L2, resulting in high L2 access latencies. In memory-intensive applications, multithreading is unable to hide such high latencies, making it critical to performance.

We discover considerable potential for data reuse within the L1 caches. We exploit this opportunity to reduce the miss traffic to the L2 cache and thereby reduce the L2 cache bandwidth demand. Therefore, we present a CCN that services the L1 load misses cooperatively via a lightweight ring network. We show that GPUs can tolerate reuse latencies gracefully up to 80 cycles, and therefore a ring topology appears to be a cost-effective solution, as it allows us to trade off higher latencies for simplicity and short wires (i.e., lower power consumption and die-area cost). We also use shadow tag memory, adjacent to each L1 data cache, to decouple the local L1 cache performance from remote L1 cache tag lookups. For applications that do not exhibit any intercore reuse, we detect the lack of sharing at runtime and prevent the L1 miss requests from incurring the CCN overhead, sending them directly to the L2 cache. For applications that exhibit reuse, our technique improves the IPC by 14.7% but is neutral to applications that show little or no reuse. We likewise reduce the traffic to the L2 cache by 29% and reduce the AML by 24%. As a result, we reduce the total core stall cycles by 26%. Alongside the preceding improvements, the CCN presents an area and energy overhead of 1.3% and 2.5%, respectively. The CCN also compares favorably to alternative techniques that address the bandwidth issue.

REFERENCES

Manuel E. Acacio, José González, José M. García, and José Duato. 2002. Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*. IEEE, Los Alamitos, CA, 1–12. http://dl.acm.org/citation.cfm?id=762761.762762

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU concurrency: Weak behaviours and programming assumptions. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 577–591. DOI: http://dx.doi.org/10.1145/2694344.2694391

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE, Los Alamitos, CA, 163–174.

L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*. 282–293.

Bradford M. Beckmann and David A. Wood. 2004. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*. IEEE, Los Alamitos, CA, 319–330. DOI:http://dx.doi.org/10.1109/MICRO.2004.21

Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*. IEEE, Los Alamitos, CA, 217–228. http://dl.acm.org/citation.cfm?id=2665671.2665705

Jichuan Chang and Gurindar S. Sohi. 2006. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. IEEE, Los Alamitos, CA, 264–276. DOI:http://dx.doi.org/10.1109/ISCA.2006.17

Jichuan Chang and Gurindar S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*. ACM, New York, NY, 242–252. DOI:http://dx.doi.org/10.1145/1274971.1275005

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, Los Alamitos, CA, 44–54. DOI:http://dx.doi.org/10.1109/IISWC.2009.5306797

Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE, Los Alamitos, CA, 343–355. DOI:http://dx.doi.org/10.1109/MICRO.2014.11

Hyojin Choi, Jaewoo Ahn, and Wonyong Sung. 2012. Reducing off-chip memory traffic by selective cache management scheme in GPGPUs. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*. ACM, New York, NY, 110–119. DOI:http://dx.doi.org/10.1145/2159430.2159443

George Chrysos. 2012. *Intel Xeon Phi Coprocessor—The Architecture*. Technical Report. Intel Corporation. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2016. Characterizing memory bottlenecks in GPGPU workloads. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC'16)*.

Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE, Los Alamitos, CA, 389–400. DOI:http://dx.doi.org/10.1109/MICRO.2012.43

Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. 2011. Bypass and insertion algorithms for exclusive last-level caches. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 81–92. DOI:http://dx.doi.org/10.1145/2000064.2000075

Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, NY, 260–269. DOI:http://dx.doi.org/10.1145/1454115.1454152

Enric Herrero, José González, and Ramon Canal. 2008. Distributed cooperative caching. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, NY, 134–143. DOI:http://dx.doi.org/10.1145/1454115.1454136

Mark A. Holliday and Michael Stumm. 1994. Performance evaluation of hierarchical ring-based shared memory multiprocessors. *IEEE Transactions on Computers* 43, 1, 52–67.

Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 15–24. DOI:http://dx.doi.org/10.1145/2304576.2304582

Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the 18th International Conference* *on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 395–406. DOI:http://dx.doi.org/10.1145/2451116.2451158

Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2016. Exploiting core criticality for enhanced GPU performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. 351–363. DOI:http://dx.doi.org/10.1145/2896377.2901468

Stefanos Kaxiras and Georgios Keramidas. 2010. SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro* 30, 5, 54–65. DOI:http://dx.doi.org/10.1109/MM.2010.82

Mohammad Mahdi Keshtegar, Hajar Falahati, and Shaahin Hessabi. 2015. Cluster-based approach for improving graphics processing unit performance by inter streaming multiprocessors locality. *IET Computers and Digital Techniques* 9, 5, 275–282. http://digital-library.theiet.org/content/journals/10.1049/iet-cdt.2014.0092.

P. Kongetira, K. Aingaran, and K. Olukotun. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro* 25, 2, 21–29. DOI:http://dx.doi.org/10.1109/MM.2005.35

Alvin R. Lebeck and David A. Wood. 1995. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, New York, NY, 48–59. DOI:http://dx.doi.org/10.1145/223982.223995

Jaekyu Lee and Hyesoon Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA'12)*. IEEE, Los Alamitos, CA, 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2012.6168947

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 487–498. DOI:http://dx.doi.org/10.1145/2485922.2485964

Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS'15)*. ACM, New York, NY, 67–77. DOI:http://dx.doi.org/10.1145/2751205.2751237

Lingda Li, Ari B. Hayes, Shuaiwen Leon Song, and Eddy Z. Zhang. 2016. Tag-split cache for efficient GPGPU cache utilization. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*. ACM, New York, NY, Article No. 43. DOI:http://dx.doi.org/10.1145/2925426.2926253

Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why on-chip cache coherence is here to stay. *Communications of the ACM* 55, 7, 78–89. DOI:http://dx.doi.org/10.1145/2209249.2209269

Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 308–317. DOI:http://dx.doi.org/10.1145/2155620.2155656

NVIDIA. 2009. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Technical Report. NVIDIA Corporation. http://www.nvidia.co.uk/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

NVIDIA. 2012. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Technical Report. NVIDIA Corporation. http://www.nvidia.co.uk/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

NVIDIA. 2014. CUDA by Example—Errata Page. Retrieved October 27, 2016, from http://developer.nvidia.com/cuda-example-errata-page.

NVIDIA. 2016. Parallel Thread Execution ISA, Version 5.0. Retrieved October 27, 2016, from http://docs.nvidia.com/cuda/parallel-thread-execution.

Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 457–467. DOI:http://dx.doi.org/10.1145/2540708.2540747

Govindan Ravindran and Michael Stumm. 1997. A performance comparison of hierarchical ring- and mesh-connected multiprocessor networks. In *Proceedings of the 3rd IEEE Symposium on High Performance Computer Architecture (HPCA'97)*. IEEE, Los Alamitos, CA, 58. http://dl.acm.org/citation.cfm?id=548716.822685

Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 86–98. DOI:http://dx.doi.org/10.1145/2540708.2540717

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE, Los Alamitos, CA, 72–83. DOI:http://dx.doi.org/10.1109/MICRO.2012.16

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 99–110. DOI:http://dx.doi.org/10.1145/2540708.2540718

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, et al. 2008. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH'08)*. ACM, New York, NY, Article No. 18. DOI:http://dx.doi.org/10.1145/1399504.1360617

Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache coherence for GPU architectures. In *Proceedings of the 2013 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'13)*. 578–590. http://doi.ieeecomputersociety.org/10.1109/HPCA.2013.6522351

John A. Stratton, Christopher Rodrigrues, I.-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign, Urbana.

David Tarjan, Jiayuan Meng, and Kevin Skadron. 2009. Increasing memory miss tolerance for SIMD cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis (SC'09)*. ACM, New York, NY, Article No. 22. DOI:http://dx.doi.org/10.1145/1654059.1654082

David Tarjan and Kevin Skadron. 2010. The sharing tracker: Using ideas from cache coherence hardware to reduce off-chip memory traffic with non-coherent caches. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'10)*. IEEE, Los Alamitos, CA, 1–10. DOI:http://dx.doi.org/10.1109/SC.2010.54

Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. 2012. CPU-assisted GPGPU on fused CPU-GPU architectures. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*. IEEE, Los Alamitos, CA, 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2012.6168948

A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu, and T. C. Mowry. 2016. Mitigating the memory bottleneck with approximate load value prediction. *IEEE Design and Test* 33, 1, 32–42. DOI:http://dx.doi.org/10.1109/MDAT.2015.2504899

Received June 2016; revised August 2016; accepted September 2016