The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches

David Tarjan and Kevin Skadron
Department of Computer Science
University of Virginia, Charlottesville, VA 22904
{dtarjan,skadron}@cs.virginia.edu

***Abstract*—Graphics Processing Units (GPUs) have recently emerged as a new platform for high performance, general-purpose computing. Because current GPUs employ deep multithreading to hide latency, they only have small, per-core caches to capture reuse and eliminate unnecessary off-chip accesses. This paper shows that for general-purpose workloads, the ability to copy cache lines between private caches captures inter-core temporal locality and provides substantial reductions in off-chip bandwidth requirements. Unlike hardware cache coherence, a sharing tracker only needs to track cache lines in the private caches *imprecisely*, because it is only a performance hint. This simplifies the implementation and is so effective at capturing inter-core reuse that the L2 can be eliminated entirely. The sharing tracker is motivated by but not specific to the GPU and could be used in other manycore organizations.**

I. Introduction

Graphics processing units (GPUs) are optimized for high throughput on workloads with abundant parallelism. They were once fixed-function hardware for 3D rendering, but demand for increasing programmability for those applications has driven GPU architectures to become more general-purpose, manycore architectures (that just happen to be embedded within a system-on-chip including various 3D-specific accelerators). The introduction of hardware and software support for general-purpose programming languages on the GPU [1], [2], [3] has allowed GPUs to become a viable platform for throughput-oriented general-purpose computing. As with other throughput-oriented organizations, GPU cores have simple pipelines and are deeply multithreaded. Instead of using hardware for ILP discovery, area is used to maximize thread parallelism. In this data-intensive era, the principles of designing for data parallelism, throughput, and latency tolerance make GPUs a useful platform for drawing more general lessons for future, general-purpose, manycore processor architectures.

D. Tarjan is now with NVIDIA Research

GPU cores can share memory, but their caches do not support hardware cache coherence. In prior generations, only a subset of memory – that which was declared read-only by the graphics APIs – was cached. NVIDIA's new Fermi architecture [4] introduces a more general-purpose cache hierarchy, but still does not support hardware cache coherence. Values in GPU L1s must be kept coherent by software. This can be achieved by the programmer or the compiler (by flushing when necessary), and techniques for compiler-controlled software coherence have been studied for over 20 years (e.g., [5], [6]).

The non-coherent nature of the L1s prevents re-using values shared among cores, yet sharing is important because many algorithms require threads to operate on overlapping regions of data structures. Allowing cores to share the L1 contents of other cores reduces off-chip bandwidth requirements. Hardware coherence would of course capture reuse [7], [8], and support for hardware coherence has been studied for decades, but hardware coherence comes at the expense of considerable complexity and power dissipation in order to support the correct semantics.

For large scale systems with hundreds or thousands of processors with dozens or hundreds of cores each, coherence is of questionable value. In the case of GPUs, since graphics workloads typically do not benefit from coherence, it is unlikely that GPUs will add the required hardware in the near future. The Cell BE [9] is another major general-purpose architecture that eschewed hardware coherence, and various multicore organizations for embedded systems also forgo hardware coherence.

Capturing reuse without hardware-managed coherence would avoid unnecessary re-loading of shared, read-only data from memory without the overheads of hardware cache coherence. This requires some alternative means by which a core finds a cache line on a miss in its private L1 cache. One option is to have a last level cache (LLC) shared among all the cores. The drawback to such a design is that an LLC of sufficient size to support the request streams from a large number of wide SIMD cores will significantly reduce the chip area available for the high throughput cores. Inclusive caching (generally required with the point-to-point interconnect needed with large numbers of cores) further exacerbates the area overhead of an LLC.

Instead, we propose the *sharing tracker*, which simplifies the directory from cache coherence approaches for use with non-coherent cache hierarchies. *The key insight is that when software is responsible for coherence, the directory becomes a predictor and a mere performance hint.* Erroneous predictions may reduce performance but do not violate memory semantics. In contrast to full coherence directories, the sharing tracker is a low-cost structure that can be sized independently of the overall cache capacity it covers, and does not have the complexities associated with cache coherence protocols. A simplified directory-like sharing tracker is able to effectively capture reuse and fill misses from other private, on-chip caches. This greatly reduces off-chip accesses. With memory bandwidth increasingly becoming the limiting factor in throughput, this can have dramatic performance benefits.

As long as the L1s have sufficient aggregate capacity for the application's working set, our results show that an L1-only organization with sharing tracking matches performance with the large L2! Eliminating the L2 can reduce cost or permit integration of additional cores. Adding the sharing tracker to a manycore CMP with only per-core caches can increase performance per mm^2 by 35%. It can also increase raw performance by reducing bandwidth contention. For example, our results show that the sharing tracker can increase performance by 5 to 12%.

The effectiveness of the sharing tracker with only small, per-core L1s is chiefly due to two factors. First, with many cores, the aggregate L1 capacity is impressively large (1 MB for 32 cores x 32 KB/core). To provide value, an L2 must be substantially larger than this and capture a larger working set that the L1s cannot contain. Furthermore, an inclusive organization, much of the L2 is "wasted" in duplicating the L1 contents. Second, a latency-tolerant design converts the the cache from a tool to reduce latency into a tool to conserve bandwidth. This means that cache misses have minimal cost as long as bandwidth is not a bottleneck. Of course, this requires sufficiently deep multi-threading to hide latency effectively. The sharing tracker's value comes from capturing inter-core reuse that would otherwise have incurred off-chip accesses. Although motivated by the GPU's software-coherent organization and evaluated in that context, the sharing tracker actually offers the opportunity to reduce coherence costs in any manycore organization where scalable hardware coherence protocols are challenging. The sharing tracker could be used to eliminate coherence altogether (with the compiler managing coherence), or the coherence hardware could transition into a simpler mode when coherence is not needed, and therefore save power.

II. Related Work

As an alternative to hardware cache coherence, which poses a number of design challenges, software-controlled coherence has been proposed as a more scalable and lower-cost solution for cc-NUMA and virtual distributed shared memory (VDSM) multicomputer organizations. A simple version of software coherence is for the programmer to manually flush caches when switching between reading and writing, or to double buffer, with separate (cached) input and (uncached) output data structures. This does not present a great burden when the sharing is infrequent and occurs in well-defined patterns. In order to support finer-grained sharing, considerable work was done in the 80s and 90s to enable the compiler to automatically manage coherence in shared-memory systems [5], [6] and to reduce the cost of network transactions for VDSM. For VDSM, the main techniques were to reduce the frequency and size of updates (e.g. Munin [10]) and reduce the latency of those updates (e.g. Shrimp [11]). These techniques generally required operating system support (to manage shared pages) and potentially hardware support (new network interfaces).

Chip multiprocessors have an advantage in this regard, because sharing can be managed natively in hardware and all cores share a common pool of global memory. Other multicore organizations take advantage of this to eschew hardware coherence, e.g. RAW [12] and Cell [9]. GPUs take advantage of shared global memory to optimize the L1 caches for data that is read-only or exhibits only coarse-grained sharing. Although details differ, GPU architectures from NVIDIA [13] and AMD [14] both support similar memory hierarchies; for more details, see the next section. Briefly, fine-grained read-write sharing and synchronization objects are expected to be localized into the PBSM (per-block scratchpad) or accessed only through global memory. Deep multithreading allows other threads to hide latency of threads stalled on global-memory access.

Bakhoda *et al.* [15] evaluate a multi-level, hardware-coherent cache hierarchy for GPUs but results are inconclusive. Our work proposes an alternative that avoids the challenges of hardware coherence.

A huge body of work has of course explored conventional hardware cache coherence organizations (Stenstrom [6] provides a good overview), and various optimizations can be built on top of a coherent organization. We briefly mention work that we believe is most closely related to our line of investigation.

Chang *et al.* [7] use *cooperative caching* to share the resources of a number of private caches on a single chip. They use a *central coherence engine* which replicates the tags of all private caches. Requests which miss in a core's private L2 cache access the coherence engine to check whether the requested cache line is in an L2 of a different core. They also add mechanisms for intelligently replicating cache lines and having evicted cache lines spill to another on-chip cache. The drawbacks of their technique are that each request needs to check a large number of tag arrays (as many as there are cores on the chip minus one), which is a power-hungry process, and that a single cache line can have copies in multiple L2s, which wastes space in the coherence engine.

Herrero *et al.* [8] build on cooperative caching with their work on *distributed cooperative caching*. They replace the replicated L2 tag arrays of the central coherence engine with a distributed, address-indexed tag array, reducing the number of tag comparisons any request has to make to determine whether a copy of its requested cache line exists somewhere on chip. Our work differs in that the sharing tracker is not a full coherency directory, substantially reducing the required hardware and eliminating the complexity of traditional coherence hardware.

Destination Set Prediction [16] assumes a cache-coherent multi-processor where each core has its own L2 cache, and each L2 has its own predictor, which predicts which other core/L2 cache has current ownership of certain cache lines. Destination set prediction was designed for workloads with low degrees of sharing between cores, such as commercial workloads. Our sharing tracker differs from destination set prediction as it useful for workloads where there can be large degrees of sharing of cache lines with irregular patterns. The sharing tracker tracks cache line information at the global level, while destination set prediction keeps track of which other cores a given core previously has exchanged cache lines with.

There has also been considerable work on caches with non-uniform access latencies [17], [18] (so called NUCA caches). NUCA caches are built from a large number of memory tiles, which are addressed by a smart controller, which can move around cache lines based on recency of access or alter the degree to which a memory tile is shared between cores. NUCA caches take a fundamentally different approach from our own, since they focus on intelligently mapping cache lines based on address or giving cores a fixed and uniform amount of sharing with a given set of other cores. The sharing tracker is purely demand driven, only restricted by the capacity and associativity of the caches it covers, and does not restrict sharing between any core anywhere on the chip.

For *modeling* GPUs, Bakhoda *et al.* [15]'s simulator *GPGPUSIM* is an execution-driven simulator which can run kernels compiled to NVIDIA's PTX assembly format and closely models a current generation NVIDIA GPU. Our simulator takes a different, lighter-weight approach, by instrumenting data-parallel applications and collecting only their data access traces. Our simulation approach is discussed further in Section V.

III. GPU Architecture and Memory Model

GPUs are optimized to provide high-throughput and to tolerate frequent long-latency accesses to graphics memory. This is because graphics workloads typically have a very large number of independent tasks (hundreds of thousands of triangles and millions of pixels per rendered frame), and data access patterns with little temporal locality. As a consequence, GPUs have adopted an architecture similar to the MTA Tera [19]. Each core is heavily multi-threaded and scheduling hardware decides each cycle which of the many threads to execute. This is necessary because threads frequently stall due to accesses to graphics memory, and many threads are needed to keep the ALU unit of a core reasonably occupied. In addition, each core uses a SIMD execution model, since graphics workloads are data parallel, with the same task executed for each vertex or pixel. A SIMD organization amortizes the area and power

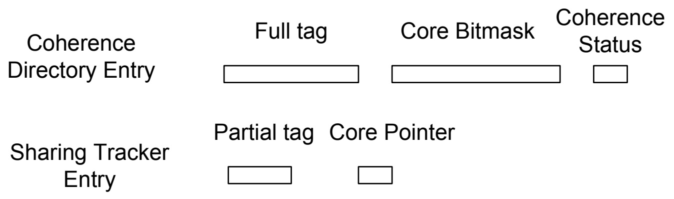


Fig. 1. A cache coherence directory entry consists of a full tag, a bitmask indicating which cores have copies of a particular cache line and a small bitfield to track the current coherency state. In contrast, a sharing tracker entry consists of a smaller partial tag and a pointer to the cache that contains a particular cache line.

overhead of a core's front end over many execution back ends, increasing the total computational power achievable within a given power and area envelope. Note that, in GPU terminology, SIMD lanes are referred to as threads and SIMD groups as warps or wave fronts. We will use the terms thread and warp throughout this paper.

GPU caches are specialized to deal with different address spaces and access patterns, which are derived from the high level graphics APIs [20]. GPUs have long provided caches for the read-only texture and constant spaces, and with Fermi, for global read-write data. The question might be asked why GPUs have any caches for data at all, since they are optimized to tolerate latency. The answer is that GPU caches are mostly meant as bandwidth savers, to avoid wasting memory bandwidth on data with locality, and not as a way to decrease latency of memory accesses.

GPU memory is non-coherent and there are no rules for ordering stores from a single core. Changes made by one core will only be guaranteed to be globally visible after a global memory fence.

As noted above, GPUs employ heavy multi-threading as a way to tolerate memory latency. When adding traditional caches, which in this context we define as supporting both reads and writes and having an access latency substantially lower than memory latency, there is an interesting balance between number of warps per core and the size of the per-core caches. More warps per core increase memory latency and performance, while increasing cache size for a given warp count will increase hit rate, decrease average memory latency and increase performance. But there is the problem that for a given cache size, increasing the number of warps per core will put more pressure on the cache, sometimes leading to a sudden jump in the required off-chip bandwidth due to cache thrashing and a *decrease* in performance. While more warps per core increase performance, this also means increasing the size of the register file to hold the larger number of threads, as well as potentially having to increase the size of caches to prevent thrashing. The best performance per unit of area is not necessarily with the maximum number of warps per core and the largest cache, as we will show in Section VII.

IV. Adapting Coherency Hardware

Current GPUs have multiple SIMD cores, with small, per-core caches. To get better performance with general-purpose workloads, we want to exploit sharing of cache lines between cores to reduce the number and latency of off-chip memory requests. One option would be to add a large, shared, inclusive LLC, which would naturally capture such re-use. But such a cache would occupy significant area, which might otherwise be devoted to more cores.

In traditional CMPs, cache coherence is used to figure out if there is a copy of a requested cache line in a cache on-chip and to request a copy. For manycore CMPs, a snoopy coherence protocol would be problematic because of the rapid rise in communication volume as the number of cores increases. A directory protocol is the better choice for such an architecture. But of course, cache coherence does much more than that, ensuring that a core receives the most up to date version of a cache line and that if one core is writing to a cache line no other core has a valid copy.

This is too much functionality for our purposes, since we want to only save off-chip bandwidth and improve latency of memory requests. We want to decompose the functionality of directory-based cache coherency hardware and keep only the parts needed for our purposes.

- Tracking the status of cache lines (shared/exclusive/etc) is not necessary, since the current programming models of GPUs allow race conditions and reads of stale data.
- Keeping track of all copies of a cache line is not necessary, since we don't have the constraint that a core must hold the only copy of a cache line for a write.
- There can be cache lines which are on chip and not tracked at all. This is allowed since stale copies of cache lines are allowed. Any copying of cache lines between cores is simply to save off-chip bandwidth, not for correctness.

With these relaxations of the requirements versus full cache coherence, we have derived a new structure from previous proposals for directory-based cache coherency hardware for CMPs [7], [8].

We call this new structure the *sharing tracker*.

A. Sharing Tracker Organization

Figure 1 shows the different units involved in a sharing tracker look-up. Note that in the following explanation we refer to all caches as being L2s, but of course the same mechanism applies if the GPU cores only have private L1 caches. The sharing tracker is organized like a shared cache, but each entry holds as data only a pointer to a private cache that contains the specific cache line. Unlike a full distributed coherence engine [8], the sharing tracker does not need to track all the cores which have a copy of a given cache line (which requires a bitmask which grows with the number of cores) or the current coherence state of a cache line.

When a L2 cache miss occurs, the sharing tracker is checked, similar to a shared L3 cache (see Figure 2). If there is

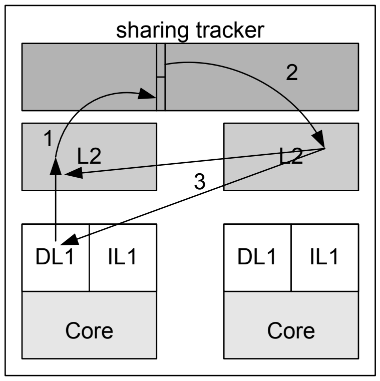


Fig. 2. On an L2 miss, the request is sent to the sharing tracker (1). The sharing tracker is queried like a shared L3 cache. On a hit in the sharing tracker, it reads out the pointer in its entry and forwards the request to the appropriate L2 cache (2). If there is an L2 hit, a copy of the cache line is then forwarded to the original L2 and core (3).

a hit in the sharing tracker, a pointer to the cache holding that cache line (called the source cache) is read from the sharing tracker. A request is sent to the source cache. The source cache then does a normal cache lookup. Note that the lookup will not necessarily hit since the sharing tracker entry can be out of date or there was a false positive hit due to the use of a partial tag. If there is a hit in the source cache, that cache then forwards the cache line to the requesting cache. The sharing tracker's entry is updated to point to the requesting core. If a cache line is evicted from the private L2 cache of a core, the sharing tracker is checked for that entry. If the sharing tracker hits on that cache line AND the core id of the sharing tracker entry matches the L2 id from whose L2 the cache line is being evicted, the sharing tracker entry is invalidated. Note that it is possible that there are one or more copies of the evicted cache line in other private L2 caches on chip, which are lost for future sharing purposes if the corresponding entry in the sharing tracker is invalidated.

Unlike a distributed cache coherency directory [8], the sharing tracker does not have to return a correct prediction. Since each prediction is checked through an L2 lookup, false positives are caught automatically. If there is a miss in the source L2 cache, the request is sent to the memory subsystem and the corresponding sharing tracker entry is invalidated. If the sharing tracker lookup hits and returns the result that the source cache equals the requesting cache we know immediately that a false positive has occurred, since the requesting cache has already done a lookup before sending the request to the sharing tracker.

We can take advantage of this fact by reducing the size of tags in the sharing tracker [21] (see Figure 1). As we show in Section VII, it is possible to substantially reduce the tag size without unduly reducing the effectiveness of the sharing tracker. This is especially important for a cache-like structure such as the sharing tracker where the tag size can be larger than the data per entry.

V. Simulator

The general goal of our simulation approach is to let us explore new architectural ideas in the manycore space quickly. Since we observed that most programs on manycores are bound by the performance of the memory subsystem and because the manycore CMPs use very simple core architectures compared to traditional speculative, out-of-order cores, we have focused our efforts on the cache and memory subsystem while modeling instruction execution with the simplest model possible.

Our custom simulator models a number of SIMD/vector cores, along with a cache hierarchy and a shared memory subsystem. We do model the SIMD nature of the memory references. The cores are modeled as having a constant CPI of one for all non-memory instructions, private L1 data caches, and our model assumes that the structures for holding outstanding memory requests are not a limiting factor. Each core can have one or multiple warps, and like current GPUs, can switch among warps on a cycle by cycle basis at no extra cost. The scheduling algorithm is round-robin, skipping warps that are waiting on memory requests. The memory reference traces are collected directly from the native applications, which are instrumented with calls to our simulator. Direct instrumentation of native applications to generate traces on the fly was preferred over gathering and storing large memory traces. This avoids the I/O and decompression overheads of normal trace-based simulators. To determine the number of instruction cycles between memory references, each application is inspected manually and the number of arithmetic and control flow instructions between memory references is passed to the simulator.

The combination of a simple core model and direct instrumentation of native applications allows the simulator to be very fast (slowdowns of just 10-30x over pure native execution are the norm) and it can consequently capture the performance on input sizes which would be prohibitively slow to simulate otherwise. This is especially important when dealing with a large number of cores and threads per core.

A. Simulated System

Our simulated system is described in Table I. We assume a CMP consisting of 32 in-order cores each supporting 32-wide SIMD execution, all running at 2 GHz, for an overall maximum execution bandwidth of 2 Teraops. Each core has a 32KB private data cache, which has 64B cache lines and is 8-way set associative. We explore whether it makes sense to add a 256KB, 16-way set-associative L2 cache to each core (similar to the proposed Larrabee [22]) in terms of area efficiency or if having smaller cores with only L1 is enough. For all caches, we model a standard LRU replacement policy. We experimented with a variety of other replacement policies, e.g. adaptations of Qureshi's work [23], [24], with no major benefit. We assume that it takes 100 cycles to access the sharing tracker, forward the request to the source cache and copy a cache line to the requesting core's cache. All cores share a 256 GB/sec memory interface, with a memory access latency of 500 cycles.

Number of cores	32
SIMD width	32
Warps per core	1 - 16
Register File size per warp	4KB
Non-memory CPI	1
Per core L1 I-cache	32KB, 8-way
Per core L1 D-cache	32KB, 8-way
(Optional) per core L2 cache	256KB, 16-way
Line size	64 bytes
L2 hit latency	20 cycles
Hit latency in remote cache	100 cycles
Off-chip bandwidth	256 GB/sec
memory latency	500 cycles
Clock speed	2 GHz

TABLE I
Details of the simulated system

Size of the physical address space supported	40 bits
size of full tag and valid bit	$15 + 1$ bits
size of bitmask and coherence state	$32 + 2$ bits
number of entries needed to cover 8MB of L2	128K
Total size of coherence directory	800KB
size of partial tag and valid bit	$10 + 1$ bits
size of L2 pointer	5 bits
Total size of sharing tracker covering 8MB	256KB

TABLE II
Comparison of a coherence directory to the sharing tracker

B. Area Model

To evaluate the trade-off between additional cores or adding an L2 cache to each core or adding structures such as a sharing tracker, we need an estimate of the chip area the different types of structures occupy.

To estimate the area of the SIMD cores, we measured the sizes of the different functional units of an AMD Opteron processor in 130nm technology from a publicly available die photo–this was the best source of area data we were able to obtain. We could only account for about 70% of the total area, the rest being x86-specific, system level circuits, or unidentifiable. We scaled the functional unit areas to 45nm, assuming a 0.7 scaling factor per generation. The sizes of the different cores were then calculated from the areas of their constituent units, scaled by capacity, data path width and port numbers. We also compared the area estimate of the L1 caches we derived from the die photo to the area estimate of Cacti 5 [25] and they were within 2% of each other.

We assume that each lane in a SIMD core has a 32 bit data path and that each thread has a total of 32 32-bit registers, so that each 32-wide SIMD warp uses 4KB of register file. We use Cacti 5 [25] to estimate the area of the per-core 256KB, 16-way set associative L2 cache as well as the other caches and cache-like structures. The area estimates from these calculations are shown in Table III.

For our calculations of area efficiency in Section VII, we also need an estimate for all the structures on a chip apart

core type	core area
32-wide SIMD, 1 warp	7
32-wide SIMD, 2 warps	7.3
32-wide SIMD, 4 warps	7.9
32-wide SIMD, 8 warps	9.1
32-wide SIMD, 16 warps	11.5

TABLE III
AREA ESTIMATES FOR A DIFFERENT VARIANTS OF A 32-WIDE SIMD CORE WITH 32KB L1S. FOR CASES WITH PER-CORE 256 KB L2S, AN ADDITIONAL 4.35 MM^2 SHOULD BE ADDED.

structure description	area (mm^2)
256 KB per-core L2	4.35
8MB LLC cache	44.65
full distributed coherence directory covering 8MB	3.42
sharing tracker covering 8MB	1.01
Area for inter-core network, IO-pads, etc.	74

TABLE IV
AREA ESTIMATES FOR CACHE-LIKE STRUCTURES AND UN-CORE UNITS.

from the cores themselves. We estimate that the SIMD cores will occupy 75% of the die area, with the other 25% used for the inter-core network, IO-pads, memory buffers, etc. We used the smallest SIMD core for this calculation and assumed, as elsewhere, that the chip would have 32 cores. The area of the cache-like structures and the non-core part of the chip are shown in Table IV.

VI. WORKLOAD

Our chosen application kernels represent a mix of application domains and memory access patterns. We have included a kernel (k-means) which is pure streaming, having no reuse of data between threads and cores. We do not expect this kernel to benefit from the sharing tracker, and use it to make sure the sharing tracker does not hurt such applications. Another set of kernels (neighbor list generation, Lennard-Jones force calculation and Gaussian filter) has data reuse between software threads, but the sharing patterns are mostly between threads that tend to access nearby data. These threads are often mapped to the same core and the L1 data caches are enough to capture most of the data reuse. We expect these kernels to show only limited benefits from the sharing tracker, as only a small fraction of memory requests will not hit in the local cache or go to global memory.

Lastly, we have also included kernels (ray tracing and DNA sequence alignment) which have both large working sets and data sharing patterns that are non-regular, meaning that threads on different cores will share data. We expect these kernels to show the most improvement out of all kernels.

Table V shows low-level details of all of our kernels.

In HOOMD (Highly Optimized Object Oriented Molecular Dynamics) [26] version 0.8, the two most computationally intensive functions in HOOMD are the Lennard-Jones potential computation and neighbor list generation, which make up over 95% of the runtime. The neighbor list function (NL) determines, for every particle, to be taken into account. Since all particles move during the simulation time frame, the neighbor list is regenerated every 10 time steps. To avoid the need to check every particle against every other particle, particles are sorted into spatial bins in a preliminary step. The Lennard-Jones function (LJ) calculates the Lennard-Jones potential for each particle for each time step as a function of its neighbor list. Both kernels are parallelized by assigning each particle to a separate thread. We run the standard HOOMD benchmark simulating a liquid consisting of 64000 particles at a packing fraction of 0.2 interacting via the Lennard-Jones force. We simulate the first 600 time steps.

MummerGPU [27] (SA) uses a suffix tree to efficiently find alignments of short DNA sequences against a reference genome. The tree is traversed from the root in a data dependent manner, with each edge holding a variable number of base pairs that must all match for the traversal to proceed to the next node. MummerGPU parallelizes its computation by mapping each input string to a thread. Similar to Schatz *et al.* [27], we run SA in the exact matching mode, matching batches of synthetic snippets of length 25, 50, 200 and 800 base pairs sampled randomly from the *Bacillus anthracis* genome ($GenBankID : NC\_003997.3$) to match against itself. Each batch contains a total of one million base pairs, with batches containing longer string containing linearly fewer samples. We report the average performance over all 4 string lengths.

From the bwfirt ray tracing (RT) framework [28], we use the provided SimpleBVH ray tracer as our test application. SimpleBVH decomposes the scene into a bounding volume hierarchy tree. Each ray traverses the tree to find the object that it hits in the scene. Bwfirt uses SimpleBVH to do path tracing through a given scene, letting rays bounce around a scene multiple times until they hit a light source. We chose bwfirt because it does not just trace primary rays. We parallelize SimpleBVH by having each thread trace a different ray through the scene. This method of parallelization provides a large number of independent tasks without the need for any communication among threads until the output of the final result. As our input, we use the conference scene with approximately 1 million triangles and set the resolution of the generated image to 1024 by 1024 pixels.

K-means (KM) mines data sets by grouping data elements into a desired number of clusters in a way that minimizes the aggregate distance from cluster centers. We use the Minebench [29] version, which randomly generates N cluster centroids, and then iteratively assigns points to the nearest centroids, calculates new centroids, and repeats until the number of points switching cluster to another falls below a pre-specified threshold. We assign each point to a thread for the distance computation and each centroid to a thread for the centroid re-computation, and run k-means with 32 clusters and with the provided input set of roughly half a million data points, each with 36 features.

For image manipulation, we use a blurring kernel (GF), which computes the 3 by 3 Gaussian blur for each pixel of the input image. Each warp is assigned an image tile consisting of 32 by 32 pixels, with threads being assigned a single row in the tile. The input is a randomly generated black and white

Kernel Name	instructions per memory op	memory access pattern
Neighbor List Generation (NL)	19	local sharing
Lennard-Jones Force Calculation (LJ)	25	streaming & local sharing
DNA Seq. Align. (SA)	7	complex sharing
Ray Tracing (RT)	15	complex sharing
K-Means (KM)	5	private re-use
Gaussian Filter (GF)	8	2D stencil

TABLE V
NUMBER OF INSTRUCTIONS PER MEMORY OPERATION AND MEMORY REUSE PATTERN FOR EACH KERNEL

image with 2048 by 2048 pixels resolution.

VII. EVALUATION

Our initial investigations showed that reducing the tags to 10 bits showed no noticeable performance drop compared to full tags. We use 10 bit tags in all the following experiments.

A. Performance Comparison

To evaluate the overall impact of adding the sharing to a manycore GPU, we first show the performance and bandwidth improvement possible by adding a sharing tracker to the base CMP with per-core L2 as described in Section V-A. In this first experiment, the L2s are maintained. The moderately large per-core L2s already capture much more of each core's working set (despite some duplication of data among L2s) than the 32 KB L1, so we expect modest benefit from adding the sharing tracker. We compute the unweighted geometric mean performance and bandwidth across the kernels from Section VI, where the performance of each kernel in each configuration has been normalized to the performance of that kernel without a sharing tracker.

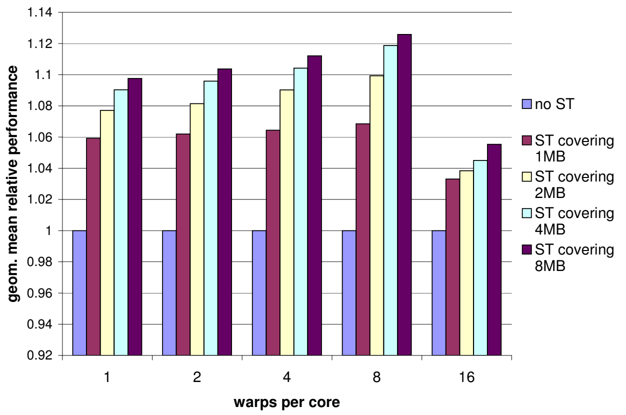


Fig. 3. Gmean performance using sharing trackers (ST) covering 0 to 8 MB of L2, normalized to no sharing tracker, as a function of warps per core.

As can be seen in Figures 3 and 4, the sharing tracker can increase performance between 3 and 12% while reducing the required off-chip bandwidth by 20 to 45%.

The kernels which benefit the most from the sharing tracker are those that are bandwidth bound and have significant sharing of data between threads on different cores. These are primarily the RT and SA kernels (shown in Figure 8 along with LJ) which both traverse very large data structures that are shared between all threads and have complex sharing patterns

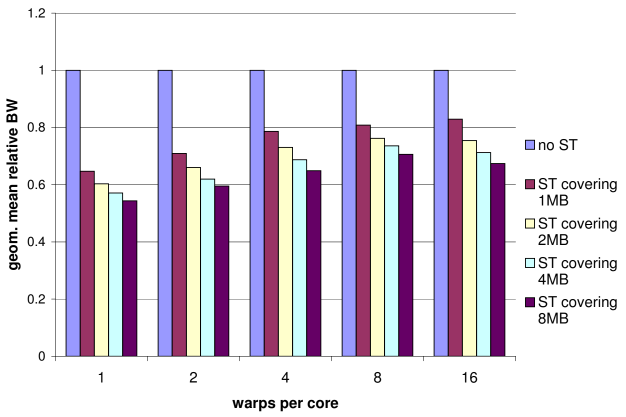


Fig. 4. Gmean off-chip bandwidth.

where widely spaced threads can access the same data. The KM, GF and NL generation kernels show no performance improvement. This is expected, as the GF and NL kernels have only local sharing of data which can be satisfied by each core's L2 caches. The KM kernel only shares a very small array between all threads and each thread touches only its private data apart from the very small global array, meaning it has no re-use which cannot be captures by the L1 caches.

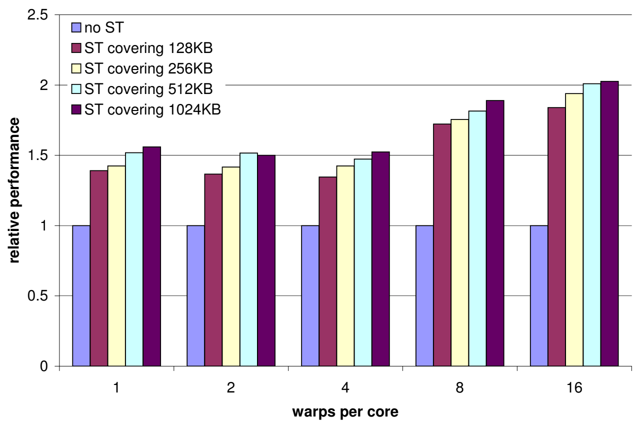


Fig. 5. Gmean performance using sharing trackers covering 0 to 1 MB of L1, normalized to no sharing tracker.

We now evaluate the performance and bandwidth savings if each core only has L1 caches. Figures 5 and 6 show the performance and bandwidth improvements possible by adding a sharing tracker covering part or all off the L1 data caches. We can see that both the performance and bandwidth improvements with a sharing tracker are greater than when

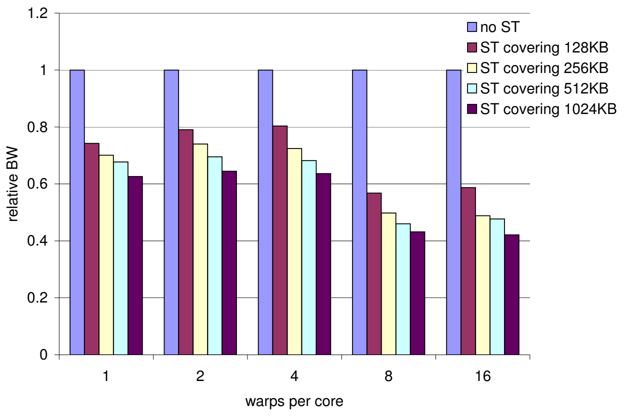


Fig. 6. Gmean off-chip bandwidth.

each core has an L2 cache. Performance improves between 50% and 102% relative to the L1-only case without the sharing tracker, but the more important observation, shown in Figure 7, is that the L1-only case with sharing tracker performs about as well as the conventional organization with the large total L2. The difference to the prior case is due to most kernels' becoming much more bandwidth and latency bound without L2. The RT, SA and LJ kernels show bigger improvements, but the real difference is that the KM and GF kernels now also improve in performance for some configurations. This is primarily because these kernels thrash their L1 caches at higher warp counts. Bandwidth savings are between 38 and 58% for similar reasons. The performance of the smallest sharing tracker we study, 128K/core, is surprisingly close to the largest sharing tracker. This is likely because that small capacity is able to capture much of the short-term reuse among cores.

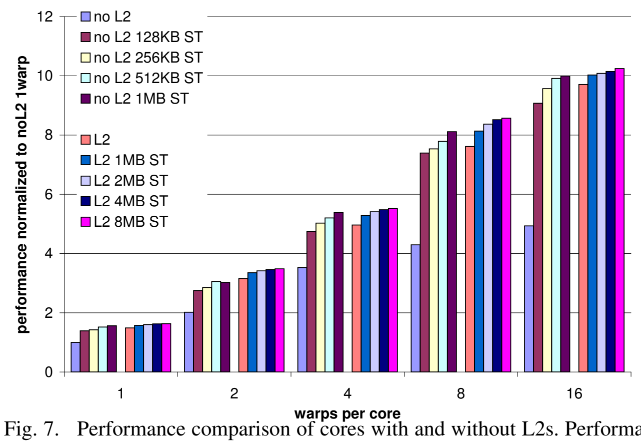
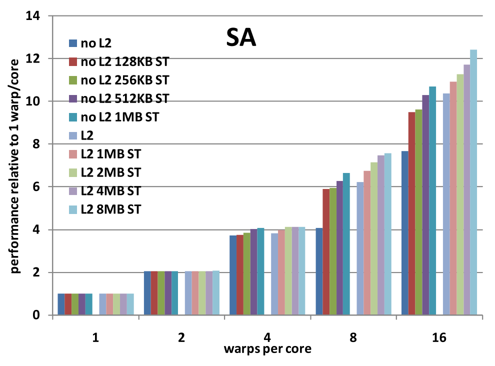
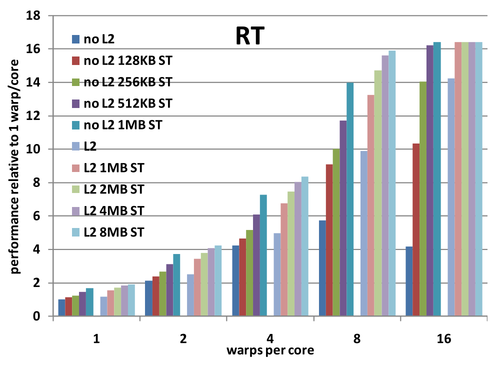


Fig. 7. Performance comparison of cores with and without L2s. Performance is the Gmean across kernels, normalized to the performance of 1 warp per core and no L2 cache or sharing tracker. The former 5 bars per configuration are without L2 and the latter 5 bars are with L2. The first bar of each group uses no sharing tracker.

Clearly the individual L1s (i.e., no sharing tracker) do not have sufficient capacity to capture each core's working set. Next we compare organizations with and without the L2. Figure 7 compares geometric mean performance across all kernels normalized to the performance of the configuration





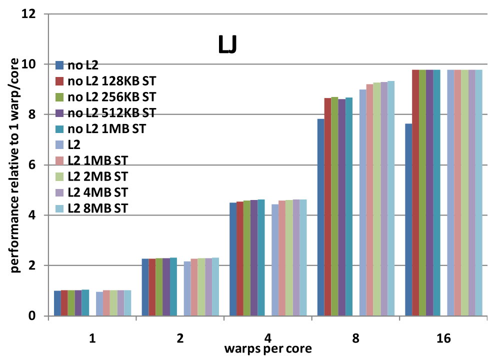


Fig. 8. Performance scaling relative to 1 warp/core for the RT, SA and LJ benchmarks.

with the smallest chip area, which is one warp per core and no L2.

We first note that if we compare the performance of cores with and without L2 cache and no sharing tracker, the relative performance benefit of L2 grows as we increase the number of warps per core. This is because more warps per core put more pressure on the caches, and the L1 starts to thrash for some kernels at 8 and 16 warps per core. It is very interesting to note that the small sharing tracker can lift the performance of the the no-L2 configuration to a level competitive with the L2 configurations. Performance drops at most a few percent even if L2 is eliminated! It is not clear how much this is due to limited long-range temporal locality in our suite of kernels, and how much due to latency tolerance with sufficient number of warps. At 16 warps, the L1-only organization with the best sharing tracker outperforms the conventional organization with 256 KB L2 per core and no sharing tracker, and is within 2.4% of the configuration with L2 and the largest sharing tracker.

B. *Performance/Area Comparison*

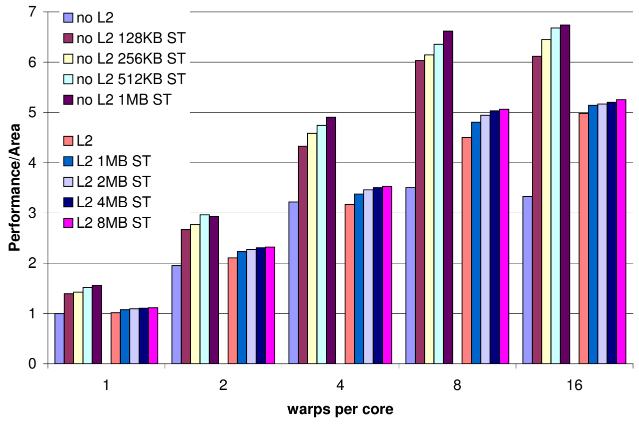


Fig. 9. Performance per mm^2 for cores with and without L2 caches as we scale the number of warps per core and the sharing tracker.

Raw performance is not the only metric architects care about. Figure 9 shows the performance per mm^2 of each configuration. For this calculation we use the area of each core configuration from Table III and add the fixed overhead of the non-core part of the chip as shown in Table IV. Here we can see that in the base case (no sharing tracker), adding L2 caches to each core makes little sense even without the sharing tracker below 8 warps per core, as performance per mm^2 of the configurations with and without L2 caches are within 0 to 7% for 1 to 4 warps per core. At 8 warps per core that difference grows to 28% and to 50% at 16 warps.

With the addition of the sharing tracker, the performance per mm^2 of the configurations without L2 cache rises much more than those with L2, making the no-L2 configuration the top choice. The advantage is 40% with 1 warp, 26% with 2, 39% with 4, 31% with 8 and 28% with 16 warps. Comparing the configuration with the highest performance per mm^2 (16 warps per core and per core L2 caches) without the sharing tracker to the one with sharing tracker (16 warps per core, no L2 caches, sharing tracker covering all of the on-chip L1 capacity), we see a 35% improvement.

To more clearly illustrate the benefit of removing the L2's from each core we plot the ratio of performance/mm^2 for each configuration with and without L2 caches and with different sized sharing trackers in Figure 10. To put it in terms of an equal-area comparison, we estimate that eliminating the L2 saves up to 10% chip area. This can be applied to improve performance of the no-L2 case, either by integrating more cores (but odd numbers of cores might be problematic, and more cores may require more memory bandwidth), or more warps per core. A good example is the comparison between 4 and 8 warps in Figure 7. The points with 8 warps without L2 are about 45% better than 4 warps with 8MB L2.

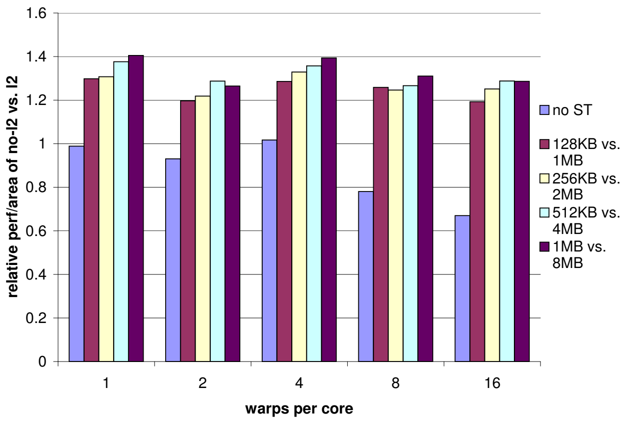


Fig. 10. The ratio of the performance per unit area of configurations with and without per core L2 cache.

VIII. Conclusions and Future Work

GPUs have recently emerged as a new platform for high-performance computing. Their current cache organization is optimized for streaming data with little temporal locality and no sharing between cores, and requires software to manage any coherence requirements. To efficiently support general-purpose workloads, better support for temporal reuse is needed. However, as long as GPUs' main sales volume remains biased toward 3D rendering applications, and these do not require cache coherence, we think it is unlikely that GPU vendors will add full cache coherence in the near future. Since cache coherence is challenging to implement at large scales in any case, software coherence is an appealing option for any large-scale multiprocessor, and some other organizations, notably Cell and RAW, have followed this approach.

This paper shows that in a throughput-oriented processor with effective latency-tolerance mechanisms, a lightweight alternative (called the sharing tracker) to a full on-chip cache coherency directory provides all of the benefits of cache coherence for sharing cache lines among multiple caches on a chip, with 4 to 20 times less area than a coherent organization. The sharing tracker allows the L2 to be eliminated entirely and still boosts performance by 3%. Even in the case where the L2 organization includes a sharing tracker, the L1-only organization with sharing tracker only sacrifices 2.4%. This conclusion of course depends on the working set. Our applications have short-term working sets that can be captured by the L1s' aggregate capacity, and long-term working sets that even a large L2 cannot contain.

The sharing tracker also reduces off-chip bandwidth by 38–58% compared to a conventional L2 organization, and increases performance/mm^2 by 35% compared to a design with only per-core caches. In terms of specific equal-area trade-offs, the sharing tracker can be used to turn area that would otherwise have been poorly utilized for a large L2 into computational resources, such as providing more warps per core. Although our results are obtained with a GPU organization, the success of the sharing tracker in that context suggests that other throughput oriented architectures should evaluate a similar approach. Generalizing our approach poses an interesting direction for future work.

As manycore CMPs increase the number of cores per chip, the latency of accessing any global structure will worsen, and the idea of the sharing tracker opens a number of avenues for future work. One way to deal with this problem is by replicating resources, but this is expensive for large structures. We want to investigate whether we can use the fact that the sharing tracker is small and does not require precise or up-to-date data, for a design that distributes multiple copies of the global sharing tracker across a CMP. This can reduce global on-chip network bandwidth and latency, potentially increasing performance. Another question is whether a hierarchical sharing tracker, where smaller sharing trackers cover a subset of cores and can resolve misses before they have to go to a global sharing tracker, can achieve the same bandwidth and latency advantages as our replicated sharing tracker. Yet another question is whether the sharing tracker argues for an inverted hierarchy in which only a very small L2 is used as a victim cache for the L1s. Finally, our investigations in this paper should be repeated with a wider range of applications and working set sizes.

Acknowledgments

This work was supported in part by NSF grant no. CNS-0509245 and CNS-0916908. We would also like to thank the anonymous reviewers for their helpful comments.

References

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, , and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *SIGGRAPH*, 2004.

[2] A. Munshi, "The OpenCL specification, version 1.0, document revision 29," Dec. 2008. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[4] N. Corp., "Nvidias next generation cuda compute architecture: Fermi," Whitepaper, 2009.

[5] H. Cheong and A. Veidenbaum, "A cache coherence scheme with fast selective invalidation," in *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988, pp. 299–307.

[6] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[7] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006, pp. 264–276.

[8] E. Herrero, J. González, and R. Canal, "Distributed Cooperative Caching," in *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 134–143.

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell processor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, 2005.

[10] J. B. Carter, J. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *In Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Oct. 1991, pp. 152–164.

[11] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "Virtual memory mapped network interface for the shrimp multicomputer," in *ISCA'94: Proceedings of the 21st International Symposium on Computer Architecture*, May 1994.

[12] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring It All to Software: Raw Machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, 1997.

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

[14] Michael Mantor, "Radeon R600, a 2nd Generation Unified Shader Architecture," 2007.

[15] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS'09: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[16] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors," in *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 206–217.

[17] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 211–222.

[18] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA substrate for flexible CMP cache sharing," in *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, 2005, pp. 31–40.

[19] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," in *ICS '90: Proceedings of the 4th International Conference on Supercomputing*, 1990, pp. 1–6.

[20] D. Blythe, "The Direct3D 10 system," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 724–734, 2006.

[21] B. Fagin, "Partial Resolution in Branch Target Buffers," *IEEE Trans. Comput.*, vol. 46, no. 10, pp. 1142–1145, 1997.

[22] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[23] M. Qureshi, "Adaptive Spill-Receive for robust high-performance caching in CMPs," in *HPCA'09: Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 45–54.

[24] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2007, pp. 381–391.

[25] Shyamkumar Thoziyoor and Naveen Muralimanohar and Jung Ho Ahn and Norman P. Jouppi, "CACTI 5.1," HP Labs, Tech. Rep. HPL-2008-20, 2008.

[26] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General Purpose Molecular Dynamics Simulations fully implemented on Graphics Processing Units," *J. Comput. Phys.*, vol. 227, no. 10, pp. 5342–5359, 2008.

[27] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-Throughput Sequence Alignment using Graphics Processing Units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/474

[28] M. Raab, L. Grünschloss, J. Hanikaz, M. Finckh, and A. Keller, "bwfirt reposit ory." [Online]. Available: http://bwfirt.sourceforge.net/

[29] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads," in *IISWC'06: Proceedings of the 2006 IEEE International Symposium on Workload Characterization*, 2006, pp. 182–188.