USING HYBRID SHARED AND DISTRIBUTED CACHING FOR MIXED-COHERENCY GPU WORKLOADS

BY

NASSER SALIM ANSSARI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Wen-mei W. Hwu

ABSTRACT

Current GPU computing models support a mixture of coherent and incoherent classes of memory operations. Workloads using these models typically have working sets too large to fit in an economical SRAM structure. Still, GPU architectures have last-level caches to primarily fulfill two functions: eliminate redundant DRAM accesses servicing requests from different L1 caches to the same line, and maintain on-chip memory coherence for the coherent class of memory operations.

In this thesis, we propose an alternative memory system design for GPU architectures better fit for their workloads. Our architectural design features a directory-like sharing tracker that allows the incoherent private L1 caches to directly satisfy remote requests for shared data. It also retains a shared L2 cache with a customized caching policy to support coherent accesses on-chip and better serve non-coalesced requests that contend aggressively for cache lines.

This thesis characterizes the novel and intriguing tradeoffs between the components of our proposed memory system design for area, energy, and performance. We show that the proposed design achieves a 22% average reduction in DRAM data demand over a standard GPU architecture with 1MB L2 cache, leading to an overall 28% reduction in the memory system energy consumption on average. Conversely, our results show that the DRAM data demand of the proposed design with 256KB L2 cache is on par with a standard GPU architecture with 1MB L2 cache, albeit at a smaller area overhead and power leakage. Our results, while drawn on motivations from the GPU realm, are not architecture-specific and can be extended to other throughput-oriented many-core organizations.

Patri meo

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Wen-mei W. Hwu, for his mentorship and support. His devotion and hard work have always been examples to follow. I appreciate the opportunity to work in his research group.

I am grateful to Marie-Pierre Lassiva-Moulin for her keen assistance and unparalleled spirit. I am also grateful to John Stratton for his invaluable insights and leadership.

I would like to thank all members of the IMPACT research group for their advice and camaraderie. IMPACT is a team in every sense of the word.

This thesis is based on unpublished manuscripts for MICRO-45 and ISCA-40. I gratefully acknowledge the contributions of John Stratton, Christopher Rodrigues, Isaac Gelado, and Jeremy Enos.

Finally, I would like to thank my parents for their love and discipline.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Throughput-oriented compute accelerators in heterogeneous computing systems have become ubiquitous targets for performance-sensitive applications. Graphics processing units (GPUs) have played a dominant role across accelerated computational domains, and have significantly influenced their software programming models. As such, programming for accelerated systems today is almost synonymous with General-Purpose Computing on Graphics Processing Units (GPGPU) [1].

GPU architectures at first were heavily influenced by their original graphics workloads. The lack of a general cache for instance was mainly justified by graphics workloads folding spatial locality into vector accesses of entire DRAM bursts in a single instruction. In addition, specialized caching mechanisms for input textures, output frame buffers, and global constants were sufficient for the graphics workloads.

Experiences with general computing, however, led designers of more recent GPU architectures to include bona fide cache systems. In a stark contrast to GPUs of only few years ago, the most recent GPUs from NVIDIA for example have up to 64KB of L1 cache private to each compute unit, and a 1536KB L2 cache shared between compute units [2]. With an aggregate L1 cache capacity on par with the L2 cache, the on-chip memory hierarchy of a GPU breaks the conventions of the CPU world, where the last-level cache is orders of magnitudes larger than the aggregate lower levels of the hierarchy, to handle the largest working set possible on-chip. Instead, GPU architects devote much less chip area to the last-level cache, because for many general-purpose workloads amenable to acceleration, the largest economical on-chip RAM structure (SRAM or EDRAM) would still be too small to hold the entire working set.

That being the case, the L2 cache in current GPU memory designs does not increase the amount of cache space per thread compared to the L1 caches: the cache capacity divided by the number of threads with access to that cache is equivalent for both cache levels. The second-level GPU cache serves primarily as a coherence point between the otherwise incoherent L1 caches, and a sharing point for L1 caches requesting the same lines. Yet sharing among private L1 caches can be facilitated by other means, including direct line transfers between them. Proposals to this end range between assuming fully coherent memory systems similar to modern CPUs, or completely incoherent systems [3]. Both extremes, however, do not address the de facto memory models of GPU programming languages at hand, the former being rather too constrained, and the latter being far more lax.

A second observation about current GPU memory designs is that their cache hierarchies are optimized for wide SIMD accesses, but handle strided and interleaved accesses less efficiently. L1 cache lines in GPUs are typically longer than L2 cache lines, and fully occupied systems have more threads than unique cache lines. In pathological scenarios, strided and interleaved accesses thrash the L1 caches and incur the cost of a full L1 cache line transaction, rather than the smaller cost of an L2 cache line transaction, for each word used. Accessing data from a thrashing L1 cache constantly requesting lines is thus worse in many cases than doing so directly from the DRAM. Previous work noted that performance can be significantly improved by leveraging instruction set architecture (ISA) features to bypass the L1 cache in these cases as directed by static compiler analysis [4]. We see little reason to rely on the compiler for such caching policy decisions when the hardware already has full knowledge of the access pattern of a memory instruction upon generating cache line requests for it.

Based on these insights, we propose a GPU memory system design with the following novel features:

- A directory-like sharing tracker compatible with real GPU memory models, including the support of thread-block self-consistency and coherent memory operations. We show that the sharing tracker, when combined with a shared L2 cache, reduces total DRAM data demand and memory system energy consumption, even when the L2 cache is as large as the L1 caches combined. These benefits amount to 9% average reduction in DRAM data demand and memory system energy consumption when the L2 cache capacity is one-eighth the aggregate

L1 cache capacity.

- A pure hardware policy for selectively bypassing the L1 caches based on the likelihood of cache thrashing for each memory access. On top of the sharing tracker, this new caching policy reduces DRAM data demand by 20%, and energy consumption by 25% on average in a memory system with 1MB L2 cache. Conversely, the sharing tracker and caching policy reduce the DRAM data demand of a memory system with 256KB L2 cache to the levels of a system with 1MB L2 cache that uses neither.

Not only are the proposed features individually useful, but also they are synergistically related. A sharing tracker essentially increases the exclusivity of the L1 and L2 caches, and thus the effective cache capacity of the system. A caching policy adaptive to cache contention makes better usage of the extra L2 cache space unlocked by the sharing tracker for accesses with poor or dynamic locality not addressed well by the L1 caches.

In terms of temporal performance, the memory system energy and data demand reductions should not come at any cost for compute-bound applications. The sharing tracker and caching policy should have a positive impact as well for systems where the aggregate L1 cache capacity outweighs the L2 cache capacity. However, for systems with relatively large L2 caches, the on-chip network bandwidth and latency must be kept under control lest the additional on-chip traffic for direct L1 cache transfers render them as the new bottlenecks.

Before going through the details of our proposed design, we discuss the architecture and memory model of current NVIDIA GPUs in Chapter 2, together with the common characteristics of the memory access patterns of GPU workloads. Chapter 3 explores the details of our proposed design and presents our architectural improvements. The effects of our design on energy, area, and off-chip memory traffic, according to our simulation methodology, are evaluated and discussed in Chapter 4. Chapter 5 highlights related work and concludes.

CHAPTER 2

GPU ARCHITECTURE, MEMORY MODEL, AND WORKLOAD CHARACTERISTICS

2.1 GPU Architecture and Memory Model

Efficiency has recently become a multi-faceted notion and goal, ranging from the ever-crucial performance efficiency to the freshly minted energy efficiency. Performance has been the most directly visible aspect of efficiency for application users. Energy efficiency, however, is becoming increasingly important across the spectrum of computing markets, from portable devices and their battery lives, to supercomputers and their electric bills. Perhaps unsurprisingly, these two aspects of efficiency are organically linked: applications with higher performance tend to use less energy in general, simply by virtue of finishing more quickly, spending less time consuming power. It is this desire for both high performance and efficient energy usage, coupled with the stagnation of processor frequencies, that led to the mass adoption of parallel and accelerated computing.

The mainstream vehicle for accelerated computing has been graphics processing units, or GPUs. GPUs began as special purpose processors, but gradually adopted more general workloads. The current generation of low-level, general-purpose accelerator programming models, including CUDA [5], OpenCL [6], and Direct Compute [7], as well as their higher-level relatives, such as OpenACC [8] and C++AMP [9], all attempt to map common hardware artifacts to software constructs in an elegant way for high performance. The programmability of any of these languages, however, is at least as complicated as developing sequential C or C++ code. Accelerator programming models and languages have primarily been the domain of performance-sensitive code regions or *kernels*, where developers are keen to invest extra time and effort to improve efficiency even after meeting all functionality criteria.

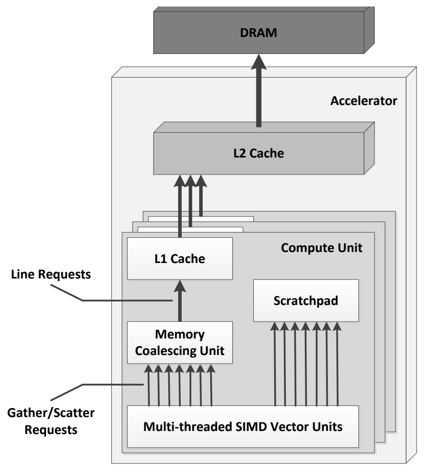


Figure 2.1: An Abstract Accelerator Architecture

Moving forward, there are two plausible perspectives to consider on the evolution of accelerated computing. One perspective assumes that accelerated code is optimized code, and thus rationalizes designing features for the utmost efficiency and performance of well-optimized programs, regardless of the performance cliffs for certain patterns of poor optimizations. The other perspective expects the average level of optimization in kernels to decrease as the barrier of entry lowers and more developers adopt accelerated programming. In this thesis, we reevaluate the GPU memory architecture, as a specimen of accelerators, and show that current designs can be significantly improved from each of these two perspectives. Therefore, we continue our background discussion with an analysis of current GPU architectures, and the resulting execution and memory access patterns and how well they suit these architectures.

Though the concepts of our thesis are language-agnostic, we use the CUDA language and terminology for our studies. One reason to choose CUDA is that its current implementations often perform better than other languages on NVIDIA's hardware [10]. It is the recent NVIDIA GPUs that have the most extensive caching among consumer GPU products, which ensures that our simulations are grounded with real hardware results.

Figure 2.1 shows our abstract accelerator architecture. We assume that the *device* is composed of a number of independent compute units (processors), attached to a single DRAM memory system. Each compute unit is equipped with a scratchpad memory and L1 cache. These are logically separate memories, though in some GPUs they inhabit the same physical storage. The last-level cache is shared by all compute units. The focus of this study is on effectively caching DRAM accesses through the different cache levels and policies.

Almost all of the current competing accelerator programming models implement a bulk-synchronous parallelism model [11] between the *host* CPU and the device. The primary construct for the device code in these models is a data-parallel, single program multiple data (SPMD) kernel comprising groups of co-scheduled threads, or *thread blocks*. Thread blocks have access to private local stores, through which their constituent threads can share data. Each thread in turn has a private data space, typically implemented as registers. All other data reside in the global memory, a space shared by all thread blocks.

When scheduled, each thread block is assigned to a particular compute unit, which executes the constituent threads to completion. A compute unit may concurrently host multiple thread blocks, and it interleaves their execution to overlap the instruction latencies of different threads. Internally, threads are not independently executed, but are bound into execution groups, or *warps*. The execution hardware schedules instructions at the granularity of a warp, with transparent mechanisms for masking off individual threads not following the same control flow path as the rest of the warp. As shown in Figure 2.1, the execution hardware essentially comprises single instruction multiple data (SIMD) vector units, and all loads and stores are actually SIMD gather and scatter operations. Requests from all SIMD lanes, or threads of a warp, are grouped by a *coalescing* unit into the smallest number of cache lines necessary to satisfy them. Memory transactions processed at the first cache level are coalesced.

In such a SIMD-like execution model, memory operations are most efficient if the SIMD lanes access adjacent memory locations, thereby generating memory requests with ideal spatial locality. When programmers follow this pattern, the SIMD execution of a memory instruction generates a single, coalesced memory transaction, and the cache hierarchy is accordingly built to accommodate such a transaction in a single line. Strided or scattered access patterns, on the other hand, generate many memory transactions to multiple cache lines at once [4, 12], exposing the poor provision of the cache hierarchy for these access patterns.

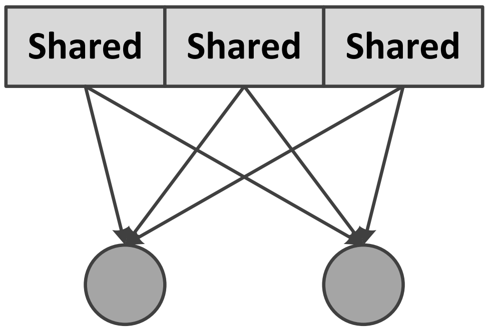
As far as memory coherence and consistency are concerned, GPU (accelerator) programming languages have very relaxed memory models. In particular, they feature three scopes over which different models hold: within a thread block, across thread blocks, and across kernel instances (or between the kernel and the host code). There is weak consistency within a thread block, as threads can barrier-synchronize with each other, forcing all their pending memory operations to complete. Threads from different threads blocks cannot, however, and there is generally *no* memory coherence or consistency between them [6]. Weak consistency holds again across kernel instances. To allow for a kernel's output to be safely used following its execution, the host code (or GPU driver) can synchronize with the device to ensure that all memory operations in the kernel have completed.

The absence of inter-block memory consistency means that there can be no global coordination within kernels. Nevertheless, thread blocks can still use atomic operations to communicate, which implies a consistency model stronger than the one described above. Atomics are an example of a set of coherent memory operations which observe a coherent memory state, supported besides the usual incoherent memory operations. Coherent accesses are visible to later coherent accesses from all threads, and therefore can be used for inter-block communication.

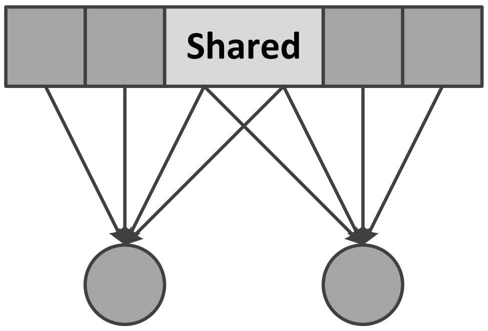
Furthermore, current GPUs, to the best of our knowledge, also provide consistency among accesses from the same compute unit, allowing all accesses performed by a thread block to be visible to all subsequent accesses from the same thread block. In this thesis, we assume a coherence (memory operation visibility) and consistency (memory operation ordering) model based on CUDA and the observed behavior of NVIDIA GPUs.

This relaxed memory model arises because GPUs eschew hardware cache coherence, and stale data in their private caches is an artifact of the lack thereof. Stale data may remain in a private cache until invalidated at a global synchronization event. A shared higher-level cache, to which the private caches write through, provides a coherent view of memory because all writes update it. Atomics, among other memory operations, achieve coherence by bypassing the private cache levels.

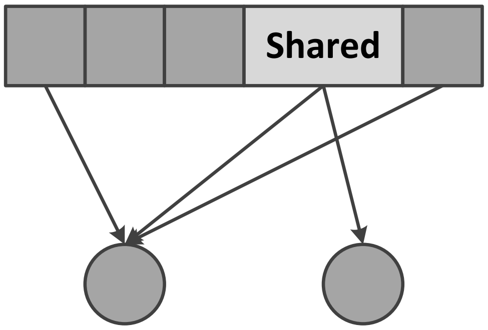
Practically, for programmers to attain predictable behavior, they need to partition memory into regions, some private to thread blocks and incoherently accessed only by the associated thread block, and others shared between



(a) Broadcast Sharing: Data Accessed by a Large Number of Threads



(b) Boundary Sharing: Data Shared by a Small Number of Threads at Tile Boundaries



(c) Dynamic Sharing: Indirect Accesses to Shared Data Have Undetermined Sharing Between Threads

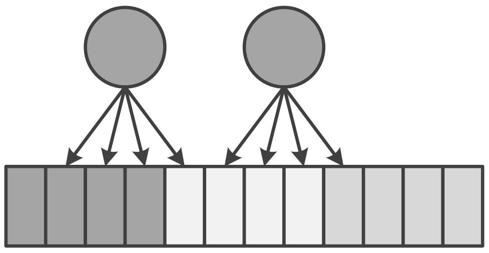
Figure 2.2: Data Sharing Patterns in GPU Computing Workloads

thread blocks that are either coherently accessed, or incoherently read only. The partitioning may change at global synchronization events.

2.2 GPU Workload Memory Characteristics

Our proposals for refining the GPU memory system are motivated by an analysis of the workloads of that system. In this section, we categorize the common patterns of interactions between GPU computing applications and the memory system, and highlight benchmarks which demonstrate those patterns.

Sharing in GPU computing workloads generally falls into one of three patterns. The first, depicted in Figure 2.2a, is a *broadcast* pattern, where a


(a) Misaligned Partitioning: Access Misalignment with Cache Line Boundaries Causes Most Cache Lines to Be Falsely Shared Between Warps

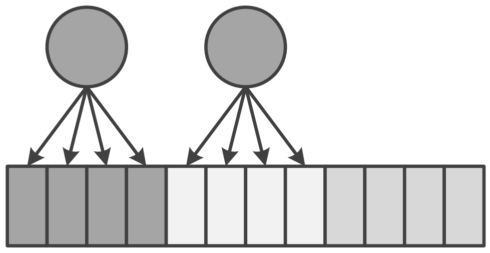

(b) Perfect Partitioning: Aligned, Coalesced Accesses Causes Each Cache Line to Be Touched By One Warp

Figure 2.3: Data Partitioning Patterns in GPU Computing Workloads (Cache Lines are Marked in Different Shades of Gray)

particular data region has many sharers in the same kernel. This includes both full and partial broadcasts. A histogramming benchmark, where each datum of the input set can affect all output elements, is an example of the former, while a tiled matrix multiplication benchmark, where each tile of input matrices is consumed by a band of threads assigned to output tiles, is an example of the latter.

Figure 2.2b shows the second pattern, *boundary sharing*, common in stencils. In this pattern, tiles of data overlap between different threads, resulting in a small number of sharers for any particular element. Boundary sharing can cause a varying degree of cache-line sharing, depending on the sizes of tiles and cache lines, but a common case of tile sizes matching cache line sizes results in a small degree of sharing for most cache lines.

Finally, *dynamic sharing*, depicted in Figure 2.2c, is a data-driven sharing pattern, resulting from statically unknown indirection, and thus a nondeterministic sharing pattern. The sharing is proportional to the access density, how many accesses are spread over how many data elements, if the accesses are random. Frequent accesses to a small lookup table for example exhibit lots of sharing, while a sparsely accessed data structure does not.

Partitioned data are data needed by only one thread. Data partitioned contiguously can still be falsely shared between threads, with a cache behavior similar to boundary sharing, if the partition boundaries are not aligned to cache-line boundaries as shown in Figure 2.3a. In the best scenario from a performance standpoint, portrayed in Figure 2.3b, partitioned data are accessed in a perfectly coalesced manner, with aligned contiguous accesses that do not straddle several cache lines. In such situations, no cache line is touched by more than one thread, and more often than not, a cache line is touched exactly once in the kernel, either read into registers to be reused as necessary, or written with the final result previously accumulated in registers. The interplay between data with non-coalesced partitioning and the cache hierarchy is more intricate and nuanced.

Besides the L1 cache, compute units in most GPU architectures are usually equipped with a scratchpad. Threads within a block can cooperate by sharing data through the scratchpad while synchronizing their execution to coordinate memory accesses. Using the scratchpad as a software-managed cache changes the access pattern visible to the cache hierarchy itself. Our preliminary experiments confirm that the number of L1 cache accesses is significantly lower for benchmarks optimized by using the scratchpad, as many redundant accesses are diverted to it. However, the accesses seen by the rest of the memory hierarchy are largely identical, as thread blocks traverse the same data in roughly the same general order. Therefore, we can conclude that the insights about a memory system design drawn using cache-optimized workloads are equally applicable for scratchpad-optimized workloads. In this thesis, we focus exclusively on cache-optimized workloads since they are the most sensitive to the changes we propose to the cache hierarchy.

CHAPTER 3

A NEW ARCHITECTURE DESIGN FOR MIXED-COHERENCY GPU WORKLOADS

3.1 Harnessing Locality in GPU Workloads

Traditional chip multiprocessors use cache coherence as a means for their processors to share data. In a coherent multiprocessor, caches provide both migration and replication of shared data to allow for transparent accesses to shared data at a small latency and minimal contention [13]. Coherent caches hence present a uniform view of memory using coherence protocols that track the state of all shared data and propagate changes throughout the system.

There are two classes of coherence protocols used in chip multiprocessors, namely snooping protocols and directory-based protocols. Snooping protocols are popular in small-scale multiprocessors because they use the existing bus to memory to interrogate the status of the caches. For many-core architectures, however, a snooping protocol poses a scalability challenge, requiring a wider bus with higher bandwidth to support a larger volume of broadcast coherence traffic as the number of processors in the system increases. A directory-based protocol is thus the practical option for these architectures.

To maintain their coherence requirements, not only do coherence protocols allow a processor to access data shared on-chip, but also they ensure it receives the most up-to-date version of that data. To this end, coherence protocols either grant a processor exclusive access to a data item before it writes that item by invalidating other copies, or update all cached copies of a data item whenever that item is written. Supporting the correct semantics of hardware coherence indeed comes at the expense of considerable complexity and power consumption.

These functionalities of hardware cache coherence often exceed the needs of scalable parallel applications, which typically comprise largely independent tasks. Accelerated applications often have no inter-block communication, for data being either read-only or private to a thread block throughout kernel execution. Existing hardware [14, 15] and previous work [3] take advantage of these workload characteristics to forgo cache coherence and its attendant complexities in accelerated systems and their programming models.

Independence notwithstanding, GPU workloads still share data, especially at the input side, between compute units, as demonstrated in Section 2.2. It is necessary for the L1 caches to have some form of coordination to avoid pulling shared data repeatedly from the DRAM. The latency-tolerant design of a GPU transforms caches from being tools to reduce memory access latency into tools to conserve memory bandwidth.

Existing GPU architectures capture inter-processor sharing either by buffering and combing requests extensively in the memory controller, or using a non-inclusive L2 cache. As an alternative to the latter, Tarjan and Skadron proposed using a sharing tracker to identify L1 caches that share a copy of a cache line and thus can satisfy remote misses to it [3]. Similar to Tarjan and Skadron's, we use a directory-like sharing tracker whose tags match those of L1 cache lines, but are instead associated with a list of compute units that have those lines of data cached. The sharing tracker is especially effective due to the large aggregate capacity of the L1 caches. Both mechanisms for capturing inter-processor locality, the L2 cache or direct transfer between L1 caches, have their advantages. In this work, we compare each fundamental design approach individually with the other and with their combination in terms of performance, area and energy.

The sharing tracker is effectively a simplified cache coherence directory that retains only the elements of functionality constructive for a GPU platform. The relaxations of GPU programming models void the need to track the states of shared cache lines (exclusive or shared for example). Moreover, neither is it necessary to track all cache lines, nor all copies of a cache line since a compute-unit is not required to have exclusive access to a line on a write. The sharing tracker enables an optimization opportunity for compute units to supply pieces of data from their L1 caches to other compute units, rather than dictate a requirement for compute units to either keep their data up-to-date or invalidate it.

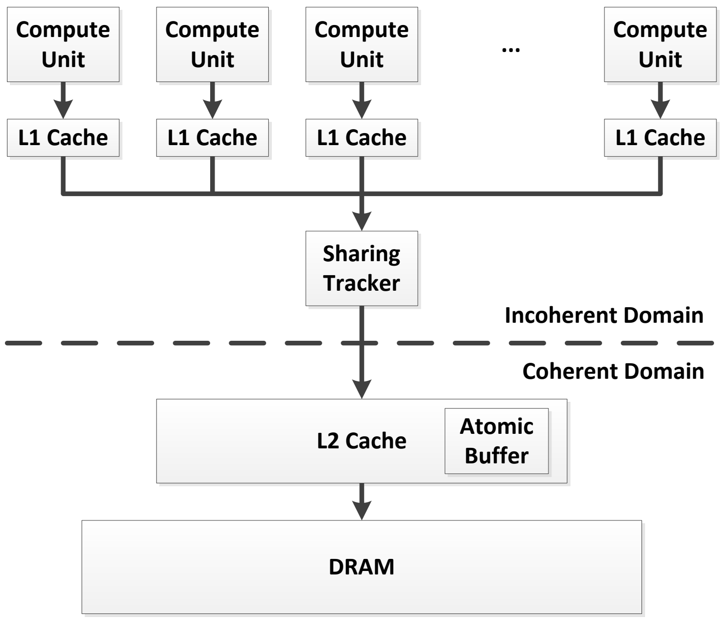


Figure 3.1: Proposed GPU Memory System Design

3.2 Memory Coherence in a Mixed-Coherence System

Our overall proposed design is shown in Figure 3.1. The memory system is divided into two domains: a coherent one comprising the DRAM and L2 cache, and an incoherent one comprising the L1 caches and sharing tracker. Incoherent memory requests access all levels of the hierarchy, while coherent requests bypass the incoherent levels to the coherent domain. The L1 caches are write-through, while the L2 cache is write-back. The L2 cache is neither exclusive nor inclusive, and the size of an L1 cache line is four times that of an L2 cache line. Therefore, each L1 cache line fetched from or written back to the coherent domain is broken into four separate requests. This configuration of cache line sizes strikes a balance between the need for an L1 cache to efficiently serve the wide vector accesses of a warp, and the need for the L2 cache to efficiently use its space for the narrow accesses of coherent operations.

Incoherent memory requests look up the L1 cache first and those that hit do not incur further memory traffic. In particular, no cache coherence or write-through traffic is generated. Otherwise, incoherent accesses go to the sharing tracker upon an L1 cache miss, and finally to the coherent memory domain on sharing tracker misses.

As described in Chapter 2, current GPU programming languages guarantee that all accesses performed by a thread block are visible to all subsequent accesses from the same thread block, but not to accesses from other thread blocks. This lack of coherence guarantees between thread blocks, as opposed to the guaranteed consistency of memory accesses within a thread block, implies that a compute unit's L1 cache may not reflect a consistent view of memory to other compute units; i.e. the most recent writes visible to them. Consequently, it is not always valid for an L1 cache to satisfy a memory request from the L1 cache of another compute unit lest it violates the memory consistency to that compute unit. Tarjan and Skadron's sharing tracker overlooks this element of memory semantics, and thus does not fully maintain the memory consistency guarantees of current GPU programming languages. Consider for example a scenario where compute units P and Q share a line in their L1 caches. Now suppose that Q writes to the line and then evicts it, while the sharing tracker continues to point to the non-updated line in P's L1 cache. If Q requests the line again, the sharing tracker may direct that request to P, and so Q ends up with stale data rather than the data it wrote to the line before it was evicted.

To solve this problem and preserve the consistency guarantees of GPU programming languages, we incorporate selective and full invalidations of cache lines and sharing tracker entries into our design. As far as the visibility of a compute unit's writes to its subsequent accesses is concerned, invalidating a sharing tracker's entry on any write to the corresponding line ensures that any possible future source of the written line listed in sharing tracker will have read the new data from the coherent memory domain. Note that cache-invalidation messages, the primary bottleneck in coherent memory architectures, are not necessary because other compute units are allowed to have a stale copy of the line as long as they do not supply it to the compute units that wrote it.

Similar to writes, coherent operations invalidate the relevant entries in the sharing tracker before accessing the coherent memory domain. However, to maintain coherence among the coherent and incoherent memory operations from the same compute unit, coherent operations also invalidate the corresponding lines in the L1 cache of that compute unit. Coherent memory operations are allocated space only in the L2 cache, and since incoherent operations to the same lines are unusual, doing so has little performance impact.

Invalidation messages to the other incoherent caches are again unnecessary since coherent operations are visible only to later coherent operations as noted in Chapter 2, and thus other compute units are allowed to access stale data incoherently.

Being incoherent, global synchronization events flush all L1 caches, and the sharing tracker, to ensure that all data is globally visible. Global synchronization is generally initiated by the host after a kernel completes, and does not occur during kernel execution in any of our benchmarks.

3.3 Memory Coalescing and Selective Cache Bypassing

The GPU memory system is optimized for coalesced, incoherent accesses, which constitute the majority of memory accesses in typical GPU workloads. It is designed to utilize the L1 caches to achieve high memory throughput for this class of memory requests.

For strided or scattered memory accesses, however, bringing a line into the L1 cache only to consume one or a few words thereof and then evict as other threads contend for cache space is wasteful of both cache space and energy. An L2 cache with four times as many lines as the L1 caches combined, assuming its size is equal to the aggregate L1 cache capacity, is a better fit for such access patterns.

We therefore propose a cache fill policy adaptive to contentious memory access patterns. To keep the hardware implementation simple, we use the number of unique L1 cache lines touched by a single dynamic warp instruction as a metric for cache contention. At runtime, a compute unit calculates the number of L1 cache lines it has available to each active warp by simply dividing the number of lines in its L1 cache by the number of warps scheduled to it. If a vector instruction touches less cache lines than those available to a warp, we deem cache contention small enough to bring the requested data into the L1 cache. Otherwise, the instruction bypasses the L1 cache to the coherent memory domain. Such instructions are allocated lines in the L2 cache. Coalesced accesses hence generate full L1 cache line requests, whereas scattered accesses that bypass the L1 caches generate only the L2 cache line requests necessary to satisfy them, thereby saving both memory bandwidth and energy.

With a sharing tracker, both the L1 caches and L2 cache facilitate sharing, and there is little advantage to requiring data to occupy both caches levels. Opting for the exclusivity between the two cache levels therein allows the L2 cache to degenerate into a small victim buffer for the lines evicted from the last L1 cache sharing them, and a coherence point for atomic operations. This reduces the size requirement for the L2 cache and improves energy and space efficiency of the overall design.

3.4 High-Throughput Atomic Operations

The throughput of atomic operations on NVIDIA's latest GPUs has been substantially improved, and for a common address it is one atomic operation per clock cycle. With the atomic operation throughput to independent addresses also significantly accelerated, atomic operations can often be processed at rates similar to generic load operations [2].

Our baseline design incorporates a small buffer for atomic operations into each memory controller or L2 cache bank to exclusively hold the last memory line accessed atomically. If an atomic operation does not hit in the buffer, the existing line is evicted back into the L2 cache to be replaced by the new line accessed by the atomic operation. Such an atomic buffer results in the kinds of atomic throughput properties of the latest NVIDIA GPUs.

3.5 Decision and Access Flow Summarized

Figure 3.2 illustrates how memory requests proceed through our proposed memory architecture. Incoherent requests are combined by the coalescing unit into L1 cache line transactions and sent to the cache. Upon a cache miss, only requests that do not contend excessively for cache lines are allocated in the L1 cache, however. Evictions from the L1 cache are reported to the sharing tracker to remove the evicting cache from the list of sharers for the evicted lines.

The L1 cache directly satisfies the transactions that hit, while a miss results in a message to the sharing tracker. A remote L1 cache access is initiated on a sharing tracker hit, and the data is forwarded to the requesting cache.

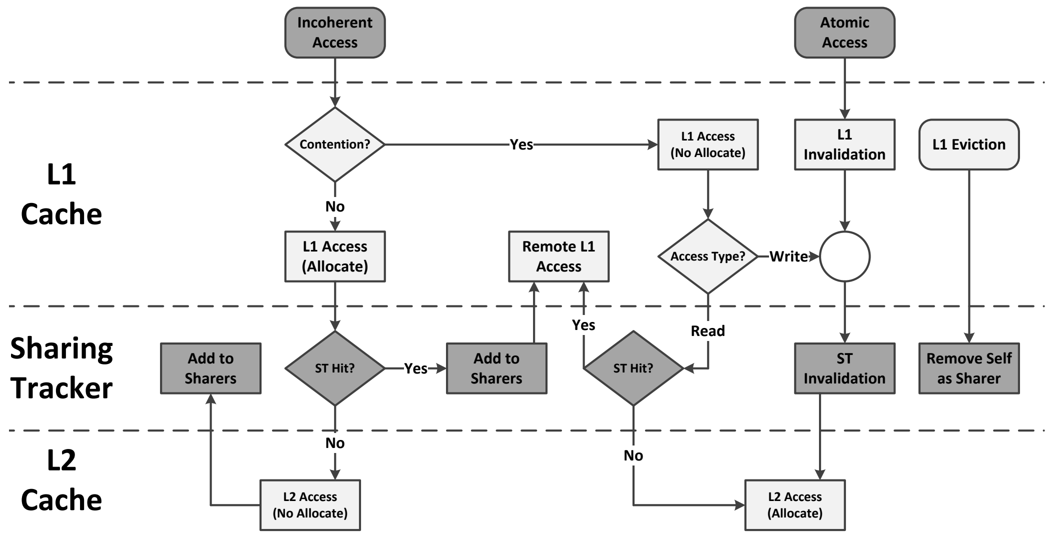


Figure 3.2: Complete Decision and Access Flow Graph for the Proposed Hybrid Shared/Distributed Cache System for a Mixture of Coherent and Incoherent Accesses

A sharing tracker miss, on the other hand, indicates the inability to find any L1 cache to satisfy the request, resulting in a message to the L2 cache. The message is broken into L2 cache line transactions that may individually be satisfied in the cache or from the DRAM. Contentious requests, designated to bypass the L1 cache, are allocated in the L2 cache on a miss. Conversely, non-contentious requests, allocated in the L1 cache, mark the requesting compute unit as a sharer of the corresponding lines in the sharing tracker when the complete lines are fetched.

As discussed in Section 3.2, the L1 caches write-through to the coherent memory domain, invalidating the sharing tracker entries for the corresponding lines along the way to retain the visibility of the writes from a particular compute unit to future reads from the same compute unit. Evicted clean lines are simply reported to the sharing tracker to purge the sharers list. Atomic and other coherent operations bypass the incoherent memory system, invalidating the L1 cache line and sharing tracker entry, and complete directly in the atomic buffer of the L2 cache.

CHAPTER 4

EXPERIMENTAL METHODOLOGY AND DESIGN EVALUATION

4.1 Simulation Framework

We use a trace-driven simulator to model the behavior of a GPU memory system. Traces are collected for all global memory and synchronization instructions by running the device code using a GPUOcelot plugin [16]. Traces also include the block and thread identifiers for each dynamic instruction.

The framework is structured into separate phases, the first of which simulates the behavior of compute units in a GPU. Based on their IDs, thread blocks are scheduled in a round-robin fashion to compute units from a *pending-block queue*. Compute units fetch thread blocks from this queue until they run out of hardware resources to schedule more blocks. During this stage, the simulator decomposes the thread blocks scheduled to a compute unit into warps, which are inserted into the *run queue* of that compute unit.

The execution of each compute unit is simulated by reading one trace entry from each warp in the run queue in a round-robin manner. A warp is retired from the run queue when all of its trace entries are consumed, and the simulation proceeds with the remaining warps in the run queue. When all warps of a thread block are retired, the compute unit fetches a new thread block from the pending-block queue, and the process is repeated until all thread blocks are simulated. The output of the first phase of the simulation is a *memory queue* for each L1 cache containing one or more coalesced line requests for each dynamic memory instruction in the trace.

The second phase of the simulation tracks the contents of the first level of the cache hierarchy. Each associated with a particular compute unit, the L1 caches read the requests from the output memory queue of their compute units in sequence. Whenever an L1 cache misses, a memory request is pushed into a memory queue to the higher levels of the memory hierarchy.

The higher levels of the memory hierarchy, the sharing tracker or L2 cache for example, are simulated in a similar way, but taking into consideration their sharing among multiple compute units. The simulator interleaves the requests from the output queues of the lower levels of the memory hierarchy, and the simulation ends when all requests are consumed. Requests that hit in the sharing tracker, and thus can be satisfied by an L1 cache, are not propagated further. Otherwise, they are pushed to the L2 cache.

While our simulation methodology does not model real hardware timings, the more simplistic approach allows us to run relatively long simulations for entire kernels, and using multiple design configurations nonexistent in real hardware.

Real GPUs, even those without general-purposes caches, have various caching mechanisms which workloads can exploit under certain restricted circumstances. Having such a variety of caching mechanisms available to general compute workload may be of benefit, but we consider those beyond the scope of this work. We would rather reexamine the design of a GPU memory system to better support compute workloads from the ground up, and leave the evaluation of specialized structures like the constant cache for future work.

4.2 Simulation Parameters and Benchmark Suite

Our simulated system is described in Table 4.1. As per NVIDIA's latest, we assume a GPU consisting of 16 compute-units, each of which has a 64KB private L1 cache, 16KB scratchpad, and 64K registers. The L1 caches are write-through, 4-way set-associative with 128B lines. The compute-units share an 8-way set-associative L2 cache comprising 8 banks and 32 byte lines, and an 8-way set-associative sharing tracker. With 1024 sets, the sharing tracker has enough entries to cover all 8K unique lines in the aggregate L1 cache space. Each sharing tracker entry has a 16-bit mask to track all L1 caches sharing a copy of the corresponding line. We explore variants of this design in terms of L2 cache size, sharing tracker bitmasks length, and number of sharing trackers in our experimental evaluation.

We chose to use the Parboil benchmark suite [17] for our analysis and experiments. We analyzed the access patterns and locality optimizations repre-

Table 4.1: Details of the Simulated System

GPU	
No. of Compute Units	16
Warp Size	32
Compute Unit	
Max. Warps	64
Register File Size	64K registers
Shared Memory Size	16KB
L1 Cache	
Size	64KB
Line Size	128B
Associativity	4
No. of Banks	1
Sharing Tracker	
No. of Sets	1024
Associativity	8
No. of Banks	8
No. of Sharers per Line	1, 16
L2 Cache	
Size	{128, 256, 512}KB, 1MB
Line Size	32B
Associativity	8
No. of Banks	8

Table 4.2: Profiles of Parboil Benchmarks

Benchmark	Kernel	Shared Data			Partitioned Data		Atomic	Performance
		Broadcast	Boundary	Dynamic	Coalesced	Non-coalesced	Operations	Bound
BFS				X	X	X	X	Bandwidth
CutCP		X			X			Balanced
Gridding	many	X		X				Balanced
Histo	prescan	X			X		X	Bandwidth
	inter				X			Bandwidth
	main	X		X			X	Bandwidth
	final				X			Bandwidth
LBM	AoS					X		Bandwidth
	SoA				X			Bandwidth
MRIQ	computePhiMag	X			X			Compute
	computeQ	X			X			Compute
SGEMM		X			X			Compute
SPMV				X	X			Bandwidth
Stencil			X		X			Bandwidth

sented in the benchmarks, and our findings are summarized in Table 4.2. Table 4.2 also notes which benchmarks use atomic operations to global memory that interact with the cache hierarchy, and categorizes each benchmark implementation as bound by compute throughput or memory bandwidth after optimization, to point out that some benchmarks may be more performance-sensitive to the behavior of the memory system than others.

Shared data is nearly ubiquitous in our benchmarks, with LBM being the exception. Broadcast sharing is the most common sharing pattern across the board, but boundary and dynamic sharing are also present. Partitioned data is also nearly ubiquitous, with perfectly coalesced partitioning being typical for these reasonably well-optimized benchmarks. Still, BFS and SPMV have misaligned partitions, the former from appending dynamically sized blocks to a queue, and the latter from compressed data of unknown alignments.

4.3 Experimental Results

To evaluate the overall impact of our proposed design on a many-core GPU, we first gauge the benefits of adding a sharing tracker to a conventional baseline GPU design. Figure 4.1 in part shows that for our workloads, augmenting L2 caches of different sizes with a sharing tracker provides DRAM demand management superior to that of a shared L2 cache individually. By essentially increasing the exclusivity between the L1 and L2 caches, the sharing tracker effectively increases the aggregate system cache capacity. The benefits of the sharing tracker are demonstrated best in systems with L2 caches relatively too small to fully capture inter-processor locality (up to 9% average DRAM data demand reduction for 128KB L2 cache). In such systems, a sharing tracker allows for a DRAM data demand comparable to that of systems with an L2 cache of at least twice the size for benchmarks like BFS, CutCP, Histo, MRIQ, SPMV, and Stencil. The one notable exception is LBM, whose large cache footprint and limited inter-processor locality result in minimal benefits from combining the L2 cache with the sharing tracker.

There are two elements to our proposed caching policy: bypassing the L1 caches on contentious memory accesses, and bypassing the L2 cache on coalesced memory accesses. The first element of our caching policy, pertinent to the L1 caches, offers LBM a significant reduction in DRAM data demand.

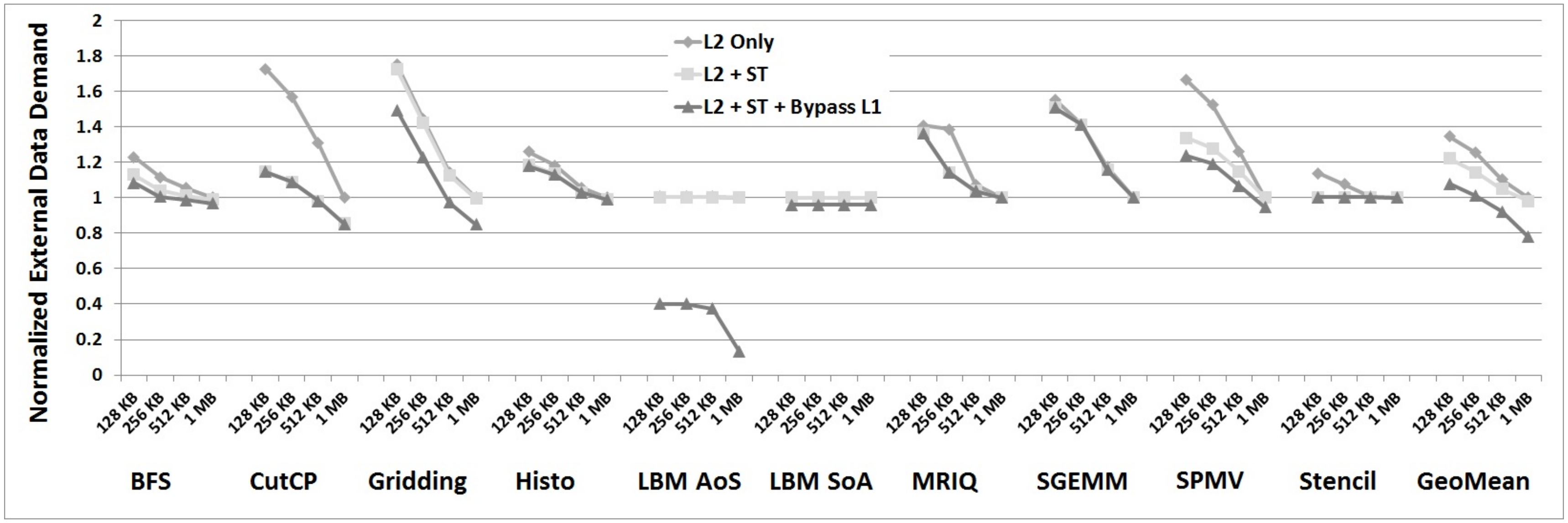


Figure 4.1: Normalized DRAM Data Demand for Different L2 Cache Sizes in a Conventional Baseline GPU Design, Our Proposed Design with a Sharing Tracker, and Our Proposed Design with a Sharing Tracker When L1 Caches are Bypassed for Non-coalesced Memory Accesses (All Results are Normalized to the Baseline Design with 1MB L2 Cache)

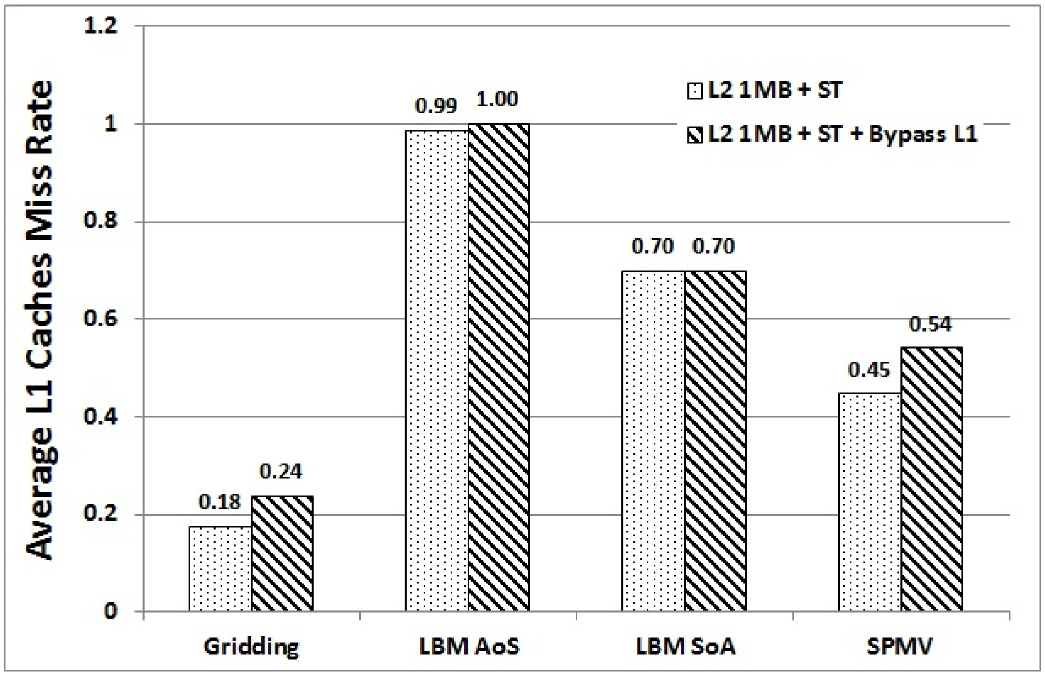


Figure 4.2: Average First Cache Level Miss Rate for Benchmarks Sensitive to Bypassing the L1 Caches for Non-coalesced Accesses

In general, benchmarks like Gridding, LBM, and SPMV with poorly coalesced or dynamic access patterns often contend aggressively for cache lines. Therefore, having those benchmarks bypass the L1 cache when they do so and instead service their requests through shorter L2 lines allows for a more efficient utilization of DRAM traffic, compared to both the baseline GPU design and the proposed design with only the sharing tracker, as demonstrated in Figure 4.1 (22% DRAM data demand reduction on average for 1MB L2 cache). So much so is the efficient utilization of DRAM traffic that the L2 cache is able to capture more inter-processor locality beyond that attainable without our caching policy, as alluded to by the drop of DRAM data demand of LBM using a 1MB L2 cache. Figure 4.2 shows that bypassing L1 caches upon contention as designated by our caching policy results in only a small increase in the average L1 cache miss rate, and thus a small penalty for memory access latency that can be well-hidden on a throughput-oriented architecture. It goes without saying that benchmarks with mostly regular access patterns, such as CutCP, SGEMM, and Stencil, exhibit no change of behavior under the first element of our caching policy.

The second element of our caching policy, bypassing the L2 cache on co-

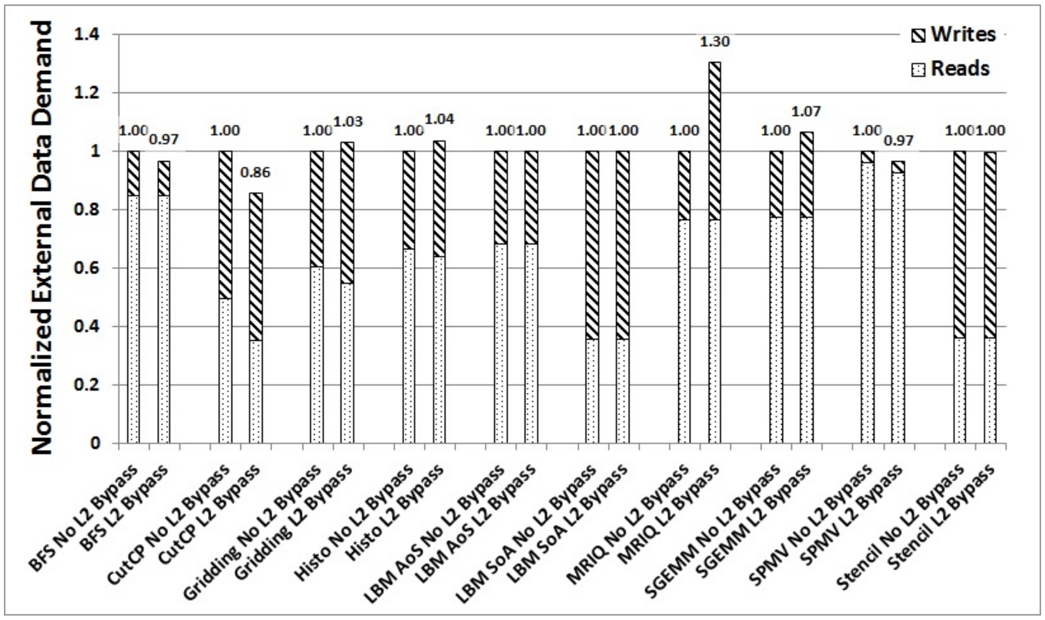


Figure 4.3: Normalized DRAM Data Demand When the L2 Cache is Bypassed for Coalesced Accesses (Both Configurations have 1MB L2 Cache, Sharing Tracker, and Bypass L1 Caches for Non-coalesced Accesses)

alesced memory accesses, further increases the exclusivity of the two cache levels, offering some extra potential for a better utilization of the cache space. Figure 4.3 elicits this potential for the BFS, CutCP, and SPMV benchmarks, yet it also exposes an issue with memory writes for Gridding, Histo, MRIQ, and SGEMM that outweighs all gains from memory reads. The problem stems from using read-write arrays in each of these benchmarks, coalesced data read from which bypass the L2 cache. When those arrays are later written through the L1 caches, therefore, they incur write misses in the L2 cache and end up being pulled again from the DRAM, possibly superfluously if they are completely overwritten. This is in part a shortcoming of our simulator, which does not keep track of dirty cache lines at the byte granularity. As a result, we exclude this element of our caching policy from our evaluation, and leave full analysis for future work. With this room for improvement in mind, our proposed design with the sharing tracker and the first element of the caching policy still furnishes significant reductions in DRAM data demand compared to a conventional baseline GPU design as shown in Figure 4.1.

Given the conventional block scheduling policies on a GPU, only a limited subset of compute units tend to share data. Tracking all L1 caches which have a copy of a particular cache line may thus be wasteful when only a small

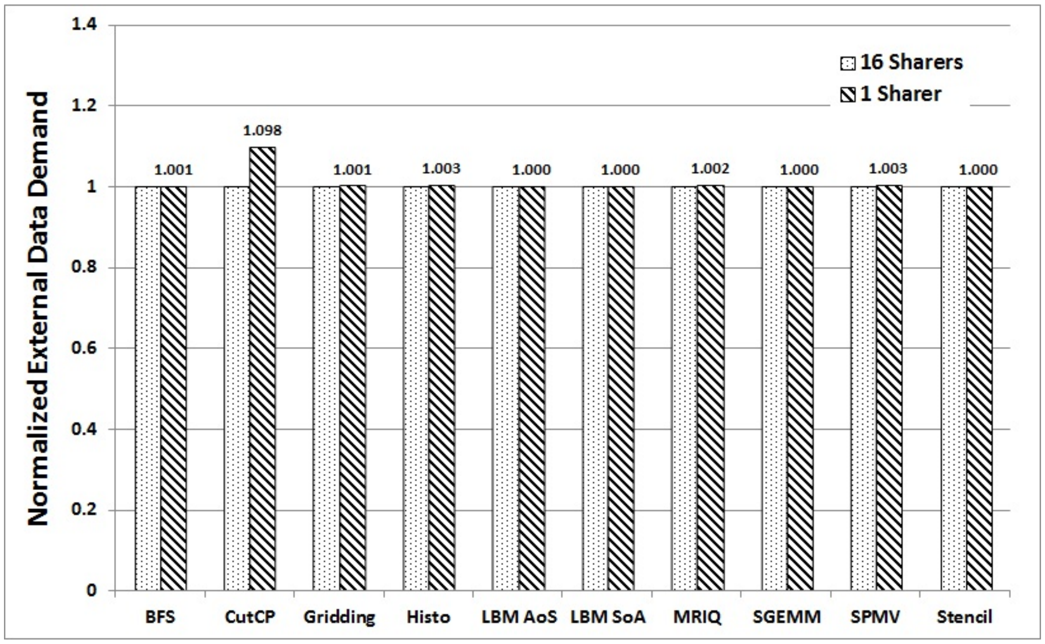


Figure 4.4: Normalized DRAM Data Demand When Tracking Different Number of Sharers (Both Configurations have 1MB L2 Cache, Sharing Tracker, and Bypass L1 Caches for Non-coalesced Accesses)

number is typically sufficient. Figure 4.4 compares the extremes of maintaining either a full map of sharers or a single-element entry. Clearly, tracking a single sharer is sufficient for all practical purposes, with the exception of CutCP for which at least a subset of sharers is required to take full advantage of inter-processor locality. It remains to say that keeping track of a subset of sharers, as opposed to one, allows the sharing tracker to alternate between different L1 caches upon responding to successive requests to a particular cache line for load balance.

As GPUs increase the number of compute-units on chip, the latency of accessing a sharing tracker will grow, potentially becoming a bottleneck. One way to deal with this scalability problem is by replicating resources, distributing multiple copies of the global sharing tracker across the GPU. Alternatively, we can leverage the GPU execution model with a distributed design of multiple smaller sharing trackers, where each covers a subset of compute units. The distributed sharing trackers can interrogate each other, but we keep them isolated in our simulations for simplicity. Apart from CutCP and SGEMM in which thread blocks scheduled to all compute units share data, Figure 4.5 shows that such an arrangement achieves the same DRAM

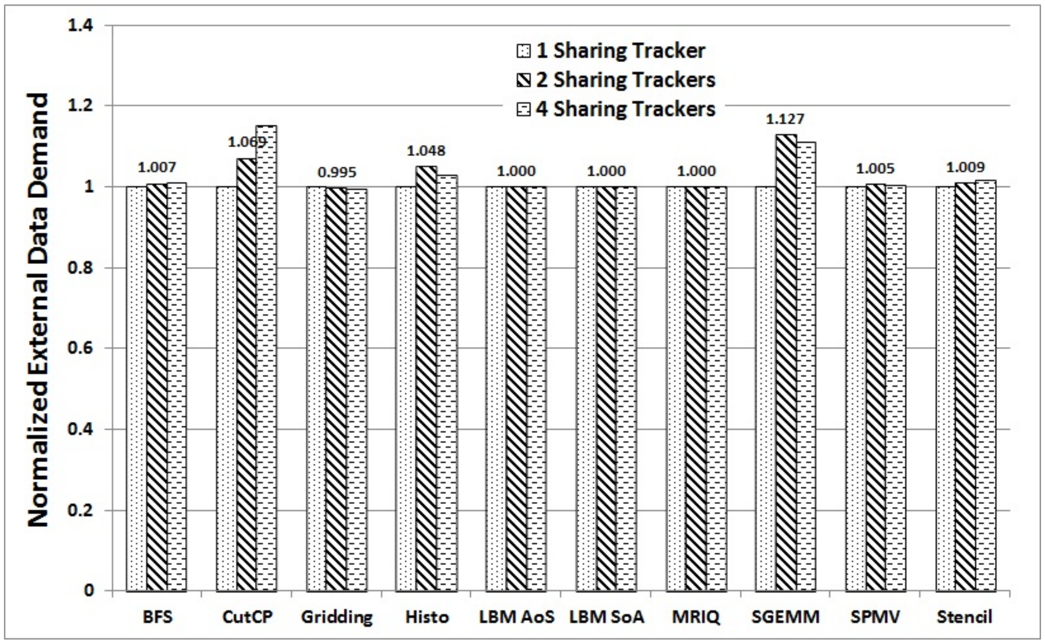


Figure 4.5: Normalized DRAM Data Demand with Distributed Sharing Trackers (All Configurations have 1MB L2 Cache, Sharing Tracker, and Bypass L1 Caches for Non-coalesced Accesses)

demand reduction advantages as a design with a global sharing tracker for most of our benchmarks at a smaller area overhead. While we leave the distributed sharing trackers with a collective number of sets equal to that in the global sharing tracker, we shorten the sharers lists in each of them to match the smaller set of possible sharers they each cover, effectively reducing the total area overhead.

Table 4.3 shows the models for area, power, and energy for the cache and sharing tracker structures of the system in a 22 nm process modeled by

Table 4.3: Power, Area, and Energy Models for the Components of a GPU Memory System in a Conventional Baseline Design and Our Proposed Design

Structure	Capacity	Line Size	Assoc.	Ports	Aggregate Chip Area	Aggregate Leakage	Line Read Energy	Line Write Energy
L1 (Baseline)	64KB	128B	4-way	1R/W, 1W	5.76 mm^2	170.38 mW	30.5 pJ	44.4 pJ
L1 (Proposed)	64KB	128B	4-way	2-R/W	5.79 mm^2	183.11 mW	30.5 pJ	44.5 pJ
1 ST	16KB	2B	8-way	1-R/W	0.07 mm^2	13.35 mW	2.3 pJ	3.9 pJ
2 STs	2×8KB	2B	8-way	1-R/W	0.05 mm^2	12.72 mW	1.7 pJ	2.9 pJ
4 STs	4×4KB	2B	8-way	1-R/W	0.07 mm^2	12.30 mW	1.2 pJ	2.1 pJ
L2 Cache	128KB	32B	8-way	1-R/W	0.46 mm^2	59.82 mW	62.1 pJ	61.5 pJ
	256KB	32B	8-way	1-R/W	0.56 mm^2	81.26 mW	67.7 pJ	67.5 pJ
	512KB	32B	8-way	1-R/W	0.81 mm^2	124.27 mW	78.5 pJ	77.2 pJ
	1MB	32B	8-way	1-R/W	1.25 mm^2	211.24 mW	99.7 pJ	96.1 pJ
Interconnect	N/A	N/A	N/A	N/A	N/A	N/A	0.144 pJ B^{-1} mm^{-1}	
DRAM (GDDR5)	N/A	32B	N/A	N/A	N/A	N/A	4480 pJ	

CACTI 6.5, which combines many of the features of CACTI 5 and 6 [18, 19]. Our proposed sharing tracker proves to be economical, both energy- and area-wise. DRAM energy costs are a best-case calculation from industry specifications [20] assuming prefect row locality for a fully saturated read-only access stream. Interconnect message costs are calculated based on data for a 0.1 V low-swing interconnect from the DARPA exascale report [21]. We exclude the energy costs of the memory controller for its design is beyond the scope of this work and would only increase the DRAM access cost further.

Based on this energy cost data, we calculate the memory system energy cost on a variety of memory system configurations as in Figure 4.6. Overall, DRAM accesses and L1 hits are the dominant components of the energy costs in almost every benchmark and configuration. Although generously estimated, the increased interconnect energy cost due to the sharing tracker is all but negligible. Therefore, the benchmarks that show significant DRAM demand improvement on the combined L2 cache and sharing tracker configuration, like CutCP, Gridding, LBM, and SPMV, also demonstrate reduced total energy cost as expected. Interestingly though, some benchmarks with little change in DRAM data demand due to the sharing tracker, BFS and Histo in particular, and SGEMM to a lesser extent, still show meaningful energy reductions due to a smaller L2 access component. For the remaining benchmarks, MRIQ and Stencil, the memory access profile with a 1MB L2 cache hardly changes in the presence of a sharing tracker, and thus the memory system energy cost is virtually the same for all configurations. However, for smaller L2 cache sizes with which a sharing tracker is shown in Figure 4.1 to be more significant for reducing DRAM data demand, the total energy cost is proportionally smaller although not demonstrated in Figure 4.6.

To summarize, DRAM access is the first-order factor in the energy cost of the memory system that utterly dominates the costs of adding a sharing tracker to the system and the resultant increase in interconnect traffic. By making a more efficient use of the two cache levels in the system, our proposed sharing tracker and caching policy eliminate some of the redundant DRAM traffic and reduce the total system energy cost (28% total energy reduction on average for 1MB L2 cache). These benefits are magnified whenever the cache space is insufficient for the purposes of the workloads hosted by the system.

The temporal aspect of performance for our proposed design obviously de-

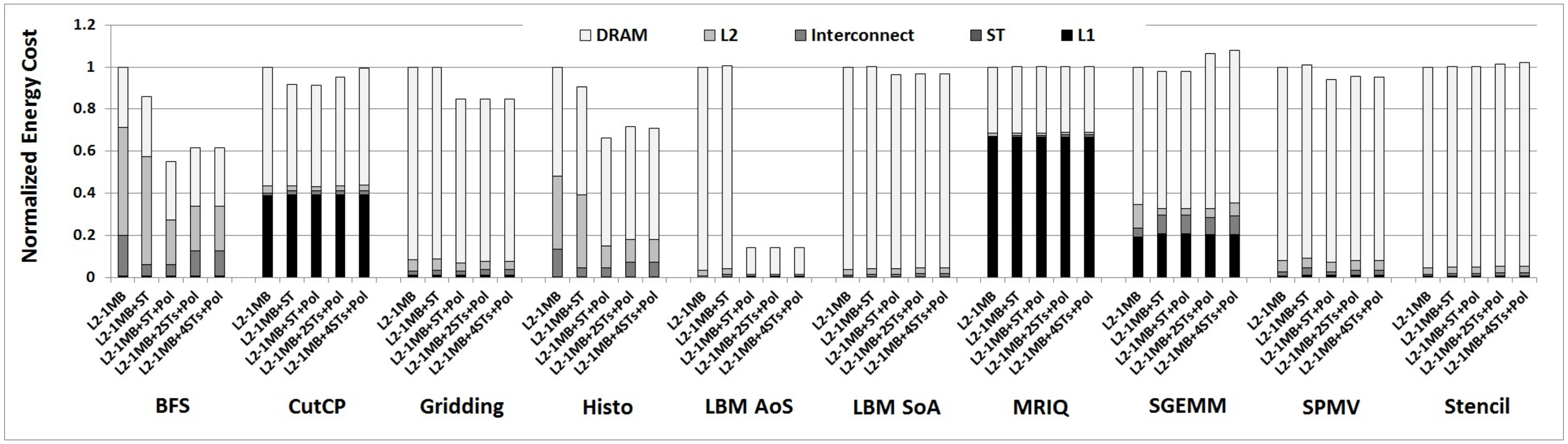


Figure 4.6: Energy Costs for Different Memory System Configurations (All Results are Normalized to the Design with an L2 Cache Only; *Pol* Means Bypassing L1 Caches on Non-coalesced Accesses)

pends on the relative latencies of remote L1 cache accesses, L2 cache accesses, and DRAM accesses. Our results suggest that the overhead of accessing the sharing tracker and remote L1 caches is likely to be hidden completely in compute-bound benchmarks, such as CutCP, MRIQ, and SGEMM, even for remote L1 cache access latencies drastically biased against our design. As for memory-bound benchmarks, LBM, SPMV, and Stencil for example, the execution times are likely to be dominated by the overhead of DRAM accesses in systems with relatively small L2 caches, and we expect our proposed design to consistently outperform the conventional baseline architecture. For systems with large L2 caches, however, the on-chip network bandwidth and latency must be kept under control in order to alleviate the overhead of accessing data from remote L1 caches via the sharing tracker.

CHAPTER 5

RELATED WORK AND CONCLUSION

A huge body of work has explored conventional hardware cache coherence organizations [22], and the optimizations that can be built on the top of which. For example, Chang and Sohi use cooperative caching to share the resources of private caches on a single chip [23]. Their Central Coherence Engine embodies a full coherence directory and engine. Herrero et al. extend the cooperative caching framework to large scale chip multiprocessors by replacing the Central Coherence Engine with Distributed Coherence Engines spread across the nodes to improve scalability [24].

A number of non-coherent shared-memory architectures have been proposed or developed [25, 14, 26, 15] as a more scalable and cost-effective alternative to cache-coherent shared-memory architectures. Coherence in such systems must be managed in software [22]. Our software coherence model, based on the software support available on Fermi GPUs [14], is similar to the task-centric memory model [25].

Caches can exploit relaxed hardware coherence requirements by tracking less information, generating fewer coherence messages, and/or not waiting for messages. Tarjan and Skadron propose a directory-like structure, dubbed a sharing tracker, to share data between a GPU's L1 caches [3] at a lower cost than larger caches. The original sharing tracker is designed for workloads with a mixture of read-shared and private data, differentiated at the granularity of cache lines, and it does not address false sharing or coherent memory operations. The Rigel architecture [25] adds fine-grained dirty bits to avoid the problem of false sharing.

Coherence decoupling in cache-coherent architectures uses data speculatively on a cache hit instead of waiting for memory coherence messages [27]. Coherence decoupling thereby reduces the average memory latency, but it is unlikely to save energy or area since it does not reduce coherence traffic or cache pressure.

A large body of work has proposed ways to combine the speed and hardware simplicity of small private caches to the sharing benefits of large shared caches, without assuming any prior knowledge of shared and private data in general. Such approaches include coordinating multiple private caches to emulate the increased sharing and capacity of a shared cache [23], and dynamically migrating lines between the banks of a shared cache to draw near the latency of a private cache [27, 28].

Another line of research fine-tunes the cache filling and eviction policies on GPUs to better utilize the available cache memory. Cache bypassing [29] has been proposed to avoid polluting the L2 cache with private data [30]. Jia et al. observe that non-coalesced accesses to the L1 cache can be slower than non-cached accesses, and propose bypassing the L1 cache to alleviate this problem, based on the estimated memory traffic of an instruction [4].

In this era of data-intensive computing, the principles of designing for data-parallelism, throughput, and latency tolerance make GPUs an ideal platform for drawing general lessons for the many-core architectures of the future. For this thesis, we propose a new memory system for GPU architectures based on an analysis of the characteristics of their workloads, and the features of their programming models. We show that for systems where the aggregate L1 cache capacity outweighs the L2 cache, an economical sharing tracker widens the temporal window of the cache hierarchy enough to capture as much inter-processor locality as an L2 cache of twice the size in a baseline system. We also show that the L2 cache with its shorter lines is better poised with an adaptive caching policy at hand to serve non-coalesced memory requests that contend aggressively for cache lines.

Our novel design with its sharing tracker and smart cache allocation policy exposes new tradeoffs for future work to assess more thoroughly. In the light of these conclusions, and given their applicability to many-core architectures in general, future work can continue to explore the roles of each level in the cache hierarchy within these next-generation architectures and for their emerging applications. It is worth considering whether the traditional role of a last-level cache is still relevant, and thus justifies raising its share of the transistor budget, or whether transistors are better spent on larger private first-level caches to which a small last-level cache plays an ancillary role.

REFERENCES

[1] [Online]. Available: http://gpgpu.org/

[2] "NVIDIA's next generation cuda compute architecture: Kepler gk110," White Paper, NVIDIA Corporation, 2012.

[3] D. Tarjan and K. Skadron, "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*, 2010, pp. 1–10.

[4] W. Jia, K. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proceedings of the International Conference on Supercomputing*, 2012, pp. 15–24.

[5] I. Buck, "GPU computing with NVIDIA CUDA," in *SIGGRAPH '07*, 2007, p. 6.

[6] Khronos OpenCL Working Group, *The OpenCL Specification*, 2009.

[7] Microsoft Corporation, "DirectCompute PDC HOL," 2009. [Online]. Available: http://archive.msdn.microsoft.com/directcomputehol

[8] [Online]. Available: http://www.openacc-standard.org/

[9] Microsoft Corporation, "C++ AMP: Language and programming model," January 2012.

[10] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of OpenCL programs," in *Fifth International Workshop on Automatic Performance Tuning*, 2010.

[11] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, August 1990.

[12] I.-J. Sung, G. Liu, and W.-M. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *Proceedings of the IEEE Innovative Parallel Computing Conference*, 2012, pp. 1–11.

[13] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface, Second Edition.* Morgan Kaufmann, 1997.

[14] "NVIDIA's next generation compute architecture: Fermi," White Paper, NVIDIA Corporation, 2009.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[16] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting GPU compute applications within GPU Ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–9.

[17] J. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W.-m. Hwu, "The Parboil benchmarks," University of Illinois at Urbana-Champaign, Urbana, Tech. Rep. IMPACT-12-01, Mar. 2012.

[18] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Laboratories, Palo Alto, Tech. Rep. HPL-2008-20, 2008.

[19] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0," HP Laboratories, Palo Alto, Tech. Rep. HPL-2009-85, 2009.

[20] Samsung, *1G GDDR5 SGRAM*, 1st ed., Samsung Corporation, Apr. 2009.

[21] "ExaScale computing study: Tehcnology challenges in achieving exascale systems," DARPA, Tech. Rep. TR-2008-13, Sep. 2008.

[22] P. Stenström, "A survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, June 1990.

[23] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *Proceedings of the 33rd annual international symposium on Computer Architecture*, 2006, pp. 264–276.

[24] E. Herrero, J. González, and R. Canal, "Distributed cooperative caching," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08, 2008, pp. 134–143.

[25] J. Kelm, D. Johnson, S. Lumetta, M. Frank, and S. Patel, "A task-centric memory model for scalable accelerator architectures," in *Proceedings of the 19th annual international conference on Supercomputing*, 2009, pp. 77–87.

[26] Y. Paek, A. Navarro, E. Zapata, J. Hoeflinger, and D. Padua, "An advanced compiler framework for non-cache-coherent multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 241–259, 2002.

[27] J. Huh, J. Chang, D. Burger, and G. Sohi, "Coherence decoupling: Making use of incoherence," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 97–106.

[28] Z. Guz, I. Keidar, A. Kolodny, and U. Weiser, "Utilizing shared data in chip multiprocessors with the Nahalal architecture," in *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures*, 2008, pp. 1–10.

[29] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu, "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338–1354, 1999.

[30] H. Choi, J. Ahn, and W. Sung, "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*, 2012, pp. 110–119.