SemCache: Semantics-aware Caching for Efficient GPU Offloading

Nabeel AlSaber and Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{nalsaber, milind}@purdue.edu

ABSTRACT

Recently, GPU libraries have made it easy to improve application performance by offloading computation to the GPU. However, using such libraries introduces the complexity of manually handling explicit data movements between GPU and CPU memory spaces. Unfortunately, when using these libraries with complex applications, it is very difficult to optimize CPU-GPU communication between multiple kernel invocations to avoid redundant communication.

In this paper, we introduce SemCache, a semantics-aware GPU cache that automatically manages CPU-GPU communication and dynamically optimizes communication by eliminating redundant transfers using caching. Its key feature is the use of library semantics to determine the appropriate caching granularity for a given offloaded library (*e.g.*, matrices in BLAS). We applied SemCache to BLAS libraries to provide a GPU drop-in replacement library which handles communications and optimizations automatically. Our caching technique is efficient; it only tracks matrices instead of tracking every memory access at fine granularity. Experimental results show that our system can dramatically reduce redundant communication for real-world computational science application and deliver significant performance improvements, beating GPU-based implementations like CULA [9] and CUBLAS [18].

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.4.2 [**Operating Systems**]: Storage Management—*Distributed Memories*

Keywords

GPU offloading; GPGPU; Communication optimization

1. INTRODUCTION

With the rise of general purpose GPU (GPGPU) programming, programmers have increasingly turned to the GPU as a cheap (both in cost and in energy) means of boosting the performance of their application. Recent years have shown that GPU implementations of linear algebra kernels [9, 18, 21, 23], graph algorithms [15], stencil codes [5], among others, can vastly outperform CPU implementations. However, while these results hold for individual kernels, it is still unclear how best to leverage GPUs to improve the performance of applications.

Consider how a developer of a large-scale computational science application might attempt to use GPU resources. One option would be to try and run the entire application on the GPU (in a sense, taking the same approach as kernel developers, but applying it to the entire program). Clearly, such a tactic is impractical. Not only are GPU programming models (*e.g.*, CUDA [17]) somewhat cumbersome, but not all portions of an application are well-suited to running on a GPU; while computation-heavy portions of an application (*e.g.*, loops performing linear algebra operations) may perform well, more control-heavy portions that do not have much parallelism may instead run *slower* on the GPU than on the CPU. Hence, our programmer may invest significant time in porting an application to the GPU only to find her efforts wasted as GPU performance fails to meet expectations.

Instead of porting the entire application to the GPU, the programmer might instead adopt a *heterogeneous* approach: portions of the application well-suited to GPU execution will be offloaded, while the remainder of the application will be run on the CPU. The programmer may attempt to use systems designed to facilitate such heterogeneous scheduling [14, 19], but these approaches require changes to the programming model, and work only for specific domains, necessitating significant porting work. Alternately, the programmer may offload portions of their code to the GPU, targeting specifically those kernels that are well-suited to GPU execution. Note, however, that a major cost in offloading computations to the GPU is *data movement overhead*: getting data to and from the GPU requires transferring data over a (relatively) slow PCIe bus, and hence data movement consumes a significant portion of the overall time required to perform operations on the GPU. While some data movement is unavoidable, when targeting an application that performs many offloadable operations, *much of this data movement is redundant.*

Consider the simple case of a series of matrix multiply operations, as shown in Figure 1. Each matrix multiply requires that the source matrices be on the GPU and the result matrix be transferred back to the CPU. However, the naïve

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

```
1 GEMM(A, B, C); //C = A * B
2 ...
3 GEMM(B, C, D); //D = B * C
4 ...
5 GEMM(C, D, E); //E = C * D
```

Figure 1: Simple example with repeated matrix multiplication

approach of transferring the sources to the GPU and the results back on every operation results in redundant communication. Some source matrices (*e.g.*, `B`) are transferred to the GPU twice, while other matrices (*e.g.*, `C` in the second operation) are transferred to the GPU even though they were computed on the GPU originally. A better approach is to transfer `B` just once, and use it for both operations, while consuming `C` directly from the GPU for the second operation.

There exist libraries of GPU kernels (*e.g.*, CULA Standard Interface) that attempt to ease the process of offloading computation to the GPU by automatically handling data movement and execution on the GPU. Because these libraries target specific operations (*e.g.*, linear algebra), using them is often as simple as replacing operations in an application with equivalent GPU versions; in fact, because computational applications are often already implemented with libraries such as BLAS [4] and LAPACK [2], CULA implementations of those operations can be used with essentially no modifications to program code and no need for GPU expertise. Unfortunately, such drop-in replacement libraries come with drawbacks. The libraries do not consider the composition of library calls, instead implementing each offloaded operation as a self-contained unit. As a result, the libraries do not consider the possibility of redundant data movement across operations (in other words, they adopt the naïve communication approach described above). *Because each operation is offloaded in isolation, the composition of operations may not be implemented efficiently.*

To correctly minimize data movement and avoid redundancy when offloading computation to the GPU, the composition of offloaded operations must be considered. While lower-level libraries (such as CUBLAS, CULA's "device interface," or MAGMA [23]) give the programmer precise control over data movement (so that, *e.g.*, he can avoid transferring matrix `B` to the GPU twice in the previous example), it is often difficult to reason about which data movement might be redundant and which might be necessary. This is especially true in large, modular applications, where operations might be quite distant from one another both in the code and during execution, and where a single piece of static code may exhibit data redundancy based entirely on when and where the code is invoked during execution (consider a method called from several places in an application that performs several linear algebra operations on matrices passed in as parameters). In such a scenario, any attempt to statically determine whether communication is necessary is doomed to failure; *simply providing low-level control of data movement is not enough to allow eliminating redundant communication.*

What is needed is an *automatic* approach to managing data movement between GPU and CPU that can *dynamically* determine whether data movement is necessary and hence provide drop-in replacements for computational libraries. Such an approach will allow programmers to achieve efficient communication for heterogeneous computing without adopting a new programming model.

1.1 Semantics-aware GPU Caching

In this paper, we propose a *semantics-aware GPU cache* to reduce redundant communication between the CPU and the GPU. At a high level, our software caching approach treats the CPU and GPU memory spaces as two caches, and uses an MSI (modified/shared/invalid) protocol to maintain coherence between them. When a method is called to execute on the GPU, the cache state of the data used by the method is inspected, and data is transferred to the GPU only if it does not already reside there. When data is modified on the CPU, the cache is used to invalidate any corresponding data on the GPU.

This type of software caching has been proposed before for other architectures, such as distributed shared memory systems [1, 7, 10, 16] and accelerator-based systems like the Cell [8, 12]. The key drawback of these prior approaches is that the granularity of caching is fixed (*e.g.*, OS pages for DSM approaches, or fixed line sizes for accelerators), and did not take into account program behavior. As a result, the cache granularity might be too big, resulting in excessive data movement, or too small, resulting in too many cache lookups and more data transfers and hence higher overhead. In contrast, our system adopts a semantics-aware approach: the granularity of our caching is determined by the semantics of the libraries being offloaded to the GPU. For example, when applying our approach to the BLAS library, rather than caching at the granularity of pages, our cache will track data at the granularity of the arrays in the application. Hence the granularity of the cache is dependent not only on the libraries in a program but also on the specific way those libraries are used in an application. By matching our cache granularity to the libraries, we can more efficiently use the space on the GPU (by caching only data that is needed for computations) and reduce caching overheads (by performing fewer cache lookups per library call). We also show how the same caching principles can leverage additional library semantics to not only save data movement, but also eliminate redundant computations.

Crucially, the cache we develop is *generic*: the system itself is not tied to any particular library. Instead, all of the semantic information is provided in the library implementation, allowing the same caching system to be reused for different libraries, in each case providing tuned cache implementations that use the correct granularity for a given library.

1.2 Contributions

This paper makes the following contributions:

- The design and implementation of SemCache, a generic GPU cache that automatically manages CPU-GPU communication and dynamically optimizes communication. It is augmented with semantic information to provide *tuned, library-specific caching.*
- A generalization of this caching solution (akin to memoization) that creates *semantic links* between data on the CPU and GPU, allowing SemCache to automatically eliminate redundant computation and translate between different layouts.

- An annotated GPU BLAS library that provides a *drop in replacement* for existing BLAS libraries that, in conjunction with SemCache, delivers optimized communication between the CPU and GPU.

- Experimental results showing, both for microbenchmarks and a large, real-world computational science application, that SemCache can dramatically reduce redundant communication, and deliver significant performance improvements, beating not only CPU implementations but also GPU-based implementations using existing, tuned libraries.

2. RELATED WORK

There are multiple libraries that optimizes the performance of Linear Algebra Kernels on GPUs. CUBLAS [18] is an implementation of BLAS [4] (Basic Linear Algebra Subprograms) on top of the NVIDIA's CUDA driver that allows access to the computational resources of NVIDIA GPUs. MAGMA [23] and CULA [9] have implemented hybrid GPU accelerated linear algebra routines (LAPACK and BLAS libraries). CULA provides a standard interface that needs no GPU knowledge in addition to the advanced interface. These approaches all focus on individual kernels; across kernels, data management must be handled by the programmer.

SuperMatrix [22] and StarSs [3] have a runtime system to solve dense linear systems on platforms with multiple hardware accelerators. Both systems use software-caching schemes to reduce data transfer cost within a single kernel. Fogue *et al.* ported the PLAPACK library to GPU-accelerated clusters [6]. They require that CPUs compute the diagonal block factorizations while GPUs compute all the remaining operations. They also store all data for a single kernel in GPU memories to reduce communication. Our caching approach is different from these libraries since we optimize communication across kernels (at the application level).

Other programming models are designed to facilitate heterogeneous scheduling. Intel's Merge [13] is a programming model for heterogeneous multi-core systems. Qilin is a generic programming system that can automatically map computations to GPUs and CPUs through off-line trainings [14]. MDR [19] is a performance model-driven runtime for heterogeneous parallel platforms. Such systems try to optimize CPU-GPU communication across the entire program. However, to use them, the programmer must rewrite their application using the specified programming model. In contrast, we are targeting existing large-scale applications, with the goal of optimizing communication without significant programmer effort.

Prior work implemented automatic data management and communication optimization systems for GPUs [7, 10, 11]. Jablin et al. have developed a fully automatic CPU-GPU communication management system (CGCM) [11]. CGCM manages data using a combined run-time and compile-time system without programmer annotations. CGCM requires static analysis (type-inference and alias analysis) because it manages data and optimizes communication at compile-time. The imprecision of static analysis limits CGCM's applicability and performance. In contrast, SemCache uses a more sophisticated run-time system that keeps tracks of data validity status and hence can better optimize communication with less overhead than a static compiler analysis.

DyManD [10] and GMAC [7] attempt to manage communication between the GPU and CPU automatically by adopting distributed shared memory (DSM) techniques [1, 16]. The two systems use the operating system's page-protection mechanism to detect when data needs to be transferred. While these techniques are fully automated, they suffer from two primary limitations compared to SemCache's approach. First, to effectively use page-based systems, the layout of data must be changed to align data with page boundaries, avoid false sharing, etc., which precludes using such systems with existing custom allocators. Second, as DSM systems, GMAC and DyManD require direct mappings between the CPU and GPU memory spaces. As a result, the amount of data shared between the CPU and GPU is limited to the GPU memory size; in fact, the largest inputs that we used in our case study (Section 7.2) cannot be handled by existing systems. Furthermore, this direct mapping precludes more complex *semantic* mappings between the CPU and the GPU, such as transforming row-major layout to column-major layout, or SemCache's computation caching (Section 5).

3. OFFLOADING LIBRARIES TO GPUS

Over the past few years, graphics processing units (GPUs) have become attractive platforms for computing. The programmable vector units on GPUs offer the potential for massive, energy efficient parallelism. There are two downsides, to GPU computing, though. First, to achieve their energy efficiency, GPU cores are very simple, and only provide performance benefits when executing carefully parallelized code. Hence, attempting to port general code to GPUs is a tedious task, and often results in ineffective code. Instead, it is more effective to execute only those portions of an application that are amenable to GPU-style parallelism, such as linear algebra code, on the GPU, leaving the remainder of the application code on the CPU. Because writing efficient implementations on a GPU is difficult even for algorithms well-suited to parallel execution, there has been a proliferation of libraries that provide GPU implementations of common linear algebra kernels (often providing the BLAS interface [4]), easing the task of offloading these operations to the GPU.

The second downside to using GPUs for general purpose computing is that most GPUs use separate memory from the CPU. In other words, the GPU uses a separate address space from the CPU, and hence the two processing units cannot readily share data. Instead, data must be explicitly transferred between the CPU and the GPU. This limitation is especially problematic when only portions of a computation are offloaded to the GPU: because both the CPU and the GPU perform operations on the same data, the data must be transferred back and forth as necessary. Worse, transferring data between CPU and GPU is *slow*, especially in comparison to the speed of each processing unit's own memory. Performing data transfers are often a significant cost of GPU computation, and there have been several approaches that have attempted to avoid even offloading computation when data transfer costs exceed the benefit of GPU computation [14, 19, 24].

The necessity for explicit data movement between GPU and CPU makes providing modular libraries that provide GPU kernels more difficult. Consider the example code in Figure 1. As discussed in the introduction, there are several

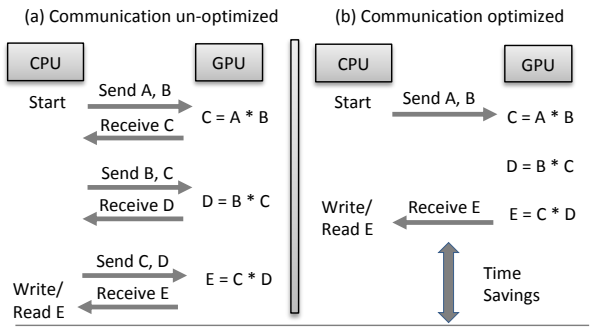

Figure 2: Communication comparison for optimized and un-optimized communication

distinct linear algebra operations performed in this example, each of which would be performed by a different library call. In the interests of modularity and encapsulation, some libraries handle communication between the CPU and GPU "under-the-hood" like CULA standard interface. While this makes using the library easier, it results in redundant communication. The library calls are implemented to execute in isolation, and as self-contained units, they assume that the data resides on the CPU. When invoking a method, a library call must (i) allocate space for the arguments and result on the GPU; (ii) transfer the arguments from the CPU to the GPU; (iii) perform the computation on the GPU; and (iv) transfer the result back to the CPU. As a result, even if multiple library calls could make use of the same data, new space is allocated and the data is transferred for each call. Hence, as we see in Figure 2(a), at each call two matrices are transferred to the GPU and one is transferred back, for a total of 9 matrix transfers.

Clearly, full encapsulation introduces too many performance problems. Instead, other library approaches, such as CUBLAS [18], give the programmer control over data allocation and movement. Hence, as in Figure 2(b), the programmer can explicitly transfer `A` and `B` to the GPU, allocate space for the results matrices in GPU memory (assuming matrices fit in the GPU memory), and operate only in GPU memory until the final result, `E`, is transferred back to the CPU. This results in the minimal amount of communication, 3 matrix transfers. Unfortunately, forcing a programmer to explicitly manage data requires the programmer to reason about the composition of GPU operations. This is a global task that may be impractical for large codes.

In fact, for highly modular codes, it may not be possible to manually manage data movement. Consider, for example, if the three matrix-multiply calls of our example occurred during different invocations of the same larger method within an application. In other words, the three matrix multiplies occurred from library calls from the same line of code, just with different arguments. Clearly, there is no way to introduce data transfer operations statically to such code to correctly transfer the matrices only when necessary. Whether or not data needs to be transferred to the GPU is a purely *runtime* property, based on what other library methods have been called, and what arguments are being passed to a particular library invocation.

The following section describes SemCache, our approach to tracking exactly this dynamic information in a manner that can be readily encapsulated into easy-to-use libraries for offloading GPU computations.

4. SemCache

This section introduces SemCache, a variable-granularity, *Semantics-aware Cache* that can be used to efficiently and easily manage sharing and transferring data between the disjoint CPU and GPU address spaces.

4.1 High Level Overview

A software cache between CPU and GPU, at a high level, is simple and intuitive. One variant, using a MSI (*m*odified, *s*hared, *i*nvalid) protocol might operate as follows: a given piece of memory (*e.g.*, a contiguous block of memory, a page, etc.) is tracked by a run-time system. The run-time tracks whether the contents of the piece of memory are currently valid on both devices (shared), valid only on the GPU (modified on the GPU, invalid on the CPU) or valid only on the CPU (invalid on the GPU, modified on the CPU). Whenever memory is read on a particular device, the cache can be consulted to determine whether the local memory has valid data; if not, communication between GPU and CPU is necessary, and the cache state is changed to shared. When a piece of memory is written on a device, the local cache state is changed to modified, and the state for the other device is changed to invalid.

Such an implementation has been used in numerous previous projects targeting different architectures, from distributed shared memory systems (*e.g.*, [1, 16]) to software caches between Cell processing units (*e.g.*, [8, 12]). The downside to prior implementations is that the granularity at which memory was tracked was constant (*e.g.*, an entire OS page, or a fixed-size block of contiguous memory). However, a fixed granularity may not be appropriate for a given application. If the granularity of the cache is too large (the blocks being tracked are too big), excessive communication will happen, both from transferring unnecessary data and from performing too many invalidations due to false sharing. If the granularity of the cache is too small, more cache lookups will be necessary for a given set of operations, and communication will be broken up into more transfers, resulting in more overhead. Unfortunately, it is difficult to tell for a given application, what the appropriate cache granularity should be, and different applications may require different granularities.

The key insight of SemCache is that when using libraries to offload computation to GPUs, *the correct granularity for a cache can be inferred*. In particular, the appropriate granularity for the cache should be the data structures operated on during offloaded library calls. Moreover, the library's semantics *directly capture what the relevant data structures are*. As a result, by tying SemCache's granularity to a library's semantics, we can track data at exactly the right granularity for a given application.

For example, when SemCache is used in conjunction with a linear algebra library, the data structures being operated on are matrices; as a result, SemCache will track data at the granularity of the matrices used in a particular application. In contrast, if SemCache is used in conjunction with a graph library, the data structures being operated on might be adjacency lists. SemCache will correctly track data at the granularity of entire adjacency lists representing the graphs being operated on.

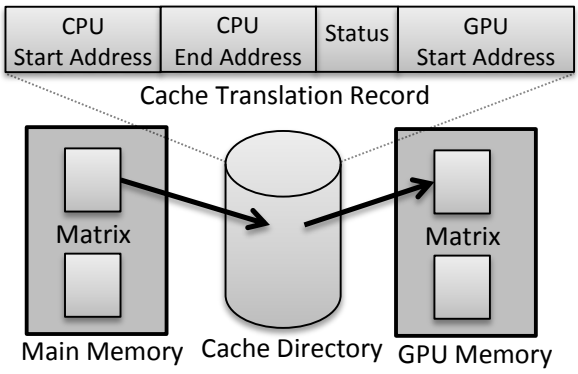

Figure 3: Structures of the Caching Directory

SemCache is composed of multiple, interlocking components: (a) a variable-granularity cache structure and interfaces for performing cache lookups, triggering data transfers, and performing invalidations; (b) a strategy for setting the granularity of the cache based on library behavior; and (c) instrumentation and protocols for tracking and maintaining the correct cache state for memory. The following subsections describe these components.

4.2 Cache Design and Structure

The basic design of SemCache is shown in Figure 3. There is a single data structure, consisting of a set of *translation records* that tracks the status of the various data structures used in a program. Note that even though data may reside on either the CPU or the GPU, it is the CPU that is in charge of maintaining the cache data, and of performing all lookups and invalidations. This is due to the basic approach used for GPGPU computation. Operations are dispatched to the GPU by transferring data (if necessary) to the GPU and invoking a single kernel method. Once the kernel method completes, control transfers back to the CPU and any necessary data is transferred back. In other words, the CPU alone is responsible for controlling execution and for transferring data between the two memory spaces. As a result, SemCache consists of a single set of translation records maintained by the CPU.

The primary data structure of SemCache is the set of translation records that maintain a mapping between CPU data and the corresponding data on the GPU, as well as the current location of the data. In a sense, SemCache serves as a translation lookaside buffer (TLB), except that its entries point to variable-length regions of memory rather than fixed-size pages. The cache entries are hence indexed by both a start address (cpu_s) and an end address (cpu_e) *of the data's location on the CPU*. Each entry also contains a status field (*status*) to keep track of the data's status. These statuses can be one of C, for valid only on the CPU, G, for valid only on the GPU, or S, for valid at both locations. Finally, the translation record contains the putative location of the same data on the GPU (gpu_s)[1].

SemCache's interface provides a number of operations. A memory range $[s, e)$ refers to start and end addresses for a memory range on the CPU.

[1]This location is putative because it is only valid if the status of the range is S or G; if the status is C, the next time the data is sent to the GPU, new space will be allocated for the data

lookup(s, e) Retrieves the translation record associated with memory range $[s, e)$. If the memory range is not currently tracked, create a new entry for the range, and set the status to C.

transferToGPU(entry) Assumes that the status of the entry is S or C. Transfers the contents of memory range $[cpu_s, cpu_e)$ on the CPU to the GPU, allocating new space on the GPU. Sets the GPU start address appropriately. Sets the status of the entry to S.

transferToCPU(entry) Assumes that the status of the entry is S or G. Transfers the contents of memory range $[gpu_s, gpu_s + (cpu_e - cpu_s))$ from the GPU back to the CPU. Sets the status of the entry to S.

invalidateOnGPU(entry) Sets the status of entry to C.

invalidateOnCPU(entry) Sets the status of entry to G.

SemCache maintains the invariant that the ranges tracked by its translation records are disjoint. If a range being looked up is a *subset* of some tracked memory range, then lookup returns the entry associated with the larger memory range. If a range being looked up spans multiple tracked ranges, SemCache*merges all the matching translation records* and creates a new record that tracks a range that spans all of the merged records.

To perform lookups and merges efficiently, SemCache maintains the entries sorted by start address. To look up the range $[s, e)$, SemCache searches for the entry with the largest start address less than or equal to s. If the end address of the found entry is less than or equal to s, SemCache creates a new entry for the range. If the end address of the found entry is greater than or equal to e, it returns the entry. If the end address of the found entry is greater than s and less than e, SemCache iterates through the subsequent entries until it finds all the entries that overlap with the current range. It then merges the ranges together, performing appropriate data movement operations so that the eventual state of the new entry is C.

Managing available GPU memory. The amount of data sent to the GPU might be too large to fit the available GPU memory. In such a situation, to allocate new data in the GPU memory, cached data must to be freed. To determine which address ranges should be freed, SemCache keeps track of the number of hits for each tracked range in the cache, and, when necessary, removes infrequently used ranges from the GPU. The simple strategy we adopted produced the minimal "miss rate" for our test applications. Note that depending on the application other polices such as least-recently-used may result in better handling of capacity misses. Multiple replacement polices can be easily integrated with SemCache and the programmer can have the option to choose between them.

4.3 Determining Granularity

SemCache by itself is simply a variable-granularity cache that supports a few methods to transfer data between the CPU and GPU. The key to SemCache's utility is that the granularity of the cache is determined by the semantics of the GPU libraries being used in a program. In particular, we note that the address ranges tracked by the cache are determined during cache lookup: if a particular range has

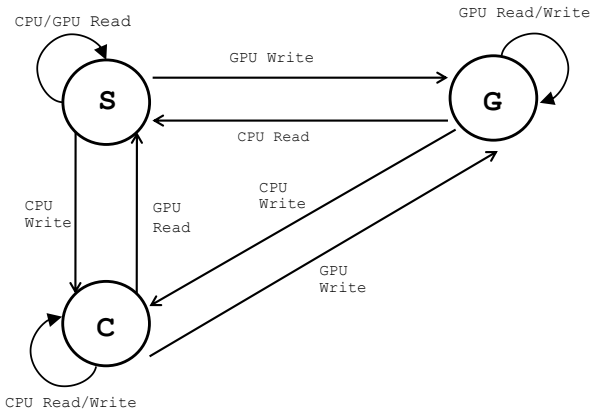

Figure 4: Write-back protocol (States: GPU/CPU/Shared)

not been seen before, a new entry for that range is created. Hence, if a library call takes as input matrices `A` and `B` and produces as output a matrix `C`, the three matrices can be individually tracked by performing lookups on their address ranges. For example, if `A` were an $n \times n$ matrix (of floats), invoking `lookup(A, A + 4 * n * n)` would cause SemCache to start tracking matrix A, and whether it existed on the GPU or not. Note that the current implementation of SemCache only tracks contiguous memory ranges; data structures that are not contiguous ranges have to be tracked with multiple entries.

4.4 SemCache Instrumentation and Protocols

The interfaces of SemCache can be used to implement a basic protocol to manage data movement between the CPU and GPU. The protocol tracks reads and writes on both devices, and transfers data when necessary. Figure 4 shows the basic protocol, which behaves similarly to an MSI coherence protocol. Data that is computed on the GPU remains on the GPU until the CPU needs to read it. Similarly, data computed on the CPU remains on the CPU until the GPU needs it. If either the CPU or GPU writes a piece of data, that data is invalidated on the other device. Adopting the terminology of Quintana-Orti *et al.*, we call this a *write-back* protocol [21]. Figure 5 shows the operations performed on the various devices to implement this protocol. Note that these methods return the translation record, as invoking methods on the GPU may require knowing the addresses where the necessary data is stored. Section 6 discusses how a library writer can use these interface methods to manage data movement for a particular library.

While `writeGPU` and `readGPU` are used before and after invocations of library calls offloaded to the GPU, in the worst case, every read and write on the CPU must be guarded by `writeCPU` and `readCPU` (as in software caching approaches for accelerators like the Cell). Section 4.5 discusses two approaches to inserting these guards.

We introduce a further protocol simplification. Because reads on the CPU are much more prevalent than writes, and because most results computed by the GPU are eventually needed on the CPU, we eagerly transfer any data *written* by the GPU during a library operation back to the CPU. This affects how library operations that modify data are handled. In the write-back protocol, `writeGPU` is invoked to invalidate the data on the CPU. In the write-through protocol, `writeGPU` is never invoked, but instead `readCPU` is immediately called to transfer the data back to the CPU. Section 6 gives a concrete example of how the implementation of a library changes based on the protocol.

```
1  //execute after writing CPU address range [s,e)
2  TranslationRecord writeCPU(s, e) {
3    entry = lookup(s, e);
4    invalidateOnGPU(entry);
5    return entry;
6  }
7
8  //execute before reading CPU address range [s,e)
9  TranslationRecord readCPU(s, e) {
10   entry = lookup(s, e);
11   if (entry.status == G) //CPU data not current
12     transferToCPU(entry)
13   return entry;
14 }
15
16 //called after a GPU method that writes [s, e)
17 TranslationRecord writeGPU(s, e) {
18   entry = lookup(s, e);
19   invalidateOnCPU(entry);
20   return entry;
21 }
22
23 //called before a GPU method that reads [s, e)
24 TranslationRecord readGPU(s, e) {
25   entry = lookup(s, e);
26   if (entry.status == C) //GPU data not current
27     transferToGPU(entry);
28   return entry;
29 }
```

Figure 5: Operations to implement write-back protocol

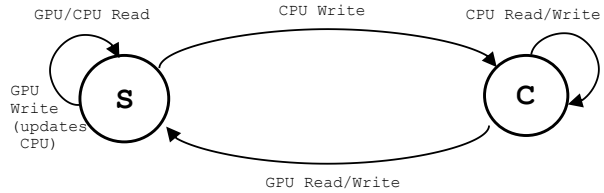

Figure 6: Write-through protocol (States: CPU/Shared)

In the write-through protocol, data is never in the G state; it can only be in C or S. The simplified protocol is shown in Figure 6. Because data is eagerly written back to the CPU, we again adopt previous terminology and call this a *write-through* protocol [21]. Note that because data is never in the G state, we no longer need to instrument CPU reads, reducing instrumentation overheads.

4.5 Instrumenting CPU Reads and Writes

SemCache provides two approaches to inserting instrumentation to implement the write-back and write-through protocols: statically-inserted instrumentation (either by the programmer or the compiler), and dynamic instrumentation using the operating system's page-protection facilities.

4.5.1 Statically-inserted Instrumentation

Conservatively, the programmer or compiler must guard every read or write on the CPU with appropriate instrumentation. In practice, because data movement between the CPU and GPU can only occur when GPU libraries are invoked, simple analyses can reduce this instrumentation over-

head. For example, any data that will never be sent to the GPU (*i.e.*, can never be passed to a method call executed on the GPU) does not need to be instrumented. Furthermore, reads or writes to array locations that occur in loops can be guarded by a single call, with the parameters determined by array analyses that determine what portions of an array are accessed in a loop. These analyses, of which many exist, are beyond the scope of this paper; we assume that such an analysis has already been performed, allowing array accesses to be efficiently guarded.

The run-time nature of SemCache tolerates instrumentation imprecision. In particular, looking up address ranges that are not shared with the GPU does not introduce extra communication, nor does invalidating the same range more than once; these operations merely introduce extra caching overhead. Conservatively invalidating an address region is also safe: while this unnecessary invalidation causes unnecessary communication, it does not affect the correctness of the program.

Note also that although we instrument particular reads and writes, as well as particular GPU operations, to perform our caching, the cache lookups, etc., are based on address ranges. As a result, program behaviors such as aliasing do not present correctness problems; the caching is performed based on the underlying memory, not the specific name given to that memory.

Even after removing unnecessary instrumentation through the above analyses, and avoiding the instrumentation of reads on the CPU with the write-through protocol, invoking `writeCPU` before every write to data that may reside on the GPU still introduces unnecessary instrumentation. For example, on a write to data that has already been invalidated on the GPU, it is redundant to look up the data and "re-invalidate" it. Developing an analysis to remove redundant instrumentation is a subject for future work.

4.5.2 Page-protection-based Instrumentation

Rather than using statically-inserted instrumentation of CPU reads and writes to drive its protocols, SemCache can also use the operating system's virtual memory system to detect when `readCPU` and `writeCPU` should be invoked. For each data structure that SemCache tracks on the CPU, SemCache sets page protection flags for all the pages the data structure spans. The page protection flags are set according to the state of the data structure. If the structure is in G state, its pages are set to `PROT_NONE`; if the structure is in S state, its pages are set to `PROT_READ`; and if the structure is in C state, the pages are set to `PROT_READ` | `PROT_WRITE`. If a CPU access triggers a page fault, SemCache invokes `readCPU` or `writeCPU` as appropriate to change the data structure's state in the cache and perform any necessary communication. The structure's page protection flags are then reset to correspond to its new cache state.

Note that although *detection* of accesses that require communication occurs at the page granularity, *communication* does not: if a structure needs to be communicated from the GPU to the CPU, SemCache transfers the entire structure, and changes the status of all of the associated pages on the CPU. This preserves SemCache's variable-granularity advantages.

The advantage of page-based invalidations over statically-inserted invalidations is that these invalidations are handled fully automatically; no additional instrumentation or compiler analysis is required. However, there are disadvantages as well: to work correctly with page-protection operations, and to avoid false sharing issues, page-based invalidations require that a program's memory layout must be changed to ensure that all data structures that may be communicated to the GPU are page-aligned and padded out to page boundaries.

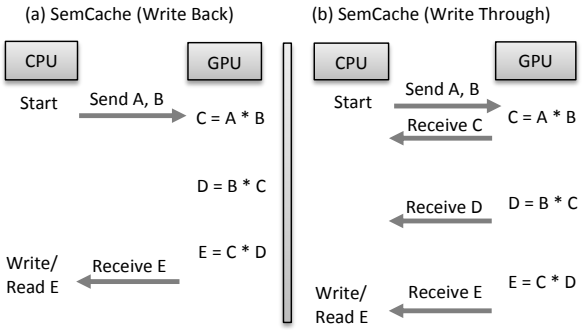

Figure 7: SemCache communication model

We note that this page-based strategy is similar to that used by DyManD [10]. However, unlike DyManD, SemCache still tracks data structures and maintains mappings between the CPU and GPU according to semantic information, rather than requiring direct memory mapping between the CPU and GPU. In addition to allowing programs whose working sets exceed GPU memory, SemCache's approach allows for *semantic links* to be formed between data on the CPU and data on the GPU, as the next section explains.

4.6 SemCache in Practice

Figure 7 shows the data movement performed by our system on the simple example of Figure 1 using the two different protocols. We note that when using the write-back protocol (Figure 7(a)), SemCache performs the minimum required data movement (*cf.* Figure 2(b)). At the first invocation of matrix-multiply, `A` and `B` are transferred to the GPU, and SemCache tracks them in S state. When `C` is computed, it is tracked in G state. Because both `B` and `C` are current on the GPU, later invocations of matrix multiply need not perform any more data transfer. Finally, `E` will be transferred back to the CPU once that matrix is read by other portions of the program. In the write-through protocol (Figure 7(b)), the amount of communication from the CPU to the GPU is minimal. However, because GPU results are eagerly communicated back to the CPU, we see that some extra communication is performed from the GPU to the CPU.

Crucially, because all of the necessary instrumentation is either automatically inserted or encapsulated in a GPU library (see Section 6), programmers can simply use SemCache-enhanced GPU libraries as drop-in replacements for their existing libraries.

5. SEMANTIC MAPPING WITH SemCache

This section discusses how the basic principles of SemCache can be extended and generalized to achieve additional savings. In particular, we describe how ancillary structures can be added to SemCache to allow it to "cache" the results of arbitrary functional computations, essentially allowing SemCache to serve as a memoizing service for GPU

computations. This facility can be used for many purposes, from avoiding expensive recomputations (*e.g.*, storing only the factorized forms of matrices on the GPU) to performing data layout transformations (*e.g.*, mapping row-major data structures on the CPU to column-major layouts on the GPU). In essence, instead of directly mapping between CPU and GPU data, SemCache can create a *semantic link* between data on the CPU and a transformed version of that data on the GPU.

To see how SemCache can create these semantic links, we note that memoization effectively maps a particular input of a function to its pre-computed output. That is, for a function $f : X \rightarrow Y$, a memoized input x is used to look up its previously-computed output y, rather than evaluating $f(x)$. If we consider x as data residing on the CPU, and y as data residing on the GPU, then we can abstract SemCache's default behavior as simply the memoization of the identity function $f(x) = x$. For a given input (*i.e.*, data on the CPU), SemCache provides the previously-computed (*i.e.*, previously-communicated) output (*i.e.*. data on the GPU). In other words, SemCache is indexed by inputs on the CPU and provides a map to the results of the identity function stored on the GPU.

We can see that there is no need for SemCache's operation to be restricted to memoizing the identity function on to the GPU. The results of other functions can be memoized as well. Consider performing matrix factorization (*e.g.*, LU factorization) as an intermediate step in equation solve (GESV), the factorization is not saved. Such factorizations on the GPU are time consuming, so repeatedly factorizing a matrix can be wasteful. Instead, we can use an extended version of SemCache to cache the *results* of the factorization on the GPU, instead of just the inputs to the factorization operation.

Figure 8 shows how SemCache is extended. The same address ranges tracked by the baseline cache are used to index into a computation cache, which stores the GPU location of the *results* of a particular computation. Since this data is computation-specific, each type of computation to be memoized will need separate lookup tables. Note that the computation structures need not separately track the status of the data. If the data in the main cache is ever invalidated on the GPU (*i.e.*, its status is changed to C), the corresponding entries in any computation caches are simply removed.

To attempt to skip performing a GPU computation on an address range $[s, e)$, SemCache takes the following steps. First, the range is found in the main cache. If the status of the range is S or G, the lookup is repeated in the computation cache, and, if a result is found, the GPU computation can be elided. If the status of the range is C, or there is no entry in the computation cache, the GPU computation is performed, the status of the range is set to S, and the computation cache is updated.

We note that the particular set of lookups, and particular data stored, is based on the semantics of the computation being cached. Using SemCache to create semantic links hence requires adding instrumentation to GPU libraries to perform the necessary lookups, etc. Nevertheless, this instrumentation can be completely encapsulated in a library, and its effects need not be visible to the programmer, preserving the library as a drop-in replacement.

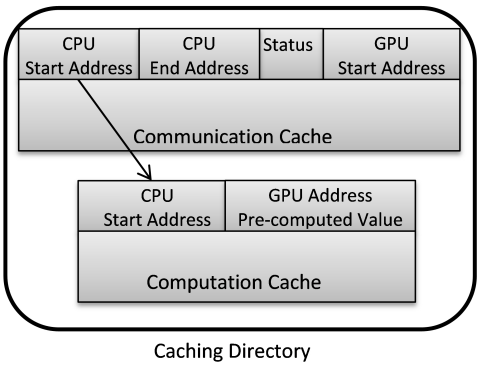

Figure 8: Caching Directory Components

```
1  cudaMalloc(A) //Allocate space on device mem.
2  cudaMalloc(B) //Allocate space on device mem.
3  cudaMalloc(C) //Allocate space on device mem.
4
5  cublasSetMatrix(A) //Move matrix A to device
6  cublasSetMatrix(B) //Move matrix B to device
7  cublasSetMatrix(C) //Move matrix C to device
8
9  cublasDgemm(TRANSA,TRANSB,M,N,K,ALPHA,
10   A,LDA,B,LDB,BETA,C,LDC)
11
12 cublasGetMatrix(C) //Get matrix C from device
```

Figure 9: Matrix multiply using CUBLAS code

6. IMPLEMENTATION

To demonstrate how SemCache can be used to improve the performance of GPU computation libraries, we use it to produce a drop-in replacement for BLAS. This allows programmers to replace BLAS calls in their code with calls to our library, automatically offloading computation to the GPU and handling memory management transparently. The GPU kernels of our library are based on the corresponding CUBLAS implementations. Our library supports either the write-back protocol or the write-through protocol, controlled by a compile-time flag. If instrumentation-based invalidation is used, then CPU reads and writes must be instrumented as described in Section 4.4. If page-based invalidations are used instead, no instrumentation need be inserted, but `malloc` calls must be modified to page-align allocations and pad to page boundaries to avoid false sharing, as described in Section 4.5.2.

Figure 9 shows the sequence of calls that would be required to use CUBLAS to perform matrix multiply, with all communication explicitly managed by the programmer. In contrast, Figure 10 shows the interface for the SemCache-enhanced version of matrix multiply.

Figure 11 shows how matrix multiply is implemented. Under the hood, we still invoke the CUBLAS matrix multiply method. However, all communication is managed by SemCache, and is only performed when necessary. When SemCache is called, the caching directory is searched for each matrix using the start and end address in the main memory. The start address is the pointer address and the end address is calculated using the matrix size. The cache keeps a record

```
1  SemCacheDemm(TRANSA,TRANSB,M,N,K,ALPHA,
2        A,LDA,B,LDB,BETA,C,LDC)
```

Figure 10: SemCache library interface

```
1  SemCacheDemm(TRANSA,TRANSB,M,N,K,ALPHA,
2       A,LDA,B,LDB,BETA,C,LDC)
3  {
4    //A stored on CPU in memory range [A, A+(M*K*8))
5    //A will be read by GPU, its state will be "S"
6    entryA = readGPU(A, A + (M*K*8));
7
8    //B stored on CPU in memory range [B, B+(K*N*8))
9    //B will be read by GPU, its state will be "S"
10   entryB = readGPU(B, B + (K*N*8))
11
12   //C stored on CPU in memory range [C, C+(M*N*8))
13   //C will be read by GPU, its state will be "S"
14   entryC = readGPU(C, C + (M*N*8))
15
16   cublasDgemm(TRANSA,TRANSB,M,N,K,ALPHA,
17       entryA.gpu_s,LDA,
18       entryB.gpu_s,LDB,BETA,
19       entryC.gpu_s,LDC)
20
21   //C was written by cublasDgemm
22 #ifdef WRITEBACK
23   //If we're using write-back, writeGPU must be
         called to invalidate, C state will be "G"
24   writeGPU(C, C + (M*N*8))
25 #else
26   //If we're using write-through, we eagerly
         communicate to the CPU, C state will be "S"
27   readCPU(C, C + (M*N*8))
28 #endif
29 }
```

Figure 11: Implementation of SemCache matrix multiply (DGEMM)

of the start and end address in the main memory for each matrix accessed using our library. If the matrix does not exist, it is transferred to the GPU and cached. A new record is created for it in the cache. If the matrix is found in the cache and it is in S state, then it is a hit and there is no need to transfer the matrix to the GPU. The matrix address in the GPU memory is taken from the translation record. This address is used to access the matrix using CUBLAS. If the matrix is not valid on the GPU (it is in C state), it is transferred to the GPU and the record in the cache is updated. After all of the matrices are transferred or located in the GPU memory, the CUBLAS call is executed. Then the result is transferred back to the main memory.

CUBLAS does not provide an implementation of general equation solve (GESV), instead only providing triangular solves for factorized matrices. While there exist several efficient GPU implementations of LU factorization [23, 25], our implementation instead implements equation solve in two steps: we compute the LU factorization on the CPU, then perform the equation solve on the GPU using CUBLAS's triangular-solve routines. We then use SemCache's computation caching capability to avoid performing the factorization whenever possible. This implementation was chosen to demonstrate SemCache's generalized memoization ability.

7. EXPERIMENTAL EVALUATION

Experiments were run on a server with 24 AMD Opteron 6164 HE Processors (1.7 GHz, 512 KB L2 cache), 32 GB memory, running 64-bit Fedora Linux, and NVIDIA Tesla C2070 card (6 GB memory) with a peak memory bandwidth of 144 GB/s. Three libraries were used: CUBLAS version 4.0, CULA Dense R15 and MAGMA version 1.2. Each test was run 3 times, distributed over a wide range of time, on an unloaded machine and the median time selected.

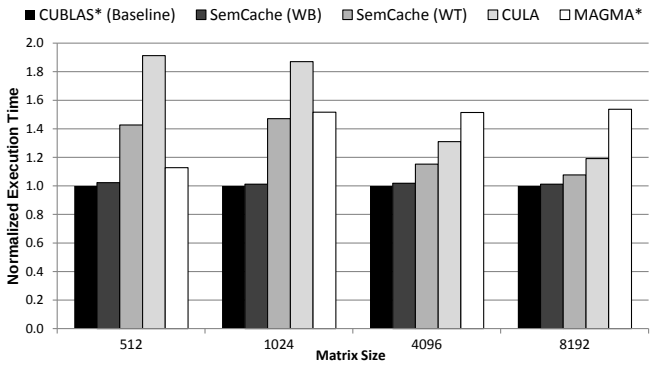

Figure 12: Test case normalized execution time *(Communication in CUBLAS and MAGMA is hand optimized)

We evaluated our library in two ways. First, we used a test case based on a series of matrix multiplications (as in Figure 1). The simplicity of the test case allowed us to perform several comparisons with other libraries, as well as test the two SemCache protocols. Nevertheless, the primary target for our work is large-scale computational applications where hand-tuning is infeasible. To study SemCache's effectiveness in this setting, we used our modified BLAS libraries (see Section 6) on a large-scale, real-world computational mechanics application, which uses finite element methods and domain decomposition to solve a structural dynamics problem.

As described in Section 4.4, SemCache can perform invalidations either with statically-inserted instrumentation at CPU reads and writes, or using a page-protection-based mechanism. We found empirically that the two approaches perform equivalently; in the experiments presented here, we use instrumentation to perform invalidations.

7.1 Test Case Performance Evaluation

Figure 12 shows the total execution time for the test case. The results are collected using CUBLAS, CULA Standard Interface (which automatically manages communication between the CPU and GPU), MAGMA and SemCache using both write-back and write-through policies. Communication in CUBLAS and MAGMA are hand tuned. The total execution time is normalized to CUBLAS execution time. The best performance is achieved by hand tuning the memory transfers using CUBLAS. CULA performance was slowed down due to the repeated unnecessary transfers. SemCache write-back performance matches the optimal communication performance using CUBLAS, but is slightly slower due to caching overhead. SemCache write-through performance is the next closest to the optimal communication performance. The slowdown is due to the eager copying back of the result to the CPU after each multiplication. MAGMA's performance varies based on the matrix size (as it uses different kernels tuned to different matrix sizes), but overall uses slower implementations than CUBLAS.

Communication savings. Figure 13 breaks down the communication performed for a medium-sized squared matrix (N=4096), distinguishing between data sent and data received. We collected data for CUBLAS, MAGMA, SemCache write-back, SemCache write-through and CULA. Hand-tuned communication for CUBLAS and MAGMA minimize the memory transfers. SemCache write-back performs exactly as much communication as hand-tuned libraries. It

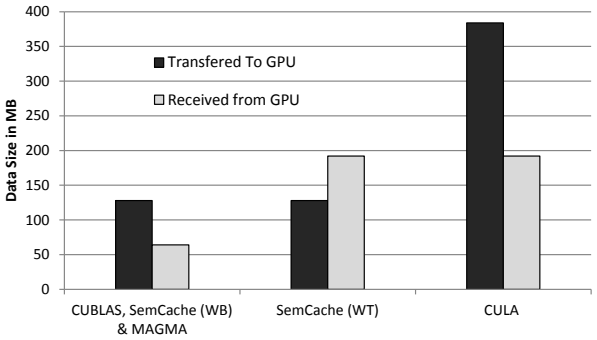

Figure 13: Test case communication results (N=4096)

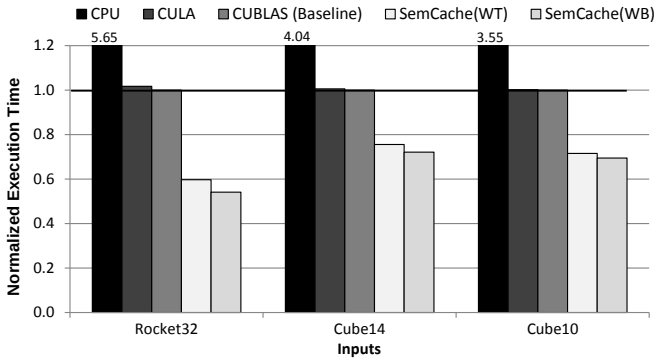

Figure 14: Testing application normalized execution time

performs the minimum amount of data transfers, as the data is already cached on the GPU and is only sent back when needed. In SemCache write-through, the data sent to the GPU is minimized. However, data is always copied back, introducing redundant communication. CULA shows the most overhead since matrices are sent to the GPU for every calculation. The results are also sent back to main memory after each calculation, introducing extra communication.

7.2 Computational Mechanics Case Study

We next tested SemCache's performance in a real-world setting. We studied a large computational mechanics application. In this application domain decomposition is used for the simulation of structural dynamics problems. Domain decomposition methods solve a boundary value problem by splitting it into smaller boundary value problems on subdomains and iterating to coordinate the solution between adjacent subdomains. Then the Newmark-beta method of numerical integration is used to solve differential equations. The application we use solves the subdomains recursively. This method was introduced by [20]. Typical structural dynamics problem include simulation of the effect of cracks in structures, or buildings under stress.

Most of the application's execution is spent performing linear algebra routines. Three main double-precision linear algebra subroutines are used: matrix multiplications (DGEMM) and scalar multiplication/vector addition (DXPY) to couple and update the subdomains results and equation solve (DGESV) to solve the system of equations at each node. Because these operations make up a large fraction of the application's computation, they are attractive targets for offloading. However, optimizing communication in this application is essentially impossible. The application has tens of thousands lines of code, and the relationship between various linear algebra operations is difficult to reason about due to recursive calls and multiple abstraction layers.

We evaluated five versions of this application. A serial CPU version that performed no offloading, a CUBLAS version with hand-inserted unoptimized communication (communication can't be optimized manually due to program abstraction), a CULA version that simply replaces all CPU BLAS calls with CULA BLAS calls, a version using our SemCache write-through library, and another version using our SemCache write-back library.

The SemCache versions of the application exploit computation caching in two ways. First, as described in Section 6, our implementation of equation solve leverages SemCache's computation-saving capabilities to memoize the results of matrix factorization. Second, the baseline CPU version of the application uses row-major storage for its matrices, while CUBLAS assumes column-major storage. SemCache thus creates a semantic link between the two representations, avoiding performing the transformation unless the data changes[2].

We used three inputs with different characteristics, ranging across various sizes: Rocket32, which has 7262 nodes and takes 246 seconds to run on the CPU; Cube14, which has 3375 nodes and takes 130 seconds to run on the CPU; and Cube10, which has 1331 nodes and takes 10 seconds to run on the CPU.

Execution time. Figure 14 shows the total execution time for the five variants of the application, across the three inputs. Run time is normalized to the CUBLAS variant. All inputs gained from three to six times speedups when running on the GPU over the CPU version. CULA and CUBLAS performance was very similar. CULA uses CUBLAS as an underlying library with a few additional optimizations. Both approaches incur the cost of extra communication. Using SemCache with write-through policy, the performance improved (30% to 40%) over the GPU CUBLAS baseline version due to the communication savings from caching. SemCache with write-back gained an additional 4–10% over write-through, as data was only transferred back to the CPU when needed. The inputs speedup ranges are different based on the structure of input and domain decomposition. Inputs whose subdomains have larger shared interfaces generate more matrices that will be repeatedly reused. As a result, caching yields more benefits.

Communication savings. Table 1 shows the amount of data transferred to the GPU. The SemCache results show the optimal communication for the application since all of the calculations were computed on the GPU and the hit rate was 100%. Using SemCache, more than 80% of the unoptimized communication is reduced. Both write-through and write-back policies reduced the size of the data sent to the GPU. Write-back policy reduced the size of the data received from the GPU. These savings are a result of eliminating redundant transfers since the data in the testing application is shared between different subdomains. The same matrices will be reused multiple times for different subdomains.

[2]Because the row-major/column-major transformation is only necessary due to an implementation detail of the original application, we factor out the transformation time for the non-SemCache versions in all our results.

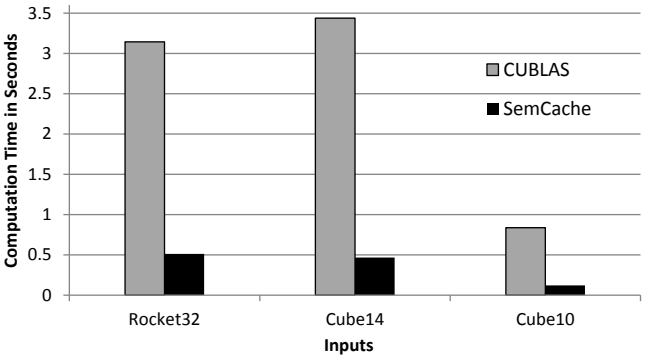

Figure 15: Computation time for factorization

Input/Library	CUBLAS		SemCache	
	Sent	Received	Sent	Received
Rocket32	23.70	5.58	2.02	2.45
Cube14	10.67	1.53	1.01	0.63
Cube10	3.01	0.33	0.29	0.13

Table 1: Size of transferred data using CUBLAS versus SemCache (in GB)

Computation caching. We evaluated the savings of performing computation saving for LU factorization. Figure 15 shows the LU factorization time on the CPU for out testing application. Using SemCache, repeated computations are eliminated since the factorized matrices are already cached. Fewer factorizations are needed, which reduces the total computation time by more than 80% .

Caching overhead. Table 2 shows the data transfer time to GPU for different inputs. The results show that the caching overhead is very low (less than 4% of SemCache total runtime). The overhead comes mainly from searching and updating the cache directory. The transfer time using our library including the caching overhead is significantly less than the transfer time for CUBLAS without caching. We note, however, that our low caching overhead is due to SemCache's variable granularity, which requires fewer invalidations and fewer lookups.

Instrumentation statistics. For our testing application, more than 10,000 lines of code and around 45 BLAS and LAPACK calls are used. No writeCPU invalidations were needed because the all of the calculations were computed on the GPU. For the write back protocol, reads were instrumented using readCPU API. Seven API calls were needed.

Table 3 shows how many times matrix multiply (GEMM), equation solve (GESV), scalar multiplication and vector addition (XPY), copy (COPY), lookup and invalidation operations were executed. The lookup matches exactly the number of matrices sent to the GPU (3 per GEMM, 2 per GESV, XPY and COPY).

Comparison with fixed-granularity approaches. One of the primary advantages of SemCache over distributed-shared memory systems is its ability to track data and perform communication with variable granularity, rather than using a fixed granularity. To quantify this benefit, we modified the page-protection version of SemCache (Section 4.5.2) to perform communication in page-sized chunks, rather than tracking entire data structures[3]. Figure 16 compares the

[3]This variant is not strictly correct, as without transferring data at matrix granularity, the semantic mapping between row-major and column-major representations cannot be maintained. Nevertheless, this variant lets us explore the penalty of page-granularity caching.

Input/Library	CUBLAS	SemCache	
	Transfer	Transfer	Caching Over.
Rocket32	11.09	0.86	0.38
Cube14	5.27	0.47	0.05
Cube10	1.32	0.12	0.023

Table 2: Data transfer time from CPU to GPU for CUBLAS versus SemCache with overhead (in seconds)

Input/Op.	GEMM	GESV	XPY	COPY	Lookup
Rocket32	6720	1209	3520	480	30578
Cube14	944	233	688	104	4882
Cube10	470	131	394	62	2584

Table 3: Operations count at runtime

CUBLAS baseline with this DSM-like approach as well as SemCache's variable-granularity approach on our case study[4]. Clearly, fixed-granularity tracking does not perform as well as SemCache.

Interestingly, the total amount of data communicated is the same for both the fixed-granularity and variable-granularity versions. The performance difference arises because fixed-granularity tracking breaks that communication into more discrete communication operations, incurring additional overhead. Clearly, taking advantage of semantic information to perform variable-granularity tracking and communication yields a notable performance benefit.

8. CONCLUSIONS

The proliferation of libraries of GPU kernels has made it easy to improve application performance by offloading computation to the GPU. However, using such libraries still introduces the complexity of managing explicit data movement. Unfortunately, when using these libraries with complex applications with multiple levels of abstraction, it is very difficult to reason about how multiple kernel invocations interact with one another, and hence avoid redundant communication. In this paper, we introduced SemCache, a semantics-driven caching technique that can tune its granularity based on the semantics of the GPU libraries in an application. SemCache can automatically detect and avoid redundant communication. We evaluated SemCache on a large, real-world application and showed that our approach can deliver significant performance improvements over state-of-the-art GPU libraries.

There are many promising avenues of future work. SemCache's computation-caching mechanism already supports mapping a data layout on the CPU to a different layout on the GPU. This facility can be extended to allow SemCache to track non-contiguous data structures, for example mapping a set of locations on the GPU to a single, contiguous, packed location on the GPU. We also plan to extend SemCache to support kernels distributed/offloaded across multiple GPUs and to automatically manage and optimize communication between the CPU main memory and several GPUs not only for a single kernel but for the entire application.

[4]Due to limitations of the page-based approach, large inputs (such as Rocket32) cannot be run.

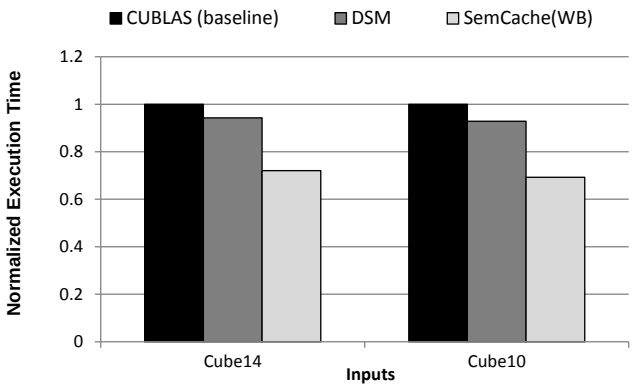

Figure 16: Testing application normalized execution time for CUBLAS, SemCache write-back and DSM

9. ACKNOWLEDGMENTS

We would like to thank Hasan Jamal, Chenyang Liu and Arun Prakash for the computational mechanics application code we used in our case study and for their support and feedback during this work. We would also like to thank the anonymous reviewers for their feedback and suggestions. This research is supported by the Department of Energy under contract DE-FC02-12ER26104. The GPU hardware we used was provided by an equipment grant from Nvidia.

References

[1] C. Amza, A. L. Cox, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK users' guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, 2009.

[4] BLAS. Basic linear algebra subprograms. `http://www.netlib.org/blas/`.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, 2008.

[6] M. Fogue, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Retargeting PLAPACK to clusters with hardware accelerators. In *HPCS*, pages 444–451, 2010.

[7] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 347–358, 2010.

[8] M. Gonzàlez, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien. Hybrid access-specific software cache techniques for the Cell BE architecture. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 292–302, 2008.

[9] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. *SPIE Defense and Security Symposium (DSS)*, pages 770502–770502–7, 2010.

[10] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165–174, 2012.

[11] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. *SIGPLAN Not.*, 47(6):142–151, June 2011.

[12] 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*, 49(4.5):589 –604, july 2005.

[13] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, Mar. 2008.

[14] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

[15] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. *SIGPLAN Not.*, 47(8):117–128, Feb. 2012.

[16] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 –60, aug. 1991.

[17] NVIDIA. CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.

[18] NVIDIA. CUDA toolkit 4.0 CUBLAS library. `http://docs.nvidia.com/cuda/cublas/index.html`.

[19] J. A. Pienaar, A. Raghunathan, and S. Chakradhar. MDR: performance model driven runtime for heterogeneous parallel platforms. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 225–234, 2011.

[20] A. Prakash and K. D. Hjelmstad. A FETI-based multi-time-step coupling method for Newmark schemes in structural dynamics. *International Journal for Numerical Methods in Engineering*, 61(13):2183–2204, 2004.

[21] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 44(4):121–130, Feb. 2009.

[22] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 121–130, 2009.

[23] P. D. S. Tomov, R. Nath and J. Dongarra. MAGMA version 0.2 users' guide. `http://icl.eecs.utk.edu/magma/`, 2009.

[24] C.-Y. Shei, P. Ratnalikar, and A. Chauhan. Automating GPU computing in MATLAB. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 245–254, 2011.

[25] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, 2008.