SemCache++: Semantics-Aware Caching for Efficient Multi-GPU Offloading

Nabeel Al-Saber, Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
nalsaber, milind@purdue.edu

ABSTRACT

Offloading computations to multiple GPUs is not an easy task. It requires decomposing data, distributing computations and handling communication manually. GPU drop-in libraries (which require no program rewrite) have made it easy to offload computations to multiple GPUs by hiding this complexity inside library calls. Such encapsulation prevents the reuse of data between successive kernel invocations resulting in redundant communication. This limitation exists in multi-GPU libraries like CUBLASXT.

In this paper, we introduce SemCache++, a semantics-aware GPU cache that automatically manages communication between the CPU and multiple GPUs in addition to optimizing communication by eliminating redundant transfers using caching. SemCache++ is used to build the first multi-GPU drop-in replacement library that (a) uses the virtual memory to automatically manage and optimize multi-GPU communication and (b) requires no program rewriting or annotations. Our caching technique is efficient; it uses a two level caching directory to track matrices and sub-matrices. Experimental results show that our system can eliminate redundant communication and deliver performance improvements over multi-GPU libraries like StarPU and CUBLASXT.

Categories and Subject Descriptors

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

Keywords

Multi-GPU offloading; GPGPU; Communication optimization

1. INTRODUCTION

Graphics processing units (GPUs) offer massive, highly-efficient parallelism, making them an attractive target for computation intensive applications. Due to the difficulty of programming GPUs, a practical option for leveraging their capabilities is to *offload* computation using libraries. For example, there are many GPU implementations of linear algebra libraries [16, 11, 20, 18], which outperform CPU implementations of popular libraries such as BLAS [8] and LAPACK [3] by taking advantage of the GPU's parallel hardware. Such GPU libraries allow existing applications written against the BLAS and LAPACK APIs to easily benefit from execution on heterogeneous platforms: most computation executes on the CPU, but invocations of BLAS methods are executed on the GPU.

This library-based offloading approach to harnessing the power of GPUs has some drawbacks. Notably, moving data back and forth between the CPU and the GPU incurs significant expense, making optimizing this communication paramount when library calls are composed. If successive library calls operate on the same data, the data should be moved to the GPU just once, rather than separately for each call, while data should only be transferred back to the CPU if a computation requires it. Such optimization is in tension with the encapsulation objectives of library-based offloading: if a programmer has to manually manage communication between the CPU and GPU, she can no longer port her program to a heterogeneous system without modification.

To help tackle this problem, over the past several years there have been several proposals to introduce automatic memory management between the CPU and single GPU, freeing the programmer from the burden of managing data movement [13, 12, 9, 17, 2]; in fact, the newest version of CUDA [15] offers *Unified Memory (UM)*, which dynamically tracks data movement between the CPU and GPU, minimizing communication. As a result of these techniques, library-based offloading is a viable option for leveraging a GPU in a heterogeneous system.

In recent years, *Multi-GPU* systems are becoming increasingly popular, with multiple GPUs available for computation offloading. Unfortunately, handling multi-GPU systems is substantially harder than managing a single GPU, as now computation and data need to be distributed across multiple GPUs. To simplify multi-GPU offloading, libraries such as CUBLASXT [16], MAGMA [20] and CULA [11], completely encapsulate communication in their library calls: prior to invoking a method, data is transferred to the GPU(s), and upon completion, data is transferred back. Such encapsulation introduces significant overheads, as much of this data movement is redundant. However, without encapsulation, managing data movement between kernels is quite difficult in multi-GPU systems.

While there have been several attempts at developing multi-GPU frameworks that can optimize communication more thoroughly, they are not well-suited to developing library replacements. They either require adopting a new programming model such as StarPU [4] and PTask [19] or require annotating every CPU data access, including those outside the offloaded library call such as StarSs [5]. The burden of rewriting an application or annotating large numbers of data accesses makes these models hard to adopt for large applications.

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3559-1/15/06 ...$15.00.
http://dx.doi.org/10.1145/2751205.2751210.

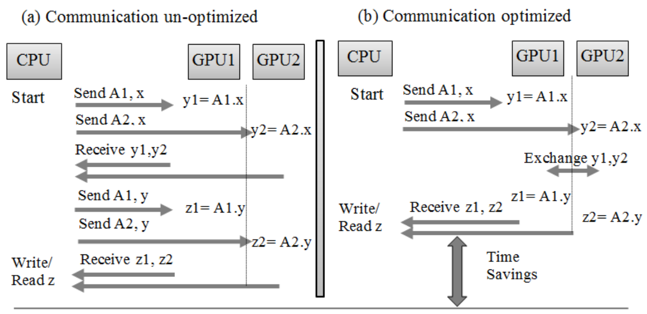


Figure 1: Communication comparison of Encapsulated Multi-GPU libraries and hand tuned communication

To understand the difficulty of managing data movement between CPUs and multiple GPUs, consider distributing a series of matrix-vector multiplications (MVMs): $y = A * x; z = A * y$. (This type of computation arises in algorithms such as Jacobi iteration.) To distribute this computation across GPUs, each operation should be decomposed. A natural decomposition is to split A horizontally into two submatrices A_1 and A_2, sending one to each GPUs. x can then be sent to both GPUs, computing $y_1 = A_1 * x$ and $y_2 = A_2 * x$.

Figure 1 demonstrates two different ways that communication could be handled for the remainder of this computation. If the MVM were fully-encapsulated, as in Figure 1(a), y_1 and y_2 would be sent back to the CPU and combined into y. When the second MVM is executed, A_1 and A_2 will be re-sent to each GPU, along with the re-composed y. A more efficient approach is to *leave* A in its decomposed form on both GPUs, as in Figure 1(b). When the second MVM is invoked, each GPU already has part of y already resident, and need only receive the portion of y they do not already have in order to complete their computation. This dramatically reduces the amount of communication. However, organizing this computation and communication correctly requires realizing that the communication of A is redundant and also that only a portion of y need be communicated. Note that the situation only becomes more complicated if the CPU requires access to the data as well: if code on the CPU (*i.e.*, not in library calls) accesses y between the two MVMs, then y must be fetched back from the GPUs, but the programmer must realize that the portions of y on the GPUs are still valid to avoid performing redundant communication for the second MVM. All in all, efficiently managing communication imposes a significant burden on the programmer.

What is needed is an *automatic* approach to managing data movement between the CPU and multiple GPUs that can *dynamically* determine whether data movement is necessary and most importantly provide a drop-in replacement library without adopting a new programming model.

In our previous work, we proposed SemCache [2], a runtime system to automatically manage and optimize communication between the CPU and a single GPU using caching. SemCache was used to build a drop-in GPU BLAS library. In this paper we propose SemCache++, an extension of SemCache that (a) enables the development of multi-GPU libraries with optimized communication by carefully tracking the coherence and matrix decomposition on multiple GPUs and (b) optimizes synchronization across library calls by continuing CPU execution past GPU library calls. This allows exploiting all devices (CPUs and GPUs) in parallel and allows to overlap communication with computation (even on a single GPU). Although both systems will have the same drop-in interface, the runtime system in SemCache++ is completely redesigned to support multiple accelerators and to allow the new features.

At a high level, SemCache++ treats the GPU memories as software caches, leaving data on the GPU even after library calls return. A SemCache++-enabled library call decomposes its computation across multiple GPUs. For example, a BLAS call will be decomposed into a series of operations over submatrices. When the computations complete, the results remain on the GPUs, rather than being transferred back to the CPU. SemCache++ tracks the location of each of the operands of the subcomputations. If another BLAS call is invoked, SemCache++ can avoid communication by directing subcomputations to GPUs that already have the necessary data. If communication is necessary, either between GPUs or because the CPU requires the results of a computation, SemCache++ can automatically perform the data transfer, updating its record of where the submatrices reside. SemCache++ exploits all GPUs in parallel. It also takes advantage of new CUDA features such as CUDA streams to overlap communication and computation, further enhancing performance. Crucially, all of SemCache++'s tracking and scheduling is encapsulated in library calls. Communication between the CPU and GPU(s) is managed transparently to the programmer, and SemCache++ libraries can be "dropped in" in place of existing linear algebra libraries without substantial program modifications.

1.1 Contributions

This paper makes the following contributions:

- The design and implementation of SemCache++, a generic multi-GPU cache that automatically manages communication between CPU and multiple GPUs at variable granularity.
- SemCache++ exploits all devices (CPUs and GPUs) in parallel, and uses CUDA streams to allow overlapping of communication and computation.
- A SemCache++-enabled multi-GPU BLAS library that provides a drop-in replacement for existing BLAS libraries.
- Experimental results showing that SemCache++ can dramatically reduce redundant communication, and deliver significant performance improvements over CUBLASXT, NVIDIA's tuned multi-GPU BLAS library.

2. BACKGROUND

2.1 Multi-GPU Drop-in Libraries

The most popular approach to leveraging multiple GPUs is to provide libraries that encapsulate the necessary decomposition and communication. CUBLASXT [16], MAGMA [20] and CULA [11] all provide subsets of BLAS and LAPACK methods that have been optimized for multiple GPUs. As described in the introduction, this encapsulation carries with it a cost: each method call is optimized in isolation, so any opportunities to identify and avoid redundant communication *across* library calls are lost. Essentially, data is "based" at the CPU, and is only distributed among GPUs for the duration of the method call, resulting in redundant communication of shared operands across method calls, and unnecessary communication of result operands when they are not necessary on the CPU.

In libraries such as CUBLASXT, this back-and-forth communication is hidden through pipelining. The computation is broken into chunks, which are distributed among the multiple GPUs. While each GPU is performing a chunk of computation, input data for the next chunk is concurrently sent to the GPU and output data from the previous chunk is retrieved from the GPU. Provided the computation is operating over sufficient data, most of the communication cost can be completely overlapped with computation. Note that the effectiveness of this overlap is dependent on properly choosing the chunk size for pipelining—CUBLASXT leaves the selection of granularity to the programmer, breaking the abstraction layer somewhat.

This pipelining strategy has a deleterious side effect: because the operands must be transferred to the GPU for *every* method call, and the implementation relies on overlapping communication with computation, the communication costs cannot be hidden for small inputs. Moreover, some linear algebra methods, such as SAXPY, simply do not contain enough computation to amortize the communication cost, regardless of how large the input data is (because communication cost grows at the same rate as computation time). Hence, such operations cannot be profitably executed on the GPU, even if there is opportunity to exploit the computation resources of multiple GPUs. As a result, CUBLASXT only provides multi-GPU implementations of BLAS Level 3 methods, while other libraries such as MAGMA also only provide a subset of LAPACK methods. The abstraction boundary imposed by the library interfaces to linear algebra routines precludes exposing communication management to programmers; the only way to support computation offloading efficiently is to automate the data management.

2.2 Multi-GPU automatic data management models

In recent years, researchers have proposed many multi-GPU frameworks to optimize communication and provide automatic data management, such as StarPU [4], FLAME [18], PTask [19], and StarSs [5]. These systems tackle the challenge of effectively managing communication between the CPU and multiple GPUs. However, they come with a different set of drawbacks than the drop-in libraries described previously. Fundamentally, these models rely on identifying all computations that operate on data that may be accessed by the GPU, allowing their runtime systems to track dependences on that data to manage communication. To support this dependence tracking, the systems either require adopting a new task-based programming model [4, 18, 19] or require annotations of all data accesses [5].

2.3 SemCache

In prior work, we proposed SemCache, a system for writing libraries suited for single-GPU offloading, backed by a variable-granularity cache to manage data movement between the CPU and GPU [2]. This section briefly summarizes SemCache's operation, as it forms the basis for SemCache++.

Like previous memory management systems for GPUs ([13, 17, 12, 9]), SemCache maintains what amounts to a distributed shared memory (DSM) between the CPU and GPU. SemCache tracks shared data at a variable granularity—as memory ranges. Memory ranges are tracked in a directory structure, along with their cache state: one of ***S***hared, ***C***PU exclusive, or ***G***PU exclusive. The granularity of a memory range is determined the first time an offloaded operation is performed on the memory. Libraries for GPU offloading use SemCache directives to register the memory ranges that are read and written by an offloaded operation. For example, in the BLAS operation DGEMM ($C+=A*B$), matrices A and B are registered as read, while C is registered as read and written.

SemCache uses page protection to determine when the CPU accesses data. Although access detection is done at page granularity, SemCache tracks and transfers data at the granularity of matrices. More fined grained systems like Cuda Unified Memory transfer matrices at the granularity of pages. For large matrices, page transfers result in significant slow downs as shown in Section 5.3.

In the single-GPU case, SemCache completely avoids any redundant or unnecessary communication. Data is only transferred between the CPU and GPU when one device or the other requires the data, and caching on the GPU allows offloaded operations to reuse data transferred by earlier operations. The main limitation of SemCache is the synchronous memory transfers which blocks the CPU until the GPU communication is done. Many features can be enabled with asynchronous transfers like: hybrid CPU/GPU parallel execution, and overlapping communication with computation. The following section explains how SemCache++ builds on SemCache to work with multiple GPUs and addresses its limitations.

3. SemCache++

This section introduces SemCache++, an extension of SemCache that supports multiple GPUs. SemCache++ automatically manages data movement and synchronization across SemCache++-enabled library calls. These libraries can thus be used as direct replacements for CPU libraries, providing the performance of hand-tuned multi-GPU implementations without breaking the abstraction boundaries of the library.

3.1 High Level Overview

A SemCache++-enabled library looks, to a programmer, like a typical CPU library. SemCache++ directives (embedded in the library code, not exposed at the interface level) specify what data (matrices) are read and written by the library call. SemCache++ libraries provide multi-GPU implementations by decomposing the computation into subtasks that operate over portions of the data (submatrices). These computations are then distributed across multiple GPUs as part of the library implementation (Section 4 discusses a concrete example of how such a library might be implemented).

SemCache++ can also support hybrid CPU/GPU execution. Since each submatrix is tracked and executed separately, the CPU can be also exploited in parallel with multiple GPUs to compute part of the result.

SemCache++ manages communication by tracking the locations of the submatrices, identifying whether it is on the CPU, or on one or more GPUs, or shared between the CPU and GPUs. Data is not eagerly communicated, but instead it is only transferred if it is needed by a computation. Because the data remains distributed after a library call completes, when a future library call is issued, the subtasks of that call can be dispatched to appropriate GPUs to reduce communication.

SemCache++ then ensures that the CPU and GPU(s) maintain a consistent view of data by transferring data back from the GPU(s) whenever the CPU requires the data. As in SemCache, SemCache++ determines when a CPU reads or writes data through the use of page protection. Note that while data on the GPUs is tracked at the granularity of decomposed inputs (submatrices), data on the CPU is tracked at the granularity of entire matrices. Thus, SemCache++ uses a two-level directory structure to track data, as described next.

3.2 Cache Design and Structure

SemCache++ uses a directory structure to track the status of data that is used during offloading operations. Figure 2 illustrates the design of this directory. As mentioned above, SemCache++ uses a two-level structure: the first level of the directory tracks matrices at the granularity they are used in library calls, while the second level is used to track the submatrices that are distributed across GPUs.

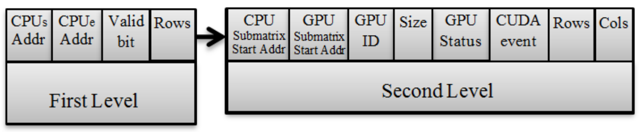


Figure 2: Structure of Caching Directory

The first level of the directory tracks the matrices that are involved in offloading operations, indexed by the CPU start address (CPU_s) and CPU end address (CPU_e). This first level also records the number of rows in the matrix (n_r), to support efficient memory transfer, as described in Section 3.2.1. Finally, the first level records whether the matrix is valid or invalid on the CPU.

SemCache++ assumes that matrices are stored contiguously in memory. When an offloadable method is invoked, the directory can be queried to see if entries exist for the matrix operands of the method. If not, an entry is created. Note that since library methods operate on entire matrices (matrices are only decomposed for distribution across multiple GPUs), lookups into the first level happen at the granularity of matrices.

The first-level entry for a matrix points to a set of *translation records* for the matrix. When a matrix is decomposed into submatrices and distributed across the GPUs, each submatrix is assigned a record in this second level. A translation record serves several purposes. First, it translates between the location of data on the CPU and the corresponding location on the GPUs, facilitating data movement between devices. Second, it keeps track of the coherence state of the data (*i.e.*, where valid copies of the submatrix reside). Finally, it tracks the ready state of the data (*i.e.*, whether the data is available for use by a task). The following sections describe these tasks in more detail.

When a task is launched to execute on a GPU, it uses SemCache++ directives to identify which submatrices are needed for the computation. If the data is already being tracked by the first level, SemCache++ checks the status of the required submatrices in the second level. If the data does not exist on the target GPU, communication is performed.

3.2.1 Translating between CPU and GPU Addresses and Transferring Data

A submatrix is a region of data within the range of a larger matrix. The submatrix may be copied to the GPU as row tiles or column tiles, the translation record stores the start address of the submatrix on the CPU as well as the number of rows and columns. The submatrices are stored contiguously on the GPU, so the translation record tracks the start address of the data on the GPU and the size of the data. Because a submatrix may be replicated on multiple GPUs, the translation record stores the GPU ID and the start address for each GPU the submatrix resides on.

This translation information is used to transfer data back and forth between the CPU and GPUs, as well as for inter-GPU transfers. Inter-GPU transfers are straightforward. If the submatrix is being moved to a GPU that does not currently have a copy of the submatrix, new space is allocated on that GPU and the translation record is updated to reflect the location of that space. When moving a submatrix from the CPU to the GPU, the row and column information stored in the translation record for the submatrix are used to generate a `cublasSetMatrix` call, which provides a single call to transfer an entire tile of a matrix to the specified GPU, allocating memory if necessary. Data tracking is done at the granularity of submatrices. When the CPU requires access to region of data computed on the GPU, only the corresponding submatrix is transferred back using a `cublasGetMatrix` call.

3.2.2 SemCache++ Coherence Protocol

SemCache++ uses a modified MSI coherence protocol to track which devices have valid copies of (sub)matrices. The states are tracked through the use of a *valid* bit in the first level entry for a matrix, as well as a *GPU Status* field for each submatrix in the second-level entry. The *CPU valid* bit in the first-level tracks whether or not the matrix is available to the CPU to speedup the lookup process. It is set when all submatrices have CPU only **C** status or shared **S** status. When the *CPU valid* bit is unset, each submatrix can have a different status and the GPU Status field in the second-level entry is used to determine the status as follows:

- **C:** Submatrix exclusive to CPU.
- **S:** Submatrix shared between CPU and GPU(s).
- **G:** Submatrix valid only on GPU(s).

The caching directory records transitions between states in the usual way, triggering communication if necessary. If a task dispatched to a GPU reads a submatrix that is not already on the GPU, then data is transferred (from the CPU if possible, as GPU-GPU communication is often slower), an entry for the submatrix is created, and the status in the second level is set to shared (**S**). SemCache++ allows multiple copies of the same submatrix to exist in the shared status; if another GPU wants the submatrix, then it receives a copy, too. However, like regular caches if a submatrix needs to be written to, all shared copies of the submatrix are discarded and only one submatrix holds a modified state (**G**).

If a matrix is *read* on the CPU while the first-level valid bit is unset, the GPU status is checked in the second level entries. If the status is **G**, submatrices are transferred from the GPU back to the CPU, the valid bit is set, and all submatrices change status to shared (**S** state). If a matrix is *written* on the CPU, then the status of the second level entries becomes **C**, with data transferred back from the GPUs if necessary.

Section 3.3 describes the CPU and GPU instrumentation that triggers the state changes in the coherence protocol.

3.2.3 Synchronization

The final purpose of the second-level translation record for a submatrix is to enable synchronization between tasks. To facilitate parallelism, and the overlap of communication and computation, tasks are launched asynchronously, using CUDA's streams. Moreover, this overlap can occur across library methods, if a second library call uses the same submatrices as the first library call.

Because tasks are launched asynchronously, and from multiple (possibly dependent) library calls, it is important that tasks do not begin to execute until their predecessor tasks complete.SemCache++ takes advantage of CUDA *events*: small kernels that can be launched to streams and act as signals. Each submatrix has an event handle associated with it, stored in the translation record. Whenever a submatrix is sent to a GPU, or when a submatrix is computed (modified) by a task, the operation is performed by dispatching the task to a stream on the target GPU. The event handle associated with the submatrix is then dispatched to the same stream using `cudaEventRecord`. The semantics of streams ensure that this event will not trigger until the previous operation (communication or computation) finishes. In other words, the event will not execute until the submatrix is up-to-date on the target GPU.

Before a communication or computation operation that needs a submatrix is dispatched, SemCache++ must make sure that the submatrix is up-to-date. The submatrix's event handle is dispatched to a stream using `cudaStreamWaitEvent`. This ensures that the operation will not commence until any previous `cudaEvent-`

Record events associated with the same handle have completed (even if those events were dispatched on different devices). Thus, tasks that require a submatrix will wait until operations that compute or transfer that submatrix complete. Essentially, SemCache++ uses events as full/empty bits, ensuring that consumers of a submatrix wait until producers complete.

3.3 Instrumentation

3.3.1 Instrumenting GPU Reads and Writes

To be able to track the status of GPU data correctly, you need to determine which data is read or written by the GPU. Prior work has used compiler analysis or programmer annotations to determine if the operation is a read or a write [13, 12, 9, 17]. Since SemCache++ focuses on libraries, it can use simple directives inserted into the library code to indicate which matrices are read and written by the GPU, as well as which submatrices are needed by tasks dispatched to various GPUs.

3.3.2 Instrumenting CPU Reads and Writes

Similar to SemCache, SemCache++ uses the operating system's virtual memory protection to automatically detect CPU reads and writes. Page protection can be used to limit access to the CPU data which has been sent to the GPU. For each submatrix that SemCache++ tracks on the CPU, SemCache++ sets page protection flags for all the pages the submatrix spans. The page protection flags are set according to the state of the data structure. If the submatrix is in G state, its pages are set to NO ACCESS; if the submatrix is in S state, its pages are set to READ ACCESS; and if the submatrix is in C state, the pages are set to READ and WRITE ACCESS.

If a CPU access triggers a page fault due to write to a no access region or a read only region, SemCache++ looks up the address that caused the page fault in the caching directory. If the translation record is found, it transfers the submatrix back from the GPU and the submatrix status becomes CPU only (C). If a CPU access triggers a page fault due to a read to a no access region, SemCache++ transfers the submatrix back from the GPU and the submatrix status becomes shared (S).

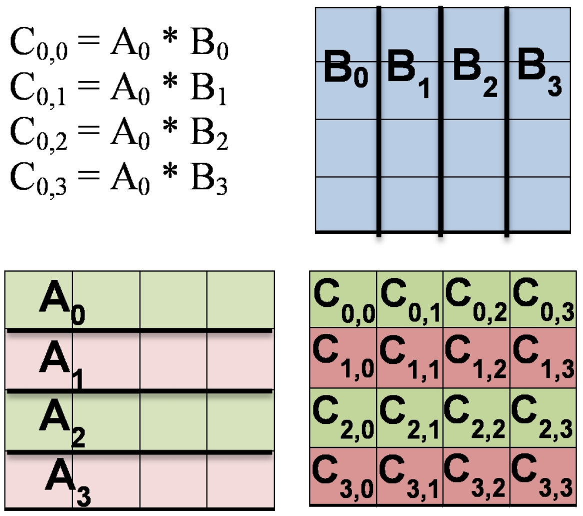
In order to avoid false sharing between matrices, memory allocation should be page-aligned and padded out to page boundaries.[1] Depending on the matrix decomposition, false sharing might also exit between submatrices which share the same page. If a single submatrix is modified by a GPU, all of the pages the submatrix spans are protected to no access. If this submatrix is invalidated, the other submatrix which shares the same page is conservatively invalidated and both submatrices are transferred back to the CPU.

Page aligned memory allocation can introduce some wasted memory which is negligible for larger data sizes. Usually, it is only profitable to offload medium to large data structures on the GPU to take advantage of the parallelism, where this overhead is minimal ($<$1% for 400x400 matrix). We note that this overhead is only introduced on the CPU side. On the GPUs, sub-matrices are allocated in variable sizes and do not have to be page aligned.

3.4 Managing Available GPU Memory

Using multiple GPUs increases the total available memory space. Kernels that do not fit in a single GPU memory can be executed on multiple GPUs. Although multi-GPU increases the caching space, data might occupy all of the free GPU memory. In such a situation, to allocate new data in the GPU memory, cached data must be freed. To determine which address ranges should be freed, SemCache++ uses least-recently-used (LRU) policy. Any data accessed on the GPU is added to the end of a queue. If the GPU memory is full, data at the head of the queue is removed. Note that depending on the application other polices can be used.

[1]While page aligning data requires some program modification, identifying allocations to modify is significantly easier than, for example, identifying data accesses to annotate.



$C_{0,0} = A_0 * B_0$
$C_{0,1} = A_0 * B_1$
$C_{0,2} = A_0 * B_2$
$C_{0,3} = A_0 * B_3$
B_0 B_1 B_2 B_3
A_0 A_1 A_2 A_3
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$

Figure 3: Matrix decomposition

4. ADAPTING A LIBRARY TO USE SemCache++

This section describes the process of building a library for multi-GPU offloading using SemCache++. First, we describe how a matrix multiply (DGEMM) call can be decomposed to distribute computations across multiple GPUs. Then we describe how SemCache++ directives can be used to perform automatic data management and synchronization.

4.1 Multi-GPU Decomposition and Scheduling

There are multiple parallel algorithms for solving matrix computations on distributed systems (*i.e.* ScaLAPACK [6]). Choosing the right algorithm depends on the underlying network and computing architecture. Many factors can be taken into consideration to determine which algorithm to use like load balancing, optimizing communication and computation-communication ratio. SemCache++ is not tied to a single algorithm, it can be used with any distribution algorithm. It can automatically cache and manage the communication with any type of these algorithms. In our implementation of DGEMM, we adopt a strategy similar to Song *et al.*, which takes advantage of locality to minimize communication [22].

DGEMM calculates $C = \alpha * A * B + \beta * C$. The distribution of the computation across N devices uses a straightforward decomposition. Each matrix is partitioned into N^2 submatrices (an $N \times N$ grid), each of which is tracked separately by SemCache++. For notational convenience, we consider that A's submatrices are grouped into N rows, $A_0, A_1, \ldots, A_{N-1}$, and B's submatrices are grouped into N columns, $B_0, B_1, \ldots, B_{N-1}$. The matrix multiplication is thus broken into N^2 tasks, with a row of A's submatrices being multiplied by a column of B's to produce a single submatrix of C. The decomposition and computation are shown pictorially in Figure 3.

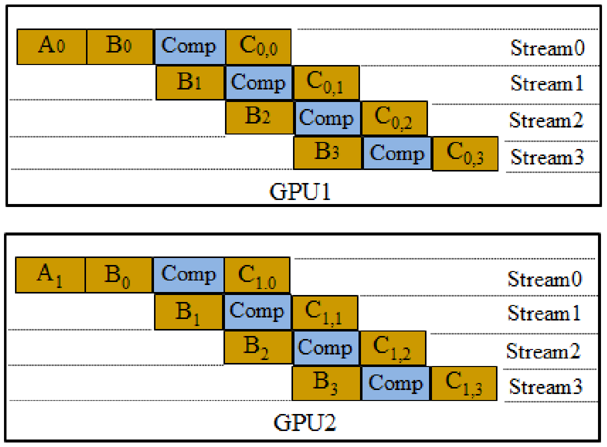


Figure 4: SemCache++ Computation scheduling

As in Song *et al.*, the computation is scheduled by (conceptually) distributing C's submatrices to the N GPUs by dividing the grid of submatrices evenly by rows. Tasks that compute each submatrix of C are then scheduled on the appropriate GPU. Independent tasks are assigned to different streams on the GPU, allowing the computation of one C submatrix to be overlapped with communicating the operands from B for the next task. Figure 4 shows how this pipelining can hide communication overheads.

Note that once the computation is completed, each GPU holds a row of A's submatrices and *all* of B. These submatrices remain on the GPUs until another device wants the data. If subsequent calls use the same matrices, mapping tasks to the appropriate GPUs can avoid communication.

For less compute intensive BLAS routines like level 1 and 2, a simple decomposition can be used. Each matrix can be split by rows into multiple sub-matrices. The number of sub-matrices is equal to the number of GPUs. Each GPU performs part of the computation. Sending the data to the GPU can be pipelined but it has little effect on the performance since the percentage of communication is much higher than the computation.

4.2 SemCache++ Directives

SemCache++'s API for identifying which computations a task needs is similar to the API defined in SemCache [2], extended to support multiple GPUs. SemCache++ requires that the programmer to specify the number of GPUs using the API method: `SemCacheDeviceSelect(devicesNumber,deviceIds)`. As in SemCache, `readGPU` is used to indicate to SemCache++ that a region of memory (in this case, a submatrix) will be read during a GPU task; the only difference is that in SemCache++, the GPU that will read the submatrix must be identified. Analogously, `writeGPU` is used to indicate that a submatrix was modified by a particular GPU after a task, potentially triggering invalidation of the submatrix on other GPUs or on the CPU.

Because it is common to distribute the entire matrices at once, SemCache++ also provides aggregate versions of `readGPU` and `writeGPU` that operate over a whole matrix, decomposing and distributing the matrix across the GPUs. These aggregate functions automatically decompose a matrix into multiple submatrices and distribute the submatrices by rows or columns to the GPUs. If the submatrix requires further decomposition, it is decomposed into N^2 submatrices. Different decomposition and distribution algorithms exist in SemCache++. Since the decomposition algorithms are not tightly coupled with SemCache++, new algorithms can be easily defined and used. `DecomposeRow` and `DecomposeCol` distribute submatrices by rows and columns, respectively. Figure 5 shows how these aggregate functions can be used to manage submatrices for matrix multiply.

```
SemCacheDgemm(TRANSA,TRANSB,M,N,K,ALPHA,
   A,LDA,B,LDB,BETA,C,LDC) {
  //A stored on CPU in memory range [A, A+(M*K*8))
  //A will be decomposed and sent to multiple GPUs, its
       state will be "S"
  entryA = readGPU(A, M, K, DecomposeRow)

  //B stored on CPU in memory range [B, B+(K*N*8))
  //B will be decomposed and sent to multiple GPUs, its
       state will be "S"
  entryB = readGPU(B, K, N, DecomposeCol)

  //C stored on CPU in memory range [C, C+(M*N*8))
  //If BETA!=0, C will be decomposed and sent to multiple
       GPUs, its state will be "S"
  entryC = readGPU(C, M, N, DecomposeRow)

  foreach GPU{
    foreach stream{
    //Perform computation on submatrix
    cublasDgemm(stream,
      TRANSA,TRANSB, Atiles ,Btiles ,K,ALPHA,
      entryA.subRecord.gpu_s,LDA,
      entryB.subRecord.gpu_s,LDB,BETA,
      entryC.subRecord.gpu_s,LDC)

    //Issue synchronization event for submatrix C
    cudaEventRecord(entryC.subRecord.sync_event, stream);
    }
  }
  //C was written by cublasDgemm
  //Each C block state will be updated to GPU only "G"
  writeGPU(C, M, N, DecomposeRow)
}
```

Figure 5: Pseudocode of SemCache++ matrix multiply (DGEMM)

Inside the `readGPU` call, a lookup in the caching directory is performed using the start and end address on the CPU and the translation record is returned if found. If data does not exist on the GPU, the matrix is decomposed using the specified decomposition algorithm and sent to multiple GPUs asynchronously as described previously. If data already exist on the device, each submatrix record is inspected as follows: If a submatrix already resides on the designated GPU, no communication is necessary. If the submatrix is not valid or not on the designated GPU, a synchronization event is issued to ensure that the submatrix is up-to-date, communication is performed and the directory state is updated appropriately. `readGPU` also page-protects the CPU page(s) containing the matrix as read-only, as discussed in Section 3.3.

Pinned memory is used to allow overlapping transfers to multiple devices in parallel, it also allows concurrent communication in both direction on Fermi GPUs. Pinned memory allocates page-locked (non-swappable) memory which enables a DMA on the GPU to request transfers to and from the host memory without the involvement of the CPU.

Once all the data is transferred, the individual tasks are executed. Note that all of these kernel invocations occur asynchronously, and hence can be executed simultaneously (there are no dependences in DGEMM). However, because subsequent library calls might use the matrix C, after each task that computes C, `cudaEventRecord` is called on the submatrix's synchronization event so that later tasks wait until the submatrix is computed.

Finally, `writeGPU` changes the state of all C submatrices to GPU modified (**G**). To ensure that CPU accesses to C wait until the computation is complete and then transfer data back from the GPUs, `writeGPU` changes the page protection on C to no access.

4.3 Using SemCache++ with Complex Data Structures

While SemCache++ provides helper methods to aid in distributed matrices across multiple GPUs, not all data structures are amenable to such predictable partitioning and distribution (*e.g.*, 3D matrices, or irregular structures such as trees and graphs). In such cases, SemCache++'s low level API (`readGPU` and `writeGPU`) can be used to distribute those data structures by invoking the appropriate methods on each address range for the data structure. It becomes the library writer's responsibility to appropriately distribute the data structure. For example, to distribute a 3D matrix, SemCache++'s methods can be called individually on the address ranges for each submatrix (multiple transfer calls are needed for non-contiguous matrices) to transfer the matrix and distribute it according to the library writer's distribution algorithm. Performing distribution using the low-level methods obviates the benefits of SemCache++'s distribution functions and multi-level state tracking, but does not preclude the use of its automatic data movement capabilities.

Distributing Sparse Matrices.

As an example of distributing more complex data structures, we have used SemCache++ to provide offloading support for sparse-matrix libraries. Sparse matrices present an interesting challenge to most systems for managing communication between the CPU and the GPU because of their complex layout: a sparse matrix in CSR form has a data array, a row sum array and a column index array. Splitting the matrix between multiple GPUs requires carefully splitting the column index array and recomputing the row sum array.

SemCache++ handles distributing sparse matrices by delegating the distribution to the library implementation. The library can split the sparse matrix representation, recalculating the row sum arrays for each submatrix as necessary. SemCache++ tracks the individual arrays representing the sparse submatrix as separate submatrices. Recall that SemCache++ tracks submatrices according to a start address, number of columns and number of rows. SemCache++'s tracking of sparse matrices hence works as for any other data structure: if a task requires accessing the sparse matrix, the library issues `readGPU` calls for each of the components of the sparse matrix, and communication is performed as necessary.

This strategy for handling sparse matrices highlights a key advantage of SemCache++'s library-integrated approach to multi-GPU offloading over other approaches. The row sum arrays that are distributed across GPUs have *different contents* than the row sum array that resides on the CPU. Nevertheless, the abstract state of the sparse array is the same: the same data is stored in two different representations, depending on whether it resides on the CPU or on the GPU. SemCache++ establishes a *semantic link* between the two representations, allowing state changes on one device (*e.g.*, changing the contents of the sparse matrix on the CPU) to be reflected on other devices (*e.g.*, by invalidating all of the sparse submatrices on the GPUs).

5. EXPERIMENTAL EVALUATION

To evaluate SemCache++, we built multi-GPU implementations of the library interfaces provided by CUBLAS and CUSPARSE (NVIDIA's single-GPU linear algebra libraries) using SemCache++ directives to manage communication and synchronization. The internal, per-GPU tasks of the SemCache++ implementations were used the single-GPU CUBLAS and CUSPARSE implementations, as described in Section 4.2.

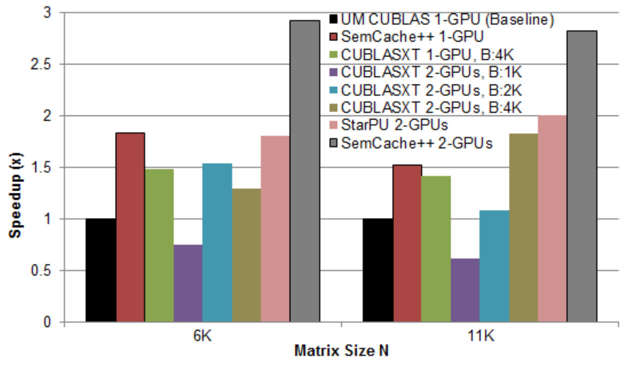


Figure 6: Speedup of microbenchmark for different matrix sizes, normalized to UM CUBLAS 1-GPU)

We evaluated three benchmarks: First, we looked at a microbenchmark that allowed us to investigate the behavior of SemCache++ as well as other multi-GPU libraries in depth. Next, we looked at two case studies of using SemCache++-enabled libraries to offload computation in two solvers: Jacobi iterative solver (which used dense matrices), and conjugate gradient (which used sparse matrices). The conjugate gradient code is taken directly from NVIDIA's CUDA benchmark suite. We compared NVIDIA's hand tuned single-GPU implementations (and UM implementations when available) with multi-GPU implementations that used SemCache++, StarPU and CUBLASXT.

We used two platforms to conduct our experiments. Most of our experiments were performed on a server with AMD Opteron Processors and 32GB memory connected via PCIe 2.0 to two NVIDIA Kepler K20 GPUs. These GPUs support compute capability 3.5 (allowing us to use NVIDIA's Unified Memory as a baseline). The second platform was used to evaluate offloading to more than two GPUs but it does not support UM. The host has AMD Opteron Processors and 64GB memory connected to eight Tesla M2090 GPUs in an external PCIe expansion chassis. While this platform let us scale to more GPUs, the external configuration of the GPUs meant that communication between the host and the GPUs was much slower. We refer to the first platform as **kepler** and the second as **tesla**.

5.1 Microbenchmark Performance Evaluation

To understand the behavior of SemCache++-enabled applications, we wrote a simple microbenchmark that performs two matrix multiplies and a DAXPY: $D = AB + AC$. Note that the two matrix multiplies share one of their operands (A), and the DAXPY operates on the results of the two multiplications. As a baseline, we used CUDA 6's unified memory along with CUBLAS to implement a communication-optimized single-GPU version of the microbenchmark. We compared this baseline to SemCache++, CUBLASXT and StarPU using one and two GPUs. Unlike SemCache++ and CUBLASXT, StarPU implementation requires rewriting the benchmark using their programming model. CUBLASXT supports multi-GPU computation by carefully overlapping communication with computation. Its performance is dependent on setting the block size for this pipelined schedule. Hence, we evaluate several different block sizes for CUBLASXT on two GPUs. These experiments were conducted on **kepler**.

Figure 6 shows the results of the microbenchmark experiment, looking at two different matrix sizes (11K$\times$11K matrices were the largest that could fit on a single GPU for the microbenchmark). We see that even on a single GPU, both SemCache++ and CUBLASXT are faster than the baseline—this is because the baseline does not

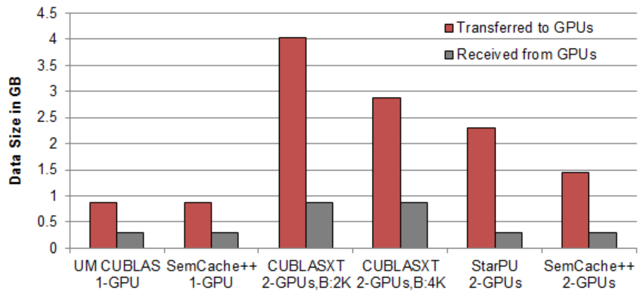


Figure 7: Microbenchmark communication results for size N=6K

overlap communication with computation, while both SemCache++ and CUBLASXT do. SemCache++ is faster than CUBLASXT because it is able to minimize communication. The A matrix is cached on both GPUs, as are the results of the DGEMMs. Hence, the DAXPY can be performed with no additional communication. In contrast, CUBLASXT, which does not leave the DGEMM results on the GPUs, must communicate the results of the DGEMMs *back* to the GPUs to perform the DAXPY.

When scaling to two GPUs, we find that SemCache++'s advantage increases: it is nearly 3× faster than the baseline, and 30-50% faster than CUBLASXT and StarPU. Although StarPU reduces redundant communication and allows overlapping communication with computation, when synchronization was used to produce correct results, overlapping was limited. Additionally, communication was not fully optimized which made it slower than SemCache++. StarPU results were similar for both: the DMDA prefetching scheduler and the eager scheduler.

We note here a further problem of CUBLASXT's reliance on computation/communication overlap: the optimal block size depends on the input matrix size. In fact, the default block size for CUBLASXT (1K) results in slower performance than a single GPU! This sort of tuning is not necessary for SemCache++, which decomposes the matrix into equal blocks regardless of input size (as described in Section 4.1). Instead, SemCache++ derives its performance improvement from avoiding redundant communication entirely.

To better understand where SemCache++'s advantages lie, we investigated two possible sources of performance improvement. First, we measured the performance of a *single* matrix multiply using SemCache++'s library and using CUBLASXT. We found that even with the optimal block size, SemCache++'s DGEMM implementation is slightly faster, about 10% for 11K matrices. We speculate this is because SemCache++ uses a simpler matrix distribution than CUBLASXT, resulting in slightly more efficient communication of the matrix operands.

The remainder of SemCache++'s performance improvement comes from optimized communication. Figure 7 shows the amount of data transferred to and from the GPU for 6K×6K matrices. SemCache++ transfers significantly less data than CUBLASXT and StarPU. Note that this figure reflects two sources of additional communication. First, StarPU's and CUBLASXT's less efficient matrix decomposition requires more communication to perform a matrix multiplication (this effect is reflected in SemCache++'s 10%-faster DGEMM than CUBLASXT). Second, in the case of CUBLASXT, matrices are re-transferred across library calls, while SemCache++ avoids this communication.

Scalability: Finally, we investigated the scalability on multiple GPUs. Figure 8 shows the microbenchmark performance on the

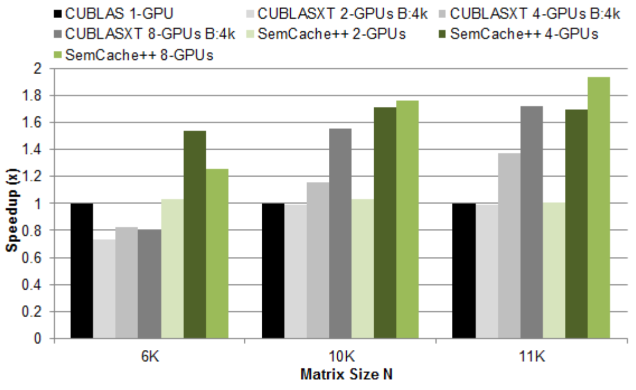


Figure 8: Microbenchmark performance on multiple GPUs for different matrix sizes, speedups with respect to CUBLAS 1-GPU)

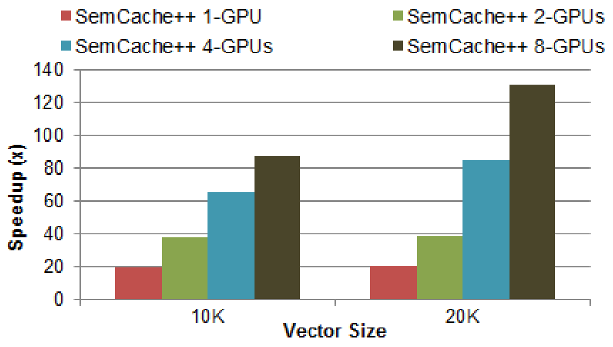


Figure 9: Speedup of Jacobi, normalized to unoptimized CUBLAS

tesla platform, running on up to 8 GPUs. For each matrix size, we show the best-performing CUBLASXT block size. Speedups are limited because communication from the host to the external GPUs is slow, and, unlike with internal GPUs that can take advantage of direct DMA transfers, with external GPUs the bandwidth is divided and hence per-GPU bandwidth decreases with scale. Nevertheless, with the largest matrices, where there is enough computation to amortize the slow communication, we see that SemCache++ is able to provide increasing performance up to 8 GPUs, and is faster than CUBLASXT running on the same number of GPUs.

5.2 Case Study(I): Jacobi Iterative Solver

The Jacobi iterative solver performs the repeated MvM computation described in the introduction. Figure 9 shows Jacobi performance for different vector sizes on the **tesla** platform. Speedups are normalized to the unoptimized CUBLAS implementation. The unoptimized version provides encapsulation; the A matrix is sent to the GPU in every iteration. Running the unoptimized CUBLAS implementation on multiple GPUs did not gain any speedups because communication cost was dominant so the results are not included in the figure. Running Jacobi using SemCache++ on a single GPU achieved 20x speedup because matrix A is cached. For large vector sizes. SemCache++ achieved linear speedups on multiple GPUs. As described in the introduction, each GPU computes part of the vector in each iteration and SemCache++ automatically sends the partial vectors to each GPU using peer to peer transfers. The communication is naturally overlapped, which minimizes the overhead. Note that unlike in our microbenchmark, the ratio of computation to communication is high enough that the slow PCIe bus does not limit scalability.

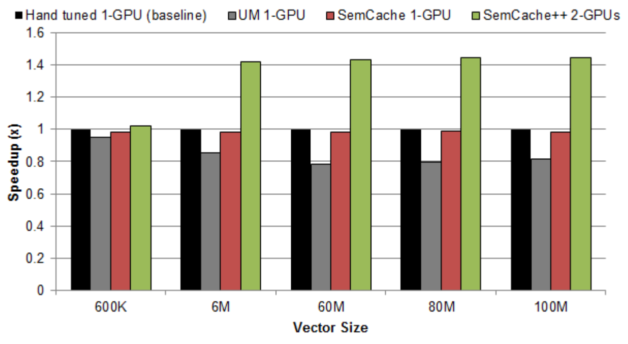


Figure 10: Speedup of CG, normalized to (Hand-tuned 1-GPU)

5.3 Case Study(II): Conjugate Gradient

NVIDIA provides two variants of conjugate gradient (CG) in its benchmark suite. In the first, communication is hand-tuned, while in the second, unified memory is used to manage communication. We use the first implementation as the baseline. Because SemCache++ enables drop-in library replacements for BLAS operations, we were able to directly use NVIDIA's unified memory code with SemCache++ to provide multi-GPU offloading.

CG uses CUBLAS and CUSPARSE libraries. Sparse matrix multiplication (SpMV) from the CUSPARSE library uses Compressed Sparse Row (CSR) format for storing matrices. The generated matrix is symmetric tridiagonal. Since the matrix is symmetric any split is balanced, we choose to split it by the number of GPUs. The rows sum is calculated per split. SpMV requires the entire vector for the matrix multiplication. Parts of the vector are computed on each GPU. SemCache++ detects from the caching directory entries that the vector is split on multiple GPUs and it automatically initiates communication to broadcast the vector to all GPUs. Each part of the vector is communicated to the other GPUs using direct GPU to GPU communication. The transfers are overlapped in both directions to double the bandwidth. It is important to note here that this is a general approach in SemCache++. It works for any SpMV kernel, there are no special optimizations done for CG.

Figure 10 shows CG performance for different vector sizes. Speedups are normalized to the hand-tuned implementation's execution time on a single GPU. SemCache performance on a single GPU is very close to hand-tuned performance, with 2% overhead due to cache lookups. Unified Memory (UM) is slower than hand-tuned because data transfers from the GPU to the CPU are done at the granularity of pages. Using SemCache++ with 2 GPUs we achieved 1.4x speedup on average over the single GPU hand-tuned baseline version for vector sizes larger than one million. Note that SemCache++ uses NVIDIA's DirectGPU capabilities when transferring data between GPUs. Nevertheless, for smaller matrix sizes, the overhead of peer-to-peer communication between GPUs limits the performance improvement. These results are consistent with the results of other multi-GPU conjugate gradient solvers [23], which found that the main limiting factor for their speedup was peer-to-peer communication.

Large problem sizes: Multi-GPU execution can not only be used to improve performance; it can also be used to run larger problem sizes than would fit on a single GPU. Splitting the matrix on 2 GPUs enabled us to run CG with sizes double the size that can fit in a single GPU. For example, we were able to run CG for a vector size of 100M on a single GPU using either single-GPU implementation, while with SemCache++ we were able to run double that size (200M) on two GPUs.

6. RELATED WORK

6.1 Programming Models

Aside from using multi-GPU enabled libraries for offloading, another approach to exploiting multiple GPUs is to use programming models that target general heterogeneous platforms. These approaches tend not to be suitable for library-based offloading, for various reasons. Kim *et al.* develop compiler tools that can automatically distribute an OpenCL kernel across multiple GPUs [14]. However, this work focuses on splitting a single kernel across GPUs, and does not consider how to optimize communication across kernels. MGPU [21] and Trilinos [10] libraries allow the programmer to specify the communication at a high level and the library automatically distributes and executes the workload on multiple GPUs. Unlike SemCache++, such libraries depends on the programmer to manually optimize communication across kernel calls.

StarPU [4], FLAME [18] and PTask [19] are task-based programming models for mapping applications to heterogeneous architectures. Such models are essentially similar: computation tasks are identified by the programmer, with dependences provided either through implicit or explicit dependence information. A runtime system then maps those tasks across the CPU and multiple GPUs, preserving dependences while trying to minimize communication. The fundamental drawback to the previous approaches is that they require writing the entire program in a task-based programming model (or using special APIs as in FLAME). Thus, these models cannot be used to provide library-based offloading, as even the non-library portions of the application must be modified to conform to the model, precluding a "drop-in" replacement for existing linear algebra libraries.

StarSs [5], OpenMP accelerator model [7] and OpenACC [1] are directive based programming languages. They require the programmer to annotate tasks with input/output information in addition to specifying the data movement. The runtime system can decompose and execute the tasks on multiple GPUs. OpenMP and OpenACC support manual caching and require the programmer to manually manage the CPU/GPU coherence. StarSs automatically caches data on the GPU for annotated tasks. Unlike drop-in replacement libraries such as those provided by SemCache++, annotations are prone to errors and they are not enough to automatically manage communication. While such models can be used to encapsulate several tasks into library calls, all computation over the data accessed during those calls must be annotated with directives, *including computations meant to execute on the CPU*. If data is cached on the GPU, any CPU access to the data needs to be annotated to maintain the coherence. Identifying such accesses for annotation is not practical for large-scale applications.

6.2 Automatic Memory Management

In the single-GPU setting, there is substantial prior work on automatic data management [13, 12, 9, 17]. Some approaches rely on compiler-assisted software coherence [13, 17], limiting applicability and scalability. Other DSM like approaches attempt to provide the appearance of a single memory space that is shared by the CPU and GPU, where data on the CPU and GPU share the same address [12, 9]. NVIDIA's Unified Memory [15] takes the same approach. Such single memory space models use the same masked address for data allocated on the CPU and the GPU to simplify address translation. If this direct address mapping is extended to multiple GPUs, each GPU needs to reserve the space for the entire matrix although only a sub-matrix is allocated on each GPU, resulting in wasted GPU memory. As a result, applications where the overall footprint is larger than a single GPU memory will not work.

Furthermore, applications where the data layout changes between the CPU and the GPU (as in the sparse matrix example) cannot be handled.

7. CONCLUSION

Multi-GPU libraries hide the complexity of decomposing data, distributing computations and handling communication manually inside library calls. Such encapsulation prevents the reuse of the data between successive kernel invocations resulting in redundant communication. In this paper, we introduced SemCache++, a semantics-aware GPU cache that automatically manages and optimizes communication between the CPU and multiple GPUs. We applied SemCache++ to the BLAS library to provide drop-in libraries that offload computation to multiple GPUs, and showed that our system can eliminate redundant communication and deliver significant performance improvements over multi-GPU libraries like CUBLASXT.

Acknowledgment: This research is supported by the Department of Energy under contract DE-FC02-12ER26104.

8. REFERENCES

[1] Openacc directives for accelerators. `http://www.openacc-standard.org`, 2011.

[2] N. AlSaber and M. Kulkarni. Semcache: Semantics-aware caching for efficient gpu offloading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 421–432, New York, NY, USA, 2013. ACM.

[3] 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.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874. Springer Berlin Heidelberg, 2009.

[5] 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, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster scalapack (dense linear solvers). *Computers, IEEE Transactions on*, 50(10):1052–1070, Oct 2001.

[7] J. Beyer, E. Stotzer, A. Hart, and B. de Supinski. Openmp for accelerators. In B. Chapman, W. Gropp, K. Kumaran, and M. MÃijller, editors, *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin Heidelberg, 2011.

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

[9] 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, New York, NY, USA, 2010. ACM.

[10] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[11] 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.

[12] 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, New York, NY, USA, 2012. ACM.

[13] 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.

[14] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[15] NVIDIA. Cuda. `https://developer.nvidia.com/cuda-toolkit`.

[16] NVIDIA. Cuda toolkit 6.0 cublas library. `http://docs.nvidia.com/cuda/cublas/index.html`, 2014.

[17] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 33–42, New York, NY, USA, 2012. ACM.

[18] 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, New York, NY, USA, 2009. ACM.

[19] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[20] P. D. S. Tomov, R. Nath and J. Dongarra. Magma version 0.2 user guide, 2009.

[21] S. Schaetz and M. Uecker. A multi-gpu programming library for real-time applications. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'12, pages 114–128, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM.

[23] M. Verschoor and A. C. Jalba. Analysis and performance estimation of the conjugate gradient method on multiple {GPUs}. *Parallel Computing*, 38(10-11):552–575, 2012.