Semantics-rich Libraries for Effective Exascale Computation
Progress Report: March 2013

DOE Award #: DE-FC02-12ER26104
Recipient: Purdue University, Sandia National Labs
PI: Milind Kulkarni (PI, Purdue University), Sam Midkiff (Purdue), Vijay Pai (Purdue), Arun Prakash (Purdue), Michael Parks (PI, Sandia National Labs)

1 Overview

This project period encompasses the ramp-up phase of SLEEC. We have hired 5 students to work on the project (4 directly funded by the project, 1 on fellowship). We have focused on four tasks over the past 6 months:

- Initial work on exploiting domain semantics and cost models to optimize locality and parallelism in computational mechanics applications (PIs involved: all)
  - PIs involved: all
  - Allows multi-scale applications to be automatically restructured for specific problem instances by exploiting semantic properties such as commutativity and associativity.
  - Demonstrated substantial improvements over domain-expert-optimized code
  - Begun work on extending approach to other multiscale applications
  - Paper under submission to ICS 2013
  - More details in Section 2
- Initial work on using annotated domain libraries in conjunction with domain-aware run-time to optimize communication for heterogeneous architectures
  - PIs involved: Kulkarni
  - Allows redundant communication to automatically be identified and eliminated while taking advantage of domain semantics to reduce run-time overheads
  - Paper under submission to ICS 2013
  - More details in Section 3
- Begun developing common IR for use in optimizing domain applications by exploiting domain knowledge
  - PIs involved: Kulkarni, Midkiff, Pai
  - Based on system dependence graphs and dependence flow graphs, with library operations represented as single nodes
  - Can apply transformations directly to this representation
  - First submission targeted for Summer 2013
- Begun work on identifying properties of Trilinos components that can be exploited to optimize applications for different target architectures

  - PIs involved: Kulkarni, Prakash, Parks
  - Looking at approaches to "swap out" solvers, preconditioners, etc.
  - Goal: automatically generate inspectors to transform code at run-time based on input and architecture.

Sections 2 and 3 provide more details on the first two tasks; the last two tasks are still in their preliminary stages.

Unexpended funds We still have funds to hire two additional graduate students; spending these funds is contingent upon finding the right students for the tasks.

2 Optimizing Computational Mechanics Codes

2.1 Summary

An important emerging problem domain in computational science and engineering is the development of *multi-scale computational methods* for complex problems in mechanics that span multiple spatial and temporal scales. An attractive approach to solving these problems is *recursive decomposition*: the problem is broken up into a tree of loosely coupled sub-problems which can be solved independently and then coupled back together to obtain the desired solution. However, a particular problem can be solved in myriad ways by coupling the sub-problems together in different tree orders. The space of possible orders is vast, the performance gap between an arbitrary order and the best order is potentially quite large, and the likelihood that a domain scientist can find the best order to solve a problem on a particular machine is vanishingly small.

We have developed a system that uses domain-specific knowledge captured in computational libraries to optimize code written in a conventional language. The system generates efficient coupling orders to solve computational mechanics problems using recursive decomposition. Our system adopts the inspector-executor paradigm, where the problem is inspected and a novel heuristic finds an effective implementation based on domain properties evaluated by a cost model. The derived implementation is then executed by a parallel run-time system (Cilk) which achieves optimal parallel performance. We demonstrate that our cost model is highly correlated with actual application runtime, that our proposed technique outperforms non-decomposed and non-multiscale methods. The code generated by the heuristic also outperforms alternate scheduling strategies, as well as over 99% of randomly-generated recursive decompositions sampled from the space of possible solutions.

This work is under submission to ICS 2013.

2.2 Preliminary findings

2.2.1 Multi-scale methods

Multi-scale methods for computational mechanics have been an area of significant research interest in recent years [17, 8, 10]. These methods allow mechanical systems, both static and dynamic, to be simulated with vastly differing spatial and temporal scales in different parts of the domain. This allows areas of interest to be investigated at fine granularity but at high computational cost, while other portions of the problem can be approximated with a much coarser-grain simulation. Multi-scale methods thus hold the promise of simulating complex systems at the necessary level

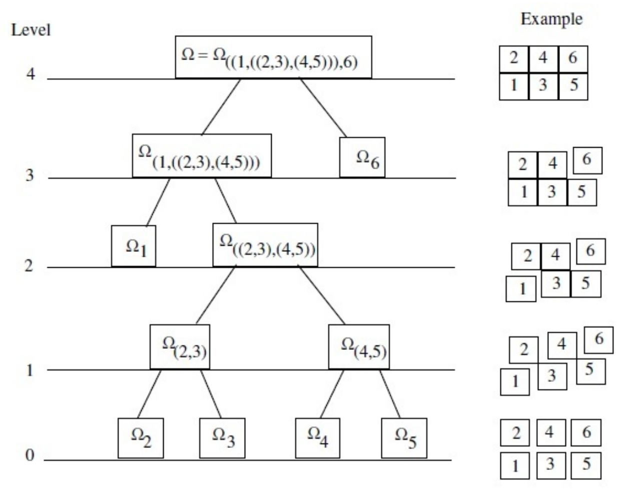
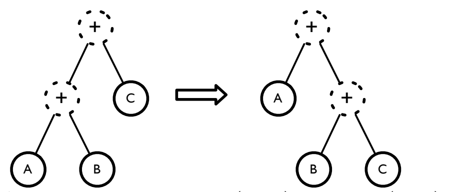


Figure 1: A Decomposed Problem Domain and Corresponding Recursive Coupling Order

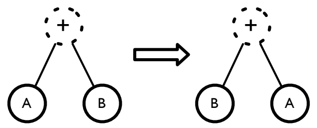
of resolution without incurring the cost of such fine-grained simulation throughout the problem domain. The multi-scale strategy is thus even more aggressive in achieving computational efficiency than well-studied strategies such as adaptive mesh refinement (AMR) that only adopt multiple spatial scales.

An attractive approach to multi-scale simulation that has been investigated in recent work is *recursive domain-decomposition* [11, 4, 3, 22, 26, 14]. A large problem is broken up into a set of loosely-coupled subdomains; these subdomains can be solved independently, except for points on potential shared *interfaces* that connect them. These shared interfaces require that the solutions of each individual subdomain be coupled to ensure that their solutions at the interface are consistent. As long as the interfaces are small relative to the subdomain sizes, it is computationally advantageous to decompose a large system versus solving the system as a single entity. For the particular case of recursive bisection adopted in this study, coupling is a pair-wise operation, and as subdomains are coupled, a solution is obtained for the combined, larger domain. Hence, by hierarchically coupling the subdomains, the solution for the overall system can be obtained. We describe this coupling order using a *coupling tree.* Figure 1 shows an example decomposition, where the problem domain is decomposed into six subdomains, which can be solved independently and coupled together according to the coupling tree in Figure 1.

A critical point about this hierarchical approach to solving multi-scale problems is that the *structure* and *topology* of the coupling tree has a significant effect on the performance of the algorithm. Because the coupling operation is both commutative and associative, a vast number of unique coupling trees are possible (945 for a problem with just 6 subdomains), making it highly unlikely for all but the simplest problems that a domain scientist will adopt an effective coupling order. Furthermore, because the various relevant parameters associated with computational costs are problem-dependent, finding the optimal coupling tree becomes even more difficult, as no single approach for finding an optimal tree coupling order may work for all different problems.



(a) Associative rotation: $(a \cdot b) \cdot c \equiv a \cdot (b \cdot c)$



(b) Commutative swap: $a \cdot b \equiv b \cdot a$

Figure 2: Possible Optimizing Operations on Trees

2.2.2 Coupling trees and semantic properties

The important properties of a coupling operation are that they are both associative and commutative. Coupling two subdomains A and B is identical regardless of order, and coupling (A B) with C is no different than coupling A with (B C). Thus, given a particular coupling tree, we can create a new, equivalent tree by re-associating the coupling operations. This is equivalent to performing tree-rotations on any pair of coupling operations, as shown in Figure 2(a) or by performing commutative swaps of the tree, as shown in Figure 2(b).

By performing a series of these commutative and associative manipulations, we see that *any* binary tree with a particular set of leaf nodes represents a valid coupling order for a given problem.

When we move to *multi-scale* trees, where different subdomains may be solved at different time steps, the constraints on coupling trees change. In particular, *all* subdomains at a given time step must be coupled together before they are coupled with other time steps. While it may seem that this property is hard to capture within the semantics of associativity and commutativity, we can do so by extending the properties of subdomains and the semantics of coupling. Each subdomain has a time step associated with it, and coupling two subdomains produces a new subdomain with a time step equal to the larger of the two time steps. We then add an additional constraint to associative rotations in coupling trees: a sequence of coupling operations are associative *only if all subdomains involved are at the same time step.* We note that given a valid starting tree, this restriction on associativity ensures that any tree derived from applying transformations will also be valid.

2.2.3 Cost Model

A cost model was developed to accurately reflect the execution time of the TreeSolve code prior to solving the coupling tree. This is done by taking the tree and basing the costs on the size of the input system and interfaces once the structure of the tree is known. This can be done by measuring the matrix size of all operations to estimate the time it will take to run during actual execution.

With this cost model, our inspector can access cost values of different trees without having to solve them. In our results, we show that the cost values correlate well with actual runtimes, enabling us to use heuristics based on the models to create effective coupling trees.

Figure 3 shows CDF of tree cost for 1000 randomly generated trees, for a given computational mechanics problem decomposed into 8 subdomains. It is observed that by picking only 1000 randomly coupled trees out of the whole tree space ($< 1\%$ of the space), the costs vary substantially, and between two coupling configurations, the costs can differ by an order of magnitude. Our work presents a way to find a configuration that obtains a near-optimal runtime among the space of coupling trees.

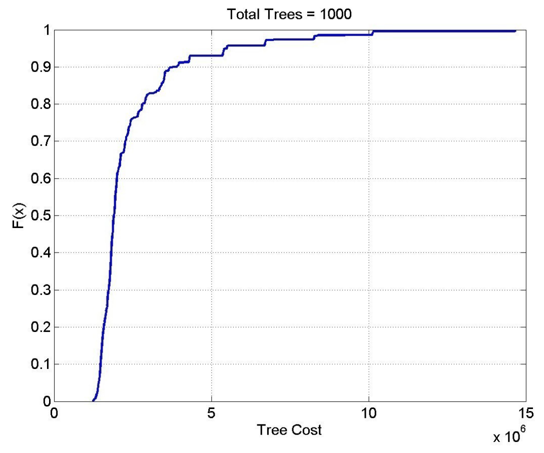


Figure 3: CDF of 1000 Random Trees based on tree costs

2.2.4 Tree-building and execution

To find an optimized coupling tree for a given problem, the inspector phase analyzes the input problem and views the partitioned mesh as an undirected graph. In the graph, each node represents a subdomain, and edges represent shared interfaces between them. In other words, the graph represents the topology of the subdomains. We use domain knowledge obtained from our cost model and apply the coupling and subdomain costs to the edge and node weights in our graph: a node's weight is calculated based on the leaf-node cost model, and is the cube of the number of equations in the subdomain. Precise edge weights are harder to determine, as they rely on the overall structure of the subtree rooted at a coupling node. However, the primary cost of a coupling operation is proportional to the size of the interface between the domains being coupled. Hence, edge weights are set to the interface size of the two subdomains connected by that edge.

We then use METIS to perform repeated bisections of this graph. METIS attempts to create balanced partitions while minimizing the weight of cut edges. The cost of a coupling operation is proportional to the interface size between the two domains being coupled. When the graph is bisected, the two sets of nodes represent the two domains that will be coupled, and the interface between those domains is exactly captured by the edges that are cut. Hence, because edge weights are determined by interface size, by minimizing the weight of cut edges, METIS naturally produces low-cost coupling operations. Second, the dominant cost in the TreeSolve algorithm is the cost of solving leaf nodes for a tree. Because these costs are exactly captured by the weights on nodes, by attempting to balance the two partitions of a graph, METIS naturally produces balanced trees. Hence, by incorporating knowledge of our domain-specific cost models into the top-down tree-building framework, we can produce low-cost, well-balanced trees.

After producing a coupling tree, the inspector transforms this tree into an execution schedule, which is passed to the executor phase for execution. Our approach provides two executors: a sequential executor and a parallel executor. Both executors uses platform-optimized BLAS [5] and LAPACK [2] routines to efficiently compute the matrix solutions for a system using the hierarchical method. For our parallel implementation, we use Cilk [6] to obtain optimal parallel performance.

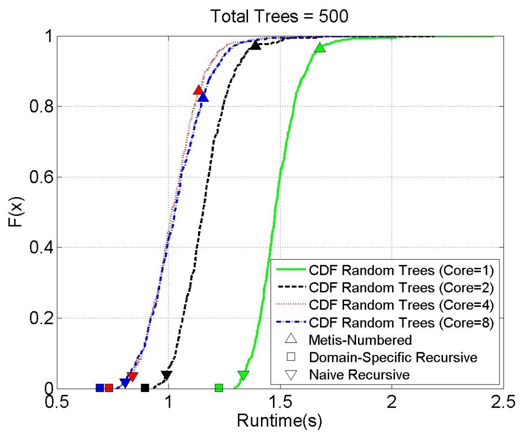


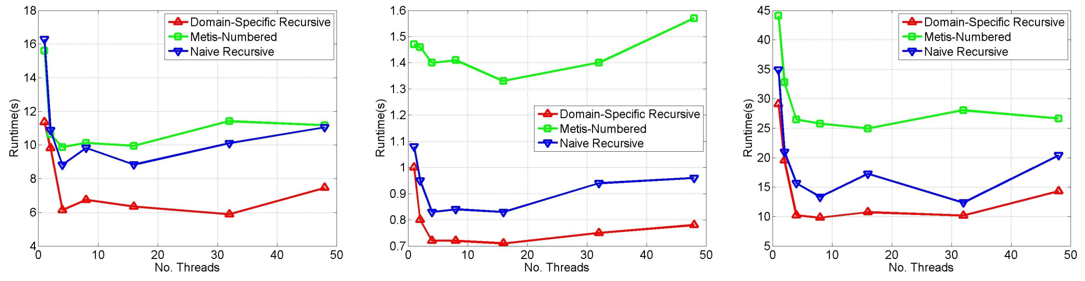
Figure 4: CDF of Runtimes on 500 Random Trees using Heuristics for 1,2,4, and 8 Cores.

2.2.5 Preliminary results

In our evaluation, we compare the execution times for running our heuristics on our three testing inputs. Given each partitioned mesh, our heuristics generate new coupling orders to solve each problem, subject to multiple timescales. In addition to our domain-specific heuristic, we evaluate a naïve recursive tree (a top-down bisection performed without regard to domain-specific cost models) and a tree based on the initial METIS-numbering, called METIS-numbered. As described in the previous section, METIS-numbered serves as a baseline; the initial coupling schedule produced after METIS is used to partition the initial mesh into subdomains. This coupling schedule also serves as the input to our inspector-executor system. The naïve recursive tree uses a top-down approach to produce a new tree, but does not incorporate the domain-specific cost models, while our domain-specific heuristic refines the top-down approach by incorporating knowledge of leaf solve and coupling costs. For each input, we evaluate all three schedules.

Results for the 32 subdomain rocket case is shown in Figure 4, for sequential execution as well as parallel execution on various numbers of threads. We compare the execution times of the domain-specific heuristic, naive recursive heuristic, and METIS-numbered heuristic, along with the 500 randomly generated trees for single and multiple threads. Here we observe that the domain-specific heuristic outperforms 99% of all trees that we tested and we achieved a significant speedup over randomly selected trees in the configuration space. In other words, despite the vast configuration space, by incorporating domain-specific knowledge, we are able to infer a very effective coupling order. We also note that the other evaluated schedules, METIS-numbered and naïve-recursive perform worse than our domain-specific schedule. We see that the baseline schedule is quite slow, while even taking advantage of a naïve top-down tree-building approach yields better results. Nevertheless, our domain-specific approach outperforms the naïve recursive schedule by 7 to 20%.

Figure 5 compares the execution times for each heuristic across different number of threads for the cube, rocket and stargrain input meshes. We note that in all cases, our top-down approach delivers noticeably better performance than the domain-expert tree and the naive recursive tree. The performance improvement values are 10% to 41% for the cube, 7% to 20% for the rocket, and 7% to 38% for the stargrain mesh. While the specific amount of improvement when using our domain-specific heuristic is problem-dependent, we see that incorporating domain knowledge



(a) 128 Subdomain Cube Mesh (b) 128 Subdomain Rocket Mesh (c) 128 Subdomain Stargrain Mesh

Figure 5: Execution times across different number of threads

provides a consistent edge across multiple inputs.

2.3 Ongoing work

This work will be extended in two ways. First, we are investigating how to apply these techniques to other domains. In particular, PI Parks is developing a multi-timescale Peridynamics implementation based on the same recursive decomposition ideas. Our optimization techniques should directly apply to this new domain.

We are also enriching the domain semantics themselves. In particular, we will investigate the use of accuracy models to allow our system to automatically determine decomposition factors (how many subdomains to use, etc.) and time steps in order to achieve particular accuracy goals while minimizing performance.

3 Optimizing Communication for Heterogeneous Architectures

3.1 Summary

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 handling explicit data movements between GPU and CPU memory spaces. Data movement is expensive, and hence communication costs need to be minimized. 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.

We built SemCache, a semantics-aware GPU cache that eliminates redundant communication between the CPU and GPU. 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 drop-in replacement library which handles communications automatically by hiding the data movements from the programmer and avoiding duplicate transfers. Our caching technique is efficient because 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 and CUBLAS.

This work is under submission to ICS 2013.

3.2 Preliminary findings

3.2.1 Motivation

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 [21, 12, 25, 24], graph algorithms [16], stencil codes [7], 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 [20] and OpenCL [18]) 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 [15, 23], 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.*

To correctly minimize data movement and avoid redundancy when offloading computation to the GPU, the composition of offloaded operations must be considered. 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.*

3.2.2 SemCache

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. These invalidations can be inserted automatically at compile-time, resulting in a hybrid static/dynamic approach to optimizing data movement.

This type of software caching has been proposed before for other architectures, such as distributed shared memory systems [1, 19] and accelerator-based systems like the Cell [13, 9]. The key drawback of these prior approaches is that the granularity of caching was 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.

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.

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.

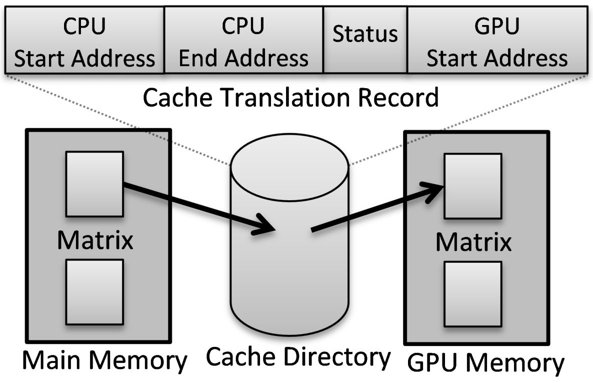


Figure 6: Structures of the Caching Directory

3.2.3 Operation

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.

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 the entry to C.

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

This interface can be used to implement various protocols, including a basic MSI protocol, as in Figure 7.

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,

[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

```
//execute after writing address range [s, e) on CPU
TranslationRecord writeCPU(s, e) {
  entry = lookup(s, e);
  invalidateOnGPU(entry);
  return entry;
}

//execute before reading address range [s, e) on CPU
TranslationRecord readCPU(s, e) {
  entry = lookup(s, e);
  if (entry.status == G) //data not current on CPU
    transferToCPU(entry)
  return entry;
}

//called after invoking a GPU method that writes [s, e)
TranslationRecord writeGPU(s, e) {
  entry = lookup(s, e);
  invalidateOnCPU(entry);
  return entry;
}

//called before invoking a GPU method that reads [s, e)
TranslationRecord readGPU(s, e) {
  entry = lookup(s, e);
  if (entry.status == C) //data not current on GPU
    transferToGPU(entry);
  return entry;
}
```

Figure 7: Operations to implement write-back protocol

SemCache*merges all the matching translation records* and creates a new record that tracks a range that spans all of the merged records.

3.2.4 Preliminary results

To evaluate the efficacy of SemCache, we used it to optimize communication in a GPU-offloaded version of the computational mechanics code described in Section 2. We evaluated four versions of this application. A CPU version that performed no offloading, a CUBLAS version that used hand-written CUBLAS code (with hand-inserted, but unoptimized communication), a CULA version that simply replaces all CPU BLAS calls with CULA BLAS calls, and a version using our SemCache write-through library, plus invalidations inserted at all writes of data that may be transferred to the GPU.

Experiments were run on a server with 24 AMD Opteron(tm) 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.

We used three test cases that are commonly seen in a structural dynamics problem. The first case is a finite element mesh of a rocket that has a crack. It is made of 7262 nodes and decomposed into 32 subdomains each subdomain has 115 elements. This test case is similar to a real world problem. Similarly, the second and third cases are shaking cubical structures with

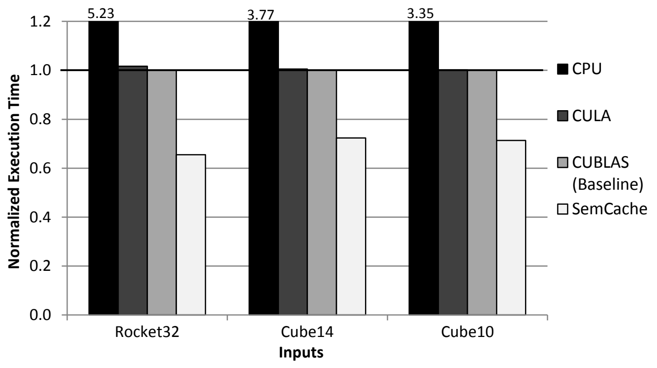


Figure 8: Testing application normalized execution time

Input/Library	CUBLAS	SemCache
Rocket32	21.86	2.87
Cube14	9.90	1.45
Cube10	2.79	0.42

Table 1: Size of transferred data to GPU using CUBLAS verses SemCache (in GB)

different properties. Cube14 is made of 3375 nodes, it has 8 subdomains each subdomain has 343 elements. Cube10 has 5 subdomains and 1331 nodes each subdomain has 200 elements.

Execution Time Figure 8 shows the total execution time for the testing application. The x-axis shows the inputs used in the experiment. The y-axis shows the normalized total execution time. Each input is executed using four different configurations: single CPU core, CULA, CUBLAS and SemCache. All inputs gained speedups of more than 70% 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 our library, the performance improved (30% to 40%) over the GPU CUBLAS baseline version due to the communication savings from caching. Inputs speedup ranges are different based on the structure of the shape and the common interface size of the subdomains. The inputs that have a larger interface size have more data shared between subdomains. The same matrices will be repeatedly reused. As a result, caching yields more benefits.

Communication Table 1 shows the amount of data transferred to the GPU. Using SemCache, more than 80% of the unoptimized communication is reduced. 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.

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 15% of 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.

Input/Library	CUBLAS	SemCache	
		Exec. Time	Caching Overhead
Rocket32	13.87	3.52	0.58
Cube14	6.99	1.14	0.067
Cube10	1.77	0.20	0.035

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

Input/Operation	GEMM	GESV	Lookup	Invalidations
Rocket32	6720	1209	22578	4065
Cube14	944	233	3298	625
Cube10	470	131	1672	367

Table 3: Operations count at runtime

3.3 Ongoing work

This work will be extended in two primary ways. First, we will target multi-GPU/multi-process systems, allowing these techniques to help optimize communication in larger-scale systems. Second, we will develop an annotation "language" for specifying how SemCache should be integrated into computational libraries. This language must be able to capture (i) which data is communicated; (ii) how the data layout changes (this will allow us to easily handle different data representations/-transformations when moving between CPU and GPU); and (iii) what functions are necessary to perform those layout changes.

References

[1] Cristiana Amza, Alan L. Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy 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] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro. A comparison of eigensolvers for large-scale 3d modal analysis using amg-preconditioned iterative methods. *International Journal for Numerical Methods in Engineering*, 64:204–236, 2005.

[4] J. K. Bennighof and R. B. Lehoucq. An automated multilevel substructuring method for eigenspace computation in linear elastodynamics. *SIAM Journal on Scientific Computing*, 25:2084–2106, 2004.

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

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the*

fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '95, pages 207–216, 1995.

[7] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine 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, Piscataway, NJ, USA, 2008. IEEE Press.

[8] J. Fish. Bridging the scales in nano engineering and science. *Journal of Nanoparticle Research*, 8(5):577–594, 2006.

[9] Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn 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, New York, NY, USA, 2008. ACM.

[10] V. Gravemeier, S. Lenz, and W.A. Wall. Towards a taxonomy for multiscale methods in computational mechanics: building blocks of existing methods. *Computational Mechanics*, 41(2):279–291, 2008.

[11] W. Hackbush. On the computation of approximate eigenvalues and eigenfunctions of elliptic operators by means of a multigrid method. *SIAM Journal on Numerical Analysis*, 16:201–215, 1979.

[12] John R. Humphrey, Daniel K. Price, Kyle E. Spagnoli, Aaron L. Paolini, and Eric J. Kelmelis. Cula: hybrid gpu accelerated linear algebra routines. *SPIE Defense and Security Symposium (DSS)*, pages 770502–770502–7, 2010.

[13] 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.

[14] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10:485–509, 2003.

[15] Chi-Keung Luk, Sunpyo Hong, and Hyesoon 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, New York, NY, USA, 2009. ACM.

[16] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012.

[17] J.G. Michopoulos, M. ASME, C. Farhat, A. Fellow, and J. Fish. Modeling and simulation of multiphysics systems. *Journal of Computing and Information Science in Engineering*, 5:198, 2005.

[18] A. MUNSHI. Opencl parallel computing on the gpu and cpu. *SIGGRAPH*.

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

[20] NVIDIA. Cuda. http://www.nvidia.com/object/cuda$_{h}ome_{n}ew.html$.

[21] NVIDIA. Cuda toolkit 4.0 cublas library. http://docs.nvidia.com/cuda/cublas/index.html, 2011.

[22] C. Papalukopoulos and S. Natsiavas. Dynamics of large scale mechanical models using multilevel substructuring. *Journal of Computational and Nonlinear Dynamics, ASME*, 2:40–51, 2007. Transactions of the ASME.

[23] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. Mdr: performance model driven runtime for heterogeneous parallel platforms. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 225–234, New York, NY, USA, 2011. ACM.

[24] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 44(4):121–130, February 2009.

[25] P. Du S. Tomov, R. Nath and J. Dongarra. Magma version 0.2 userss guide, 2009.

[26] Y. Saad and J. Zhang. BILUTM: A domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 21:279–299, 1999.