Efficient Heterogeneous Execution on Large Multicore and Accelerator Platforms: Case Study Using a Block Tridiagonal Solver[☆]

Alfred J. Park, Kalyan S. Perumalla

Computational Sciences and Engineering Division
Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831

Abstract

The algorithmic and implementation principles are explored in gainfully exploiting GPU accelerators in conjunction with multicore processors on high-end systems with large numbers of compute nodes, and evaluated in an implementation of a scalable block tridiagonal solver. The accelerator of each compute node is exploited in combination with multicore processors of that node in performing block-level linear algebra operations in the overall, distributed solver algorithm. Optimizations incorporated include: (1) an efficient memory mapping and synchronization interface to minimize data movement, (2) multi-process sharing of the accelerator within a node to obtain balanced load with multicore processors, and (3) an automatic memory management system to efficiently utilize accelerator memory when sub-matrices spill over the limits of device memory. Results are reported from our novel implementation that uses MAGMA and CUBLAS accelerator software systems *simultaneously* with ACML [5] for multithreaded execution on processors. Overall, using 940 nVidia Tesla X2090 accelerators and 15,040 cores, the best *heterogeneous* execution delivers a 10.9-fold reduction in run time relative to an already efficient parallel *multicore-only* baseline implementation that is highly optimized with intra-node and inter-node concurrency and computation-communication overlap. Detailed quantitative results are presented to explain all critical runtime components contributing to hybrid performance.

Keywords: Tridiagonal Solver, Linear Algebra, GPU, Accelerator, Heterogeneous Execution, Memory Management

1. Introduction

The availability of accelerator cards on compute nodes of large parallel platforms opens both opportunities and challenges: there is potential for exploiting the niche computing power of the accelerators, but it is unclear if and how

[☆]This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

Email addresses: parkaj@ornl.gov (Alfred J. Park), perumallaks@ornl.gov (Kalyan S. Perumalla) an overall performance gain may be posted by an application. With data dependencies across processors, blocked waiting times may overshadow the time gained from fast computation on a local accelerator. It is unclear if and to what extent the speed gains from accelerator-based computation can be effectively *combined* with the computational power of the host multicore processor. It is also unclear how the memories of host processor (system memory) and that of the accelerator (device memory) can be used in good combination in a *distributed* algorithm. All these point to the need for an *empirical* evaluation of actual implementation to uncover the efficacies of different schemes for combining accelerator use with multicore-based concurrent execution.

Here, we explore the design space of heterogeneous execution which is a gainful combination of computation on accelerator device cards with traditional multithreaded computation on multicore host processors, in a large parallel computing installation. A complex solver algorithm based on recursive cyclic reduction is used as the application in which this heterogeneous execution is implemented and tested.

1.1. Tridiagonal Cyclic Reduction Solver

Block tridiagonal matrices arise in several important applications such as magneto-hydrodynamics simulations [21]. In these applications, solutions to multiple right hand side vectors are computed during simulation, and the matrix itself is modified at different points in the simulation. The applications involve a system of linear equations $Ax = b$ in which A is an $NM \times NM$ matrix, x is an $NM \times 1$ column vector, and b is the $NM \times 1$ right hand side (RHS) vector. For simplicity, we assume only a single RHS vector, although solutions for multiple RHS vectors can be obtained at the same time. A is structured as a block tridiagonal matrix composed of lower-, main- and upper diagonal blocks, each block being of size $M \times M$ (blocks are assumed to be potentially dense). Thus, A is viewed as an $N \times N$ matrix of $M \times M$ blocks, its rows being referred to as *block-rows*. Each block-row r contains a lower diagonal block L_r ($1 < r \leq N$), a main diagonal block D_r ($1 \leq r \leq N$), and an upper diagonal block U_r ($1 \leq r < N$).

While general-purpose sparse solvers may be employed to solve such systems, customized solvers that are optimized to account for the specific structure are vastly more efficient and scalable. A parallel solver called BCYCLIC [20] based on a cyclic reduction algorithm has been recently reported to scale to hundreds of processors on multicore platforms, and customized for multi-threaded operation (at block-level) on multicore nodes, communicating over MPI. We utilize a rewrite, called BLOCKTRI, of the BCYCLIC software prototype, into which different sub-solvers can be easily incorporated. The parallel solver is structured such that any local solver (e.g., GotoBLAS [19]) can be used for linear algebra operations on the $M \times M$ blocks, while the overall parallel algorithm deals with the inter-processor dependencies and computation-communication overlap.

Using a divide-and-conquer approach, the original problem with N block-rows is divided into two sub problems. The first half is formed by the even numbered block-rows, and the second half is formed by the odd numbered block-rows. The first half is then expanded in terms of the variables in the second half, giving a problem half the size of the original problem. The rewriting results in the following computational operations to be performed on the blocks: (a) one factorization of the $M \times M$ diagonal block matrix, (b) two solve operations on $M \times M$ right hand sides using the computed factorization, (c) two matrix-matrix products on $M \times M$ matrices, and (d) one matrix-vector product on a $M \times M$ matrix. Standard linear algebra operations are used to perform these operations on $M \times M$ matrices within the confines of a single compute node, and these can be driven in a distributed fashion over communication interfaces such as MPI. Between each level of recursion, the newly computed values of the bordering rows are exchanged by each processor p with its neighboring processors $p' = p \pm 1$. The recursion ends when $N = 1$ after $l = \lceil \log_2 N \rceil$ levels, and recursion is unwound easily at each recursion level giving partial segments of x, which, at the end of unwinding of the recursion, gives the full solution vector x. For parallelism, block-rows are assigned to processes in contiguous segments. For example, with P MPI processes, the first $n_p = \lceil \frac{N}{P} \rceil$ block-rows are assigned to MPI process 0, the next n_p to process 1, and so on. Factors are retained across recursion levels, because they are reused in a later solve (the application may invoke multiple solves per factor, and not all RHS vectors are known in advance).

Diagonal blocks are assumed to be nonsingular at every recursion level. This makes it sufficient to perform pivoting at block level. Typically the tridiagonal matrices are preconditioned prior to use, which helps ensure this assumption is met. Please see [20] for complete details of the algorithm and multicore implementation.

1.2. Potential for Accelerated Execution

While our previous implementation is optimized for multicore, distributed (multi-node) execution, newer computation platforms are now becoming available in which each node is augmented with accelerators such as nVidia Tesla and AMD FireStream. It is now well-known that certain linear algebra operations can be completed significantly faster on the accelerator cards than on the multicore host processor(s). To illustrate, Figure 1 compares the sample run times of BLOCKTRI on a *single* node with $M = 512$. In this simple experiment, all linear algebra operations are performed exclusively in CPU mode (using MPI across cores within the node and multithreaded linear algebra library ACML [5]), and exclusively in accelerator mode (using MAGMA [23, 7] and CUBLAS [1]). The experiment is executed on a node containing 16 integer cores with 8 FPUs using 4 MPI ranks (2 threads per cpu process). Clearly, the potential for reduction in overall run time is evident. However, this is performance on only one node, and for small values of N and M. It is unclear how best to structure the execution such that the best (minimal) overall run time is obtained in a distributed execution on thousands of cores and hundreds of accelerator cards.

A spectrum of possibilities exists in how to structure the parallel execution to accommodate the accelerators. The default is the CPU-only execution in which multicore execution is used on every node. There are c processor cores on each node, and each node executes P_c MPI tasks, with t threads per task. When an accelerator is added to the mix, we

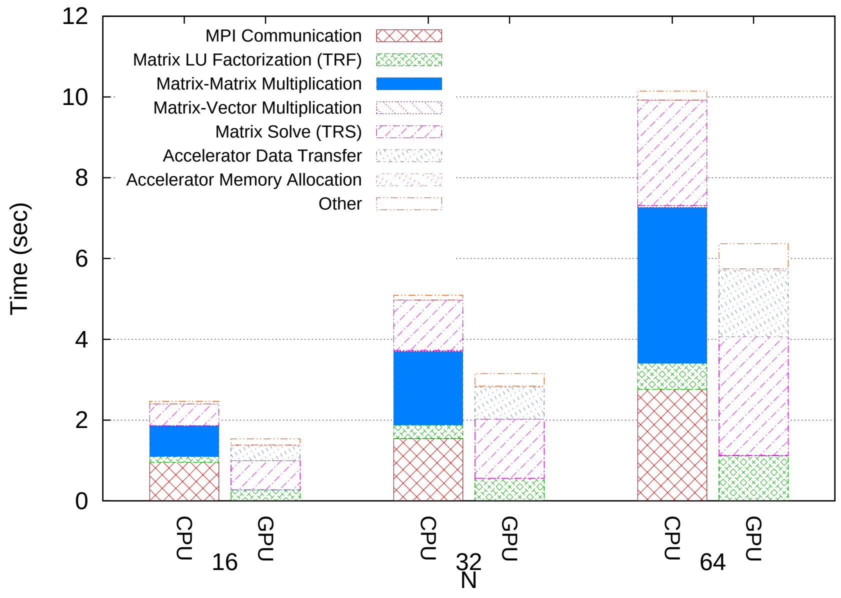

Figure 1: Potential performance improvement with accelerators

consider three possibilities: (1) GPU-only, in which only one MPI process is used per node, and only the accelerator is used for all linear algebra, and no CPU is used, (2) CPU+GPU in which two MPI processes are used per node, with one task using the accelerator and the other task using all the CPU cores with t threads, (3) SGPU+CPU, in which K MPI processes are used per node, with the first $P_A = K - 1$ tasks sharing the accelerator, and the last task using all c CPU cores with t threads. Each option has its own advantages and disadvantages, the net effect of which is not evident without actual implementation and experimentation.

1.3. Challenges and Design Considerations

Current accelerator-based systems contain disparate general purpose CPUs that contain one or more processing cores and one or more specialized accelerators, usually a compute-oriented graphics processing unit or GPU. As these elements are separate, they exclusively access their own memory subsystems.

Most recent computer architectures contain a fast, high bandwidth interface between the processor and system memory. Accelerator-based systems also contain their own local memory with a very fast, ultra high bandwidth interface. Any communication between the processor and the accelerator is done over a slower interface, and in modern architectures is usually PCI Express.

Since accelerators have their own memory subsystem and cannot directly access data stored in main system memory and vice versa, data must be copied back and forth to be visible on the respective context. This creates multiple challenges in efficiently porting the application to utilize the accelerator:

1. Accelerator memory must be explicitly managed
2. Data must be copied before and after computation to be visible to either the processor or accelerator
3. Accelerator memory is typically smaller than system memory; thus, not all data items can be kept on the accelerator for the lifetime of a program
4. Certain accelerator runtime systems allow multiple processes to access the same accelerator simultaneously, yet there is no automatic approach to manage memory between such processes.

In the context of the block tridiagonal solver, calls to linear algebra routines (BLAS and LAPACK) do not have a specific context, and hence their intermediate results stored on the device cannot be directly referred to in subsequent operations (e.g., factorization using TRF followed, arbitrary time later, by solve using TRS). Thus, an unoptimized accelerator-based solver would have to copy all data from the host to the device before the linear algebra operation and then the results from the device to host after the operation completes. Obviously, this can result in inefficient performance as input/output data may be copied up to four times if host paged memory is used, which is the usual default.

In the process of designing and implementing the spectrum of possibilities in heterogeneous execution, we uncovered the following needs: (1) unobtrusively and flexibly associate dynamic mappings from host memory to device memory, (2) dynamically manage device memory while maximizing the lifetime between host-device-host transfers of data on device, (3) sharing the computational capacity and memory of the accelerator across multiple tasks on the same node.

1.4. Contributions

The following are some of the contributions of this work:

- We empirically evaluate the design space in the combined use of multicore processors and accelerators within a distributed algorithm
- In the process, we evolve generalized principles (with respect to algorithmic and system effects) to guide in moving multicore-only implementations to heterogeneous executions
- We develop a generalized framework for memory management across host and accelerator memory subsystems by the definition of a new, minimally-invasive interface
- We show that a naïve heterogeneous execution of CPU plus accelerator can slow down rather than speed up execution

- Upon a sequence of improvements, we show that a more appropriate heterogeneous execution is most often faster than any homogeneous execution with only CPUs, and also faster than homogeneous execution with only accelerators

- We report excellent scaling results from large executions; in the largest case, we gainfully utilize 940 accelerator cards in combination with 15,040 cores within a single distributed execution.

1.5. Organization

The rest of the paper is structured as follows. Section 2 discusses in detail the implementation of the heterogeneous solver, along with the design of a new library to automatically manage accelerator memory and allow transparent shared access to a single accelerator resource per node. A performance study evaluating the key points in the parameter space for the solver is presented in Section 3. This is followed by a discussion on related work in Section 4 and concluding remarks and future work in Section 5. For the remainder of the text, the words GPU or device are used interchangeably with accelerator.

2. Efficient Hybrid Solver Implementation

When adapting the solver from CPU-only mode to accelerator-enhanced modes, it quickly became clear that a clear interface is needed to shield the implementation from accelerator-specific libraries and to be able to optimize the data movement transparently. The original CPU-only code (written in FORTRAN 90) maintains all its data structures in dynamically allocated arrays organized by recursion levels; these arrays are passed by the solver to the $M \times M$ block solvers by reference. Moreover, the operations arrive at the linear algebra package in different, non-trivial order, depending on the number of recursion levels, number of tasks, and number of solves per factor. To meet all these needs, and to retain the full flexibility of using arbitrary combinations of the CPU- and accelerator-based computation, we developed a lean interface. Upon refinement, we realized that its functionality closely approaches that of virtual memory: if accelerator device memory is viewed as physical memory and the host memory is seen as a disk, the device memory is essentially reused to house the system memory, dynamically, as needed by the application. We refined this interface, called `libaccelmm`, and realized the new heterogeneous functionality of the solver over this interface.

Here, we briefly digress from the solver per se, and focus on the design and features of the library interface, implemented as `libaccelmm`. Following that, we describe the memory-related concepts in detail.

2.1. `libaccelmm` Interface Overview

`libaccelmm` is a compact library that replaces most of the accelerator's native API for managing memory and addresses all of the aforementioned design challenges in Section 1.3. The library is generally applicable to a wide range of applications that benefit from the reuse of computation results on the accelerator. `libaccelmm` manages certain aspects of memory mapping between the host and accelerator where complexities are abstracted away from the application while allowing the device memory space to appear much larger than the accelerator total memory space. The only constraint is that the data needed for any given kernel computation must fit completely within device memory. This is a reasonable constraint because it provides the most flexibility without incurring the overheads of a full-blown virtual memory view across main memory and device memory. Moreover, `libaccelmm`, manages memory and synchronization issues on a *per node* basis, potentially independent of the number of accelerators and concurrent processes sharing any number of accelerators. Note that accelerator-based host memory mapping (i.e., Zero-Copy) techniques, provide performance benefits in certain situations, but are conceptually very different from our techniques presented here (discussed later in Section 4 on related work).

Table 1 summarizes the four major functions[1], each of which will be described in detail in the following sections.

Table 1: `libaccelmm` API summary

API Call	Purpose
`CREATE_ENTRY()`	Create (or find an existing) mapping between host and device memory
`MARK_ENTRY()`	Set host or device memory as valid or invalid for a mapping
`LOCK_ENTRY()`	Set device memory as locked (un-evictable) or unlocked (evictable)
`SYNC_ENTRY()`	Synchronize memories: transfer from host to device or device to host
`DELETE_ENTRY()`	Forcefully delete an entry, synchronizing device to host if necessary

2.2. Accelerator Memory Mapping

GPU accelerator programming involves creating a context on the device and then explicitly creating memory space on the device to hold data pertinent for computation kernels. The accelerator runtime expects the application developer to explicitly manage all aspects of memory management. For the block tridiagonal matrix solver, subroutines for linear algebra operations are called during the forward cyclic reduction and backward substitution phases. A straightforward

[1]`DELETE_ENTRY()` is provided for completeness; it is not directly used by the application.

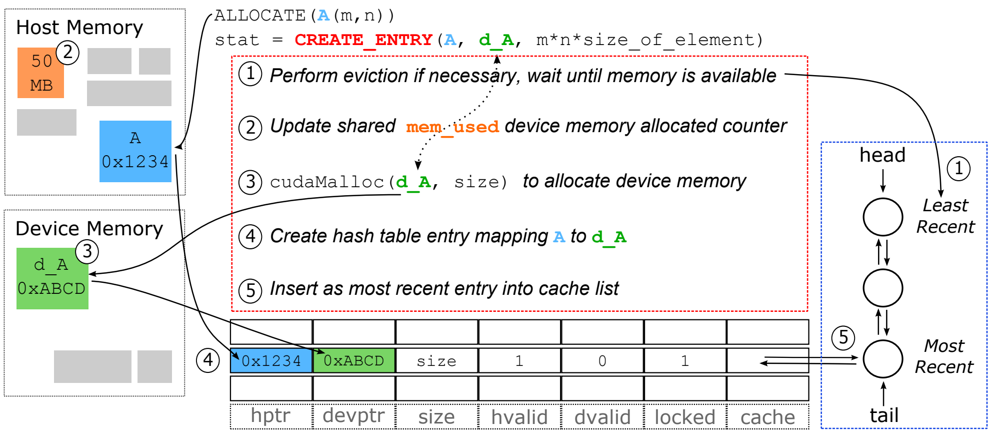
Host Memory
50 MB
A 0x1234
Device Memory
d_A 0xABCD
ALLOCATE(A(m,n))
stat = CREATE_ENTRY(A, d_A, m*n*size_of_element)
1 Perform eviction if necessary, wait until memory is available
2 Update shared mem_used device memory allocated counter
3 cudaMalloc(d_A, size) to allocate device memory
4 Create hash table entry mapping A to d_A
5 Insert as most recent entry into cache list
0x1234
0xABCD
size
1
0
1
hptr
devptr
size
hvalid
dvalid
locked
cache
head
Least Recent
Most Recent
tail

Figure 2: CREATE_ENTRY() function

implementation would mostly follow the template of accelerator memory allocation, host to device copy, compute, device to host copy, and accelerator memory deallocation. The accelerator version of the block tridiagonal solver, however, can be significantly improved by keeping as much useful data on the device as long as possible, without having to shuffle the data back and forth.

The central feature of `libaccelmm` is the automatic management of logical links between host memory and device memory to enable the application (in this case, the solver) to keep useful, completed computations on the device, avoiding redundant, inefficient copies back to the host when unnecessary. A hash table is used to map host pointers to device pointers. Additional information is stored for each hash entry: e.g., a flag to track if the host or device data is out-of-date and other bits required for efficiently managing the memory (which will be described in detail later). The first call made to `libaccelmm` is to create an entry using a host pointer to the accelerator, if one does not already exist. The `CREATE_ENTRY()` function provides this functionality as shown in Figure 2.

Prior to `CREATE_ENTRY()`, the application allocates host memory as usual. As shown in Figure 2, the FORTRAN 90 call to `ALLOCATE()` is performed on `A` with the indicated dimensions $m \times n$. Next, `libaccelmm` is invoked via `CREATE_ENTRY()`. Here, the host memory pointer `A` is given along with a pointer to device memory `d_A` that will be automatically filled by `libaccelmm` with the proper device pointer. Additionally, the size of `A` is given to `CREATE_ENTRY()`.

Once `CREATE_ENTRY()` is invoked, the first step is to perform any necessary evictions on the accelerator to free up space if device memory is full. The details of this operation will be discussed in section 2.4. Once sufficient space is available, a variable in shared memory visible to all processes sharing the same accelerator is updated in step 2, with the amount of memory to be taken by this allocation. In step 3, the accelerator runtime call to allocate memory on the device is invoked; here `cudaMalloc()` is called with the resulting device memory pointer stored in `d_A`. Once the

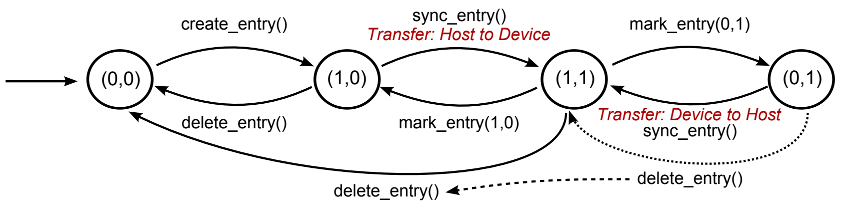

Figure 3: Accelerator memory management state transition diagram

memory is successfully allocated, then, in step 4, an entry in the hash table is added for this mapping. The variable `hvalid` represents the validity of data on the host, `dvalid` denotes the validity of data on the device memory, `locked` and `cache` fields will be explained in section 2.4.

If the node has accelerators that are being shared between processes, then the `mem_used` accumulator must be stored in shared memory, along with a semaphore to serialize accelerator memory allocations. This will prevent race conditions and runtime failures when concurrent processes are attempting to allocate accelerator memory simultaneously. When `CREATE_ENTRY()` returns, the application can then synchronize the memory between host and device as discussed next.

2.3. Accelerator Memory Synchronization

Although `libaccelmm` abstracts away most of the details of explicit accelerator device memory allocation and deallocation, some details are still the responsibility of the application. There are two cases when `libaccelmm` must be notified: (1) host data has been modified (e.g., by communication), and (2) device data has been modified (e.g., due to kernel computations). In the first case, the device memory now has an out-of-date copy of the mapped host memory and must be updated at some point before a future kernel invocation uses the associated data. In the second case, the device memory may need to be copied back to the host at some point in the future if the host requires up-to-date data. For example, in the tridiagonal solver, some data must be sent to neighboring nodes; therefore, any host memory that has associated device memory must be up-to-date before the data is sent.

To cope with these situations, `libaccelmm` provides two additional interface routines: `MARK_ENTRY( host_pointer, hvalid, dvalid )` and `SYNC_ENTRY( host_pointer )`. The purpose of `MARK_ENTRY()` is to "mark" a host to device pointer mapping with the proper bit flags, `hvalid` and `dvalid` (see Figure 2). For example, in the block tridiagonal solver, when neighboring nodes send updated values, the data in the host memory is modified. `libaccelmm` needs to know that any associated data held in the accelerator mapped to the updated host memory should be marked as invalid. Thus the call to `MARK_ENTRY(host_pointer, 1, 0)` would accomplish this. Conversely, once an accelerator kernel completes and has modified data in device memory, the call `MARK_ENTRY(host_pointer, 0, 1)` would signify as such.

`SYNC_ENTRY()` is used to perform the actual data transfer, if necessary, between host and accelerator memories. Figure 3 shows the state transition diagram and any transfer invocations for the two-bit validity states of an entry as (h, d), where h and d denote validity of the data in host and accelerator memory, respectively. As illustrated in the figure, calls to `SYNC_ENTRY()` converge the validity state into full consistency between host and accelerator memories while calls to `MARK_ENTRY()` diverge the validity state. After any data transfer is completed, the destination validity bit is flipped to `TRUE` to note that the data is consistent between the host and accelerator memories.

Special care must be taken when `DELETE_ENTRY(host_pointer)` is invoked. If the device memory is valid, but the host memory is not, then a `SYNC_ENTRY()` is automatically called to copy data from the accelerator back to the host before the entry in the mapping is removed as shown in Figure 3.

2.4. Accelerator Memory Eviction and Replacement

Over the course of the application execution, there may be instances where the accelerator memory reaches capacity and entries must be removed to make space for the next operation on the device. In the block tridiagonal solver, device memory would be needed to contain the data needed for the next linear algebra operation while the device memory is full due to occupancy by previous results left on the device for reuse, which must be evicted from the device memory to make room for the new operands. Note that eviction does not affect correctness, but only affects performance (analogous to virtual memory). Two fields, `locked` and `cache`, for each mapping in the hash table are used for this feature as shown in Figure 2. `locked` is a bit to denote if the associated accelerator memory can be deallocated under memory pressure. The `cache` entry is a pointer to the mapping's position within a doubly-linked linked list for selecting evictions based on a Least Recently Used (LRU) or Most Recently Used (MRU) scheme.

`libaccelmm` provides an API call `LOCK_ENTRY(host_pointer, value)` to allow applications to specify when to lock a host-to-accelerator mapping to prevent an eviction from taking place for the associated device memory. When `CREATE_ENTRY()` is invoked, the mapping is automatically locked as the device memory will be presumably used soon for a computation. The rationale behind exposing the locking interface is to allow the application to notify when associated device memory is non-critical for computation. For example, during a matrix-matrix multiplication, typically three areas of host memory (and thus their device memory counterparts) are required to complete the BLAS routine. The program would encounter an error if the memory management system were allowed to evict those memory regions before or during the linear algebra operation. Once the computation on the accelerator completes, then `LOCK_ENTRY(host_pointer, 0)` is invoked to notify `libaccelmm` that the device memory is free to be reused for other allocations. Note that the use of `LOCK_ENTRY()` is only needed with MRU replacement policy schemes or multi-process executions sharing the same accelerator. The LRU replacement policy will naturally reorder memory entries such that older entries are reclaimed first. Therefore, in the context of a single process exclusively using the

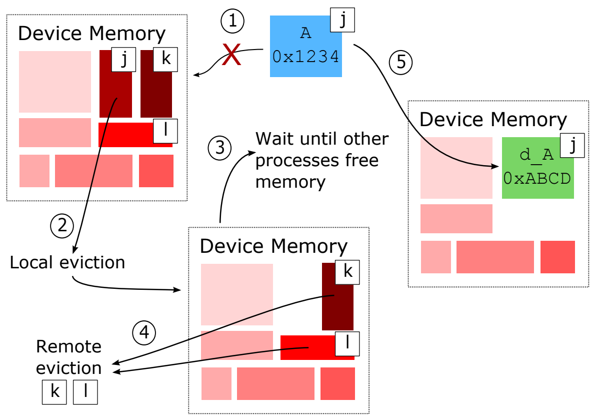

Figure 4: Multi-process device memory eviction (older allocations are darker)

accelerator, if somehow an entry that needs to be reclaimed is recent according to LRU ordering during an allocation request, then the program would have run out of device memory regardless anyway.

When a single accelerator is shared among multiple processes, there is increasing probability that the device will exhaust its free memory space because, assuming a memory-balanced problem, each process will divide the device memory across K processes sharing the accelerator. In the case of the block tridiagonal solver, memory is not equally balanced among all processes; hence, a simple $1/K$ hard limit per process imposes an excessively restrictive environment and can lead to performance penalties resulting from sub-optimal memory management.

`libaccelmm`'s handling of memory mappings allows a flexible amount of memory to be held on the accelerator at any given time with a restriction to architectural limitations such as the necessity of having all required data on the device for kernels that need them during computation. Due to this limitation, deadlocks are inherently possible with any accelerator memory management system, but the risk is lessened compared to a hard $1/K$ limit.

Figure 4 illustrates the algorithm for evicting memory to free space needed under a LRU replacement policy during a `CREATE_ENTRY()` allocation. In the first step, process `j` attempts to allocate device memory but cannot due to memory exhaustion. In the second step, the process attempts to free enough memory that it has allocated locally within its own context. There may not be enough memory that is available to be reclaimed locally, possibly due to memory that is currently locked or simply that other processes have most of the accelerator memory purposed for their own use. Process `j` then proceeds to step 3 where it waits for other processes to free memory. In this example, processes `k` and `l` are able to service the memory pressure relief request by evicting one old allocation not needed in step 4. Process `j` is then able to allocate space needed on the accelerator in step 5.

3. Performance Study

To evaluate the solver with the incorporated accelerator routines, we solved several problem sets by varying the key parameters, and instrumented the solver with timing metrics and counters to obtain a detailed log of all key performance information. The empirical evaluation not only gives the timing improvements on the overall solution times, but also provides insight into the memory behaviors with `libaccelmm`. Let P_C be the total number of processes that use CPU-based matrix operations (using different number of threads) and P_A be the total number of processes that use the accelerators. The total number of processes, P, used by the tridiagonal solver is $P = P_C + P_A$.

The performance parameter space for the distributed solver is dominated by two components, the dense block size M and the number of block rows N. M determines the size of the square matrices used in individual linear algebra routines: LU factorization, matrix-matrix and matrix-vector multiplication, and solve. The number of block-rows N dictates the length of the "periodic doubling" cyclic reduction phase. The number of reduction steps required is $\lceil \log_2 N \rceil$. N and M together define a specific problem size, which can be solved with different values of P, which in turn is varied as composition of P_C and P_A. Let a parameter α be defined such that $N = \alpha \times P$. For each composition of P, an α can be determined to obtain the N corresponding to different modes between purely CPU-only execution and a heterogeneous CPU plus accelerator execution that solve the same problem sizes.

In our performance study, N and M are varied to within the constraints of memory available per node; we also limited M based on the limited computer time available for experiments. Two distinct scenarios are chosen at various scale levels for N and M: in the scenario where N is large, a relatively smaller value of M chosen, and, conversely, when M is large, a relatively smaller value of N (equivalently, α close to unity) is chosen.

The scalar data type for the matrix can be varied at compile time to single precision, double precision real, or double precision complex. Majority of the performance data reported here is with double precision complex matrices. A few additional results are included with double precision real matrices, to illustrate capability.

3.1. Experimental Setup

The current version of `libaccelmm` is based on the CUDA [2] GPU runtime and libraries. The design of the block tridiagonal solver allows the accelerated BLAS and LAPACK routines to be easily switched between MAGMA [23, 7], CUBLAS [1] for certain BLAS routines such as matrix-matrix multiplication, Cray's libsci_acc, and CULA [3]. The CPU BLAS and LAPACK routines are provided through AMD's Core Math Library (ACML) 5.1.0 [5] with FMA4 and OpenMP multithreading support. All software was compiled using GNU gcc 4.6.2 with the optimization flags `-O3 -march=bdver1`. For the performance study, CUBLAS 4.0 was used for BLAS operations, and MAGMA 1.1.0 for LAPACK operations.

Table 2: Experimental Setup

Figure Label	Description	Total MPI Processes	Total Accelerators	Threads per CPU Process
CPU	4 CPU processes per node	$4 \times nodes$	0	2
CPU+GPU	1 CPU and 1 accelerator process per node	$2 \times nodes$	$nodes$	8
GPU	1 accelerator process per node	$nodes$	$nodes$	8
SGPU+CPU	3 accelerator processes and 1 CPU process per node	$4 \times nodes$	$nodes$	8

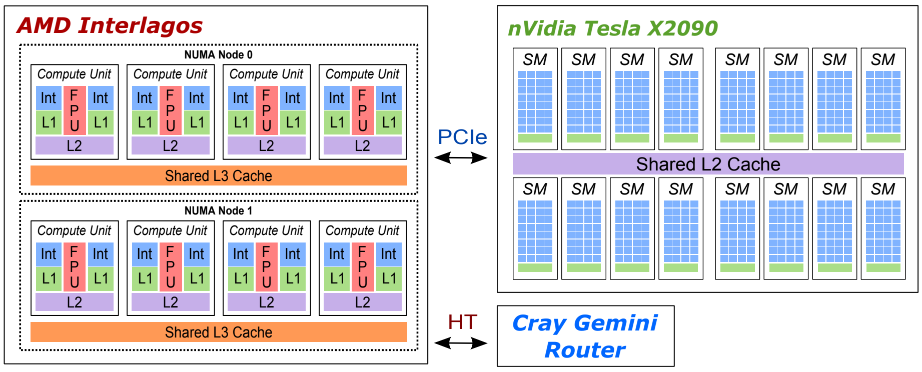

Figure 5: TitanDev Cray XK6 node layout

The performance study was performed on *TitanDev*, which is a development platform for the *Titan* supercomputer (hosted at the Oak Ridge Leadership Computing Facility, USA), using Cray XK6 nodes as shown in Figure 5. As of this writing, *TitanDev* contains 15,360 cores across 960 nodes. Each compute node consists of one 16-core AMD Interlagos processor with 32 GiB of memory and one nVidia Tesla X2090 accelerator connected via PCI Express. The nVidia Tesla X2090 is a Fermi-based device with 16 streaming multiprocessors (SM) each with 32 CUDA cores for a total of 512 CUDA cores. Each accelerator has roughly 5.25 GiB of available memory with ECC enabled.

As shown in Figure 5, each AMD Interlagos processor contains two NUMA nodes, each with four “compute units.” Each compute unit contains two integer cores and a shared FPU. Consequently, although the processor technically contains 16-cores, in the context of the block tridiagonal solver, the computational throughput will be limited by the eight FPUs. Thus, each node consists of 16 cores sharing 8 FPUs.

We empirically determined that for CPU-only executions with ACML, executions using four MPI processes per

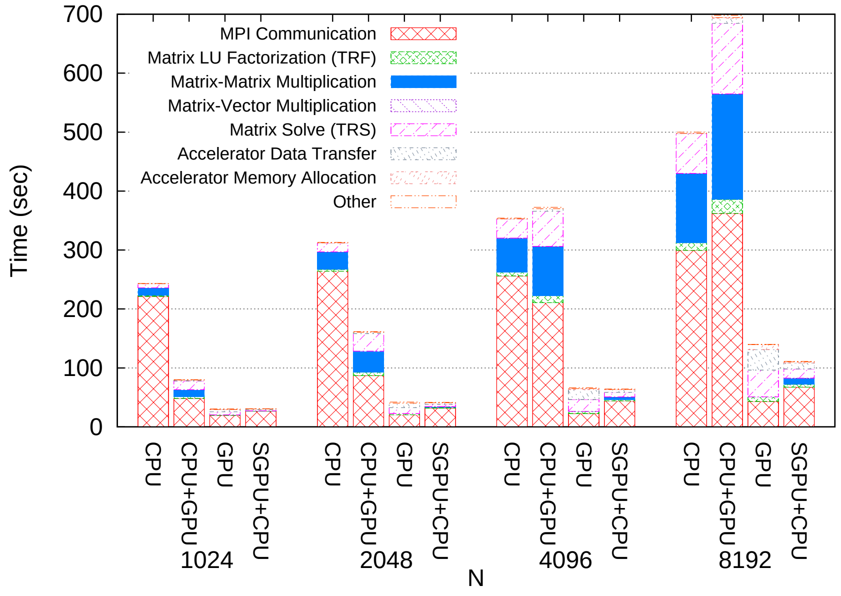

Figure 6: 256 nodes, large block size ($M = 2048$, complex)

node, each process containing two threads, gave the best performance. This approach uses a one-to-one mapping between threads and available FPUs on the processor. Consequently, we have chosen this configuration as our baseline benchmark. For heterogeneous executions with both CPU and accelerator, the MPI process responsible for CPU-based block-level linear algebra is created with eight threads to fully utilize all of the FPUs on each processor per node. A summary of the process layout and abscissa label descriptions used in the plots are described in Table 2. Unless otherwise noted, in all runs with accelerators, the LRU replacement policy is used during evictions. We note here that due to some issues with the accelerator runtime used (CUDA 4.0) such as kernels being limited to consuming no longer than one second of wall clock time, we limited the SGPU+CPU executions to three concurrent accelerator processes per node to avoid exceeding the kernel run time limits.

In addition to the block tridiagonal tests with the following configurations, we also performed individual test scenarios with very large block sizes (e.g., $M = 4096$) to ensure that `libaccelmm` was reporting similar bandwidth numbers as found in the bandwidth test program in the CUDA Toolkit.

3.2. Results

As discussed earlier, a natural approach for maximizing the performance gain from accelerators is to increase the computation to communication ratio in the linear algebra routines. Since each dense $M \times M$ block is the smallest matrix size passed to any linear algebra operation, for larger values of M, we can increase the utility of the accelerators.

Figures 6, 7, 8 show the wallclock runtime of the block tridiagonal solver when $M = 2048$ using double precision

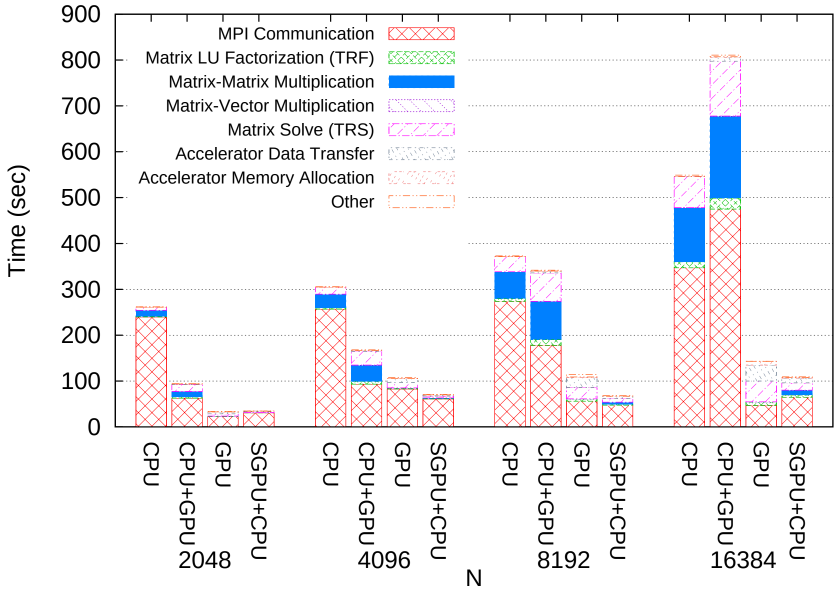

Figure 7: 512 nodes, large block size (M = 2048, complex)

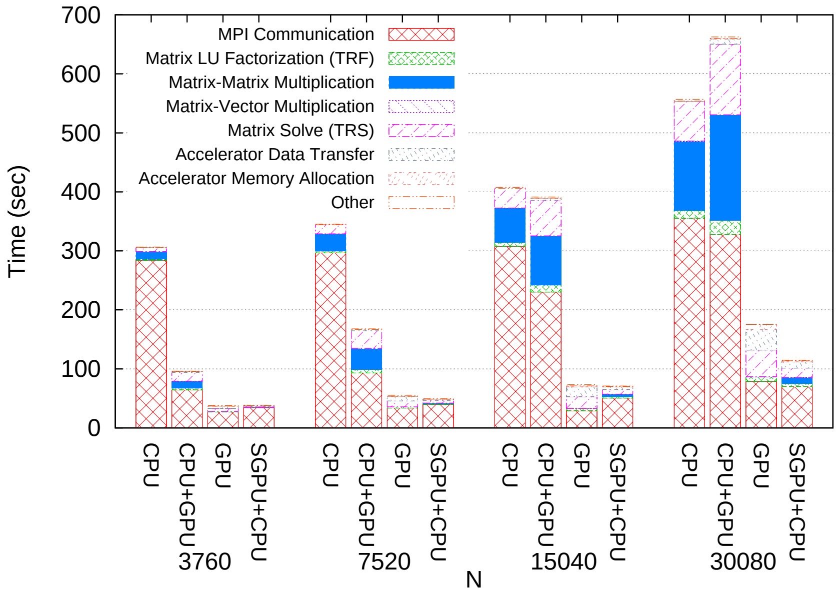

Figure 8: 940 nodes, large block size (M = 2048, complex)

complex matrices for different numbers of block rows at 256, 512 and 940 nodes, respectively. The runtimes are partitioned in terms of each individual BLAS routine in order to show the impact of the various scenarios on the time for each routine.

We observe that the overall time to solution for the block tridiagonal solver is reduced by simply adding an accelerator in addition to the CPU for smaller block row sizes. However, at the largest N block rows, performance drops considerably for the CPU+GPU case. This can be explained due to a load imbalance appearing at the largest scales of N row blocks, where during the cyclic reduction phase, each level is dominated by the time taken for the CPU to perform linear algebra operations. With more levels to compute, it takes proportionally more steps to reach a reduction level where the GPU performs all of the linear algebra operations. Thus the overall load becomes more imbalanced as N (number of row blocks) increases. From the data, it is clear that it would be a misconception to expect that simply adding an accelerator into the computation would deliver performance gains.

Note that, due to the recursive divide-and-conquer nature of the algorithm, some processors at a recursion level become idle until the next level of recursion (with half the problem size) complete. Thus, the time spent waiting to receive messages from the next level of recursion is included in the communication time (e.g., in the time for `MPI_Waitany()`).

When we remove the CPU from the computation and, instead, offload all linear algebra computations to the accelerator, we see a marked improvement in time to solution. Since the accelerator can efficiently parallelize even large dense block sizes as in this test case of $M = 2048$, this is consistent with expectations. However, this seems sub-optimal: reserving the CPU solely for inter-node synchronization and data distribution operations would be a waste of the CPU computational capabilities, and hence there may be loss of some additional, potential performance gains.

Bearing this in mind, the SGPU+CPU test case reincorporates the CPU into the linear algebra operations while increasing the load on the GPU, but reduces the work the CPU must perform. Although the total work per node remains the same for any given N, in the SGPU+CPU case (instead of a 50% work split between the accelerator and CPU) the work distribution is shifted where more work is given to the accelerator (i.e., a 75% to 25% split). We observe that for all scales, and at nearly all N row block sizes, the SGPU+CPU case comes close to, or even outperforms, the pure accelerator-only (GPU) test case. This, in effect, is a load balancing mechanism to attempt to ensure that enough work is fed to the accelerator so both the accelerator and CPU are kept busy for roughly equivalent amount of walltime per level. With increased amount of accelerator processes per node, the number of levels of recursion required to reach an accelerator-only level is reached in fewer number of steps since the proportional amount of work given to the CPU is reduced by a factor equal to the number of accelerator processes.

Table 3 shows the accompanying factor of performance improvement for the test cases at $M = 2048$ using complex matrices. The data presented in the performance improvement tables portrays a progressive cumulative gain factor between different improved methods. The third column shows the factor of improvement achieved by moving to a pure, accelerator-only solution (GPU). The fourth column shows the factor of improvement by moving from a pure accelerator-only algorithm to a fully heterogeneous solution of incorporating multiple simultaneous accelerator

Table 3: Factor Improvement for Large Block Sizes $M = 2048$, Complex

nodes	N (Block Rows)	CPU to GPU Gain Factor	GPU to SGPU+CPU Gain Factor	Overall SGPU+CPU Gain Factor
256	1024	7.97	0.99	7.88
	2048	7.44	1.01	7.51
	4096	5.33	1.03	5.50
	8192	3.59	1.25	4.50
512	2048	7.79	0.97	7.55
	4096	2.84	1.53	4.32
	8192	3.26	1.68	5.49
	16384	3.85	1.31	5.04
940	3760	11.0	0.99	10.9
	7520	6.25	1.11	6.99
	15040	5.56	1.03	5.73
	30080	3.18	1.53	4.86

processes sharing a GPU with the CPU aiding in the computation as well (SGPU+CPU). In nearly all cases, the heterogeneous SGPU+CPU execution results in significant factor of improvement with increased performance over an accelerator-only approach. Gain factors range from 4.32× in the worst case at 512 nodes to 10.9× at 940 nodes in the best case. Interestingly, the factor of improvement numbers quantify the effect of increasing N on time to solution. Gain factors of a pure accelerator-only approach and a fully hybridized approach (column 3 and 4 respectively) move in opposite directions as the execution is scaled. An accelerator-only approach loses efficiency gains because of the inability to keep all of the required data on the accelerator as N increases due to increasing number of levels required by the cyclic reduction phase. This is clearly observed in the data where increasing amount of time is required for accelerator data transfers as N increases.

Next, we explore the parameter space in the opposite direction by holding the dense block size constant at a smaller size of $M = 256$ while greatly increasing the number of block rows, N, from 32K to nearly 1 million (962K). Figures 9, 10 and 11 show the time to solution for 256, 512 and 940 nodes, respectively.

At $M = 256$ using complex data type, the matrix size is $256 \times 256 \times 16$ bytes in size, or 1 MiB. These matrices can be computed rather quickly by the CPU, perhaps by being able to completely fit them in the L2 cache of each computation unit of the AMD Interlagos processor, and several such matrices in the shared L3 cache within each NUMA node. Thus we see for certain scenarios in Figure 10 that the CPU performance is competitive to accelerator-based approaches at certain N block rows.

As discussed previously, with larger amounts of cyclic reduction levels given higher N, the advantage of the

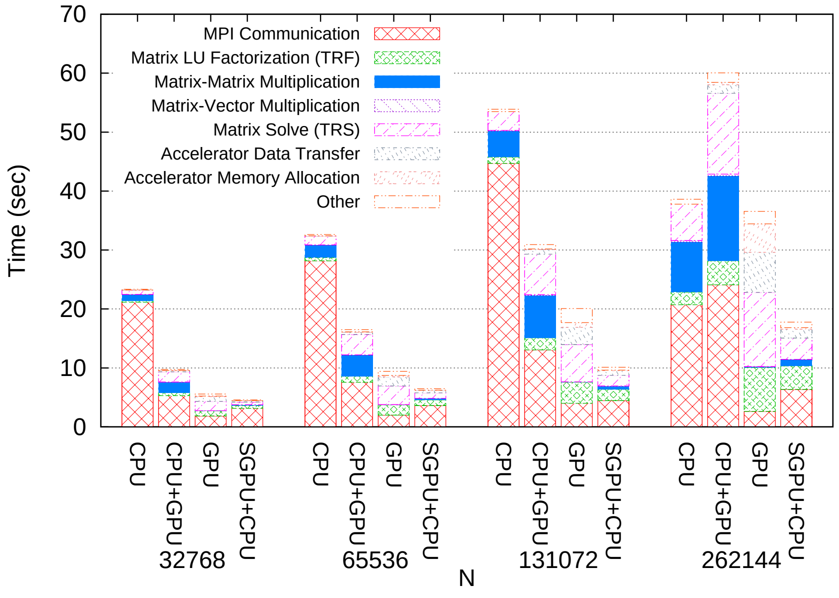

Figure 9: 256 nodes, large block rows (M = 256, complex)

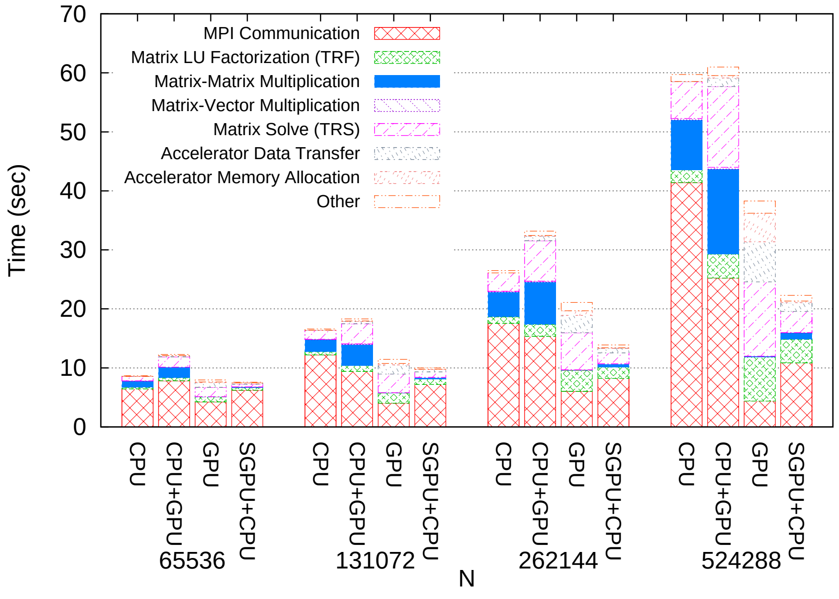

Figure 10: 512 nodes, large block rows (M = 256, complex)

CPU+GPU approach is diminished as seen across most node scales. With reduced arithmetic compute intensity due to smaller M, we see smaller factor of improvement gains in Table 4 of accelerator-only (GPU) executions compared to CPU-only. Similar to the large block size test scenario, nearly all SGPU+CPU test cases result in performance improvement over just accelerator-only executions.

For completeness and to exercise a different data type other than complex at similar scales, Figure 12 shows

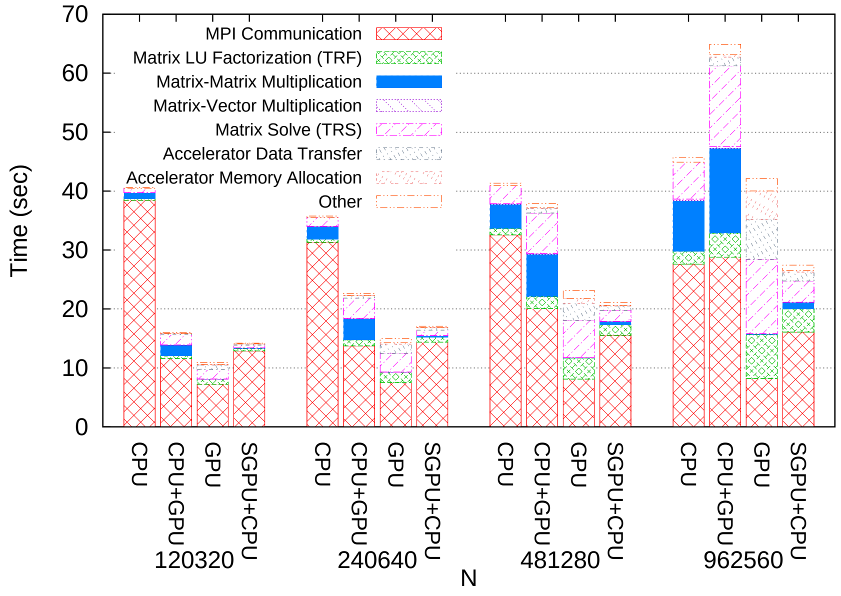

Figure 11: 940 nodes, large block rows (M = 256, complex)

Table 4: Factor Improvement for Large Block Rows M = 256, Complex

nodes	N (Block Rows)	CPU to GPU Gain Factor	GPU to SGPU+CPU Gain Factor	Overall SGPU+CPU Gain Factor
256	32768	4.18	1.22	5.10
	65535	3.46	1.46	5.05
	131072	2.69	1.98	5.33
	262144	1.06	2.06	2.18
512	65535	1.09	1.05	1.15
	131072	1.45	1.15	1.67
	262144	1.26	1.52	1.91
	524288	1.55	1.72	2.66
940	120320	3.71	0.77	2.86
	240640	2.39	0.88	2.09
	481280	1.79	1.10	1.97
	962560	1.09	1.53	1.67

performance of the block tridiagonal solver using double precision real numbers in matrices with a block size M = 256. We observe performance improvements similar to that of complex data type at M = 256. As the size of the matrices is reduced in half, further improved CPU performance is seen due to more matrices being able to fit into the L2 and shared L3 caches of the CPU.

The quantitative performance gains by leveraging the memory management facilities provided by `libaccelmm` for

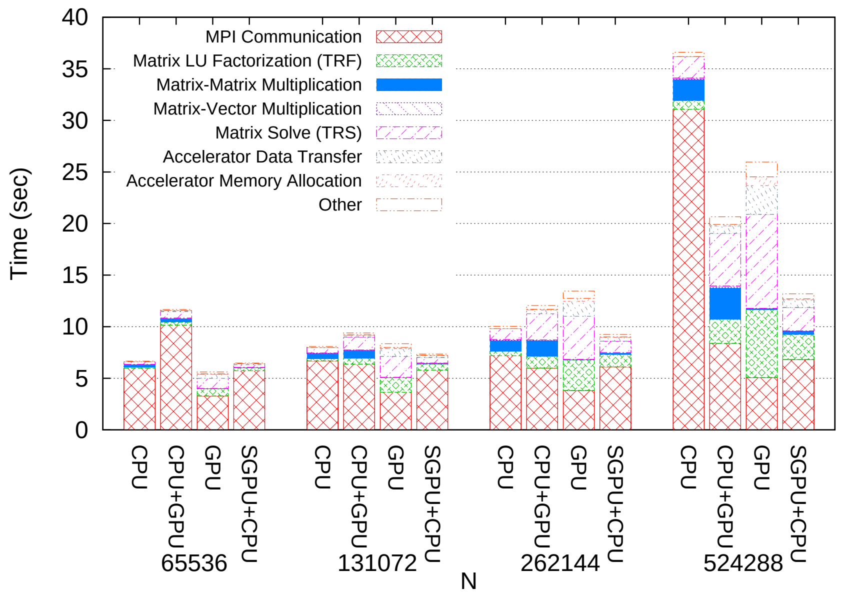

Figure 12: 512 nodes, large block rows (M = 256, double)

Table 5: `libaccelmm` memory management with LRU replacement policy (Complex, 940 nodes, GPU-only)

M (Block Size)	N (Block Rows)	Memory Map Hit Rate	Memory Map Eviction Rate	Memory Allocation Reduction per GPU	Memory Transfer Reduction per GPU	Gain Factor over No Memory Management
1024	3760	66.0%	0.0%	2.93x	3.05x	1.20
	7520	69.1%	0.0%	3.26x	3.51x	1.22
	15040	71.1%	0.0%	3.52x	3.93x	1.25
	30080	72.4%	0.0%	3.71x	4.26x	1.57
	60160	69.9%	6.8%	3.25x	3.20x	1.25
2048	3760	66.0%	0.0%	2.93x	3.05x	1.18
	7520	69.1%	0.0%	3.25x	3.49x	1.11
	15040	67.8%	7.9%	3.01x	2.83x	1.14
	30080	66.8%	19.0%	2.89x	2.45x	1.08

dense block sizes M of 1024 and 2048 are shown in Table 5. We see that up to N = 30080, the memory management system is able to keep the entire computation on the device giving an average memory reduction per accelerator of 3.71×, a memory transfer reduction per accelerator of 4.26× and a performance improvement of 1.57×, compared against an execution that employs no memory management. As memory pressure builds and the eviction algorithm is invoked in the N = 60160 case, a drop off in memory allocation, transfer reduction, and gain factor is observed,

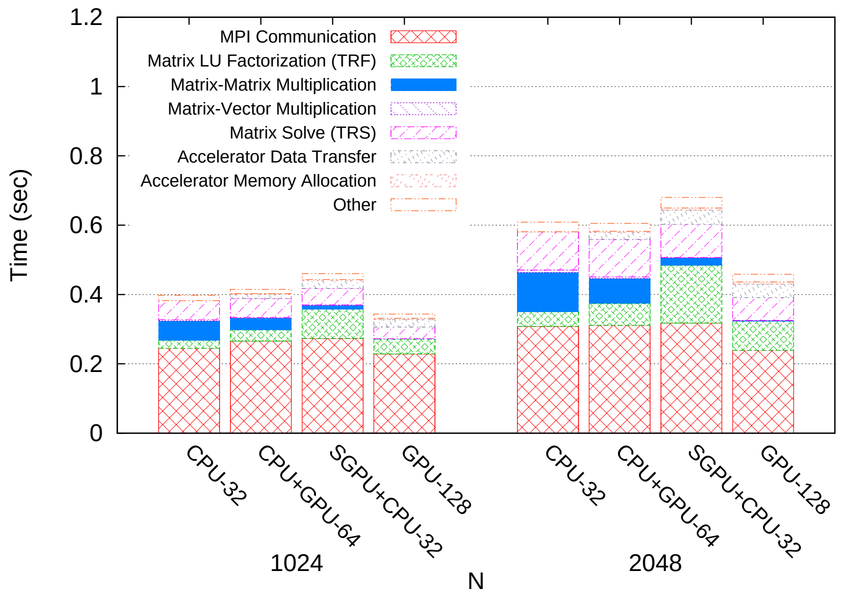

Figure 13: SIESTA Data (M = 234, double)

as expected. When block sizes are doubled as shown in the third row of Table 5, performance gains are somewhat reduced but are still appreciable, ranging from 8% to 18%.

Note that, when the GPU is shared, the exact memory metrics cannot be precisely reported due to uncontrollable runtime variations. To avoid perturbation factors in the multi-process, shared-accelerator execution, the results of a GPU-only single-process run are observed.

3.3. Matrices from SIESTA

Magnetohydrodynamic (MHD) equilibrium codes are important for a wide range of plasma applications. They are used for design and analysis of tokamaks and stellarators, reconstruction of plasma states from experimental data and initialization of extended MHD time-dependent codes. SIESTA (Scalable Iterative Equilibrium Solver for Toroidal Applications) [21] is an iterative MHD equilibrium solver that enables the exploration of a wide range of new scenarios to be simulated in support of all the aforementioned areas.

The block solver is essential for the numerical efficiency of the SIESTA code which computes high-resolution MHD equilibria in the presence of “magnetic islands.” Most of the computational time in SIESTA is spent in the inversion of large block matrices (N > 100, M > 300) which are repeatedly applied as preconditioners to accelerate the convergence of lineralized MHD equations toward an equilibrium state. Eventual simulations for plasmas in the International Thermonuclear Experimental Reactor (ITER, with T ~ 15 keV, a ~ 2 m, B ~ 5 T) will require even larger spatial resolution resulting in greater block row numbers and sizes. Therefore, the efficient parallel inversion and

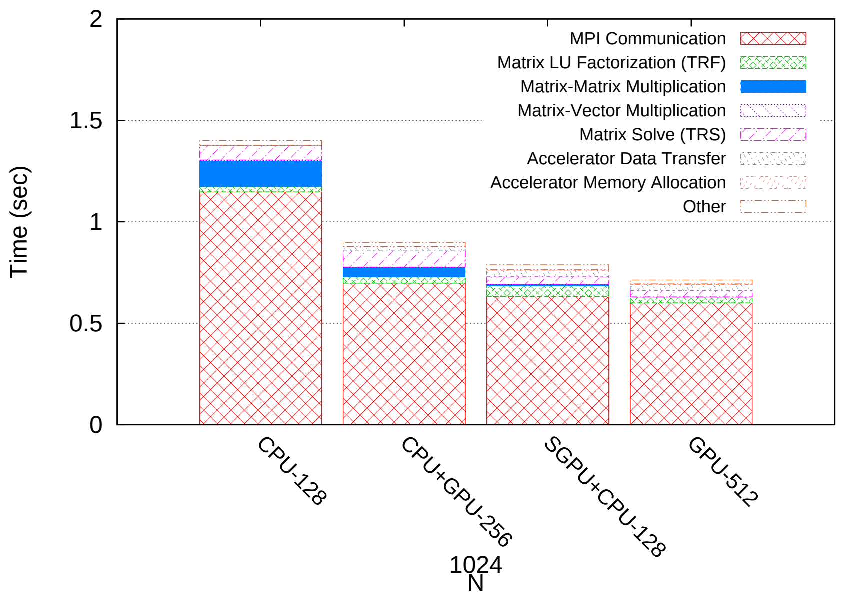

Figure 14: SIESTA Data (M = 528, double)

storage of the factors of A (i.e., a block tridiagonal matrix consisting of large, dense blocks) is essential in SIESTA.

To study the effectiveness of the solver on the matrixes that SIESTA uses, we use the exactly same solver as described in previous sections, but we execute the solver on non-synthetic matrix data obtained from actual SIESTA simulations.

We generated three real data sets from SIESTA and evaluated the performance of CPU, accelerator and mixed heterogeneous executions. Due to the static nature of the matrix configuration mapped to the number of MPI processes, the number of nodes is not kept constant in contrast to prior synthesized large scale matrices. Here, since the number of MPI processes is kept constant, the total number of nodes changes according to the execution type; this is denoted by the trailing number along the X-axis labels.

In Figure 13, matrix sizes of $M = 234$ with $N = 1024$ and $N = 2048$ are evaluated. The CPU and SGPU+CPU cases are run across 32 nodes, while the CPU+GPU is run on 64 nodes and the GPU only test case is run on 128 nodes. Due to the block size and number of block rows being relatively small, the addition of an accelerator does not result in large performance improvements. However, the accelerator only scenario improves execution performance due to fast matrix-matrix operations, although this requires 4× the number of nodes.

Figure 14 shows performance for block sizes of $M = 528$ and $N = 1024$ row blocks. Due to the larger block size, the benefits of utilizing an accelerator are more apparent. We observe that by doubling the amount of nodes from 128 in the CPU case to 256 in the CPU+GPU case, the execution time decreases by nearly 1.64×. Further performance gain is achieved by quadrupling the number of nodes to 512 in the GPU-only case, resulting in about 16% runtime reduction over CPU+GPU. However, we can *retain the same number of nodes* as the CPU case by effectively sharing the accelerator in the SGPU+CPU case and achieve a time to solution that is 1.81× faster.

3.4. Runtime Analysis and Validation

With the aid of Vampir, further detailed runtime analysis on the tridiagonal solver was performed. The traces allowed for validation of the communication patterns along with runtime behavior such as computation and communication overlap as observed quantitatively in Section 3.2. Due to trace file sizes and compute time considerations, the following trace-based runtime analysis was confined to the scenario of the largest N block rows at 256 nodes with complex data type (i.e., $N = 262144$) as shown in Figure 9. Current Vampir limitations prevented tracing of the SGPU+CPU scenario and is therefore absent from the analysis presented here.

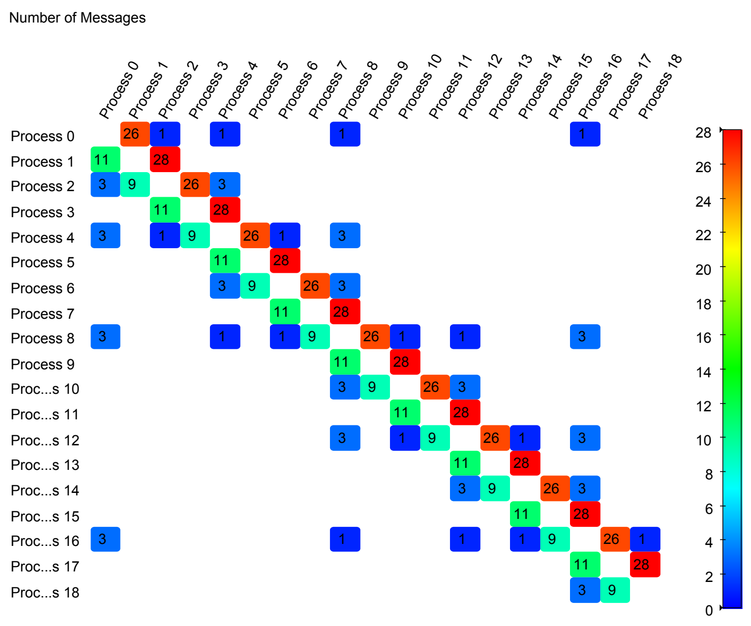

Figure 15: 256 nodes, CPU-only, communication matrix

Table 6: 256 node, CPU-only message counts from trace

Message Size (KiB)	Message Count
4	21463
1024	20440
Total messages	41903

Figure 15 shows a visualization of the runtime trace of communication between processes. The rows represent sender processes and the columns represent receiver processes. The trace shows the general pattern of messaging intensity between neighboring processes as illustrated by the superdiagonal and subdiagonals along the matrix. Table 6 shows number of MPI messages sent and the corresponding message size, as reported by the Vampirtrace tool. The 4 KiB messages are vector-length messages (i.e., 256×16 bytes), while 1024 KiB messages are lower or upper matrices (i.e., $256 \times 256 \times 16$ bytes).

In the forward solve part of the solver, the cyclic reduction portion is broken down into two distinct phases, S_1 and S_2. The first parallel portion, S_1, is where the number of block rows exceeds the number of processors available, or $N > P$. This phase will always have processors that have work to do. The second phase of the cyclic reduction, S_2, begins when $N \leq P$ and continues until only one row block remains. The total number of messages sent can be formulated for each portion of forward and backward solve phases. Due to the varying computation patterns depending upon odd or even partitions, which is dictated by N and P, the following formulas are restricted to power-of-two N and P, which corresponds to the pattern shown in Figure 15.

$$S_1 = (P - 1)\log_2 \frac{N}{P} \tag{1}$$

Equation 1 shows the total amount of message *exchanges* in S_1. There are $\log_2 \frac{N}{P}$ steps where there the number of block rows exceeds the number of processors. This is then multiplied by $P - 1$ even-odd row boundaries for the number of message exchanges.

$$S_2 = \sum_{i=1}^{\log_2 P} 2(\frac{P}{2^i} - 1) + 1 \tag{2}$$

When there are more processors than work available per processor, the number of steps becomes $\log_2 P$. As the forward solve progresses, the number of processors is halved each time and each even row sends updates to their odd neighbors which is shown in Equation 2.

$$S_{fm} = 2(S_1 + S_2) \tag{3}$$

$$S_{fv} = S_1 + S_2 \tag{4}$$

$$S_f = S_{fm} + S_{fv} \tag{5}$$

For each message exchange, three messages, each containing two matrices (Equation 3) and one vector (Equation 4), are sent to neighboring processes. S_f represents the total amount of messages sent during the forward solve portion as shown in Equation 5. For the 256 node, CPU-only scenario shown in Figure 15, where $N = 2^{18}$ and $P = 2^{10}$, results in $S_1 = 8184$ and $S_2 = 2034$. Thus, $S_{fm} = 20440$, $S_{fv} = 10220$ and $S_f = 30660$.

$$S_b = S_1 + S_2 \tag{6}$$

For the backward solve portion of the solver, the number of message exchanges are identical as the forward solve (except that senders become receivers and receivers become senders) as we are unrolling the recursion substituting for unknowns. However, there are no matrices exchanged, but only vectors are exchanged, as represented in Equation 6. Thus, for $N = 2^{18}$ and $P = 2^{10}$, $S_b = 10220$. The total amount of messages sent during the forward and backward solve is $S_f + S_b = 40880$. The difference between Vampirtrace reported 41903 messages and 40880 is 1023, which is the exact number of messages used during solution verification (i.e., $P - 1$). Comparing against Table 6, $S_{fm} = 20440$ matches the 1024 KiB message count for the number of messages containing matrices. $S_{fv} + S_b = 20440$ and adding $P - 1 = 1023$ verification vector messages gives a total of 21463, which matches the 4 KiB message count in Table 6.

The individual amount of messages sent on a per processor basis can also be validated as shown in Figure 15. During the S_1 phase in a power-of-two N and P, processes always send to their next "highest" neighbor three messages. This phase consists of $\log_2 \frac{N}{P}$ steps, thus for $N = 2^{18}$ and $P = 2^{10}$, this results in $3 \times 8 = 24$ messages. In the case of process 0, there are no additional messages sent during the S_2 phase of the forward solve. However, during the backward solve, messages are sent from process 0 to processes 2^i where $i = \log_2 P - 1 \rightarrow 0$. This pattern is shown in the first row of Figure 15 where a single message is sent to each power-of-two process. This results in 25 messages sent to process 1. The final message is sent during the verification portion of the program matching the 26 shown in the communication matrix.

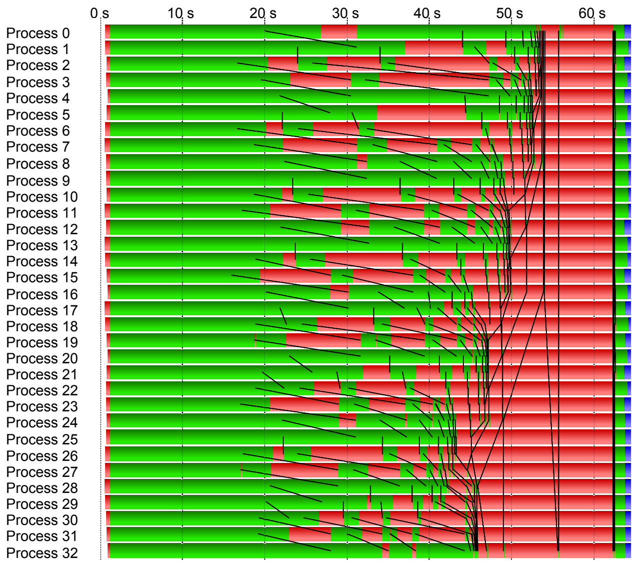

Figure 16: 256 nodes, CPU-only timeline, zoomed in

Figure 16 shows a zoomed in view of the computation and communication patterns of the solver execution for the first 32 processes. The green areas represent application activity while the red areas represent MPI communication time (spent in non-blocking send/receive and in blocking "wait all" synchronization). The black lines represent messages being sent from one process to another[2].

The overview shows the general cyclic reduction occurring over time until the forward reduction completes and then backward solve represented as previous unknown values being distributed back out as messages. The final amount of time in green represents solution verification time. For this particular scenario, the random number generation and matrix population took approximately 13 seconds of the initial application runtime[3]. The zoomed in runtime breakdown shown in Figure 16 provides a higher resolution view of the computation and communication behavior. Here the communication and computation overlap is clearly visible where MPI messages are sent to neighbors and computation continues.

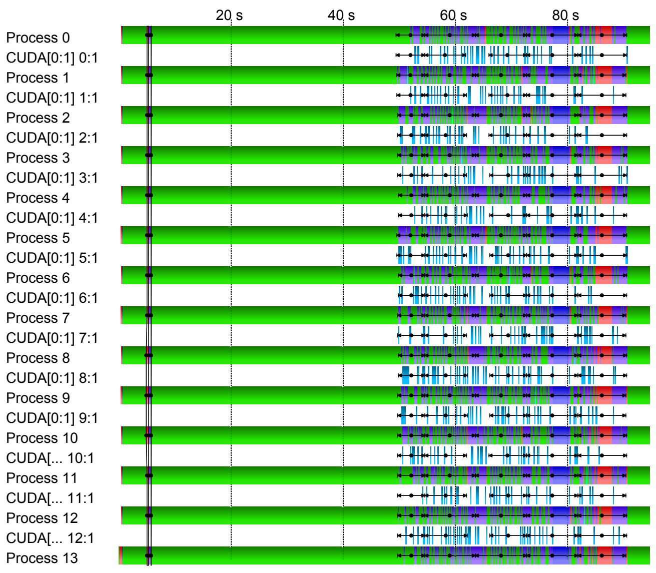

Figure 17: 256 nodes, GPU-Only timeline, zoomed in

Figure 17 shows the timelines for accelerator-only execution. Approximately the first 49 seconds of execution can be ignored due to generation of random matrix[4]. A relatively well balanced execution is observed, with no CPU processors computing linear algebra operations. This is shown in detail in Figure 17, where the areas shaded blue represent CUDA kernel time and the purple areas represent CUDA runtime calls such as memory copy and allocation.

[2]Due to the large amount of messages being sent in groups and limited resolution of the timeline, not all messages sent are shown in the Vampir graphical output.

[3]These initialization times are not counted as application time in Section 3.2 which would unfairly bias application time compared to MPI communication time.

[4]Initialization time due to random number generation is 4× slower in this case compared to CPU-only due to only 1 CPU core used in this case (see Table 2).

It is observed that the amount of time spent in MPI waiting for messages is drastically reduced even during the S_2 phase of the forward solve as the time to finish matrix operations on 256×256 matrix sizes are very fast. This is cross-verified in the timing breakdown shown in Figure 9 where the total MPI Communication time represents only a small fraction of the total aggregate time.

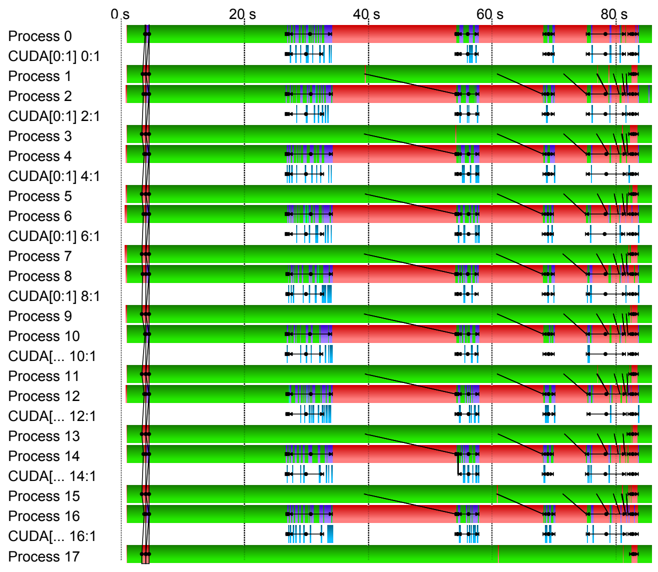

Figure 18: 256 nodes, CPU+GPU timeline, zoomed in

The heterogeneous CPU+GPU computation mode is shown in Figure 18. Approximately the first 26 seconds of execution can be ignored as initialization time for random number generation and matrix population[5]. The overall execution seems to exhibit a structured execution pattern representing computation followed by communication in cycles. However, from Figure 18, it is evident that the apparent non-overlapping execution is, in fact, a load balance problem. Here, even-numbered processes represent CPU processes which act as proxies for the accelerator to launch computation kernels and send and receive MPI messages; they do not perform any computation. Odd-numbered processes are CPU processes which perform linear algebra operations. As described in Table 2, each node contains two (one even and one odd) of these processes.

It is clearly visible that the accelerator processes quickly complete execution and is left to idle wait until the next portion of the cyclic reduction and corresponding update from neighbor processes. Meanwhile, the CPU processes are almost always busy. This shows the performance culprit behind simply adding an accelerator into an application for heterogeneous execution without properly analyzing load imbalance computation artifacts. The SGPU+CPU execution mode, which is possible due to `libaccelmm`, remedies this situation by balancing the arithmetic load of the

[5]Initialization time due to random number generation is $2\times$ slower in this case compared to CPU-only due to 2 CPU cores used in this case (see Table 2).

solver by shifting work away from the CPU and on to the accelerator. This allows the solver to gainfully use the CPU in addition to the accelerators.

4. Related Work

Work on sparse nonlinear systems on GPUs [15], LU factorization on GPU clusters [13], and general hybridized linear algebra routines [8] including tridiagonal solvers on GPUs have been studied and optimized in recent literature [9, 18, 30, 25, 12, 33]. In contrast to these prior works, our work is focused on *block* tridiagonal matrices, of which tridiagonal matrices are a small special case with block size equal to unity that are solved using a hybrid solver utilizing heterogeneous resources of many nodes containing both CPUs and accelerators. The hybrid solver is also based on an algorithm different from those previously presented for block tridiagonal matrices, using latency hiding techniques to improve computation and communication overlap across distributed processors. These same benefits are extended to the heterogeneous accelerator based implementation presented in this work.

Some recent work also is aimed at exploiting the processing power of multicore processors with GPU-based accelerators, spanning a wide variety of problems and domains (e.g., solving boundary value problems for second-order ordinary differential equations [31] and simulation of agent-based models [6]). In [6], a multi-level latency-hiding scheme is used to maximize computation and communication overlap to improve the speed of simulation. This is accomplished as a trade-off by duplicating some computation as to offset communication latencies. The memory issues we discussed here were, however, apparently not encountered. Other recent work proposed a “waterfall” model-based energy-conscious algorithms for mapping computation on heterogenous platforms [26]. A hierarchical matrix partitioning algorithm has been proposed for automatically load balancing matrix operations on large heterogenous computing systems [11].

Hybrid execution across CPUs and GPUs has been applied to quantum chemistry applications [17] showing significant performance gains through parallelization of both the orbital distribution scheme where each processor holds a small subset of orbitals containing all coefficients (which is useful for nodes with accelerators as variable number of orbitals and/or wave function coefficients can be stored per processor) and variable repartition. These performance gains are possible due to the linear algebra operations representing the most expensive parts of the code for large systems. The iteration loop is broken down into components and independently transferred to the GPU and computed in an optimal way to save on memory copy time. The authors find that an N:1 (CPU:GPU) execution binding mode, where more than one processor shares a single GPU, does not cause any runtime issues in their specific usage pattern, since each GPU only tends to be half loaded during calculations.

Similarly, the work shown in [14] describes performance gains of hybridizing coupled-cluster with single and double excitations (CCSD) and configuration interaction with single and double excitations (TD-CISD). These methods can be expressed as a series of dense matrix-matrix operations. Generally, the partitioning works such that terms scaling to $< 5^{th}$ power are done on the CPU while 6^{th} power terms are done on the GPU. However, better performance is achieved where "easy" work is done on the GPU where all inputs are already on the device or fast memory bandwidth can be exploited on the GPU. Other computations where the size of the intermediate is too large must be avoided on the GPU. Essentially, the performance problem is solving the load balancing issue between two different types of processors (i.e., CPU and GPU) used for execution with different capabilities.

Simulations of multiphase compressible flows [28] are parallelized across both CPUs and GPUs showing good speedup through the use of "wavelet blocks," task-based parallelism on the CPU, and asynchronous management of the GPUs. Dynamic load balance is achieved through a work stealing algorithm which schedules ghost reconstruction tasks asynchronously for GPUs.

A technique of pipeline-based parallelism for image processing is used in [27] where the GPU performs FFT computations to extract phase from fringe patterns and then hands off the frame to a CPU threads in a pipeline. The process is fast enough to achieve higher than realtime requirement of 30fps.

A parallelized breadth-first search is explored in the context of sharing the execution between the CPU and GPU in [22]. At each level of the algorithm, the best implementation is dynamically chosen from serial, two different multicore methods, and a GPU method. The GPU algorithm is initiated if there is an exponential growth in the number of nodes in each level. Furthermore, the GPU method uses an optimization where only the contents of the current-level queue is copied to the GPU and then the entire level array is reconstructed as this is faster than copying the entire level array.

In [10], GPUs are used as coprocessors for simulating high-fidelity wireless propagation models. Ray tracing is used to obtain accurate measurements for these models where which is computationally intense. Simulation is done on the CPU, but the ray tracing computation is offloaded via a work queue system mapped to GPU devices. Idle CPU processors can service the queue as well leading to heterogeneous execution.

There has been some related work with managing GPU memory. memCUDA [24] reduces the burden of separate memory semantics through the use of pragmas leading to compact and less error prone code. memCUDA differs from our work in that it does not offer multi-process support for sharing a GPU, support decoupled semantics of marking and synchronizing memory, and requires a full suite of support software to function compared to `libaccelmm`'s single link-time library.

GMAC [16] is an Asymmetric Distributed Shared Memory (ADSM) library that attempts to hide some of the programming complexities of utilizing accelerators similar to `libaccelmm`. However, GMAC only provides coherence on the host, but not the accelerator, with the rationale that synchronous coherence would impose large performance penalties or hardware burdens. `libaccelmm` performs user-directed lazy synchronization where data transfer is only performed when absolutely necessary.

The MYO [29, 32] runtime provides transparent data sharing between the CPU and accelerators. Although this implementation allows for shared data structures between heterogeneous compute processors, the heavyweight implementation must touch the entire system stack, including the OS layer and application.

5. Conclusions and Future Work

The problem of introducing accelerator-based execution into a traditional multicore, multi-node algorithm is studied here in the context of a scalable block tridiagonal solver. An efficient implementation approach has been described, and a new device-host memory management framework is designed to aid in heterogeneous program execution. Performance results from execution on up to 940 accelerator cards and 15,040 cores show that a simplistic use of accelerators via an added MPI task is sub-optimal on large matrix sizes; a carefully chosen load-balanced, heterogeneous assignment is shown to deliver the best performance, in which multiple tasks share the accelerator on each node, while a single task per node utilizes all the CPU cores using multi-threading. Relative to the CPU-only baseline, the shared-accelerator scheme is observed to deliver over 10-fold decrease in overall solution time in some of the best configurations. The memory management interface and implementation we developed to move the CPU-only solver to this heterogeneous execution is elegant and flexible, and we believe it might be useful in other applications as well, in which accelerators are to be integrated.

Some important observations were made from the performance study. First, small-scale optimizations do not always translate to net performance gains at the highest levels of scale. Secondly, and perhaps the most relevant for applications being ported to use accelerators, addition of an accelerator to perform computation (e.g., simply offloading work to an accelerator) does not always improve performance. The problem distills down to the objective of equalizing the load between all cores of the node and the accelerator(s), which is a non-trivial problem.

There are multiple related and orthogonal directions for future work in both areas of improving the hybridized block tridiagonal solver and `libaccelmm`. For the block tridiagonal solver, the reduction in the number of MPI processes sharing a GPU on the node can alleviate contention issues and improve throughput. Further integration with the eviction mechanism and reception of MPI messages of the block tridiagonal solver can also improve efficiency. Migrating certain allocations within the block tridiagonal solver to use page-locked (i.e., pinned) system memory can also increase the throughput of host to accelerator data transfers. Performance effects of different BLAS/LAPACK vendor libraries, larger M block sizes, multiple accelerators per node, and further increasing the shared ranks per node can be examined.

Regarding `libaccelmm`, further enhancements to the memory management system can be explored such as creating a custom allocator with a managed memory pool to improve speed and help offset fragmentation with application-directed hints and batched memory transfers to improve memory synchronization speed. Portability and generalization enhancements can be made by supporting OpenCL [4].

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract No. DE-AC05-00OR22725.

References

[1] Cublas: Cuda basic linear algebra subroutines, `http://developer.nvidia.com/cublas`, 2012.

[2] Cuda: Compute unified device architecture, `http://www.nvidia.com/object/cuda_home_new.html`, 2012.

[3] Cula: Gpu-accelerated linear algebra libraries, `http://www.culatools.com`, 2012.

[4] Opencl: The open standard for parallel programming of heterogeneous systems, `http://www.khronos.org/opencl`, 2012.

[5] Acml: Advanced micro devices core math library, `http://developer.amd.com`, 2013.

[6] B.G. Aaby, K.S. Perumalla, S.K. Seal, Efficient simulation of agent-based models on multi-gpu and multi-core clusters, in: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools '10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 2010, pp. 29:1–29:10.

[7] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, S. Tomov, Lu factorization for accelerator-based systems, in: Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on, pp. 217 –224.

[8] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Chapter 34 - a hybridization methodology for high-performance linear algebra software for gpus, in: W. mei W. Hwu (Ed.), GPU Computing Gems Jade Edition, Morgan Kaufmann, Boston, 2012, pp. 473 – 484.

[9] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The plasma and magma projects, Journal of Physics: Conference Series 180 (2009) 012037.

[10] S. Bai, D. Nicol, Acceleration of wireless channel simulation using gpus, in: Wireless Conference (EW), 2010 European, pp. 841 –848.

[11] D. Clarke, A. Ilic, A. Lastovetsky, L. Sousa, Hierarchical partitioning algorithm for scientific computing on highly heterogeneous cpu &#43; gpu clusters, in: Proceedings of the 18th international conference on Parallel Processing, Euro-Par'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 489–501.

[12] A. Davidson, Y. Zhang, J.D. Owens, An auto-tuned method for solving large tridiagonal systems on the gpu, in: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 956–965.

[13] E. D'Azevedo, J. Hill, Parallel lu factorization on gpu cluster, Procedia Computer Science 9 (2012) 67 – 75.

[14] A. DePrince, J. Hammond, Quantum chemical many-body theory on heterogeneous nodes, in: Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, pp. 131 –140.

[15] V. Galiano, H. Migalln, V. Migalln, J. Penads, Gpu-based parallel algorithms for sparse nonlinear systems, Journal of Parallel and Distributed Computing 72 (2012) 1098 – 1105.

[16] I. Gelado, J.E. Stone, J. Cabezas, S. Patel, N. Navarro, 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 '10, ACM, New York, NY, USA, 2010, pp. 347–358.

[17] L. Genovese, M. Ospici, T. Deutsch, J.F. Méhaut, A. Neelov, S. Goedecker, Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures, The Journal of Chemical Physics 131 (2009) 034103.

[18] D. Goddeke, R. Strzodka, Cyclic reduction tridiagonal solvers on gpus applied to mixed-precision multigrid, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 22–32.

[19] K. Goto, R. Van De Geijn, High-performance implementation of the level-3 blas, ACM Trans. Math. Softw. 35 (2008) 4:1–4:14.

[20] S. Hirshman, K. Perumalla, V. Lynch, R. Sanchez, Bcyclic: A parallel block tridiagonal matrix cyclic solver, Journal of Computational Physics 229 (2010) 6392 – 6404.

[21] S.P. Hirshman, R. Sanchez, C.R. Cook, Siesta: A scalable iterative equilibrium solver for toroidal applications, Physics of Plasmas 18 (2011) 062504.

[22] S. Hong, T. Oguntebi, K. Olukotun, Efficient parallel graph exploration on multi-core cpu and gpu, in: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 78–88.

[23] M. Horton, S. Tomov, J. Dongarra, A class of hybrid lapack algorithms for multicore and gpu architectures, in: Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing, SAAHPC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 150–158.

[24] H. Jin, B. Li, R. Zheng, Q. Zhang, W. Ao, memcuda: Map device memory to host memory on gpgpu platform, in: C. Ding, Z. Shao, R. Zheng (Eds.), Network and Parallel Computing, volume 6289 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2010, pp. 299–313.

[25] H.S. Kim, S. Wu, L.w. Chang, W.m.W. Hwu, A scalable tridiagonal solver for gpus, in: Proceedings of the 2011 International Conference on Parallel Processing, ICPP '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 444–453.

[26] W. Liu, Z. Du, Y. Xiao, D. Bader, C. Xu, A waterfall model to achieve energy efficient tasks mapping for large scale gpu clusters, in: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pp. 82–92.

[27] Y. Miyamoto, A. Wada, T. Iizuka, T. Suzuki, T. Nakayama, K. Yanagawa, S. Aoki, Y. Ozaki, T. Toriu, M. Kawana, T. Nishino, M. Takeda, Realtime 3d profilometer using gpu and multicore cpu, Digital Holography and Three-Dimensional Imaging, in: Digital Holography and Three-Dimensional Imaging, Optical Society of America, 2011, p. DTuC10.

[28] D. Rossinelli, B. Hejazialhosseini, D.G. Spampinato, P. Koumoutsakos, Multicore/multi-gpu accelerated simulations of multiphase compressible flows using wavelet adapted grids, SIAM Journal on Scientific Computing 33 (2011) 512–540.

[29] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, A. Mendelson, Programming model for a heterogeneous x86 platform, in: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09, ACM, New York, NY, USA, 2009, pp. 431–440.

[30] C.P. Stone, E.P.N. Duque, Y. Zhang, D. Car, J.D. Owens, R.L. Davis, Gpgpu parallel algorithms for structured-grid cfd codes, in: Proceedings of the 20th AIAA Computational Fluid Dynamics Conference, 2011-3221.

[31] P. Stpiczynski, J. Potiopa, Solving a kind of bvp for odes on heterogeneous cpu + cuda-enabled gpu systems, in: Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on, pp. 349 –353.

[32] S. Yan, X. Zhou, Y. Gao, H. Chen, G. Wu, S. Luo, B. Saha, Optimizing a shared virtual memory system for a heterogeneous cpu-accelerator platform, SIGOPS Oper. Syst. Rev. 45 (2011) 92–100.

[33] Y. Zhang, J. Cohen, J.D. Owens, Fast tridiagonal solvers on the gpu, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, ACM, New York, NY, USA, 2010, pp. 127–136.