G-Charm: An Adaptive Runtime System for Message-Driven Parallel Applications on Hybrid Systems

R.Vasudevan, Sathish Vadhiyar
SuperComputer Education and Research Centre
Indian Institute of Science
Bangalore, India
vasudevan@ssl.serc.iisc.in,
vss@serc.iisc.in

Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kale@illinois.edu

ABSTRACT

The effective use of GPUs for accelerating applications depends on a number of factors including effective asynchronous use of heterogeneous resources, reducing memory transfer between CPU and GPU, increasing occupancy of GPU kernels, overlapping data transfers with computations, reducing GPU idling and kernel optimizations. Overcoming these challenges require considerable effort on the part of the application developers and most optimization strategies are often proposed and tuned specifically for individual applications. In this paper, we present *G-Charm*, a generic framework with an adaptive runtime system for efficient execution of message-driven parallel applications on hybrid systems. The framework is based on CHARM++, a message-driven programming environment and runtime for parallel applications. The techniques in our framework include dynamic scheduling of work on CPU and GPU cores, maximizing reuse of data present in GPU memory, data management in GPU memory, and combining multiple kernels. We have presented results using our framework on Tesla S1070 and Fermi C2070 systems using three classes of applications: a highly regular and parallel 2D Jacobi solver, a regular dense matrix Cholesky factorization representing linear algebra computations with dependencies among parallel computations and highly irregular molecular dynamics simulations. With our generic framework, we obtain 1.5 to 15 times improvement over previous GPU-based implementation of CHARM++. We also obtain about 14% improvement over an implementation of Cholesky factorization with a static work-distribution scheme.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*
; D.3.4 [**Programming Languages**]: Processors—*Optimization, Run-time environments*

Keywords

GPU, CHARM++, Optimizations, Hybrid execution, Data management, Combining kernels

1. INTRODUCTION

GPU-based hybrid systems have become highly prevalent for high performance computing with about 62 systems in the Top500 [1] list, including the #1 Titan system of Oak Ridge National Laboratory, being powered using accelerator/co-processor technology. However, effective use of a GPU system for high performance requires overcoming several challenges including effective asynchronous use of heterogeneous resources, reducing memory transfers between CPU and GPU, increasing occupancy of GPU kernels, overlapping data transfers with computations, reducing GPU idling and kernel optimizations. Overcoming these challenges require considerable effort by the users. Many optimization strategies have been proposed and tuned for specific applications [14, 15, 6, 10, 5].

In this paper, we present *G-Charm*, a generic framework with an adaptive runtime system for efficient execution of message-driven parallel applications on hybrid systems. The framework is based on CHARM++ [7], a message-driven programming environment and runtime for parallel applications. CHARM++ is an object oriented parallel programming framework in which the application domain is partitioned among several migratable objects called *chares*, and the chares are distributed among allotted processors. By employing over-decomposition, dynamic load balancing using migration of chares, and asynchronous communications overlapped with computations, CHARM++ has been used to provide high performance for different scientific applications including NAMD [10], a molecular dynamics application, ChaNGa [5], a cosmological simulator and ParFUM [9], a framework for unstructured mesh applications. These principles are highly needed for executions on hybrid systems consisting of a number of heterogeneous CPU and GPU units.

Obtaining high performance for CHARM++ message-driven parallel applications on GPU systems is challenging. CHARM++ chares are usually small in size dealing with small subdomains to promote data locality and cache benefits, and to provide communication-computation overlap among chares. An existing GPU task library for CHARM++, *HybridAPI* [17], does not provide a way to automatically merge units of data across multiple chares into blocks that would provide good occupancy on the GPU. The user must perform this agglomeration manually [6], or expect poor occupancy if each chare issues its own kernels. Invocations of kernels by each chare also results in large number of kernel invocations. Also, in HybridAPI, it is the users' responsibility to specify when to allocate, copy, and delete GPU buffers, so as to reuse buffers over execution of multiple kernels. It is also the users' responsibility to schedule work on either GPU or CPU. This places significant burden on the programmer since the code for making scheduling decisions has to be written in addition to the main logic.

As an example, consider the case of a 2D Molecular dynamics (MD) simulations. In this application, two types of chare objects are used. The molecules or particles are distributed among a 2D array of chares called *patches*. Force calculations between particles present in a pair of patch chares is calculated by an element of a chare array called *compute object*. The particle coordinates change only at the end of an iteration. A patch needs to be transferred to GPU only once for an iteration, and can be used for force calculation with each of its neighbors. In HybridAPI, ensuring that the patch is transferred only once to the GPU, and merging the data for separate kernels to invoke a smaller number of kernels with better occupancy require significant programming effort by the user. Finally, in HybridAPI, the user has to decide if the force calculation for a compute object has to be done on either CPU or GPU.

Our G-Charm framework includes important set of techniques for achieving high performance on GPUs in a transparent and generic manner. G-Charm performs dynamic scheduling of work units to CPU or GPU based on the current loads and the estimated run time of the work units on CPU and GPU. Our framework also dynamically combines multiple kernels corresponding to the work units to reduce the number of kernel invocations. The G-Charm runtime system also minimizes CPU-GPU data movement by keeping track of the location of data of *GPU chares* in GPU device memory and avoiding redundant data transfers for work units. G-Charm also performs data management to maximize the reuse of most frequently used data in the GPU device memory. Our techniques are generic and require minimal overhead from the application programmers.

We have presented results using our framework on Tesla S1070 and Fermi C2070 systems using three classes of applications: a highly regular and parallel 2D Jacobi solver, a regular dense matrix Cholesky factorization representing linear algebra computations with dependencies among parallel computations and a highly irregular molecular dynamics simulation. We performed comparisons with the HybridAPI framework [17]. We also compared our adaptive executions of Cholesky factorization with MAGMA [14], an implementation of Cholesky factorization for GPUs that perform static distribution of work and data units to CPU and GPU cores. With our generic framework, we obtain 1.5 to 15 times improvement over HybridAPI and about 14% improvement over MAGMA.

Following are the primary contributions of our paper.

1. A novel runtime system for message-driven parallel applications on GPUs.
2. Strategies to automatically compose and dynamically schedule work to CPU and GPU units, and maintain load balance.
3. Novel techniques to manage GPU global memory automatically for minimizing CPU-GPU data transfers and maximizing the reuse of data in GPU memory.
4. Demonstration of improvements with our generic adaptive techniques over static strategies for three important classes of applications.

The rest of the paper is organized as follows. Section 2 gives background on CHARM++ and an existing GPU management framework in CHARM++ called HybridAPI upon which our runtime system has been implemented. Section 3 describes the different components of the G-Charm framework. In Section 4, the various optimization strategies applied by our framework have been discussed. Section 5 presents the experimental results for Jacobi, Cholesky and Molecular dynamics applications comparing G-Charm with existing methods. Section 6 presents related efforts. Finally, in Section 7, we conclude and present some future works.

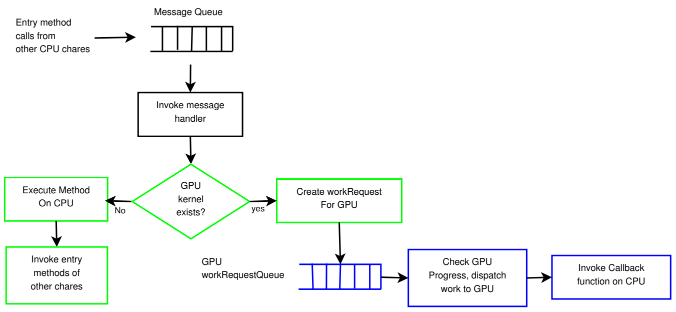
2. BACKGROUND

CHARM++ is a message-driven object oriented parallel programming framework based on C++ [7]. A parallel application written using CHARM++ divides the data among an array of migratable objects called *chares*. The chares are mapped to physical processors, and can be migrated among processors by the CHARM++ runtime system to provide load balance. Typically, the number of chares are much larger than the number of physical processors, resulting in over-decomposition. The chares are associated with specialized methods called *entry* methods. Entry methods of a chare object can be invoked from chares present in same or other processors. Remote entry methods invoked by a chare are queued as messages in a message queue at the destination processor. An instance of CHARM++ runtime system runs on each processor. The runtime system dequeues a message and invokes the corresponding chare's entry method upon arrival of all inputs of the entry method from the other chares. Thus, while input data for a chare is communicated from a remote processor, a processor can perform computation on some other chare for which inputs have already arrived. This enables CHARM++ to effectively overlap communication with computations.

HybridAPI [17] is a GPU execution framework available in CHARM++. In this framework, application programmers create a work request structure specifying the data to be transferred between CPU and GPU before and after kernel invocation, the kernel to be invoked and a callback function provided by the user that will be invoked on the CPU once the work request is processed by the GPU. The HybridAPI runtime system performs the necessary data transfers based on the specifications in the work request structures. The runtime system also checks for the progress of GPU operations related to data transfers and kernel executions, and effectively overlaps kernel execution of one work request with output data transfer for the previous work request and input data transfer for the next work request. Figure 1 depicts the control flow of a HybridAPI based CHARM++ application. In the figure, the green, blue and black boxes denote operations that are performed by the application programs, HybridAPI and the CHARM++ runtime system, respectively.

3. G-CHARM ARCHITECTURE

As mentioned, using HybridAPI without manual agglomeration of kernels and data, and manual data management leads to a large number of kernels, large number of CPU-GPU data transfers, and low GPU occupancies. Our G-



Entry method calls from other CPU chares
Message Queue
Invoke message handler
GPU kernel exists?
No
yes
Execute Method On CPU
Create workRequest For GPU
Invoke entry methods of other chares
GPU workRequestQueue
Check GPU Progress, dispatch work to GPU
Invoke Callback function on CPU

Figure 1: Charm++ Application Control Flow

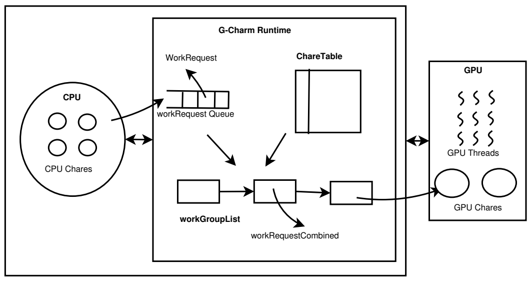


Figure 2: G-Charm architecture

Charm framework addresses these limitations. G-Charm is based on HybridAPI and consists of a runtime that performs various optimizations including minimizing data transfers, data management, dynamic scheduling and work agglomeration. The overall architecture is illustrated in Figure 2.

The application begins execution with the creation of chare objects. Each chare operates on a subset of data and executes its entry methods to update its own data on the arrival of input data from other chares and also to invoke entry methods of other chares. When a chare needs to invoke a kernel on the GPU, it creates a *workRequest* object and invokes a scheduler function in G-Charm runtime that performs dynamic scheduling of the workRequest to either CPU or GPU.

Otherwise, the workRequest is enqueued to GPU workRequestQueue. The queued workRequest object is processed by the G-Charm runtime. Specifically, the runtime checks the data region in the application domain represented by the workRequest data, and tries to avoid redundant data transfers to GPU by transferring only the data not already present in GPU. The G-Charm runtime then adds the workRequest to a node of a linked list called *workGroupList*, in which each node represents a set of workRequest objects that can be combined. G-Charm periodically combines workRequest objects from this list and creates objects of type *workRequestCombined*. The G-Charm runtime then schedules these objects for GPU execution. Thus the G-Charm runtime performs work agglomeration dynamically by combining kernels of multiple work requests for GPU execution. Once a workRequestCombined object finishes execution on the GPU, the runtime system transfers the output data back to the CPU and then invokes a callBack function on the CPU for each workRequest. The callBack functions enable the CPU chares to proceed with further computations by making use of the result computed on the GPU.

4. OPTIMIZATIONS

4.1 Dynamic Scheduling

G-Charm dynamically decides the allocation of a chare to either CPU or GPU for tasks for which kernel functions exist for both CPU and GPU. The workrequest object, after creation, is passed to the G-Charm runtime to check if it can be executed on the GPU. The decision is based on the estimated times of the corresponding task on CPU and GPU. The estimated time of a task in a time step is obtained as the average time taken by the task in the previous time steps.

The G-Charm runtime stores the average time taken by each kernel on GPU and CPU. Initially, the estimated time for each kernel on CPU and GPU is set to zero. The first workRequest for each kernel type is assigned to GPU and the second kernel is assigned to CPU to obtain the initial estimate for CPU and GPU execution time for each kernel. The estimation is for a single CPU or GPU kernel that is associated with a fixed amount of data (number of chares). For example, in a Jacobi application, a kernel corresponds to a certain rows of chares. When combining multiple GPU kernels (described in Section 4.4), the time for the combined kernel is scaled suitably based on the number of chares.

As mentioned in Section 3, the G-Charm runtime maintains a workGroupList where each node represents a set of workrequest objects that will be combined for execution on GPU. For the current workRequest for which the G-Charm runtime decides the allocation to CPU or GPU, the runtime compares the estimated CPU time of the current work request with the combined time of all the work requests in the workGroupList. The combined time is calculated by summing up the estimated GPU times of the workRequest objects in the list. If this combined time exceeds the estimated CPU time for the current work request, then the work request is assigned to CPU. Before execution of the current workrequest on the CPU, the *workRequestCombined* objects formed using the nodes in the workGroupList are scheduled for GPU execution using different CUDA streams. The control is then returned to the CPU chare to process the workRequest on the CPU. Thus, the G-Charm runtime attempts to maximize asynchronous execution of tasks on both GPU and CPU.

If the estimated time of the current work request is more than the combined estimated time, it is assigned to the GPU. In this case, it is added to the workRequestQueue and processed by the G-Charm runtime, which optimizes data transfers and adds it to the workGroupList for combined execution of multiple work requests on the GPU. The objective is to avoid GPU idling due to assigning a work request that can consume large amount of time on the CPU.

In the Molecular Dynamics (MD) example, G-Charm maintains the estimated average execution time of force computations between a pair of patches for CPU and GPU. These estimated times are updated after every execution on the corresponding unit, and are used for dynamic allocation to either CPU or GPU.

4.2 Memory Management

Each chare is associated with a region of data in the application domain. A chare maintains N_{buf} buffers, where N_{buf} is the total number of input and output data for the application. All the N_{buf} buffers of the application are divided among the chares. At a given instant, when the GPU work queue has a certain number of chares for scheduling to GPU, the G-Charm runtime allocates the maximum number of chare buffers that can fit into the GPU device memory. We divide the GPU device memory into two parts: a large *primary region* where memory management is performed, and a small scratch space where chare buffers for a workRequest are freed once the kernel of the workRequest completes. We use this scratch space when consecutive free memory cannot be found for a workRequest buffer in the primary region. For our work, we use 80% of the GPU device memory as the primary region and the rest as the scratch space.

For matrix applications, the G-Charm runtime divides GPU global memory into N_{buf} *buffer pools*. Each *buffer pool* is divided into *slots* of equal size. A *slot* is a placeholder for a chare buffer in GPU memory, and the size of the slot is equal to the size of the chare buffer. The size of a buffer pool i depends on the size of the primary region, $primarySize$, and the size of the chare buffer i, $sizeBuffer_i$, as shown in Equation 1.

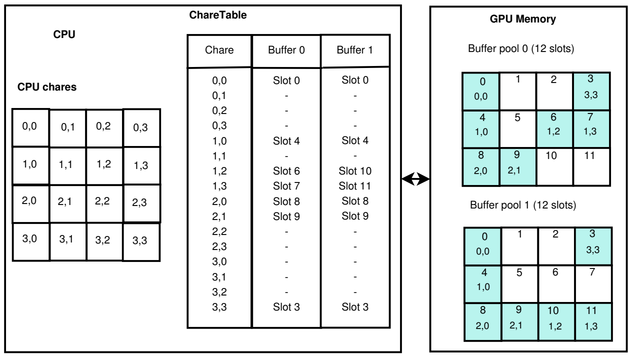
$$bufPoolSize_i = Min(N_{chares} \times sizeBuffer_i,\ \frac{primarySize}{N_{buf}}) \tag{1}$$

where, N_{chares} is the total number of chares. In the equation, the first term denotes the case when all chare buffers of all chares can be accommodated in the primary region, and the second term denotes the case when the primary region is insufficient to accommodate all the chare buffers and is shared among the N_{buf} buffer pools. In the former case, each buffer pool will be divided into N_{chares} slots, while in the latter case, the number of slots will be less than N_{chares} and memory is managed using replacement and migration of chare buffers between CPU and GPU. The allocation of a large buffer pool for storing buffers from different chares for a given input or output array allows for dynamically combining multiple kernels of multiple chares into a single kernel for GPU execution, as discussed in Section 4.4 and also supports coalesced data access.

The G-Charm runtime keeps track of the mapping of chare buffers to slots in the device memory using a *chare table*. An entry in the table corresponding to a chare indicates the slots in the device memory that are occupied by each of the buffers in the chare. The runtime also periodically frees GPU memory, reclaiming the space used by chares that have not been used recently. Figure 3 illustrates the concepts of buffer pools, slots and chare table with $N_{buf} = 2$, $N_{chares} = 16$, and only 12 slots have been allocated in each buffer pool due to insufficient GPU memory. The figure indicates that the buffers in chares (0,0), (1,0), (1,2), (1,3), (2,0), (2,1) and (3,3) are currently in GPU memory occupying the respective slots in both the buffer pools.

4.3 Minimizing Data Transfers

In G-Charm, a workRequest object contains the indices of the chare buffers representing subregions in the application domain. Chares typically need data from other chares for their computations. When a workRequest for a chare is created, the G-Charm runtime uses the buffer indices of the workRequest to check if the buffers are already located in the GPU primary region due to the prior execution of kernels of other chares on the GPU (e.g., data generated from previous iterations). If located, the buffers are not copied to GPU, thus facilitating reuse of buffers and minimizing data transfers. In the MD example, if a buffer containing particle coordinates for a patch is present in GPU memory, it is reused by the workRequest objects corresponding to force calculations with the neighboring patches, thus avoiding redundant transfers.



Chare	Buffer 0	Buffer 1
0,0	Slot 0	Slot 0
0,1	-	-
0,2	-	-
0,3	-	-
1,0	Slot 4	Slot 4
1,1	-	-
1,2	Slot 6	Slot 10
1,3	Slot 7	Slot 11
2,0	Slot 8	Slot 8
2,1	Slot 9	Slot 9
2,2	-	-
2,3	-	-
3,0	-	-
3,1	-	-
3,2	-	-
3,3	Slot 3	Slot 3

Figure 3: Chare table, Device Slots and GPU Buffer pools. In each slot in the buffer pools, the top entry corresponds to the slot number and the bottom entry denote the chare currently occupying the slot. Shaded slots are currently occupied.

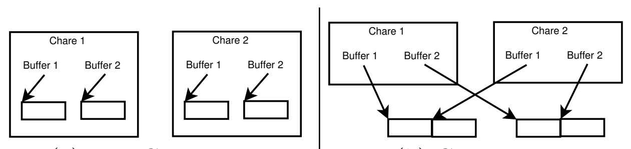
If the buffers are not located in the GPU primary region, the G-Charm runtime finds the required number of free slots in the buffer pool. Chare buffers for a chare with chare index i are mapped to GPU slots using a simple hash function, $slot = i\ mod\ N_{slots}$, where N_{slots} is the number of slots in the buffer pools (12 in Figure 3). If a slot is already occupied, then the runtime scans all the slots in the buffer pool to check for a free slot, and assigns to a chare buffer. If a free slot is not found, memory from scratch space is assigned. Memory allocated from the scratch space is freed on completion of the workRequest.

The G-Charm runtime also uses LRU policy for buffer replacement in the GPU slots. Whenever the percentage of free slots becomes less than a threshold (in our work, 10%), the buffers corresponding to the least recently processed chares are reclaimed by the runtime, thus enabling the recently used chare buffers to stay in the GPU primary region for as long as possible. In addition to the strategies for reducing the amount of CPU-GPU data transfers by data reuse, G-Charm also reduces the data transfer latencies by combining workRequests of multiple chares into a single large kernel, and transferring contiguous data needed by the chares in a single step instead of multiple individual transfers. This is discussed further in the next section.

4.4 Combining Kernels

Combining multiple kernels of different chares into a single large kernel results in smaller number of kernel invocations, smaller CPU-GPU data transfer costs and larger GPU occupancy. Our G-Charm framework dynamically selects the workRequests objects of different chares for combining into a single kernel. In G-Charm, for each workRequest object, information is maintained about whether the kernel in the workRequest object reads from or writes to each chare buffer. This allows the G-Charm runtime to resolve dependencies among workRequests. workRequests for the same task that are independent can be potentially combined into a single kernel.

We refer to the chares with adjacent indices as *adjacent*



(a) Non Contiguous (b) Contiguous

Figure 4: Types of Memory Allocation

chares. In CHARM++ applications, adjacent chares are assigned adjacent regions in the application domain. The simple hash function described in the previous section allows allocation of the buffers of adjacent chares to adjacent slots or a continuous region in the GPU device memory irrespective of whether the buffers are stored contiguously in the CPU memory or not. This in turn enables combining multiple kernels of the different chares into a single kernel with single set of input data corresponding to large continuous regions in GPU memory instead of passing multiple sets of data corresponding to the buffers of the different chares located in different non-contiguous regions. Since G-Charm combines kernels only when the corresponding data regions are contiguous in GPU memory, the combined kernel operates on contiguous buffers.

The buffers for all the workRequest objects that are combined will have to be transferred to GPU memory before kernel invocation, if they are not already present in the GPU memory. If each buffer is transferred to GPU separately, though the number of kernels invoked decreases as a result of combining or agglomeration, the average number of memory transfers per kernel invocation increases. This results in GPU idling since the memory transfer time per kernel invocation is more than the kernel execution time. In order to reduce the memory transfer overhead, the buffers will have to be combined on the CPU before transferring to GPU.

G-Charm allows the chares to share memory allocated in the main chare in shared memory systems. In these systems, buffer i of individual chares can be either allocated within each chare at different locations in CPU memory as shown in Figure 4(a) or a single large buffer can be allocated in the main chare and then the individual chares can access their buffers using offsets into the contiguous region as shown in Figure 4(b). The latter approach is advantageous when combining multiple workRequests of different chares since the buffers needed by all the chares can be sent from a contiguous region in CPU memory using a single CPU-GPU data transfer, instead of multiple individual transfers. For example in Figure 4(b), instead of transferring Buffer 1 corresponding to Chare 1 and Chare 2 separately, a single buffer starting at address of Buffer 1 of Chare 1 and of size equal to the combined sizes of Buffer 1 in Chare 1 and Chare 2 can be transferred in a single step.

The G-Charm runtime maintains a workGroupList into which workRequest objects are added. Each node in the list contains a set of workRequest objects that do not have data access conflicts and can be executed concurrently. The workRequest objects of a node are eventually combined to form a single combinedWorkRequest and executed with a single kernel execution. The G-Charm runtime executes the workRequestObjects in the order maintained in the workGroupList. A workRequest, wr, is inserted into the workGroupList based on the following conditions.

1. If wr reads from buffer B, wr is inserted after the last node containing a workRequest that writes to this buffer. This is done to preserve Read After Write (RAW) data integrity constraint.
2. If wr writes to the buffer B, wr is inserted after the last node containing a workRequest that uses this buffer for read or write operations. This is done to preserve Write After Read(WAR) and Write After Write(WAW) data integrity constraints.
3. If there exists a node in the list that contains workRequests with same kernel as wr, then wr is added to the same node, otherwise a new node is inserted in the list and wr is added to it.

The G-Charm runtime periodically calls a *combine* routine to create *workRequestCombined* objects from the nodes of the workGroupList and enqueue them for GPU execution.

In the MD example, the workRequests for different compute objects do not have dependencies since the force calculations between different pairs of patches can be performed independently. Hence any set of workrequest objects can be combined into a large kernel that deals with multiple pairs of patches. However, in linear algebra operations like Cholesky decomposition, a column may be updated by several other columns, and the workrequests corresponding to these column updates have dependencies. In this case, the G-Charm runtime combines workrequest objects corresponding to a single column, since the updates of elements within a column can be performed independently.

4.5 Programming Interface

The GPU environment supported by G-Charm is CUDA. The application programmer writes a separate GPU CUDA kernel and CPU Charm++ function for each task, and these are compiled separately using *nvcc* compiler and *charmc* program, respectively, and then linked using *charmc* program. G-Charm provides some functions for use by applications to provide information to G-Charm runtime for memory management and dynamic scheduling. Each chare object in the chare array that owns a part of data has to initialize G-Charm runtime by invoking the *initGM* method with various information including chare array dimension, the number of chares along each dimension, the number of buffers in each chare (N_{buf}), if the application involves matrix operations, row and column dimensions of the submatrices assigned to each chare and whether chares share CPU memory. One instance of the G-Charm runtime runs per processor. Since each instance has to be initialized and the chares are distributed among all processors, each chare is required to call *initGM* method and only the first invocation on each processor actually initializes the G-Charm runtime. The initialization involves GPU memory allocation including allocation of buffer pools and slots, and initialization of the chare table.

The application also calls a *checkGPURun* function to decide whether a workRequest has to be run on CPU or GPU. If this function returns true, the G-Charm runtime adds the workrequest to workRequestQueue and invokes the GPU kernel specified in the workrequest. If the function returns false, the user program explicitly invokes the corresponding CPU function.

5. EXPERIMENTS AND RESULTS

We demonstrate the benefits of our G-Charm framework with three classes of applications: a highly regular and parallel 2D Jacobi solver, a regular dense matrix Cholesky factorization representing linear algebra computations with dependencies among parallel computations and highly irregular molecular dynamics simulations. The experiments were run on a Tesla cluster with each node of the cluster containing 4 quad-core AMD Opteron 8378 cores and one Tesla S1070 GPU, and a Fermi cluster with each node containing one 4-core Intel Xeon W3550 processor and one Tesla C2070 GPU. The Tesla S1070 GPU system is composed of 4 GPUs with each GPU made up of 240 GPU cores. The Fermi C2070 GPU system is composed of a single GPU consisting of 448 GPU cores. All our experiments were performed on single GPU systems.

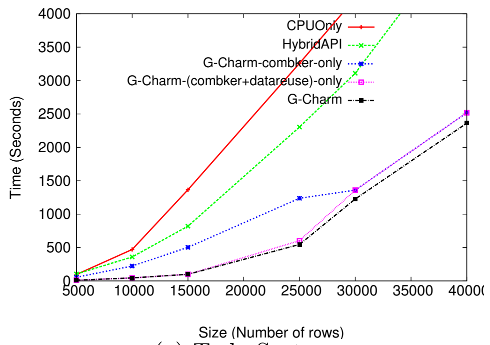
5.1 Jacobi application

This is a standard 2-D Jacobi Poisson's equation solver involving 5-point stencil computations. Optimizing such stencil computations on GPU systems is essential for high performance of large-scale multi-dimensional grid problems [4, 16]. In the CHARM++ Jacobi implementation, the 2-D grid is divided among a 2D array of chares. Each chare initializes its own data and in each iteration exchanges boundary information to its four neighboring chares, performs computations of local data and sends error values to the main chare. The main chare is responsible for computing the overall residual error for initializing the next iteration.

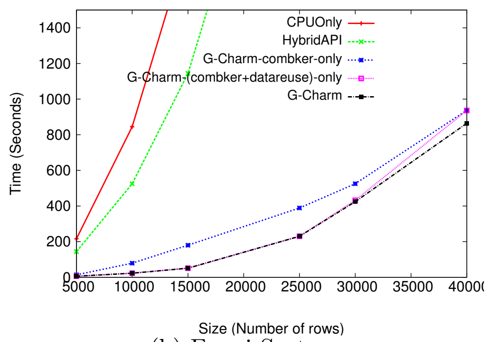
Figures 5(a) and 5(b) compare the performance using G-Charm with the performance using the other approaches, namely *CPUOnly* and *HybridAPI*, on both the Tesla and the Fermi systems. The CPUOnly approach denotes the execution using only a CPU core. We have included comparison to CPU only version, since an inefficient implementation on GPU can result in even slower performance than CPU only execution due to GPU overheads including kernel invocations and memory transfers. In the HybridAPI framework, each chare, after receiving the boundary data from the neighboring chares, creates a workRequest object and enqueues it into GPU workRequest queue for GPU computations of the local data. The G-Charm framework automatically performs asynchronous computations of parts of the 2-D domain on both the CPU and the GPU cores, manages the data, and combines multiple kernels. In order to isolate the advantages obtained using each of the three primary optimizations performed by G-Charm, we also show the results obtained by *G-Charm-combker-only*, involving only the optimization of combining kernels and reducing the number of kernel invocations, *G-Charm-(combker+datareuse)-only*, involving only the optimizations of combining kernels and data reuse, and *G-Charm*, also involving asynchronous CPU executions.

The average speedup of G-Charm over HybridAPI is about 10. The primary benefit with G-Charm is due to the combined kernel execution, as seen in the difference between *G-Charm-combker-only* and *HybridAPI* results. Combining kernels not only reduces the number of kernel invocations but also reduces the total number of data transfers due to concatenating and sending the large data needed for the combined kernel in a single step. The *G-Charm-combker-only* version gives a speedup of 2 in Tesla and 10 in Fermi System since data transfer rate between CPU and GPU is much larger in Fermi (144 GB/sec) compared to Tesla system (12.8 GB/sec). However, in the *G-Charm-combker-only* version, the same data can get transferred back and forth between the CPU and GPU across different iterations. As the matrix size becomes larger, the amount of data transfer increases. The *G-Charm-(combker+datareuse)-only* version, involving data reuse and management, gives an average benefit of about 5 times over the HybridAPI version in Tesla and about 16 times in Fermi systems. *G-Charm-(combker+datareuse)-only* gives best performance when the matrix completely fits in GPU as shown for sizes up to 15000×15000 in Tesla and 25000×25000 in Fermi. For matrix size of 40000×40000, the performance of *G-Charm-(combker+datareuse)-only* is the same as *G-Charm-combker-only*, i.e. without data reuse, since for this size, only one GPU chare could be accommodated in GPU at a time and the data corresponding to the chare has to be swapped out of the GPU memory to make way for another GPU chare execution. The optimization due to asynchronous CPU executions brings about 6% additional improvement in performance. Asynchronous CPU utilization has been employed only when a matrix cannot be fit completely in GPU memory.

We also compare the G-Charm and the HybridAPI approaches in terms of the number of kernels invoked, the amount of data transferred, and the percentage of time CPU is utilized for useful computations during application execution. Figure 6 shows these results for the Fermi system. Similar results were obtained on the Tesla system. We find that G-Charm invokes about 30 times lesser number of kernels, and transfers about less than half the amount of data than the HybridAPI for matrix size 40000×40000 for which G-Charm has maximum number of migrations as well as maximum number of kernel invocations. We also find that



(a) Tesla System



(b) Fermi System

Figure 5: Jacobi Execution Time

G-Charm, due to the optimization of asynchronous CPU executions, gives about 5% increased CPU utilization over HybridAPI. In our HybridAPI implementation, matrix update with neighbor values is performed on GPU and error calculation is performed on CPU. These error calculations constitute about 45% CPU utilization in the HybridAPI implementation. In G-Charm, asynchronous CPU utilization is employed only when matrices do not fit in GPU memory. Figure 6(c) shows the increase in CPU utilization for G-Charm as the matrix size increases.

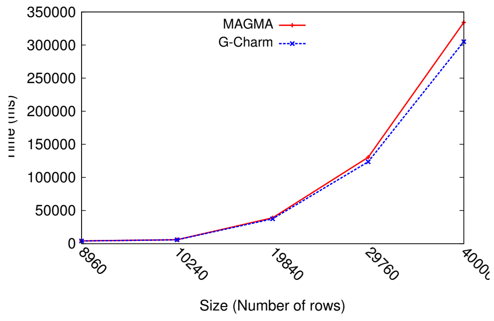
The execution profiles of the different implementations on the Tesla system are illustrated in Figure 7. This figure shows the execution timeline for GPU and CPU for matrix size 25000 × 25000 for the first ten iterations for all implementations. The patterns are found to persist across all iterations. The CPU time shown in this figure corresponds to only performing useful computations. Figure 7(a) shows HybridAPI execution which involves large number of kernel invocations and memory transfers. The continuous CPU-GPU memory transfers for large number of small kernels in HybridAPI can be seen by the almost continuous green bar. Figure 7(b) illustrates the profile of *G-Charm-combker-only* in which the combination of kernels results in reduced number of CPU-GPU data transfers as shown by the vertical lines in the bars. The optimization related to data reuse is shown in Figure 7(c) in which the frequency of data transfers is smaller. This is illustrated by the larger gaps between the data transfers. Finally, Figure7(d) illustrates asynchronous CPU-GPU executions, in which the CPU is sufficiently utilized.

5.2 Cholesky Decomposition

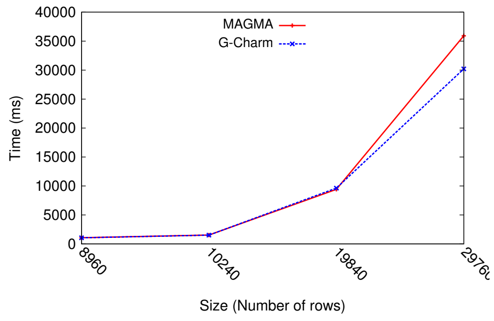
Cholesky decomposition factorizes a real, symmetric, positive definite matrix $A = LL^T$, where L is a lower triangular matrix. Cholesky factorization primarily involves updates or multiplication (*cmod*), division (*cdiv*) and factorization steps. We have implemented a subcolumn Cholesky factorization using G-Charm in which the factorization step is performed on CPU, and the *cdiv* and *cmod* steps are performed on GPU.

We compare the G-Charm implementation of double precision Cholesky factorization with MAGMA [14], a highly tuned dense linear algebra library for GPUs. In G-Charm implementation, a main chare allocates memory for the entire input matrix and creates a set of chares, with each chare being responsible for factorization of a portion of the input matrix. Chares in a column perform the *cmod* operation using all the previous columns to update their submatrices. Once *cmod* is complete for the diagonal chare element of that column, it proceeds to the factorization step. After factorization, all the chares below the diagonal chare perform the *cdiv* operation. The application begins with *cmod* operation for the first column and ends with *cdiv* operation for the last column.

Figure 8 shows the results with G-Charm and MAGMA on Tesla and Fermi systems. We have tested for matrix sizes up to 40000×40000 in Tesla and 29760×29760 in Fermi system, the maximum sizes that can be accommodated in the CPU memory of these systems. The Tesla and Fermi systems can accommodate matrices of size up to 19840×19840 in the GPU device memory. We find that up to these matrix sizes, both MAGMA and G-Charm give similar results since for these smaller matrices, GPU data management functionality of G-Charm is not exercised, and MAGMA also performs careful data management and composition of large kernels. For larger matrix sizes, G-Charm performs up to 9% and 15% better than MAGMA for double and single precision computations, respectively. The higher performance is because G-Charm performs efficient GPU memory management, reduced number of kernel invocations, and reduced CPU-GPU data transfers by reuse of GPU data, while MAGMA performs large amount of CPU-GPU data migrations for these large matrix sizes.



(a) Tesla System

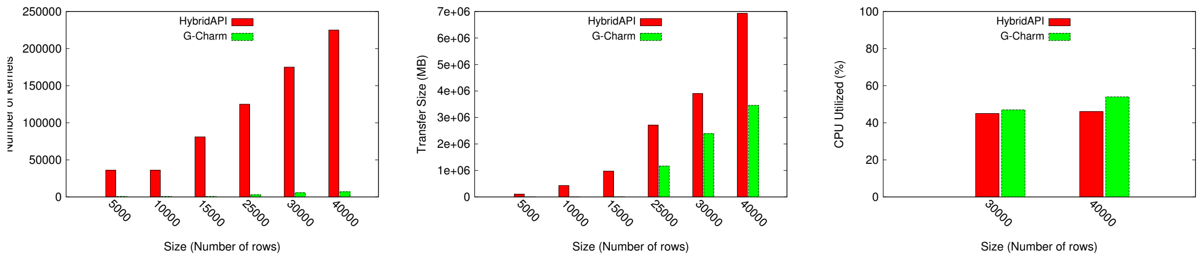


(b) Fermi System

Figure 8: Cholesky Decomposition Execution Time

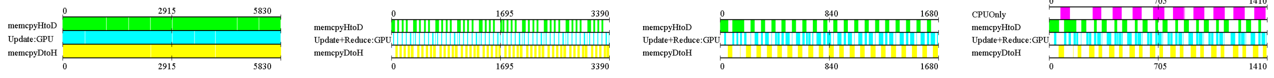
We also performed comparisons with HybridAPI and StarPU [3] frameworks for Cholesky factorization. We obtained about 8x speedup with G-Charm over HybridAPI. Similar to G-Charm, StarPU [3] also performs dynamic scheduling of work to both CPU and GPU, and data management of GPU data. We found that StarPU gave very large execution times. For example, for matrix size of 29760 × 29760, the execution time with StarPU was 137 seconds on the Fermi system, while the execution times with G-Charm and MAGMA were 31.8 and 34.7 seconds respectively. The large execution times with StarPU is because StarPU invokes *cdiv* and *cmod* operations for each individual block of the matrix resulting in 11991 kernel invocations while G-Charm invokes a single kernel for contiguous chares resulting in only 1825 kernel invocations.

We also compared the G-Charm and the MAGMA approaches in terms of the number of kernels invoked, and the amount of data transferred. Figure 9 shows these results for the Fermi system. We find that G-Charm invokes about 2.5 times lesser number of kernels for matrix size 29760×29760, since it allocates adjacent chares in adjacent GPU slots and invokes a single kernel to process all chares that are con-



(a) Number of Kernel Invocations (b) Amount of CPU-GPU Data Transfer (c) CPU Utilization

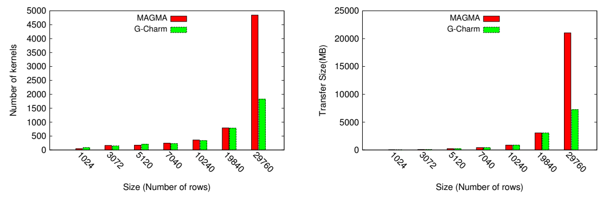
Figure 6: Jacobi Execution Statistics on the Fermi System



(a) HybridAPI (b) G-Charm-combker-only (c) G-Charm-(combker+datareuse)-only (d) G-Charm

Figure 7: Jacobi Execution Profile on the Tesla System

tiguous in GPU memory. For smaller matrices, we find that both MAGMA and G-Charm invokes similar number of kernels. G-Charm also transfers about 3 times lesser data than MAGMA due to dynamic data reuse and shared memory implementation when matrix size does not fit entirely in GPU memory. This efficient data management and reuse in our generic G-Charm framework results in improved performance of G-Charm (31810 milliseconds) over MAGMA (34740 milliseconds) for matrix size 29760 × 29760.



(a) Number of Kernel Invocations (b) Amount of CPU-GPU Data Transfer

Figure 9: Cholesky Decomposition Execution Statistics on the Fermi System

5.3 Molecular Dynamics

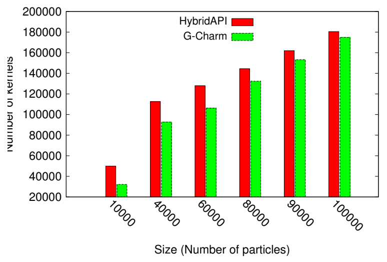
We have also conducted experiments on a non-matrix application, namely, molecular dynamics. Molecular dynamics is a highly irregular application for simulating the interactions between molecules over a period of time. We consider a two-dimensional molecular dynamics application in which the 2D space is partitioned into patches. Each patch owns the particles present in the region. In each timestep, force on each particle due to other particles within a cutoff distance is calculated and the position of the particles are updated. Particles migrate to neighboring patches according to new positions and the application proceeds to next timestep. This is repeated for a fixed number of timesteps.

In the Charm++ implementation, a *compute object* calculates force between a pair of patches, as mentioned in Section 1. The entry method *interact* takes two vectors of particles belonging to two patches and updates force components of each particle. The widely-used NAMD [10, 12] molecular dynamics framework based on Charm++ also adopts a similar parallelization scheme based on compute objects. The *interact* method has been implemented as a CUDA kernel for the HybridAPI and G-Charm implementations. The G-Charm framework automatically performs asynchronous computations of interaction calculations on both the CPU and the GPU cores.

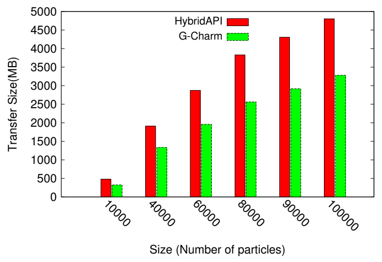
Figure 10 compares the performance using G-Charm with the performance using the other approaches, namely *CPUOnly* and HybridAPI, on both the Tesla and the Fermi systems. The *CPUOnly* approach denotes the execution using only a CPU core. We find that the average performance improvement of G-Charm over the cpu-only version is about 33% and over HybridAPI is about 21%.

Figure 11 shows the number of kernels invoked, the amount of data transferred, and the percentage of time CPU is utilized during application execution, for the cpuOnly, HybridAPI and the G-Charm implementations on the Fermi system. We find that G-Charm invokes about 20% lesser number of kernels and transfers about 30% lesser data than HybridAPI. We also find that G-Charm gives 2.5 to 5 times increased CPU utilization over HybridAPI. Both in HybridAPI and in G-Charm, updating particle coordinates after each iteration and subsequent migration of particles were performed on CPU. These contribute to nearly 10% of CPU time for HybridAPI. In G-Charm, apart from these computations, force calculation for some compute objects are dynamically assigned to CPU for asynchronous execution, resulting in improved CPU utilization.

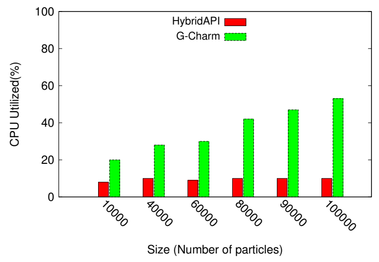
Figure 12 shows the profiling output of G-Charm and HybridAPI executions for 10 iterations on the Fermi system. We find that GPU remains idle for some time after every iteration during which CPU updates particle coordinates and migrates particles to other chares. This idle time can be reduced by providing kernel function for updating particle coordinates. We find that the primary reason for higher per-



(a) Number of Kernel Invocations

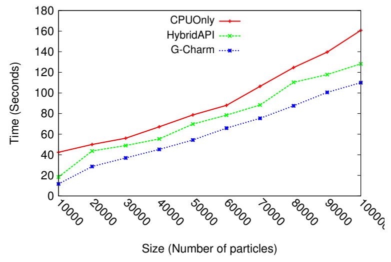


(b) Amount of CPU-GPU Data Transfer

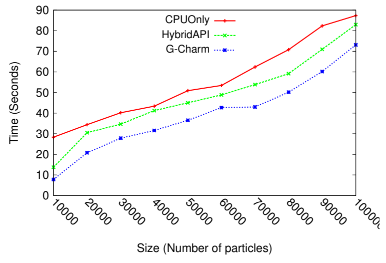


(c) CPU Utilization

Figure 11: Molecular Dynamics Execution Statistics on the Fermi System



(a) Tesla System



(b) Fermi System

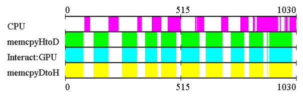
Figure 10: Molecular Dynamics Execution Time

formance of G-Charm in molecular dynamics application is due to the efficient use of CPU for useful computations.

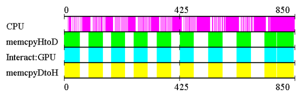
5.4 Discussion

In general, we find that combining kernels is beneficial not only in reduction of number of kernel invocations, but also in reducing the latencies of CPU-GPU data transfers. Data management and reuse of GPU data can provide significant benefits for large problem sizes in which the entire data cannot fit in the GPU memory. Finally, asynchronous CPU-GPU executions can provide added benefits, but the extent of the benefits is application-dependent and specifically depends on the CPU-GPU performance ratio obtained for the computations and the ability to estimate the performance to apportion the appropriate work to CPU.

Overall, we find higher gains of 10X and 8X in Jacobi and Cholesky applications, respectively, and smaller gains of about 21% in molecular dynamics application with G-Charm over HybridAPI implementation. The higher gains in Jacobi application is due to non dependency among chares in an iteration combined with very less CPU-GPU data transfers, and the higher gains in Cholesky factorization is due to more opportunities for efficient data management. The smaller gain in molecular dynamics application is because of non-shared memory implementation, memory copy overheads and less data reuse.



(a) HybridAPI



(b) G-Charm

Figure 12: Molecular Dynamics Execution Profile on the Fermi System

6. RELATED WORK

The AMM system [11] aims to reduce memory transfer between CPU and GPU automatically by avoiding transfer of non stale data between CPU and GPU, eager transfer of data to CPU and transfer of GPU only data to CPU by performing compiler analysis. Compiler analysis has limitations in identifying all non-stale data. Our framework, by relying on the user input, can reduce the amount of transfers of such data. Our framework also reduces the number of kernel invocations by agglomeration of kernels and the associated data. As shown in our results, combining multiple kernels provides clear performance benefits. The work by Ashwin Prasad et al. [13] maps a MATLAB basic block to either CPU or GPU at compile time using estimated run time of the block obtained through profiling. In our approach, work assignment to CPU and GPU is determined dynamically at runtime using information such as average times taken by CPU and GPU to process a kernel, and the number of messages waiting to be processed in CPU and GPU queues.

Some recent efforts have focused on efficient GPU computations of stencil computations [4, 16]. The work by Bandishti et al. [4] partitions the iteration space using hyperplanes for concurrent and load balanced executions. The work by Venkatasubramanian et al. [16] avoid synchronization delays at the end of the iterations using highly asynchronous computations. Our work is on providing a general framework based on asynchronous message-passing principles. Heterogeneous tiled algorithms implemented for Cholesky and QR factorizations [14] uses a static partitioning scheme to effectively utilize all CPU and GPU cores. In our work, the heterogeneous tile sizes are determining dynamically by agglomerating data and kernels. NAMD [10] is a widely used scalable molecular dynamics application that has GPU extensions [12] for calculating short range non bonded forces. However, this work does not attempt to reduce non-stale data and does not combine multiple kernels to reduce the number of kernel invocations.

Our runtime system is similar to StarPU [3] in which the application programmer creates CPU and GPU kernels and the runtime system decides whether to schedule a task to CPU or GPU. StarPU also automatically manages and moves data between CPU and GPU [2]. Our framework, in addition to data management, also automatically performs asynchronous CPU-GPU executions of non-dependent tasks and agglomeration of work.

Our work is closely related to the work by Kunzman [8] that has developed a unified programming model for abstracting different types of accelerators, with the runtime system performing various tasks such as load balancing, work agglomeration and data management . In this work, the user has to explicitly specify if a given data should be persistent in GPU memory across kernel invocations to avoid redundant data transfers, while in our work, such data management is performed automatically by the runtime system.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a framework called G-Charm for efficient execution of message-driven parallel applications on hybrid systems. G-Charm focuses on optimizing memory transfers to GPU by reusing data present in GPU memory, reducing the number of kernels by invoking a single kernel on the combined data of multiple chares and reducing CPU and GPU idle times by dynamic work distribution. By conducting experiments with three different applications belonging to three different categories, we found that the adaptive strategies in G-Charm provide a speedup of about 1.5 to 15 over a previous GPU-based implementation of CHARM++, and about 14% improvement over a highly tuned dense linear algebra routine. Currently, we have focused on effectively using single CPU and GPU for applications. In our future work, we plan to scale G-Charm to multi-GPU systems. We also plan to improve our programming abstractions, and explore more classes of large applications.

8. ACKNOWLEDGMENTS

This work is supported partly by DOE via DE-SC0001845, by NSF via the grant HECURA-0833188, and by CUDA Research Centre award from NVIDIA to SuperComputer Education and Research Centre, Indian Institute of Science.

9. REFERENCES

[1] Top500 supercomputer sites. `http://www.top500.org/lists/2012/11/`.

[2] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, and W. m. Hu. Faster, Cheaper, Better–a Hybridization Methodology to Develop Linear Algebra Software for GPUs. *GPU Computing Gems*, 2, 2010.

[3] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling Stencil Computations to Maximize Parallelism. In *In Proceedings of ACM/IEEE Supercomputing Conference (SC)*, 2012.

[5] P. Jetley, F. Gioachin, C. Mendes, L. Kalé, and T. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *In Proceedings of IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2008.

[6] P. Jetley, L. Wesolowski, F. G. L. Kalé, and T. Quinn. Scaling Hierarchical N-body Simulations on GPU Clusters. In *In Proceedings of ACM/IEEE Supercomputing Conference (SC)*, pages 1–11, 2010.

[7] L. Kalé and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. *Parallel Programming using C++*, pages 175–213, 1996.

[8] D. Kunzman. *Runtime Support for Object-based Message-driven Parallel Applications on Heterogeneous Clusters*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[9] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kalé. ParFUM: A Parallel Framework for Unstructured Meshes for Scalable Dynamic Physics Applications. *Engineering with Computers*, 22(3):215–235, 2006.

[10] C. Mei, Y. Sun, G. Zheng, E. Bohm, L. Kalé, J. James, and C. Harrison. Enabling and Scaling Biomolecular Simulations of 100 Million Atoms on Petascale Machines with a Multicore-Optimized Message-Driven Runtime. In *In Proceedings of ACM/IEEE Supercomputing Conference (SC)*, 2011.

[11] S. Pai, R. Govindarajan, and M. Thazhuthaveetil. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-Assisted Runtime Coherence Scheme. In *In the Proceedings of the 21st international conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 33–42, 2012.

[12] J. Phillips, J. Stone, and K. Schulten. Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters. In *In the Proceedings of IEEE/ACM Supercomputing Conference (SC)*, pages 1–9, 2008.

[13] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors. In *In the Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 152–163, 2011.

[14] F. Song, S. Tomov, and J. Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and Multi-GPU Systems. In *In the Proceedings of IEEE/ACM Supercomputing Conference (SC)*, pages 365–376, 2012.

[15] S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Computing*, 36(5):232–240, 2010.

[16] S. Venkatasubramanian and R. Vuduc. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In *In Proceedings of the 23rd international Conference on Supercomputing (ICS)*, pages 244–255, 2009.

[17] L. Wesolowski. An Application Programming Interface for General Purpose Graphics Processing Units in an Asynchronous Runtime System. Master's thesis, University of Illinois at Urbana-Champaign, 2008.