Scaling Hierarchical N-body Simulations on GPU Clusters

Pritish Jetley[†], Lukasz Wesolowski[†], Filippo Gioachin[†], Laxmikant V. Kalé[†] and Thomas R. Quinn[*]

[†]Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

[*]Department of Astronomy
University of Washington
Seattle, WA 98195, USA

Abstract— **This paper focuses on the use of GPGPU-based clusters for hierarchical N-body simulations. Whereas the behavior of these hierarchical methods has been studied in the past on CPU-based architectures, we investigate key performance issues in the context of clusters of GPUs. These include kernel organization and efficiency, the balance between tree traversal and force computation work, grain size selection through the tuning of offloaded work request sizes, and the reduction of sequential bottlenecks. The effects of various application parameters are studied and experiments done to quantify gains in performance. Our studies are carried out in the context of a production-quality parallel cosmological simulator called ChaNGa. We highlight the re-engineering of the application to make it more suitable for GPU-based environments. Finally, we present performance results from experiments on the NCSA Lincoln GPU cluster, including a note on GPU use in *multistepped* simulations.**

Index Terms—N-**Body Simulations, Barnes-Hut Algorithm, General Purpose Graphics Processors, Performance Analysis**

I. INTRODUCTION

In recent years, the GPU has received widespread acceptance as the accelerator of choice for computation-intensive applications. The use of GPUs to accelerate applications as varied as biomedical imaging, molecular dynamics and stochastic financial modeling has been detailed in the literature. However, the results of adaptation to this relatively new class of architecture vary from stellar for some applications to mediocre for others. Whereas applications dominated by floating-point arithmetic with simple layout and regular memory access patterns have done well on these platforms, significant effort is required to obtain appreciable speedups for applications that exhibit irregular parallelism. To date, relatively few studies have been carried out on the scalability of applications on large clusters of these devices.

In this paper, we analyze the performance of Barnes-Hut simulations on GPU clusters. The various tasks associated with the Barnes-Hut procedure are split among the CPU and the GPU. The parallel construction and traversal of the global Barnes-Hut tree is done by the CPUs. The traversal procedure produces lists of computations that are offloaded to the GPU. Therefore, all of the actual force calculation is performed on the GPU. The efficient use of GPUs as offload devices requires the study of different parameters of the Barnes-Hut

application. In this paper, we focus on aspects such as the layout of force computation kernels, the balance between tree traversal and force evaluation, the overlap of CPU work with GPU work, the removal of serial bottlenecks on the CPU and considerations for multi-time resolution simulations. While the effect of such factors has been examined in detail for shared and distributed memory systems, this work discusses their effects in the context of clusters of GPUs. Mathematical models for performance are used where appropriate. In addition, we provide empirical data from a hierarchical N-body simulator called ChaNGa. We establish a lower bound on the time taken to compute forces in this manner of dividing responsibility for tasks between the CPU and GPU. Finally, we demonstrate the efficacy of the techniques discussed by highlighting performance results on up to 256 GPUs using a variety of cosmological data sets.

The rest of the paper is organized as follows. We begin with brief descriptions of related work and the software infrastructure on which ChaNGa is based. Next, we analyze the performance of the Barnes-Hut algorithm on GPU clusters along various dimensions. Finally, we highlight the utility of these optimizations in the form of scaling results from the simulation of various data sets using ChaNGa.

II. RELATED WORK

Warren and Salmon [1] were among the first to design a scalable parallel simulator based on the $O(N \lg N)$ Barnes-Hut algorithm. Grama *et al.* [2] have presented an analysis of different parallel formulations of the Barnes-Hut procedure.

More recently, Kawai *et al.* [3] and Makino *et al.* [4] have demonstrated the use of specialized hardware to obtain good speedups over the traditional CPU-based approach. However, they used the $O(N^2)$, all-pairs algorithm which does not scale well with the number of particles. The all-pairs algorithm has also been adapted to the GPU. Implementations showing appreciable speedups over traditional CPUs have been provided by Nyland *et al.* [5] and Belleman *et al.* [6], among others.

Hamada *et al.* [7] have provided an efficient implementation of the Barnes-Hut algorithm for a custom-built GPU cluster. A large data set, exceeding 1.5 billion particles, was used in that work to demonstrate good performance and performance/price

results using MPI and CUDA. We assess the strong-scaling performance of GPU-assisted N-body simulations by using small data sets, and demonstrate that through a series of careful optimizations, good scaling can be achieved even with small sets of astronomical data. Moreover, we provide an analysis of the various factors that affect the scaling performance of a production-quality simulator such as ChaNGa. Our use of techniques such as quadrupole moments and gravitational softening yields greater physical precision than the code outlined in that work. Since the data sets used in our experiments are physically accurate representations of astronomical systems and feature in the computational astronomy literature, the results presented are relevant to both the astrophysics and computational science communities.

Lashuk *et al.* [8] have created scalable algorithms for the FMM technique and present results on similar counts of GPUs as described in this paper. While the force calculation methods used differ from the ones employed here, we note that the authors present speedup comparisons between the GPU-augmented and CPU-only versions of the code by restricting the number of processes per socket (4 CPU cores) to one. Moreover, artificial data sets are used to obtain uniform distributions of particles, and only weak-scaling results are presented.

Aubert and Teyssier have used a cluster of GPUs to perform radiative transfer calculations coupled with a tree-based AMR code called RAMSES [9]. They demonstrate good speedups over a CPU-only version. However, the finite difference calculations carried out on the GPU in that code are significantly different from the pairwise force computations performed in the GPU kernels studied in this paper. In particular, the irregular nature of the data parallelism in tree-based codes makes it harder to attain good speedups.

III. GENERAL PURPOSE GPUs AND CHARM++

We used NVIDIA's CUDA [10] technology to develop the GPU code for ChaNGa. CUDA devices are programmed using C with extensions for expressing parallelism on the GPU and utilizing GPU-specific hardware units. As an accelerator device, a GPU has limited control capabilities. CUDA reflects this fact by requiring programmers to write CPU code for managing the units of parallel GPU code, which are known as *kernels*.

In order to benefit from GPU acceleration, a program must be decomposed into a large number of concurrently schedulable units. Furthermore, the high density of execution units on the GPU comes at a cost of limited instruction scheduling and dispatch logic which is shared across a group of eight execution units called a *Streaming Multiprocessor* (SM) in CUDA terminology. In practice, this dictates that groups of threads must execute the same code in lockstep fashion or suffer poor performance. In CUDA GPUs these groups, known as *warps*, consist of 32 threads. If threads within a warp diverge on a branch, the full warp is serially executed on each branch path, with threads converging into a single execution path only after the branch is finished.

CUDA requires the organization of warps into larger units called *blocks*. Threads are assigned to SMs in units of blocks and can only communicate with other threads in the same block. Communication across blocks requires termination of the GPU kernel and data transfer into CPU memory where the required data manipulation can be performed. These issues limit the applicability of GPUs primarily to data parallel applications, and usually require significant program modifications when porting applications to use a GPU.

CUDA gives users access to several on-chip memory and computational resources which are unique to GPUs. Each SM has a large *register file* which provides fast storage that is shared among all its threads. Register use is indirectly controlled by the user through variable declarations inside kernels. In addition to a register file, each SM also contains a modestly sized *shared memory* unit. Shared memory is as fast as registers and can be used to communicate among threads in the same block. Shared memory is 16-way banked, requiring threads within a half-warp to access data on different banks to yield full bandwidth. *Constant memory* provides single-cycle access to immutable data as long as all threads in a half-warp access the same value. *Texture memory* is cached on-chip and optimized for 2D spatial locality. Any data not stored in one of the above units must be read from a relatively large *global GPU memory*. Global GPU memory latency is on the order of 400-600 cycles, but its bandwidth is high compared to CPU memory as long as values accessed by neighboring threads are located in close proximity in memory. Global memory latency is dynamically overlapped with useful work done by thread blocks that are ready to execute.

A. CHARM++

CHARM++ [11] is a message-driven parallel language implemented as a C++ library. CHARM++ programs consist of collections of objects called *chare objects*, which execute in response to messages received from other chare objects. While it is the programmer's responsibility to partition a program into a number of chare objects, the CHARM++ adaptive runtime system performs the mapping of objects to processors. CHARM++ objects communicate through asynchronous messages using the familiar C++ syntax of invoking a function on an object. The runtime system keeps track of the physical location of chare objects and handles the low-level details of sending and receiving messages on the network. CHARM++ is built on top of a communication layer called Converse which supports most hardware and network architectures in use today. CHARM++ shares the message driven execution model with the Actors paradigm [12].

CHARM++ applications are typically written to have significantly more chare objects than the number of processors used during execution. The presence of multiple objects on a single processor allows for automatic overlap between computation and network communication, since if one object is waiting for a message, the runtime system can schedule objects whose messages have already been received.

B. CHARM++ *GPU Manager*

The purpose of the CHARM++ GPU Manager is to simplify the management of GPUs in CHARM++ programs while providing good GPU and CPU utilization. It is possible to use CUDA directly in Charm++, and applications have been written which do so to good effect [13]. When using GPUs in a Charm++ program, one needs to be careful to ensure that the CPU is not blocked when transferring data between CPU and GPU memories or calling kernel functions. Since Charm++ programs often have several messages per processor enqueued for execution, blocking the CPU on every GPU operation is undesirable. Second, users need to be able to share GPUs among chare objects such that the objects do not synchronize explicitly with each other when using the GPU. Since chares often do not have a prescribed order of execution, synchronizing would be difficult and probably detrimental to performance. Use of the CUDA stream construct and polling functions provides a partial solution to these issues. For every CUDA kernel call and data transfer, users can indicate in which stream the operation should execute. CUDA operations within a single stream execute in order while a kernel execution and a GPU data transfer operation which happen in different streams can theoretically overlap in execution. The most natural way to utilize streams in CHARM++ is to use a different CUDA stream parameter for every chare executing on the GPU. Users can poll for completion of operations within a single stream. This provides a way to determine whether all GPU operations belonging to a particular chare have finished executing.

The above approach unfortunately suffers from some performance and usability problems. First, it requires periodic polling calls for each chare which is using the GPU to check whether work for a particular stream has completed. This reduces code clarity and is tedious for the user. Second, while the number of chares which have work to be executed on a particular GPU may be large, only one chare's kernel can execute at a time on the GPU. The large number of calls to periodic functions which perform the polling will waste CPU cycles. Finally, while CUDA can theoretically overlap kernel execution with data transfer in concurrent streams, in practice the most natural usage patterns of streams greatly limit the possibility of overlap between kernel execution and data transfer. CUDA hardware consists of a compute engine and a DMA engine which can operate concurrently. Stream operations are assigned to the appropriate engine in FIFO order as they are encountered in the program. A typical usage scenario for a stream is to submit three operations one after another: data transfer into the device, kernel execution, and transfer of results out of the device. While the kernel is executing, the data-transfer-out operation will remain at the head of the DMA engine queue, stopping data transfers in other streams from executing. To prevent this, one would have to insert an additional polling call for completion of kernel execution, and only then transfer data out of the device before scheduling another polling call.

The CHARM++ GPU Manager is a library designed to address the above issues by automating the management of GPUs[14]. Users of GPU Manager define *work requests* which specify the GPU kernel and any data transfer operations required before and after completion of the kernel. The system controls the execution of the work requests submitted by all the chares on a particular processor. This allows it to effectively manage execution of work requests and overlap CPU-GPU data transfer with kernel execution. In steady-state operation, GPU Manager overlaps kernel execution of one work request with data transfer out of GPU memory for a preceding work request and the data transfer into GPU memory for a subsequent work request. This approach avoids blocking the DMA engine by only submitting data transfers when they are ready to execute. When using GPU Manager, the user does not need to poll for completion of GPU operations. The system manages execution of a work request throughout its life cycle and returns control to the user upon completion of a work request through a *callback object* specified by the user per work request. Another advantage of using GPU Manager is that the system polls only for a handful of currently executing operations, which avoids the problem of multiple chares all polling the GPU when using CUDA streams directly. GPU Manager has options for recording profiling data for kernel execution and data transfer which can be visualized using the CHARM++ Projections profiler.

IV. CHANGA

ChaNGa [15] (CHARM++ N-body Gravity Solver) is an iterative N-body simulator written in CHARM++. ChaNGa distinguishes itself from other codes through several production-quality features. These are essential for state-of-the-art cosmological simulations, and include canonical, comoving coordinates with a symplectic integrator to efficiently handle cosmological dynamics [16], individual and adaptive time-steps, periodic boundary conditions using Ewald summation, and Smooth Particle Hydrodynamics (SPH) [17] for adiabatic gas. The gravitational softening is consistent with the spline kernel softening used in SPH [18]. ChaNGa also uses quadrupole expansions which provide a more efficient force evaluation than monopole expansions at the force accuracies required for cosmological simulations [19]. Therefore, ChaNGa has many of the features of the widely used, state-of-the-art cosmology simulation codes, GASOLINE [20], and GADGET [21]. These features are in contrast to existing GPU implementations which typically only use monopole expansions, do not use periodic boundary conditions or comoving coordinates, and use a non-local gravitational softening. In addition, ChaNGa employs several optimizations, such as the prefetching of remote data, use of a software cache to reduce average access time of remote data, and prioritized execution to overlap requests for remote data with useful computation. These optimizations have enabled ChaNGa to scale to 32,768 cores [22].

Below, we describe the various phases of each iteration of ChaNGa. This will give some context to the optimizations discussed in § VI.

Domain decomposition. Particles are decomposed onto CHARM++ objects called *tree pieces* using one of many decomposition strategies. This operation is similar to a parallel sort of particles across all tree pieces. The tree pieces are assigned to processors by the CHARM++ runtime system. GPUs are excluded from this phase since there is not enough computation per particle to justify the transfer and kernel invocation costs associated with GPU use. In the strong scaling studies that we conduct, there are significantly fewer than a million particles per processor–less than the threshold amounts above which the best GPU algorithms outperform CPU-based sorting codes.

Tree construction. Once the particles have been partitioned among tree pieces, a distributed Barnes-Hut tree is constructed, with each tree piece holding a portion of the Barnes-Hut tree that is *local* to it. This phase is characterized by irregular memory accesses and very little computation per tree piece beyond the recursive summation of multipole moments of sibling nodes. Furthermore, ChaNGa employs a fine-grained algorithm for tree construction, where the latency of exchange of shared node information is overlapped with useful work. Given this issue of grain size, it is more beneficial to perform tree construction on the CPUs.

Tree traversal. The computation of gravitational forces is preceded by a traversal of the distributed Barnes-Hut tree by each tree piece. In ChaNGa, the cost of traversing the tree is amortized over several local particles by grouping them into *buckets*. These buckets form the leaves of the Barnes-Hut tree. For each bucket of local particles (the *target* bucket), an *interaction list* is obtained, listing the nodes and particles that act as *sources* of gravitational force on it. Each tree piece performs this traversal in two parts. The *local* traversal is conducted on that portion of the Barnes-Hut tree that is local to the tree piece (i.e. its local tree). The *remote* traversal operates on the remainder of the distributed Barnes-Hut tree, leading to communication between tree pieces in the form of requests for remote nodes and particles. Since communication cannot be initiated by the GPU directly, assigning this task to it would entail repeated memory transfers between the CPU and the GPU. Due to this issue, tree traversal is performed on the CPU.

Force computation. Gravitational forces may be computed in parallel across all particles. In fact, each interaction of the particle with a tree node (a *particle-node* interaction) or another particle (a *particle-particle* interaction) may be computed in parallel with all others. Moreover, the gravitational force calculation routines exhibit a high intensity of floating point operations. These factors make force calculation an ideal candidate for execution on the GPU. In order to ensure a large enough grain size for gravity computation kernels on the GPU, ChaNGa was enhanced with a *work agglomeration module*. This module collates the interaction lists of multiple buckets into a single *work request* (WR), which is then transferred to the GPU for execution.
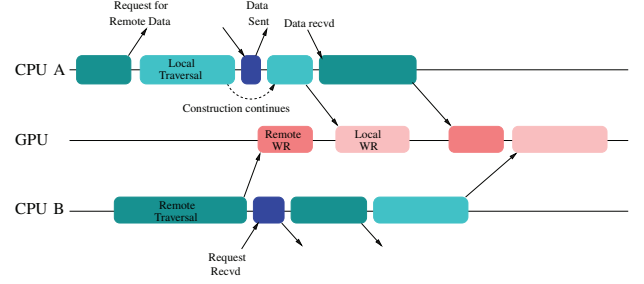

Fig. 1. Division of tasks between the CPU and the GPU. Here, two CPUs share a GPU. Remote and local tree traversals are performed on the CPU to construct lists of interactions. These are offloaded to the GPU for computation of forces. CPU and GPU work can be overlapped.

Ewald summation. Forces in simulations with periodic boundary conditions are handled using the Ewald summation technique in a manner similar to the reduced cell multipole method of Ding *et al.* [23]. For a given particle we first calculate the direct non-periodic forces due to all particles in the fundamental cube (i.e. the simulated universe) and a number of periodic replicas, usually the 26 neighbors. We then calculate the forces due to the Ewald sum of the multipole moments of the root cell of the fundamental cube. Our approach differs from that work in that we explicitly calculate the Ewald sum of the multipole moments rather than representing them with a small number of particles. The Ewald calculation involves a sum over nearby replicas and a sum over Fourier terms. The nearby replica terms are modified to exclude the forces which were included in the direct calculation. Since these summations only depend on the multipole moments of the root cell, no communication is needed for this part of the force [24].

We offload all Ewald summation onto the GPU. The organization of the algorithm into a real space component and a Fourier space component suggests a division of the GPU work into two kernels. As the two phases of the calculation use different data structures, placing them into separate kernels decreases the register usage per thread and allows more threads to be present on the GPU at the same time. To further reduce register pressure, we used constant memory to store a set of values required during the execution of the real space component of Ewald summation. For the Fourier space kernel, we used constant memory to store a large table of values precomputed on the CPU. Since in both cases the values accessed in constant memory were required by all threads within a warp at the same time, the broadcast capabilities of the constant cache were fully utilized. Both kernels further benefited from the use of fast GPU implementations of math functions.

Figure 1 illustrates the execution of various tasks on the CPU and the GPU. The figure shows the timelines of two CPU cores offloading multiple work requests to a single GPU. Also shown is activity related to remote and local traversals (aquamarine and light blue colored bars, respectively) and the exchange of node and particle data between processors (dark blue bars.) During their remote traversals, CPUs A and B send requests for particle and node data to each other. It can also be

seen that the traversals of the two processors have intervening dark blue boxes. This represents the periodic suspension of traversal work in order to satisfy any pending requests for data from remote processors. Once a threshold number of interactions has been accrued during the traversal, the CPU sends the GPU a work request. A work request has a name that corresponds to the type of traversal that generated it. For instance, a local traversal generates local work requests. Notice that the division of computation work into work requests allows overlap between traversal work on the CPU and work request execution on the GPU. Since the size of each WR can be controlled, this approach also provides an effective way to limit the amount of GPU memory used in computing forces.

During the course of a CPU-only simulation, most of the time is spent in force computation. In particular, preliminary studies showed that more than 90% of the time was spent in gravity computation on several benchmarks, even when using up to 1,024 processors. In our case, this time is consumed in the Barnes-Hut algorithm. Therefore, in the remainder of this paper we will consider only this phase of the simulation.

V. EXPERIMENTAL SETUP

The experiments and performance tests described in the remainder of the paper use various astronomical data sets of varying properties and size:

cube300. A low resolution simulation of a cosmological cube 300 Mpc on a side with 30% dark matter and 70% dark energy. Only 110,592 particles are used. The particles are moderately clustered on small scales, becoming more uniform on larger scales.

lambs. Final state of a simulation of a $71 Mpc^3$ volume of the Universe with 30% dark matter and 70% dark energy. Nearly three million particles are used. This dataset is highly clustered on scales less than 5 Mpc, but becomes uniform on scales approaching the total volume. Three subsets of this data set are obtained by taking random subsamples of size thirty thousand, three hundred thousand, and one million particles, respectively.

hrwh_lcdms. Final state of a $90 Mpc^3$ volume of the Universe with 31% dark matter and 69% dark energy realized with 16 million particles. This dataset is used in [25], and is slightly more uniform than *lambs*.

lambb. Same physical model as *lambs* except that it is realized with 80 million particles.

These data sets are small compared to state of the art cosmological N-body simulations currently being performed. In the experiments below we focus on the strong scaling of these data sets rather than weak scaling to larger data sets for a couple of reasons. First, many astrophysical problems, e.g. globular clusters or nuclear regions of galaxies, involve solving an N-body system for 10,000 dynamical times requiring millions of time steps. Hence data sets where time steps can be completed in an order of a second are needed. Second, because communication overheads become more significant as the work per processor decreases, and the amount of communication only grows logarithmically with problem size for the tree algorithm, strong scaling is harder to achieve than
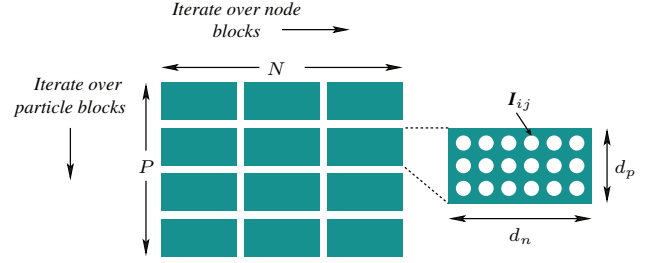

Fig. 2. Pictorial depiction of the organization of force computation kernel.

weak scaling. Furthermore strong scaling for a small problem on a modest number of processors will indicate how well the code will perform with large problems on the 100,000 core machines expected in the near future.

Experiments were performed on two machines. The single core, single GPU tests were conducted on a quad-core workstation equipped with an Intel Core 2 Quad Q6600 processor running at 2.4 GHz, augmented with an NVIDIA GeForce 8800 GTS card comprising 128 streaming processors. Strong scaling results were obtained on the NCSA's Lincoln GPU cluster, which comprises 192 Dell PowerEdge 1950 servers, each hosting dual Intel Core 2 Quad (Harpertown) 2.33 GHz processors with access to a total of 16 GB of memory. An NVIDIA S1070 Computing System is shared between two nodes, resulting in a total of 8 cores and 2 GPUs per node. Nodes are connected via an Infiniband SDR network. OS interference and system noise prevented the use of all 8 cores per node: our experiments were conducted with only 7 CHARM++ processes instantiated per node. This strategy leaves one core per node free for the execution of OS daemons and periodic OS tasks, yielding better performance than using all 8 of the cores available per node.

VI. TUNING GPU PERFORMANCE

Whereas the promise of massive amounts of hardware parallelism provides a compelling case for their adoption, the efficient use of GPUs in sophisticated variants of the Barnes-Hut procedure requires the investigation of several performance issues and significant development effort. In this section, we study the key performance characteristics of these simulations. Further, we quantify the benefits yielded by careful consideration of application parameters through experiments done with the ChaNGa N-body simulator.

A. Kernel Organization

A scalable parallel simulator spends most of its time in force computation routines. Therefore, adapting the structure of the force computation kernels is key to ensuring good performance. Consider the calculation of forces on a bucket of P particles due to a list of N nodes. Interactions between a particle p_i and a node n_j can be represented as elements of an *interaction matrix* $\mathbf{I}$, such that each interaction $\mathbf{I}_{ij}$ denotes the calculation of the force on *target* p_i due to *source* n_j. Each $\mathbf{I}_{ij}$ can be computed in parallel. However, the number of concurrent threads available per block is typically much smaller than the total number of interactions: values for N
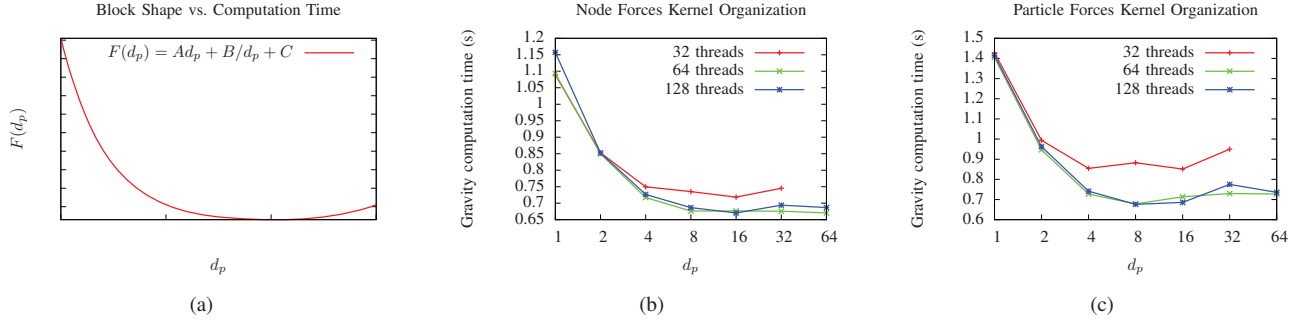
Fig. 3. Variation of computation time with kernel layout: 3(a) models this time whereas 3(b) and 3(c) show empirical results.

are typically in the thousands, and P is on the order of tens of particles. Therefore, $\mathbf{I}$ is divided into individual tiles that are executed one-at-a-time by a block of threads of size $T < NP$. A block of threads is organized logically as a rectangle of length $d_n \ll N$ and breadth $d_p < P$, with the condition that $d_p \times d_n = T$, i.e. $d_n = T/d_p$. This arrangement is depicted in Figure 2. Threads in row i operate upon the data of particle p_i, whereas those in column j share the multipole expansions of node n_j. A single thread in row i (respectively, column j) issues the load of particle p_i (node n_j) into shared memory. These threads are called *row*- and *column-leaders*, respectively.

Furthermore, stores to global memory are limited by computing partial forces on a batch of d_p particles until all node interactions are exhausted. Then, each row-leader accumulates the acceleration of its corresponding particle and updates it in global memory. This yields an optimal number of stores (exactly P) to global memory. However, loads of nodes must now be repeated across different batches of particles. If the size of a particle's coordinate information is s_p bytes and that of a node's multipole moments is s_m bytes, the total number of bytes loaded from global memory is:

$$L_{total} = Ns_m \lceil P/d_p \rceil + Ps_p$$

It may appear that performance can be maximized by increasing d_p at the expense of d_n. However, increasing d_p can have a negative effect on the performance of the gravitational force kernels, as explained below. If s_v is the size of the data structure that stores a particle's (variable) acceleration and potential, the total amount of shared memory required per block for this tiling scheme, M_{total}, is given by:

$$M_{total} = Ts_v + Ts_m/d_p + s_pd_p$$

The greater the value of M_{total}, the fewer the blocks that can be assigned for parallel execution to each multiprocessor of the GPU. Therefore, the amount of hardware concurrency utilized actually diminishes with the increase of d_p. To estimate the combined effect of these forces, we model force computation time F as a linear combination of L_{total} and M_{total}. The shape of this curve for constant T is depicted in Figure 3(a). In modeling force computation time as $F(d_p) = Ad_p + B/d_p + C$, we use placeholder constants A, B and C instead of actual values obtained by curve-fitting, since we only wish to obtain

an idea of the variation of execution time with d_p. The minimal value for computation time was obtained empirically, as detailed later. The graph illustrates that the minimum occurs at an intermediate value of d_p rather than at either extreme. For small values of d_p, accesses of node moment information account for much of the execution time. Since M_{total} grows linearly with d_p, it dominates execution time for large d_p. To ascertain the optimal value of d_p, we did experiments on the *cube300* data set on a single CPU core with one GPU. Our conjectures about the variation of execution time with d_p were borne out for the various values of T examined. Figures 3(b) and 3(c) show the variation of execution time with d_p. The expected trough is visible for different T values. The best performance was achieved with a total of $T = 128$ threads per block, and dimensions $d_p = 16$, $d_n = 8$.

B. Balancing Tree Traversal and Force Computation

There are two main parts to the calculation of forces in the Barnes-Hut procedure. The first is the traversal of the global tree to accumulate a list of interactions. Recall that these lists are accrued for buckets of particles. Then, there is the calculation of forces using the interaction lists. The amount of time spent in either routine can be controlled by tuning the average size of a bucket. Increasing the average size of a bucket reduces the number of leaves in the global tree, thereby reducing the number of internal nodes in it. This reduces the time spent walking the tree. On the other hand, increasing the bucket size can raise the amount of computation performed for the following reason. Let r_o be the *opening radius* of node n. This radius is proportional to the size of n, and describes a sphere S centered at its center of mass. If the bounding box of a bucket b intersects with S, n must be *opened*, i.e. its children must be evaluated when computing forces for particles in b. This has a dual effect: (1) since bounding boxes of buckets are bigger, b is more likely to intersect with S thus forcing the opening of n; (2) since buckets are located higher in the tree, when opening n we could find that it is a bucket, and therefore we would need to compute directly with all its particles. This increases the total number of interactions recorded, as shown in Figure 4(a). Note that although the *total* number of interactions increases, the number of particle-node interactions remains roughly the same. This is because while effect (1) increases the number of particle-node interactions,
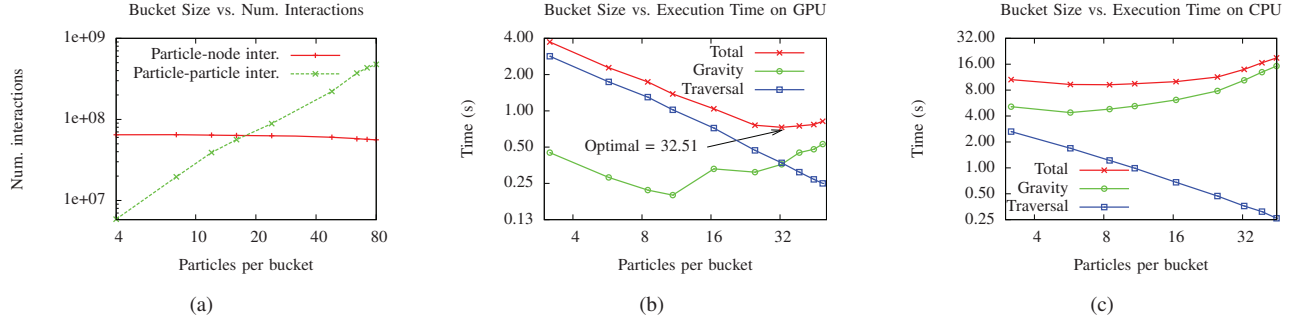
Fig. 4. Single-GPU runs on the *cube300* (100k particles) data set; 4(a): total number of interactions increases rapidly with bucket size; 4(b): tree traversal time decreases, and GPU computation time increases with larger buckets; 4(c): a similar effect is observed on the CPU.

effect (2) decreases it.

The tradeoff engendered by the average bucket size is depicted in Figure 4(b). The experiments presented were performed on the *cube300* data set on a single core with a single GPU. It must be noted that the tests were performed with no overlap between CPU work (tree traversal) and GPU work (force evaluation). Computations meant for the GPU were buffered until the tree was traversed completely for all buckets. This helped isolate the issue of work balance from that of overlap between the CPU and GPU (discussed in § VI-C.) In the graph, the time taken to traverse the Barnes-Hut tree is shown by the blue curve. It was obtained by switching off force calculation in ChaNGa. The green curve was generated by measuring the total time taken on the GPU for calculation of forces due to both particle-particle and particle-node interactions. The red curve depicts the total time taken to complete all Barnes-Hut tasks (dominated by the sum of CPU tree traversal and gravitational force calculation on the GPU.) For small buckets, tree traversal accounts for most of the execution time. This changes with an increase in bucket size. Time taken by the GPU to calculate forces decreases initially due to fewer, more efficient thread blocks. Total time falls until the optimal value for average bucket size, determined to be 32.51. Past this point, the increasing cost of extra particle-particle computation dominates and total computation time rises. On Lincoln, an average bucket size of 44.92 worked best across the range of data sets. The Ewald computation was independent of bucket size and required 120 ms for completion.

Figure 4(c) depicts the effect of bucket size on CPU execution time. As before, tree traversal time decreases with an increase in bucket size. Force calculation time also decreases until 5.7 particles per bucket. At this point, the extra computation due to larger buckets hindered performance. Ewald computation took a constant time of about 3.2 seconds, implying that GPU use sped up the operation by 25 times.

C. Overlapping CPU and GPU Work

Using the GPU as an acceleration co-processor in the described fashion, the only tasks designated to it are the computation of forces on particles due to lists of interactions and the Ewald correction. Tasks such as tree traversal and the construction and serialization of interaction lists are consigned

to the CPU. To allow the overlap of CPU work with that done on the GPU, the entire space of interactions is split into individual work requests. As noted in § IV, work requests (WRs) are created by the CPU and specify the sources of gravitational force (tree nodes or particles) with which target particles interact. Work requests are self-contained and independent of each other. Therefore, the construction of interaction lists for one request can be overlapped with the computation of forces with lists from a previous one.

However, the division of work into individual requests comes at a cost. The transfer of a WR incurs the overheads of serialization and memory transfer. An optimal balance of work between the CPU and GPU ensures that the work done by one is overlapped with that performed by the other, without incurring a prohibitive penalty for this division of computation work into WRs.

Figure 5 presents the variation in computation time with sizes of the key types of work request. The parameters studied are the sizes of local node (LN), remote node (RN) and local particle (LP) computation requests. Tests were conducted with the *lambs* data set on 14 CPU cores with 4 GPUs and 112 cores with 32 GPUs, to examine the effects of these parameters at different grain sizes. The results presented in Figures 5(b) and 5(c) respectively indicate that the effects on computation time are similar at both scales, but less pronounced with finer grain sizes.

Consider the 14 core, coarse-grained case shown in Figure 5(b). The red curve illustrates the effect of the size of local node (LN) requests on computation time. For this curve, LN was varied whereas RN and LP were held constant. Observe that computation time increases with a decrease in the WR size (i.e. more WRs.) The reason for this is as follows. In ChaNGa, computation on local nodes and particles (termed *local work*) is assigned a lower priority than computation that involves particles and nodes received from remote processors (so called *remote work*). Therefore, remote work dominates the initial part of an iteration, and the latency of requests for remote data that are generated during the remote work is overlapped with useful local work. This strategy works well for the CPU-only version. However, in the CPU/GPU version, the CPU must serialize the data required for each WR. In particular, the serialization of local WRs generated to mask remote data access latencies delays the CPU's response to
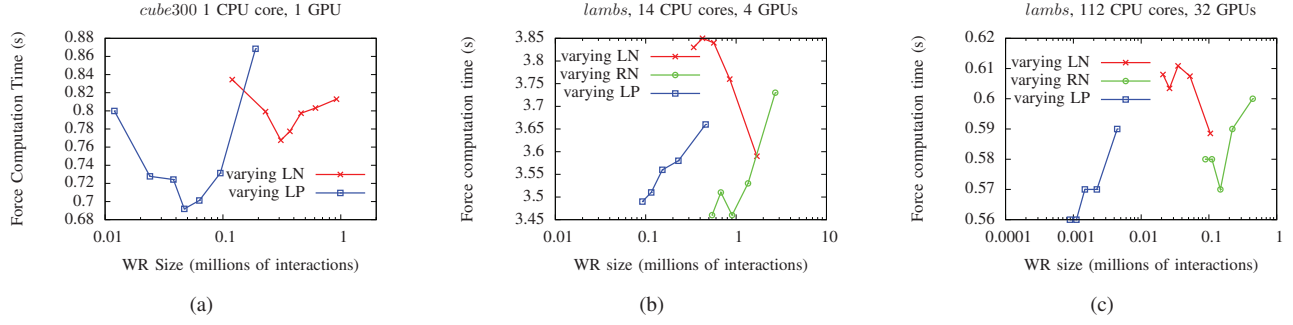
Fig. 5. WR sizes affect the overlap of tree traversal on CPUs with force computation on GPUs. The effect is greater with more particles per core.

requests from other processors. This holds up the processors requesting data, leading to a drop in performance. The effects of this serialization overhead become more prominent with an increase in the number of local WRs (corresponding to smaller LN values.) The red curve shows that up to a 7% increase in performance can result from the use of large LN values.

The green curve shows the effect of varying the size of remote node (RN) WRs. In this case, LN and LP are kept constant. Large values of RN hinder performance, since they delay the offload of WRs to the GPU, causing it to idle during the early portion of the iteration. We were able to reduce computation time by more than 7% by setting RN appropriately.

Lastly, the blue curve demonstrates the variation in computation time with changes in the size of local particle WRs (LP). In this set of experiments, LN and RN were kept constant. As can be seen, LP does not affect computation time significantly in the two *lambs* experiments, since node computations (both local and remote) dominate execution time. To obtain a more comprehensive assessment, we studied the effect of these parameters in *cube300* simulations on a single CPU core accompanied by a single GPU, completely eliminating remote work. The results of these tests are shown in Figure 5(a). Particle computations accounted for about 70%, and node computations about 20% of GPU time, so that the effect of LP on execution time was more marked than that of LN. The tradeoff between increased overlap due to *smaller* WR sizes and greater offload efficiency due to *larger* WR sizes is clearly demonstrated. The optimal values for LP and LN were found to increase performance by 20% and 10%, respectively.

D. Reducing Serial Overheads

In this subsection, we estimate a lower bound on the execution time achieved by dividing tasks related to the Barnes-Hut procedure in the way described. We then compare the scaling performance of ChaNGa to the ideal scaling profile and study the reasons for practical discrepancies between the two. This demonstrates the overheads of GPU use.

Let T_{cpu}^t denote the time taken for CPU tree traversal and T_{gpu}^f, that to calculate forces on the GPU. In addition to these, the simulation incurs an overhead T_{gpu}^{ovhd} due to the serialization of interaction lists and their transfer to the GPU. When there is overlap between CPU and GPU work, the total time taken to compute gravitational forces using GPUs is: $T_{gpu} = max(T_{cpu}^t, T_{gpu}^f) + T_{gpu}^{ovhd}$. To obtain an upper bound on the speedup, we assume that the GPU is not the bottleneck, and that CPU work and that done on the GPU are overlapped perfectly. Therefore, the time taken to complete gravity calculation by offloading force computations to GPUs with full efficiency would be $T_{gpu}^* = max(T_{cpu}^t, T_{gpu}^f) = T_{cpu}^t$, and the overhead of GPU use is given by the difference $T_{gpu}^{ovhd} = T_{gpu} - T_{cpu}^t$.

Experiments were performed to determine this overhead for ChaNGa on the *lambs* (3 million particles) data set. Results are shown in Figure 6. The orange curve with upward-facing triangles shows the scaling of the CPU version of ChaNGa ('*lambs*-CPU'). The green curve with circles ('*lambs*-GPU') shows the scaling of our first attempt at a CPU/GPU version. It can be seen that this CPU/GPU version only provided speedups over the CPU-only version of ChaNGa at lower core counts. In fact, barely any improvement in execution time was obtained past 112 cores (and 32 GPUs.) The use of GPUs was detrimental at 448 cores.

We studied the breakup of various tasks in ChaNGa to identify obstacles to scaling. The amount of computation offloaded to each GPU decreases as the number of GPUs increases. Therefore, force computation on the GPU is unlikely to be the bottleneck to scaling. Recall that the tasks performed by the CPU include the traversal of the Barnes-Hut tree and the construction of interaction lists to be shipped to the GPU. Even though the proportion of locally-available parts of the global tree diminishes with an increase in cores [15], tree traversal time ($T_{cpu}^t = T_{gpu}^*$) scales reasonably with the number of cores. This is because the number of buckets for which each core must traverse the tree falls with an increase in the total number of cores. The time to traverse the Barnes-Hut tree is depicted by the blue curve labeled 'Traversal' in Figure 6. It decreases with the number of cores, and therefore doesn't add the growing overhead that is characteristic of the green *lambs*-GPU curve.

Next, we tested the amount of time taken to transfer lists of interactions to the GPU. The asynchronous transfer of these lists to the GPU involves the use of a special allocator called `cudaMallocHost`. This function returns a buffer of page-locked memory of specified size. It was observed that the cost of using `cudaMallocHost` and its counterpart to free
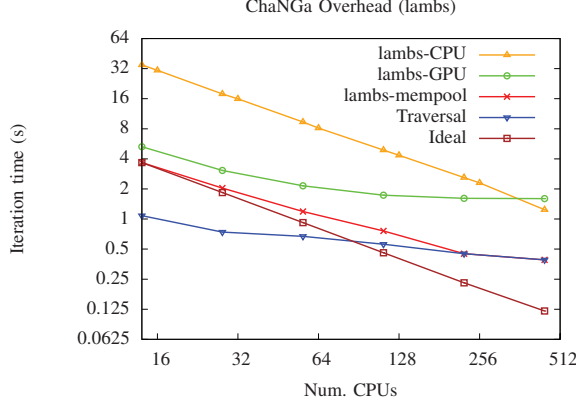
Fig. 6. Overheads in parallel simulations of the *lambs* data set.

Fig. 7. A comparison of the performance of CPU-only and CPU/GPU versions of ChaNGa.

allocated lists of interactions, `cudaFreeHost`, dominated as we scaled to larger core counts. The actual cost of these operations comes from the fact that requests to allocate and free buffers of page-locked memory result in communication between the CPU and GPU. Moreover, if a GPU is busy at the time of issue of such a request, the CPU stalls until the GPU becomes idle, at which point, presumably, the GPU is available to participate in the memory request.

To overcome this overhead, a scheme was devised to fulfil requests for page-locked memory from a pool of preallocated buffers. This reduces the time taken to fulfil memory requests considerably, since all communication between the CPU and the GPU is done at startup, and the allocation and freeing of buffers involve straightforward pointer manipulations. The performance of the version of ChaNGa that uses this memory allocator is presented in Figure 6 by the red curve marked *lambs*-mempool. As can be seen, the curve in red scales far beyond the one in green. This behavior was seen across all data sets considered. Therefore, savings due to the use of pooled memory are key to strong scaling performance.

Observe that the difference between the total time taken by the *lambs*-mempool version (red) and the tree traversal time (in blue, equivalent to T_{gpu}^*) diminishes as the number of processing elements increases. In fact, at 224 cores/64 GPUs and beyond, there is so little force calculation work per GPU that the time taken by CPUs to perform tree traversal and list construction dominates. Figure 6 also shows the ideal performance relative to 14 cores and 4 GPUs (brown curve marked 'Ideal'.) Notice that the execution times depicted by the *lambs*-mempool curve are close to the ideal values up to 56 CPUS/16 GPUs, indicating an efficient implementation.

VII. RESULTS

We now present results from tests conducted to assess the performance of the GPU version of ChaNGa under strong scaling conditions on the Lincoln GPU cluster. Figure 7 compares the performance of the CPU-only and CPU/GPU versions of ChaNGa on the *lambs*, *hrwh_LCDMs*, and *lambb* datasets. These are abbreviated as 3m, 16m and 80m, respectively. The performance of the CPU-only version with the 80m,16m and
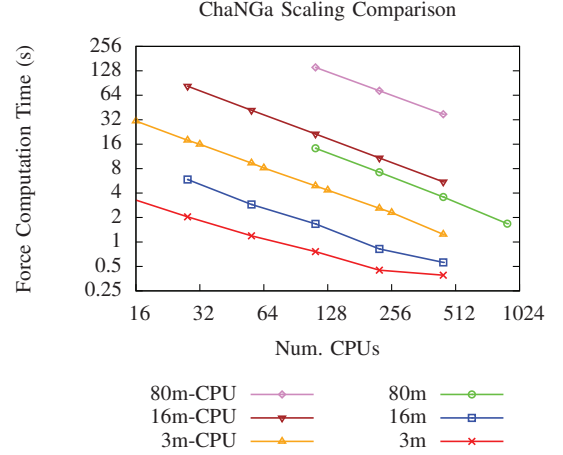
3m data sets is shown by curves marked 80m-CPU, 16m-CPU and 3m-CPU, respectively. It is noteworthy that the CPU-only version of ChaNGa has been carefully optimized and uses SIMD instructions to achieve good performance [22]. Table I compares performance of the two versions of ChaNGa in a textual format. Speedups are listed for the various data sets in columns marked S_n, which is the ratio of the time taken to compute forces on n CPU cores and the time taken on n cores together with $2n/7$ GPUs. The factor of $2/7$ comes from the fact that 7 cores per Lincoln node are used in our experiments, i.e. $n/7$ nodes, with 2 GPUs per node. The table also lists performance in terms of single precision Gflop/s. This measure was obtained by dividing the total number of floating point operations performed during force computation by the time taken to complete this procedure. This includes the time spent in traversing the tree and constructing lists, communication of remote data between processors and various overheads associated with the use of GPUs.

The smallest of the data sets, 3m, exhibits a good scaling profile up to 224 cores/64 GPUs. Efficiency suffers at 448 cores/128 GPUs, but the simulation still scales up to that point. By using 4 GPUs in addition to 14 CPUs, we were able to speed up simulation times by 9.5 times. This speedup drops with increasing numbers of cores and GPUs, as the amount of work done per GPU falls. The simulation exhibited an average rate of 538 Gflops/s with 448 cores/128 GPUs.

The 16m data set has a more uniform distribution of particles than the 3m and 80m systems. This reduces the amount of load imbalance across processors, so that greater speedups may be obtained over the CPU-only version. Values of S_n for this data set remain appreciable across the range of processors. We were able to obtain a speedup of about 14 over the CPU-only version at lower core and GPU counts, and 9.82 at 448 cores/128 GPUs. With this configuration, the simulation maintained an average rate of 1.79 Tflop/s.

Finally, the largest of the systems studied, *lambb*, demonstrated excellent scaling up to 896 cores/256 GPUs. Whereas superlinear speedups are obtained when scaling from 448

CPUs/GPUs	3m		16m		80m	
	S_n	GF	S_n	GF	S_n	GF
14/4	9.5	57.17				
28/8	8.75	102.84	14.14	176.43		
56/16	7.87	176.31	14.43	357.11		
112/32	6.45	276.06	12.78	620.14	9.92	450.32
224/64	5.78	466.23	13.21	1262.96	10.07	888.79
448/128	3.18	537.96	9.82	1849.34	10.47	1794.06
896/256					-	3819.69

TABLE I

SPEEDUPS (S_n) AND SINGLE PRECISION GFLOPS/S (GF).

cores/128 GPUs to 896 cores/256 GPUs, this is likely an artifact of the values selected for WR sizes with the 448 core/128 GPU configuration. Once again, our implementation yielded a 9.92-10x speedup over the CPU-only version on up to 448 cores/128 GPUs. The simulation sustained an average of 3.82 Tflop/s on 896 cores/256 GPUs. However, we were unable to conduct tests on an 896-core, CPU-only version, which is why the corresponding S_n value is left blank. If we were to extrapolate performance of the CPU-only version by assuming no loss in parallel efficiency upon doubling the number of processors from 224 cores, we would still obtain a speedup of 11.15 times with the CPU/GPU version.

A. Considerations for Multistepped Simulations

Modern gravitational simulators must deal with particle systems that exhibit vast variations in particle density, and consequently, high dynamic ranges of timescales. In *multistepped* simulations, this feature of self-gravitating systems is exploited to obtain large algorithmic gains in performance. Instead of computing forces on all particles at each time step, forces are evaluated at every time step only on the particles present in the most dense regions. The forces on particles in less dense regions are updated less frequently, and only occasionally are the forces computed for all particles in the system. This force evaluation scheme can lead to dramatic reductions in computational cost, while maintaining accuracy of simulation.

In GPU-supported multistepped simulations, data must be transferred to and from the GPU at each time step. These data include the coordinate and multipole information of both the *target* particles, on which forces are evaluated, and *source* particles, which contribute to the forces incident on the target particles. Note that the former constitute a subset of the latter. For time steps in which there are relatively few target particles, the memory transfer and GPU kernel startup overheads dominate execution time. These overheads are amortized over a larger number of particles in time steps where there are more target particles.

This effect is illustrated in Figure 8, which depicts performance of the CPU-only (blue) and CPU/GPU version (red) on a multistepped simulation of the *cube300* data set on a single processor. The amount of computation per time step varies with the number of target particles. Therefore, the computational power of the GPU compensates for the cost of CPU-GPU transfers and kernel invocation only beyond 250-500 target particles per processor. Therefore, it is more
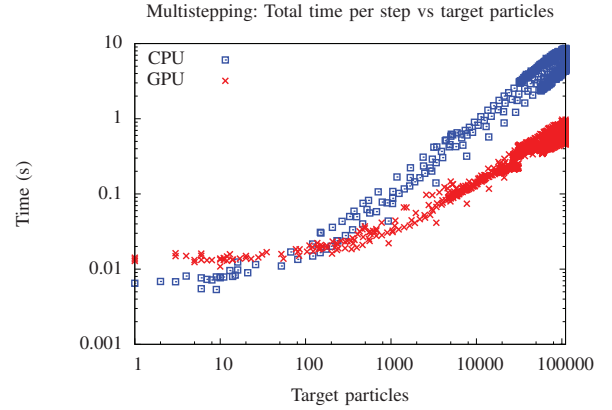

Fig. 8. Time per step measured as a function of number of target particles. The use of the GPU for as an offload device is feasible only in time steps with more than 250-500 target particles.

efficient to conduct tree traversal *as well as* force calculation on the CPU for time steps that have fewer than 500 target particles per processor.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we studied various application characteristics of Barnes-Hut simulations and their impact on the scaling performance on clusters of GPUs. The key characteristics identified were optimal kernel organization, favorable balance of work between the CPU and GPU, the overlap of tree traversal on the CPU with force routine execution on the GPU, and the removal of serial bottlenecks such as page-locked memory allocation requests. Furthermore, we provided a lower bound on execution time given the division of concerns between CPU and GPU. We incorporated these optimizations into a production-quality simulator called ChaNGa and performed strong-scaling tests on realistic data sets, demonstrating good scaling results.

Future work will focus on the adaptation of SPH and FMM methods to the GPU. We will also explore the design of a pipelined GPU tree traversal to alleviate the CPU bottleneck at scale. Finally, we leave the study of load imbalance in GPU-based multistepped simulations to future work.

REFERENCES

[1] M. Warren and J. K. Salmon, "Astrophysical n-body simulations using hierarchical tree data structures," in *In Proceedings of Supercomputing*, 1992, pp. 570–576.

[2] A. Grama, V. Kumar, and A. Sameh, "Scalable parallel formulations of the barnes-hut method for n-body simulations," in *In Proceedings of Supercomputing '94*, 1994, pp. 439–448.

[3] A. Kawai, T. Fukushige, and J. Makino, "$7.0/Mflops astrophysical *N*-body simulation with treecode on GRAPE-5," in *SC '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 1999.

[4] J. Makino, E. Kokubo, and T. Fukushige, "Performance evaluation and tuning of grape-6 - towards 40 "real" tflops," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 2.

[5] M. H. L. Nyland and J. Prins, *Fast N-body simulation with CUDA*. Addison Wesley, 2007, pp. 677–695.

[6] R. G. Belleman, J. Bdorf, and S. F. P. Zwart, "High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda," *New Astronomy*, vol. 13, no. 2, pp. 103 – 112, 2008.

[7] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[8] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[9] D. Aubert and R. Teyssier, "Reionization simulations powered by GPUs I: the structure of the Ultraviolet radiation field," *ArXiv e-prints*, Apr. 2010.

[10] NVIDIA, *CUDA 2.0 Programming Guide*. NVIDIA Corporation, Santa Clara, CA, USA, 2008.

[11] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[12] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[13] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to gpu-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[14] L. Wesolowski, "An application programming interface for general purpose graphics processing units in an asynchronous runtime system," Master's thesis, Dept. of Computer Science, University of Illinois, 2008, http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[15] F. Gioachin, A. Sharma, S. Chakravorty, C. Mendes, L. V. Kale, and T. R. Quinn, "Scalable cosmology simulations on parallel machines," in *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[16] T. Quinn, N. Katz, J. Stadel, and G. Lake, "Time stepping N-body simulations," *ArXiv Astrophysics e-prints*, Oct. 1997.

[17] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics - Theory and application to non-spherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, Nov. 1977.

[18] J. J. Monaghan and J. C. Lattanzio, "A refined particle method for astrophysical problems," *Astronomy and Astrophysics*, vol. 149, pp. 135–143, Aug. 1985.

[19] C. Power, J. F. Navarro, A. Jenkins, C. S. Frenk, S. D. M. White, V. Springel, J. Stadel, and T. Quinn, "The inner structure of ΛCDM haloes - I. A numerical convergence study," *Monthly Notices of the Royal Astronomical Society*, vol. 338, pp. 14–34, Jan. 2003.

[20] J. W. Wadsley, J. Stadel, and T. Quinn, "Gasoline: a flexible, parallel implementation of TreeSPH," *New Astronomy*, vol. 9, pp. 137–158, Feb. 2004.

[21] V. Springel, "The cosmological simulation code GADGET-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, pp. 1105–1134, Dec. 2005.

[22] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[23] H.-Q. Ding, N. Karasawa, and W. A. G. III, "Atomic level simulations on a million particles: The cell multipole method for coulomb and london nonbond interactions," *The Journal of Chemical Physics*, vol. 97, no. 6, pp. 4309–4315, 1992. [Online]. Available: http://link.aip.org/link/?JCP/97/4309/1

[24] J. G. Stadel, "Cosmological N-body Simulations and their Analysis," Ph.D. dissertation, Department of Astronomy, University of Washington, March 2001.

[25] K. Heitmann, P. M. Ricker, M. S. Warren, and S. Habib, "Robustness of Cosmological Simulations. I. Large-Scale Structure," *ApJSup*, vol. 160, pp. 28–58, Sep. 2005.

[26] C. Catlett and *et al.*, "TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications," in *HPC and Grids in Action*, L. Grandinetti, Ed. Amsterdam: IOS Press, 2007.