Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters

James C. Phillips,[*‡] John E. Stone,[*‡] and Klaus Schulten[†§]
[*]Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL, 61801
[†]Department of Physics, University of Illinois at Urbana-Champaign, Urbana, IL, 61801
[‡]The authors contributed equally. [§]To whom correspondence should be addressed.

Abstract—**Graphics processing units (GPUs) have become an attractive option for accelerating scientific computations as a result of advances in the performance and flexibility of GPU hardware, and due to the availability of GPU software development tools targeting general purpose and scientific computation. However, effective use of GPUs in clusters presents a number of application development and system integration challenges. We describe strategies for the decomposition and scheduling of computation among CPU cores and GPUs, and techniques for overlapping communication and CPU computation with GPU kernel execution. We report the adaptation of these techniques to NAMD, a widely-used parallel molecular dynamics simulation package, and present performance results for a 64-core 64-GPU cluster.**

I. Introduction

GPU-accelerated algorithms have demonstrated speedups of 10- to 100-fold, resulting in significant improvements to overall application performance, including several recent applications in molecular modeling [1], [2], [3], [4], [5]. Such impressive performance gains present a software development challenge in that it is seldom practical and never desirable to completely restructure a major legacy application for a novel technology, no matter how promising. While the role of the GPU as an adjunct to a traditional processor allows the progressive porting of the most expensive operations, the beast must be fed work in large chunks to amortize startup overhead. Furthermore, a successful acceleration effort for a parallel application will place greater strain on communication, comparable to dropping a generation back in interconnect technology.

In this paper, we review GPU technology and the CUDA programming system and their impact on high-performance clusters, describe their application to the widely-used message-driven parallel molecular dynamics program NAMD, and report performance results and insights from a GPU-accelerated cluster.

A. GPU Hardware Overview

Graphics processing units have become a ubiquitous component of modern computers due to the increasing sophistication of video games and windowing systems. Modern GPUs are massively parallel programmable devices designed for extremely high computational throughput, achieved by multiplexing tens of thousands of threads onto hundreds of processing units—an ideal match to the needs of computer graphics workloads. Unlike mainstream processor architectures, GPUs dedicate very little die area to caches, instead spending the majority of their logic on arithmetic units and hiding memory latency with multithreading. GPUs use multiple SIMD processing cores to achieve high arithmetic unit density by reducing the area taken for control logic while also mitigating the impact of branch divergence. To keep arithmetic hardware productive, GPUs use multiple independent high-bandwidth memory systems working together in concert. GPU memory systems typically include a large high-latency global memory, a small cached constant memory, and a large spatially cached read-only "texture" memory with multidimensional addressing, hardware filtering and interpolation. The availability of additional features such as shared access register files, additional read/write memory caches, and support for atomic memory operations vary by GPU make and model.

AMD and NVIDIA have begun producing GPUs tailored for computing applications. State-of-the-art compute-oriented GPUs achieve aggregate floating point performance levels approaching one half teraflop, have on-board memory capacities of up to 2 GB, and bandwidth up to 100 GB/sec. Current GPUs support PCI Express 2.0 x16 host interfaces, providing bidirectional host-GPU communication bandwidths of up to 6.4 GB/sec. Many recent GPUs also provide hardware support for asynchronous DMA transfers, allowing applications to overlap GPU computations with host-GPU data transfers. These new compute-oriented GPUs have also been customized for use in data center and high performance computing environments where high reliability, thermal monitoring, fault detection, and high density rack mount form factors are a necessity. NVIDIA has begun producing external 1U rack mount accelerators, containing four GPUs and an independent power supply, that can be cabled to one or two host machines. While first generation GPU computing products were restricted to single precision floating point arithmetic, both AMD and NVIDIA are now shipping GPUs with double precision arithmetic and significantly increased floating point performance.

B. GPU Computing with CUDA

GPU software development tools have evolved rapidly in the past several years, closely tracking advances in GPU hardware architecture towards increased flexibility. The earliest applications of GPUs for computation were based on the use of existing graphics software interfaces for programmable shading. In these early efforts, computations were performed by "drawing" results to off-screen display buffers and reading them back to the host encoded in an output image. This model of GPU computing was soon superseded with higher level programming abstractions such as the stream programming model implemented in BrookGPU [6] and Sh [7]. Stream programming was a more appropriate abstraction for GPU-based computation than the existing graphics programming interfaces and was the basis for early GPU acceleration successes in computational biology [8], [9], [10]. Subsequent to BrookGPU, Sh, and other research efforts, several commercially supported general purpose GPU development toolkits have become available, in particular CUDA [11], RapidMind [12], and more recently Brook+ [13]. These development tools expose more of the GPU hardware capabilities, have expanded the range of supported GPU devices, provide CPU-based emulation and debugging, and some can also generate code for execution on multicore CPUs, decreasing the time required to develop kernels that run on both GPU and CPU hardware [14].

The work described in this paper is based on the CUDA GPU programming toolkit developed by NVIDIA. A full overview of the CUDA programming model is beyond the scope of this paper but it is worth mentioning a few of its key characteristics to improve the clarity of subsequent discussions. Nickolls et al. provide a detailed discussions of the CUDA programming model in [15]. A number of successful molecular modeling applications using CUDA are described in [1], [2], [3], [5]. The CUDA programming model decomposes work into *grids* of *thread blocks* that are concurrently executed by a pool of SIMD *multiprocessors*. Each thread block contains up to 512 threads, which are executed by the processing units within a single multiprocessor. Since each multiprocessor executes instructions in a SIMD fashion, each thread block is executed by running groups of threads, known as *warps*, in lockstep. Future CUDA-compatible GPUs may contain many times the number of multiprocessors in current GPUs. The virtualization of processing resources provided by the CUDA programming model allows applications written with existing GPUs to scale up with future hardware designs. Well-written CUDA programs should be able to run unmodified on future hardware, automatically making use of increased processing resources.

The release of CUDA 1.1 introduced new programming interfaces with improved support for asynchronous kernel execution and new features allowing host-GPU DMA operations to proceed concurrently with kernel execution on appropriate hardware. The CUDA 1.1 streaming API allows applications to queue host-GPU DMA operations and GPU kernel invocations into one or more "streams." As long as the stream queue is not full, control is immediately returned to the caller after operations are added to the queue. Operations in stream queues are processed one at a time in first-in-first-out order. Experiments with CUDA 1.1 and a range of contemporary device drivers revealed that the maximum number of operations that could be queued without blocking the CPU ranged from 16 to 24 depending on the model of GPU. This had turned up as a potential performance limiter in some of our previous work using asynchronous kernels [5]. Recently released device drivers (version 177.67) have increased the asynchronous queue size substantially to well over 100, completely eliminating blocking behavior for many common cases. Since operations within a stream are processed asynchronously, the host CPU is free to continue with other work while the GPU continues processing the active streams. Host code can explicitly synchronize with a stream and wait for completion or poll the status of a stream periodically without blocking. Asynchronous host-GPU DMA operations have one additional requirement that host buffers must be located in "pinned" memory regions that are marked non-pageable to the host operating system by the `cudaMallocHost()` routine.

In addition to the asynchronous streaming interfaces, CUDA 1.1 added an event recording API that applications can use to check the progress of individual operations within asynchronously executing streams. The event management interface allows event tags to be generated and inserted into a stream. As operations within the stream are processed, events inserted within the stream are recorded with timestamps. This information can be used to track execution progress in streams containing many operations, and to determine the execution time of individual operations within the stream. Timing information can subsequently be used to predict time of stream completion, to assist with scheduling additional work on the GPU. The asynchronous stream and event management interfaces provided in CUDA 1.1 address shortcomings of the original CUDA interface for message-driven parallel programs, and they form the basis for several of the key NAMD performance improvements that will be described below.

C. GPU Cluster Considerations

With the introduction of GPUs tailored for computing applications, it has become practical to begin building clusters containing GPUs for accelerating scientific computations. The earliest published work in this area began before compute-oriented GPUs and software development tools were available. Fan et al. describe a GPU-accelerated cluster that achieved an application speedup of 4.6 vs. a CPU cluster for a large parallel flow simulation based on a lattice Boltzmann model [16]. Since then, GPUs have evolved significantly, most notably with peak GPU GFLOP ratios relative to CPUs increasing substantially from the factor of 6.6 reported by Fan et al. in 2004 to over a factor of ten in early 2008. Current ratios of GPU memory bandwidth vs. CPUs are still close to the factor of ten they reported.

Recent work by Göddeke et al. explores the scalability, price/performance, power consumption, and compute density for FEM simulations running on the 160 node "DQ" GPU cluster at Los Alamos National Laboratory [17]. Using previous generation Quadro FX 1400 GPUs, Göddeke et al. doubled performance with only a 20% increase in power consumption relative to a CPU based cluster. They estimate a similarly beneficial performance/power ratio for a hardware configuration based on a Quadro FX 4500, yielding a tripling of performance with only a 30% increase in power consumption. Their results indicate a continuing trend of improved price/performance and performance/Watt ratios for GPU accelerated clusters.

The work by Fan et al. and Göddeke et al. predates the availability of compute-oriented GPU devices, multi-GPU-capable host systems, and recent software-based performance improvements and asynchronous execution capabilities that became available in CUDA 1.1. It is likely that they would achieve even better results with current generation GPU hardware and software. In the present work, we have leveraged these capabilities to achieve an approximate doubling in performance over our previous results [1], [2], as will be described further below.

While GPUs have been shown to provide worthwhile performance acceleration yielding benefits to both price/performance and performance/Watt, several areas of GPU hardware and software could still be improved in pursuit of even higher performance, higher reliability, and widespread adoption. One potential issue with deploying clusters consisting of multi-GPU nodes involves the interaction between node and processor assignment performed by batch queuing systems, and the selection and allocation of GPU devices by applications. At present, queuing systems and message passing infrastructure are unaware of the existence of GPUs and other accelerators, as is the host operating system. Applications cannot simply assume a one-to-one mapping of CPU cores to GPU devices unless the cluster was specifically built that way. Applications must determine for themselves how many GPUs to allocate, and users must select node/CPU core allocation parameters for their jobs to avoid allocating CPUs lacking a matching GPU that might otherwise sit idle during a run. This lack of GPU/accelerator awareness can pose a problem if unrelated jobs are executed on a node, as multiple processes may attempt to use the same GPUs. The addition of special "exclusive open" or device reservation APIs for GPU programming toolkits could alleviate some of these problems. In practice, these complexities can be avoided by designing GPU clusters with a simple one-to-one CPU core to GPU mapping, and only allowing allocation of entire nodes at a time.

Another complication involved in utilizing clusters with multiple GPUs per node involves the interaction between host-GPU DMA operations and hosts with non-uniform memory architecture (NUMA). None of the existing GPU programming toolkits provides a means of mapping GPUs, memory allocations, and CPU cores accounting for the topology of the PCI express buses and NUMA memory system. This is a concern since performance penalties occur for DMA operations to/from remote memory pools associated with a different CPU, as a result of multiple CPU interconnect traversals. Presently, applications must determine the correct assignment of GPUs to CPU cores, complicated by underlying batch queuing and message passing systems used to launch jobs.

A potential interoperability problem that exists with user-mode DMA mechanisms used by accelerator devices, and networking equipment involves the lack of standardization of software interfaces for "pinning" pages of virtual memory to physical pages. User-mode message passing libraries and accelerators use pinned memory to increase DMA performance by eliminating the need for intermediate buffers, or to pin and unpin regions of memory on-the-fly. The use of pinned memory buffers can allow a well-written code to achieve zero-copy message passing semantics via RDMA. The lack of a standard mechanism for managing virtual memory pinnings among user-mode accelerator and message passing libraries can create problems at runtime, often resulting in data corruption or deadlock as memory pinnings created by one user-mode subsystem are silently removed by another. Early experiments with simple MPI codes sending and receiving messages directly from pinned memory regions allocated using `cudaMallocHost()` brought this potential interoperability problem to light. While individual vendors could work together to make their products interoperable, this is a problem that needs to be solved for all networking and accelerator products that make use of user-mode DMA, and one that would benefit from standardization. A further refinement would involve the development of a standard mechanism for performing DMA and RDMA operations directly between accelerator devices, bypassing the host entirely. Such an interface could conceivably allow RDMAs from one GPU device directly to another GPU on a different host, greatly reducing the number of host-device copies involved in exchanging data.

Detection and recovery from hardware errors is a significant concern for long running simulations, particularly those running on hardware configurations containing thousands of cores. Glosli et al. describe the successful use of application-assisted error recovery in a 2.8 CPU-millennia molecular dynamics simulation of the Kelvin-Helmholtz instability on up to 212,992 CPU cores of the BG/L machine at Lawrence Livermore National Laboratory [18]. Sheaffer et al. recently reported on the problem of detecting and correcting soft errors on GPUs, using various hardware and software redundancy techniques [19]. They point out that soft error rates for arithmetic units are expected to increase significantly over the next several generations of GPU hardware. Existing GPU programming environments lack mechanisms to inform applications of uncorrected soft errors, so applications must presently attempt to detect errors themselves. Scientific codes that are expected to execute for millions of processor hours often contain internal self-consistency checks, but these usually only detect errors of significant magnitude, potentially leaving small errors undetected. Once errors are detected, restarting from a checkpoint presents an additional challenge. GPU software interfaces operate largely in user-mode, avoiding kernel context switch overhead that would reduce performance. This streamlined interface makes it difficult for GPU device drivers to save and restore the full GPU hardware and application state. For graphics applications this is not a significant problem since graphics libraries can be made to cache state internally and re-send this information on demand, such as when a windowing system redraw event occurs. Due to their generality, GPU computing applications as a whole don't have a well-defined minimal state vector to save and restore and GPU programming toolkits don't currently provide checkpoint/restart interfaces. As such, GPU accelerated applications must perform application-level checkpointing for themselves. Though a system-level mechanism for checkpointing GPU-accelerated applications isn't provided presently, it may be a future possibility as this problem has been solved for other hardware with user-mode interfaces, such as InfiniBand [20].

D. NAMD and Charm++

NAMD[1] [21], [22] is a highly scalable parallel program for the molecular dynamics simulation of large biomolecular systems. Distributed free of charge since 1995 [23], [24], NAMD is recognized as the leading software for running such simulations on large parallel machines, having demonstrated scaling to thousands of processors in a 2002 paper [25] that received a Gordon Bell Award. More recently, a 100-million-atom NAMD simulation was specified as a model problem for the NSF Track 1 petascale solicitation and as an acceptance test for the resulting Blue Waters machine. Applications of NAMD range from simulations of small proteins on microsecond timescales [26] to large protein aggregates (e.g., viruses) for more modest times [27].

NAMD is based on the Charm++ parallel programming system and runtime library[2] [28], [29] developed by the Parallel Programming Laboratory at the University of Illinois. Charm++ supports a "message-driven object" programming style where the computation is decomposed into objects that interact by sending messages to other objects on either the same or remote processors. Messages are asynchronous and one-sided; a particular method is invoked on an object whenever a message arrives for it. This programming style effectively hides communication latency and is naturally tolerant of the system noise of a machine [30], [31], [32]. Charm++ also supports processor virtualization [33], which allows each algorithm to be written for an ideal number of parallel objects that are then dynamically distributed among the actual number of processors on which the program is run.

In a molecular dynamics simulation, a collection of atoms interact through a set of forces based on atomic physics and quantum chemistry. Forces calculated based on atomic coordinates at one timestep are used to update the coordinates for the following timestep, producing a trajectory at the rate of one microsecond per billion steps. Since the forces for each step must be calculated in series, only a single complete force calculation is available for parallelization at a time.

The parallel decomposition strategy used by NAMD treats the simulation cell (the volume of space containing the atoms) as a three-dimensional patchwork quilt, with each patch of sufficient size so that only the 26 nearest-neighboring patches are involved in bonded, van der Waals, and short-range electrostatic forces. More precisely, the patches fill the simulation space in a regular grid and atoms in any pair of non-neighboring patches are separated by at least the "cutoff" distance at all times during the simulation. The patches contain dynamic data about each atom, such as coordinates and forces. Each hydrogen atom is stored on the same patch as the atom to which it is bonded, and atoms are reassigned to patches at regular intervals. The number of patches varies from one to several thousand and is determined by the size of the simulation, independent of the number of processors. Additional parallelism may be generated through options that double the number (and halve the size) of patches in one or more dimensions.

When NAMD is run, patches are distributed as evenly as possible, keeping nearby patches on the same processor when there are more patches than processors, or spreading them across the machine when the number of patches is equal to, or fewer than the number of processors. A larger number (roughly 14 per patch) of compute objects, responsible for calculating atomic forces due to interactions either within a single patch or between neighboring patches, are then distributed across the processors. The distribution scheme attempts to minimize communication by grouping compute objects tied to the same patches together on the same processor, which is often the home processor of neither patch. Near the beginning of the simulation, and periodically (every several thousand timesteps), a load balancer measures the amount of work performed by each compute object, and uses the measured data to redistribute compute objects in order to balance the workload between processors.

II. Adapting NAMD to GPU Clusters

When CUDA was first announced in November 2006 we began an immediate evaluation of the technology for NAMD and other applications in molecular modeling[3] [1]. While the GPU provides an incredible leap in single-node performance, distributed memory parallelism is still required for most simulations. Attaining good parallel scaling is actually more challenging when GPUs are employed, as a formerly compute-bound application is now dominated by communication and previously negligible overhead.

In accelerating NAMD with CUDA, we sought to preserve the parallel structure of the existing code to the greatest extent possible. A naive approach would have simply replaced the force calculation of each individual compute object with an equivalent CUDA kernel invocation. This would have resulted in poor performance due to both the overhead of the kernel invocation and the difficulty of extracting sufficient parallelism from a single patch-pair to engage all of the GPU multiprocessors. Instead, a single compute object was created within each NAMD process to aggregate all of the assigned patch-pair calculations for the GPU. On receiving atomic coordinate data from all of the assigned patches, this object uses only a small number of CUDA library calls and kernel invocations to copy the data to the GPU, calculate all of the required forces, and copy the results to the CPU. Only short-range nonbonded

[1]http://www.ks.uiuc.edu/Research/namd/

[2]http://charm.cs.uiuc.edu/research/charm/

[3]http://www.ks.uiuc.edu/Research/gpu/

forces are calculated on the GPU; long-range electrostatic and bonded forces and coordinate updates remain on the CPU.

A. GPU Force Calculation Kernel

Each thread block in our kernel invocation calculates the forces on the atoms in one patch due to atoms in either the same or a neighboring patch. Thus, every patch-internal or patch-pair compute object corresponds to one or two CUDA thread blocks, respectively. The kernel copies the atoms from the first patch in the assigned pair to block-local shared memory and loads the atoms from the second patch into thread-local registers. All threads iterate in unison over the atoms in shared memory, accumulating forces for the atoms in registers only, which are then written to GPU global memory. Since the forces between a pair of atoms are equal and opposite, the number of force calculations could be cut in half, but the extra coordination required to sum forces on the atoms in shared memory outweighs any savings.

NAMD uses constant memory to store a compressed lookup table of bonded atom pairs for which the standard short-range interaction is not valid. This is efficient because the table fits entirely in the constant cache and is referenced only for a small fraction of pairs. The texture unit is used to interpolate the short-range interaction from an array of values, which fits entirely in the texture cache. The dedicated hardware of the texture unit can return a separate interpolated value for every thread faster than the potential function could be evaluated analytically. A more complete description of the nonbonded kernel may be found in [1].

We note that the nonbonded force calculation as implemented on the GPU is much less work-efficient than the CPU version. This is because forces are only calculated between pairs of atoms within a cutoff distance. While searching only neighboring patches reduces the complexity of the search to linear time in the number of atoms, over 90% of tested pairs will fall outside of the cutoff distance. Given the 32-way SIMD control structure of the GPU, most threads will be idle while a few perform the actual force calculation. NAMD on the CPU employs other methods for increasing the performance of the pair search, such as testing the distances between heavy atoms before considering their bonded hydrogens, and reusing a list of atoms found within an extended cutoff distance for several steps. Neither method maps well to NAMD on the GPU, yet a factor of ten speedup compared to the CPU is still observed.

B. Overlapping GPU and CPU Execution

Although the nonbonded force calculation is greatly accelerated, the runtime of the GPU kernel is still a limiting factor on performance. Therefore, overlapping GPU and CPU execution can provide an easy performance gain. Of greater impact in parallel runs, the effectiveness of message-driven execution for the communication-intensive particle-mesh Ewald (PME) long-range electrostatics calculation may be affected by the long-running GPU calculation. One source of NAMD's parallel efficiency is the dynamic overlap of the multi-stage PME communication with nonbonded force calculation. This is accomplished by prioritizing PME messages and slicing the other calculations thin enough that PME work is processed almost immediately. A long-running GPU call on any one processor can therefore significantly delay all of the other processors.

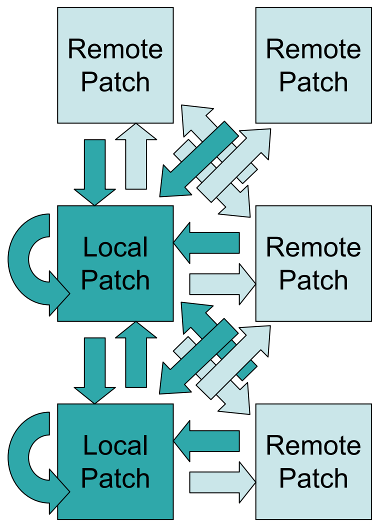

Fig. 1. Separation of GPU workload into remote and local force calculations to allow overlap of GPU calculation with force communication. Boxes represent local patches present on and remote patches accessed by a single process. Arrows represent calculation of forces on atoms in the patch pointed to due to atoms in the patch pointed from. GPU force calculations for remote patches (light arrows) are performed and the forces returned to the CPU before force calculations for local patches (dark arrows) are started, allowing local force calculations to overlap with the exchange of forces between nodes.

CUDA 1.0 allowed very limited asynchronous execution through its serialized API, busy-waiting for any transfer from the GPU. The CUDA 1.1 streaming API queues memory transfers and kernel invocations for asynchronous execution and allows probing for completion. With this API, the CUDA compute object returns once the GPU operations have been queued, allowing PME and other calculations to proceed on the CPU. By using an existing Charm++ API to insert a periodic probe for GPU completion into the Charm++ message loop, we ensure that nonbonded forces are processed nearly as soon as they are complete.

Overlap efficiency is still sensitive to the relative priorities of different messages and calculations. CPU-only NAMD prioritizes PME-related messages first, followed by coordinate exchange, force calculation, and finally force return. With nonbonded forces moved to the GPU, slightly overloaded or otherwise late-starting processors would be tied up by a backlog of PME messages, not receiving coordinates and starting the GPU calculation until PME was completed. By increasing the priority of coordinates and GPU invocation above that of PME, all processors effectively overlap GPU execution with PME and other force calculations.

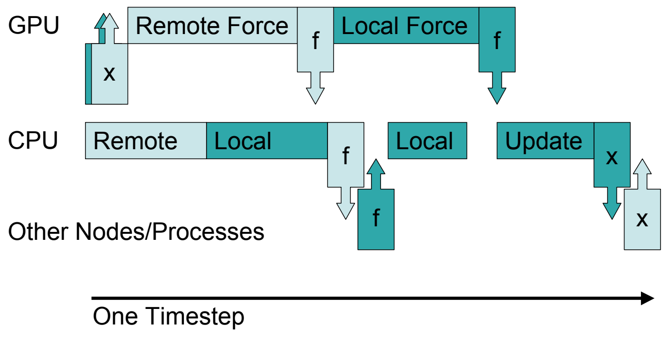

Fig. 2. Time sequence of CPU and GPU activity in a single process, showing overlap of GPU calculation and data transfer with both CPU calculation and communication. Generating and communicating results for remote patches (lighter color) is prioritized over local patches (darker color) on both the CPU and the GPU. Boxes with arrows show communication of coordinates (x) and forces (f) with the GPU and other processes. Collected forces are used to update local atom coordinates for the next timestep.

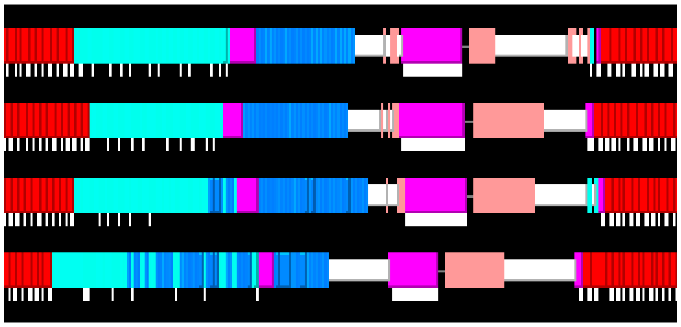

Fig. 3. Timeline of one timestep for four processes of GPU-accelerated NAMD as shown by the Charm++ performance analysis and visualization tool Projections. Colored bars indicate activity while descending white lines show message generation. Notable features, from left to right: coordinate update and network coordinate sends (red), network coordinate receipt and compute object queuing (teal), GPU data preparation and stream launch (magenta), remote and local CPU work (blue), CPU idle (white), GPU remote force notify and network sends (magenta), network force receipt (orange), CPU idle (white), GPU local force notify (very small magenta), coordinate update and network coordinate sends for following timestep (red).

C. Overlapping GPU Execution and Communication

As described above, PME communication is readily overlapped with force calculation on the GPU. The remaining communication is the return of forces calculated on the GPU to patches on other processors. CPU-only NAMD prioritizes compute objects returning forces to off-processor patches before purely local calculations. Purely local calculations may begin before off-processor coordinates are received, and continue after off-processor forces have been sent.

Using this same prioritization on the GPU is impractical because a running purely local kernel could not be interrupted when the higher priority work was ready to execute. As an alternative, we recall that separate thread blocks are used to calculate forces for each of the patches in a patch-pair compute object. Therefore, although a compute object may require coordinates from both an on-processor and an off-processor patch, the forces returned to the two patches may be calculated on the GPU at different times.

Thus we divide the GPU force calculation into two separate kernel invocations, as illustrated in Figs. 1 and 2. When coordinates from all required on-processor and off-processor patches are available these are copied to the GPU as above. The force kernel is first invoked on the GPU to calculate only forces on atoms from off-processor patches. After the resulting forces are copied to the CPU, the CUDA stream continues with a second force kernel invocation for on-processor patches while a CUDA event timer probe in the Charm++ message loop triggers the overlapped communication of forces to off-processor patches.

III. Performance and Scaling Results

Experiments were performed on the NCSA GPU cluster,[4] consisting of 16 Hewlett Packard xw9400 workstations each with two dual core 2.4 GHz Opteron 2216 processors and 8 GB of memory (2 GB/core). Each node hosts two external NVIDIA QuadroPlex model IV units containing two Quadro FX 5600 GPUs each, resulting in 4 GPUs per node with a cumulative total of 64 GPUs across the entire cluster. The QuadroPlex units are functionally equivalent to the more recent compute-specific Tesla D870 product in terms of their behavior and performance as compute accelerators. Although the cluster also contains Nallatech FPGA accelerators, they were not used for the work described in this paper. The cluster uses an SDR InfiniBand interconnect, consisting of a 24-port Cisco (Topspin) model TS-120 InfiniBand switch, with Mellanox Technologies MT25204 InfiniHost III Lx adapters installed in each node with a PCIe x4 electrical interface. The cluster software environment consisted of 64-bit RedHat Enterprise Linux Server 5.1, NVIDIA CUDA 1.1, Intel C/C++ 10.0, and MVAPICH2 version 0.9.8-15 [34]. Due to a hardware failure only 15 nodes (60 cores, 60 GPUs) were available for our tests. NAMD is capable of running on any number of processors, so we were able to obtain performance results using all of the remaining nodes.

The Charm++ performance analysis and visualization tool Projections[5] [35] was used to verify the overlapping of CPU and GPU execution and communication. A typical timestep is shown in Fig. 3, demonstrating the interleaving of CPU computation with network and GPU communication. Given visual confirmation that overlap was occurring as expected, we ran performance and strong scaling tests for two benchmarks, the large but increasingly typical “STMV” system [27] (Table I) and the smaller “ApoA1” system [25] (Table II). The local and remote blocks/GPU listed in the table are the number of thread blocks for the local and remote kernel invocations. As one would expect, the ratio of remote to local work increases with processor count. Half-sized patches were used for the smaller ApoA1 system to ensure a sufficient number of blocks to keep

[4]http://www.ncsa.uiuc.edu/Projects/GPUcluster/

[5]http://charm.cs.uiuc.edu/research/parallel_perf/

TABLE I
GPU-ACCELERATED NAMD PERFORMANCE ON 1.06M-ATOM "STMV" BENCHMARK (12 Å CUTOFF WITH PME EVERY 4 STEPS).

CPU Cores & GPUs	4	8	16	32	60
GPU-accelerated performance					
Local blocks/GPU	13186	5798	2564	1174	577
Remote blocks/GPU	1644	1617	1144	680	411
GPU s/step	0.544	0.274	0.139	0.071	0.040
Total s/step	0.960	0.483	0.261	0.154	0.085
Unaccelerated performance					
Total s/step	6.76	3.33	1.737	0.980	0.471
Speedup from GPU acceleration					
Factor	7.0	6.9	6.7	6.4	5.5

TABLE II
GPU-ACCELERATED NAMD PERFORMANCE ON 92K-ATOM "APOA1" BENCHMARK (12 Å CUTOFF WITH PME EVERY 4 STEPS).

CPU Cores & GPUs	4	8	16	32	60
GPU-accelerated performance					
Local blocks/GPU	2802	1131	492	216	98
Remote blocks/GPU	708	624	386	223	136
GPU s/step	0.051	0.027	0.015	0.008	0.005
Total s/step	0.087	0.048	0.027	0.018	0.013
Unaccelerated performance					
Total s/step	0.561	0.284	0.146	0.077	0.044
Speedup from GPU acceleration					
Factor	6.4	5.9	5.4	4.3	3.4

the GPU busy, allowing the GPU runtime per step to scale down to the 10 ms range. Figure 4 demonstrates that the ratio of atom and processor count primarily determines simulation performance, as our two benchmarks yield overlapping curves both with and without GPU acceleration.

As would be expected, parallel scalability is worse for the faster, accelerated code than for the slower CPU-only version. The GPU can do nothing to accelerate inter-node communication, and thus as communication increases with the number of processors the ratio of GPU to total s/step drops and the impact of the GPU fades. This results in a degradation of the acceleration factor from a peak of 7.0 for STMV on a single node to a low of 3.4 for ApoA1 on the complete cluster. Tuning of formerly irrelevant CPU-bound critical-path code should increase the achieved acceleration factor somewhat. The performance improvement obtained is certainly useful, for 8 GPUs nearly match the performance of 60 CPU cores. Early results from running the STMV simulation on an 8-node cluster of pre-production GT200-based GPUs (running at a reduced clock rate) yielded an overall acceleration factor of 9.0, with an overall runtime of 0.370 s/step and a GPU runtime of 0.180 s/step. The 8 pre-production GT200 GPUs match the performance of 72 CPU cores.

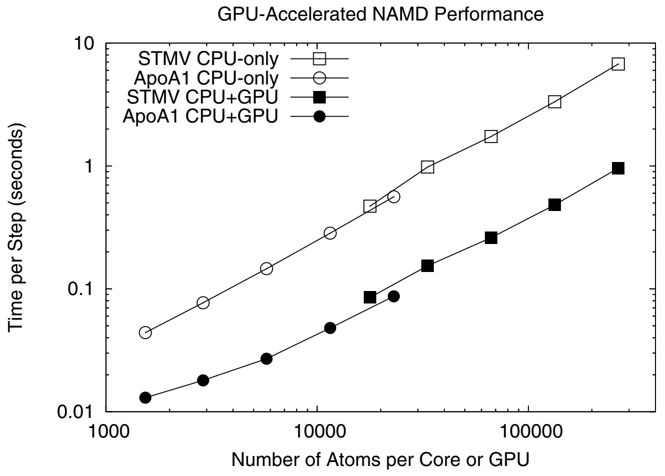

Fig. 4. NAMD performance as a function of atoms per core or GPU. The overlapping curves for large (STMV, 1.06M atoms) and small (ApoA1, 92K atoms) simulations both with and without GPU acceleration demonstrate that the ratio of atoms to processors is a primary factor in observed performance.

IV. DISCUSSION

The overlapped GPU execution pattern we have described for NAMD can be adapted for use in other parallel programs. Message-driven execution can be emulated in purely SPMD message-passing interfaces such as MPI through the use of a message polling and dispatch loop. In such an implementation a main event loop continuously polls for incoming messages, dispatching them to handler routines based on the incoming message tag [30]. Asynchronous GPU I/O and kernel invocations can be incorporated into this type of event loop by periodically polling for GPU asynchronous event completion status in the main event loop. By timing and storing the average execution time for a particular GPU kernel, a predictive strategy can be used to avoid polling GPU completion status too frequently, polling the GPU only when approaching the predicted execution time.

Figure 3 shows significant idle time when the CPU has completed all of its available work and is waiting for results from the local GPU. Many of these results are in fact already calculated but are not transferred to the CPU until the kernel invocation has completed for the entire grid. Refining the partitioning of the GPU force calculation from two classes (remote forces and local forces) currently to a series of kernel invocations on smaller grids, alternating with data transfers, would allow the CPU to stay busy as results would begin to flow from the GPU much earlier. Splitting these fine-grained kernel invocations into a pair of independent streams would further increase efficiency by allowing the GPU to overlap data transfer in one stream with calculation in the other. Since CUDA kernel invocations do not overlap, for these techniques to be effective the force calculation kernel must provide enough blocks that dividing them among several invocations does not cause excessive loss of efficiency. The block counts in Tables I and II suggest that this is will only be the case for small GPU counts or very large simulations.

An alternative approach would be to have a single long-running kernel invocation generate result sets in a prioritized order and have the CPU poll GPU memory to determine when result sets are ready to transfer. This may be acceptable when the CPU is otherwise idle, but if the CPU has other duties then either CPU-based tasks will be slowed by the overhead of frequently polling the GPU or pending data transfers may back up while the CPU is otherwise engaged. It is unfortunate that it is not possible to write directly to CPU memory or otherwise transfer data off of the GPU from within a CUDA kernel, which would greatly reduce demands on the CPU from this approach.

To better accommodate fine-grained message-driven execution we suggest several enhancements to the CUDA model. The GPU execution model of CUDA is simple: run a single kernel invocation for all blocks of the specified grid until completion. Multiple processes and multiple streams within a process share a physical GPU through coarse first-in first-out queuing, giving the bulk of the resources to the most aggressive program. The more refined behavior of the GPU when actually drawing to the screen suggests that the hardware is more flexible than CUDA exposes. The monolithic grid invocation should be extended with a block-granularity work queue. In this model a single kernel invocation would be passed groups of independent blocks in many small stages, interspersed with data transfers to be completed before or after all following or preceding blocks had been executed. This would allow the continuous streaming of data through the GPU at a much finer level than currently feasible. Additionally, by letting the programmer assign priorities to groups of blocks or perhaps only entire streams, the GPU could be kept busy at all times without delaying urgent calculations.

V. Conclusion

The ultimate goal for any accelerator technology is to turn a parallel application into a serial one with sufficient total performance. More realistically, a successful accelerator will simply change the performance-limiting factor from computation to communication. The latency tolerance and overlap capability of a message-driven programming style are therefore of particular value for accelerated applications, but only if sufficient work can be aggregated to keep the accelerator busy. We have provided in this paper an example of an application benefiting from this novel programming style, resolving issues that will likely arise in the acceleration of many other legacy programs. We have also proposed several improvements and extensions to GPU programming interfaces that would allow these techniques to be more efficiently implemented and improve the usability of GPU-accelerated clusters in general.

Acknowledgments

This work was supported by the National Institutes of Health, under grant P41-RR05969. Performance experiments were made possible by a generous hardware donation by NVIDIA, NSF CNS grant 05-51665, the National Center for Supercomputing Applications (NCSA), and the University of Illinois through an IACAT grant. We are grateful to University of Illinois Professor Wen-mei Hwu and to NVIDIA Chief Scientist David Kirk for introducing us to the CUDA technology and for many valuable discussions since. We would also like to thank Mike Showerman and Jeremy Enos of NCSA and Massimiliano Fatica of NVIDIA for hardware and software support during our development and benchmarking runs.

References

[1] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *J. Comp. Chem.*, vol. 28, pp. 2618–2640, 2007.

[2] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, 2008.

[3] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *J. Chem. Phys.*, vol. 227, no. 10, pp. 5342–5359, 2008.

[4] I. Ufimtsev and T. Martinez, "Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008.

[5] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W.-M. W. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *CF'08: Proceedings of the 2008 conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 273–282.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM Press, 2004, pp. 777–786.

[7] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 787–795, Aug. 2004.

[8] M. Charalambous, P. Trancoso, and A. Stamatakis, "Initial experiences porting a bioinformatics application to a graphics processor." in *Panhellenic Conference on Informatics*, 2005, pp. 415–425.

[9] D. R. Horn, M. Houston, and P. Hanrahan, "ClawHMMER: A streaming HMMer-search implementation," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 11.

[10] E. Elsen, V. Vishal, M. Houston, V. Pande, P. Hanrahan, and E. Darve, "N-body simulations on GPUs," Stanford University, Stanford, CA, Tech. Rep., Jun. 2007, http://arxiv.org/abs/0706.3060.

[11] "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," NVIDIA, NVIDIA, Santa Clara, CA, USA, 2007.

[12] M. McCool, "Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform," in *GSPx Multicore Applications Conference*, Oct./Nov. 2006.

[13] Advanced Micro Devices Inc., "Brook+ SC07 BOF session," in *Supercomputing 2007 Conference*, Nov. 2007.

[14] J. Stratton, S. Stone, and W. mei Hwu, "MCUDA: An efficient implementation of CUDA kernels on multi-cores," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-08-01, March 2008. [Online]. Available: http://www.gigascale.org/pubs/1278.html

[15] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[16] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 47.

[17] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, and S. Turek, "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," *Parallel Comput.*, vol. 33, no. 10-11, pp. 685–699, 2007.

[18] J. N. Glosli, K. J. Caspersen, J. A. Gunnels, D. F. Richards, R. E. Rudd, and F. H. Streitz, "Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[19] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 55–64.

[20] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-transparent checkpoint/restart for MPI programs over Infiniband," *International Conference on Parallel Processing, 2006.*, pp. 471–478, Aug. 2006.

[21] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater scalability for parallel molecular dynamics," *J. Comp. Phys.*, vol. 151, pp. 283–312, 1999.

[22] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *J. Comp. Chem.*, vol. 26, pp. 1781–1802, 2005.

[23] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kalé, R. Skeel, K. Schulten, and R. Kufrin, "MDScope – A visual computing environment for structural biology," *Comput. Phys. Commun.*, vol. 91, no. 1-3, pp. 111–134, 1995.

[24] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kalé, R. D. Skeel, and K. Schulten, "NAMD – A parallel, object-oriented molecular dynamics program," *Int. J. Supercomp. Appl. High Perform. Comp.*, vol. 10, pp. 251–268, 1996.

[25] J. Phillips, G. Zheng, S. Kumar, and L. Kale, "NAMD: Biomolecular simulation on thousands of processors." in *Proceedings of the IEEE/ACM SC2002 Conference, Technical Paper 277*. IEEE Press, 2002.

[26] P. L. Freddolino, F. Liu, M. Gruebele, and K. Schulten, "Ten-microsecond MD simulation of a fast-folding WW domain," *Biophys. J.*, vol. 94, pp. L75–L77, 2008.

[27] P. L. Freddolino, A. S. Arkhipov, S. B. Larson, A. McPherson, and K. Schulten, "Molecular dynamics simulations of the complete satellite tobacco mosaic virus," *Structure*, vol. 14, pp. 437–449, 2006.

[28] 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.

[29] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming Petascale Applications with Charm++ and AMPI," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman & Hall / CRC Press, 2008, pp. 421–441.

[30] A. Gursoy and L. Kalé, "Performance and modularity benefits of messagedriven execution," *Journal of Parallel and Distributed Computing*, vol. 64, pp. 461–480, 2004.

[31] S. Pakin, V. Karamcheti, and A. A. Chien, "Fast messages: Efficient, portable communication for workstation clusters and mpps," *IEEE Parallel Distrib. Technol.*, vol. 5, no. 2, pp. 60–73, 1997.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 256–266, 1992.

[33] L. V. Kalé, "The virtualization model of parallel programming: Runtime optimizations and the state of art," in *LACSI 2002*, Albuquerque, October 2002.

[34] W. Huang, G. Santhanaraman, H. W. Jin, Q. Gao, and D. K. Panda, "Design of high performance MVAPICH2: MPI2 over InfiniBand," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 43–48.

[35] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.