Dynamic Load Balancing on Single- and Multi-GPU Systems

Long Chen†, Oreste Villa‡, Sriram Krishnamoorthy‡, Guang R. Gao†

†Department of Electrical & Computer Engineering
University of Delaware
Newark, DE 19716
{lochen, ggao}@capsl.udel.edu

‡High Performance Computing
Pacific Northwest National Laboratory
Richland, WA 99352
{oreste.villa, sriram}@pnl.gov

Abstract

The computational power provided by many-core graphics processing units (GPUs) has been exploited in many applications. The programming techniques currently employed on these GPUs are not sufficient to address problems exhibiting irregular, and unbalanced workload. The problem is exacerbated when trying to effectively exploit multiple GPUs concurrently, which are commonly available in many modern systems. In this paper, we propose a task-based dynamic load-balancing solution for single- and multi-GPU systems. The solution allows load balancing at a finer granularity than what is supported in current GPU programming APIs, such as NVIDIA's CUDA. We evaluate our approach using both micro-benchmarks and a molecular dynamics application that exhibits significant load imbalance. Experimental results with a single-GPU configuration show that our fine-grained task solution can utilize the hardware more efficiently than the CUDA scheduler for unbalanced workload. On multi-GPU systems, our solution achieves near-linear speedup, load balance, and significant performance improvement over techniques based on standard CUDA APIs.

1 Introduction

Many-core Graphics Processing Units (GPUs) have become an important computing platform in many scientific fields due to the high peak performance, cost effectiveness, and the availability of user-friendly programming environments, e.g., NVIDIA CUDA [20] and ATI Stream [1]. In the literature, many works have been reported on how to harness the massive data parallelism provided by GPUs [4, 13, 19, 23, 25].

However, issues, such as load balancing and GPU resource utilization, cannot be satisfactorily addressed by the current GPU programming paradigm. For example, as shown in Section 6, CUDA scheduler cannot handle the unbalanced workload efficiently. Also, for problems that do not exhibit enough parallelism to fully utilize the GPU, employing the canonical GPU programming paradigm will simply underutilize the computation power. These issues are essentially due to fundamental limitations on the current data parallel programming methods.

In this paper, we propose a task-based fine-grained execution scheme that can dynamically balance workload on individual GPUs and among GPUs, and thus utilize the underlying hardware more efficiently.

Introducing tasks on GPUs is particularly attractive for the following reasons. First, although many applications are suitable for data parallel processing, a large number of applications show more task parallelism than data parallelism, or a mix of both [7]. Having a task parallel programming scheme will certainly facilitate the development of this kind of applications on GPUs. Second, by exploiting task parallelism, it is possible to show better utilization of hardware features. For example, task parallelism is exploited in [22] to efficiently use the on-chip memory on the GPU. Third, in task parallel problems, some tasks may not be able to expose enough data parallelism to fully utilize the GPU. Running multiple such tasks on a GPU concurrently can increase the utilization of the computation resource and thus improve the overall performance. Finally, with the ability to dynamically distribute fine-grained tasks between CPUs and GPUs, the workload can potentially be distributed properly to the computation resources of a heterogeneous system, and therefore achieve better performance.

However, achieving task parallelism on GPUs can be challenging; the conventional GPU programming does not provide sufficient mechanisms to exploit task parallelism in applications. For example, CUDA requires all programmer-defined functions to be executed sequentially on the GPU [21]. Open Computing Language (OpenCL) [15] is an emerging programming standard for general purpose parallel computation on heterogeneous systems. It supports the task parallel programming model, in which computations are expressed in terms of multiple concurrent tasks where a task is a function executed by a single processing element, such as a CPU thread. However, this task model is basically established for multi-core CPUs, and does not address the characteristics of GPUs. Moreover, it does not require a particular OpenCL implementation to actually execute multiple tasks in parallel. For example, NVIDIA's current OpenCL implementation does not support concurrent execution of multiple tasks due to the hardware limitations.

To address the problem of achieving dynamic load balance with fine-grained task execution on GPUs, in this paper, we make the following contributions.

- We first identify the mechanisms to enable correct and efficient CPU-GPU interactions while the GPU is computing, based on the current CUDA technology. This provides means for building uninterrupted communication schemes between CPUs and GPUs.
- Based on the above contribution, we introduce a task queue scheme, which enables dynamic load balancing at a finer granularity than what is supported in existing CUDA programming paradigm. We also study the optimal memory sub-system locations for the queue data structures.
- We implement our task queue scheme with CUDA. This implementation features concurrent host enqueue and device dequeue, and wait-free dequeue operations on the device. We evaluate the performance of the queue operations with benchmarks.
- As a case study, we apply our task queue scheme to a molecular dynamics application. Experimental results with a single-GPU configuration show that our scheme can utilize the hardware more efficiently than the CUDA scheduler, for unbalanced problems. For multi-GPU configurations, our solution achieves nearly linear speedup, load balance, and significant performance improvement over alternative implementations based on the canonical CUDA paradigm.

The rest of the paper is organized as follows. Section 2 presents a brief overview of the research on load balancing and task parallelism on GPUs. Section 3 describes the CUDA architecture. Section 4 presents the design of our task queue scheme. Section 5 discusses implementation issues and benchmarking results of queue operations. Section 6 evaluates our task queue scheme with a molecular dynamics application on single- and multi-GPU systems. Section 7 concludes with future work.

2 Related Work

Load balance is a critical issue for parallel processing. However, in the literature, there are few studies addressing this issue on GPUs. The load imbalance issue of graphic problems was discussed in [10, 18], and authors observed that it is of fundamental importance for high performance implementations on GPUs. Several static and dynamic load balancing strategies were evaluated for an octree partitioning problem on GPUs in [6]. Our work differs from this study in several ways. First, in the former study, the load balancing strategies were carried out solely on the GPU; the CPU cannot interact with the GPU during the execution. Second, the former study only investigated single-GPU systems. Our work is performed on both single- and multi-GPU systems, and can be easily extended to GPU clusters.

A runtime scheduler is presented for situations where individual kernels cannot fully utilize GPUs [12]. It extracts the workloads from multiple kernels and merges them into a super-kernel. However, such transformations have to be performed statically, and thus dynamic load balance cannot be guaranteed. Recently, researchers have begun to investigate how to exploit heterogeneous platforms with the concept of tasks. Merge [16] is such a programming framework proposed for heterogeneous multi-core systems. It employs a library-based method to automatically distribute computation across the underlying heterogeneous computing devices. STARPU [2] is another framework for task scheduling on heterogeneous platforms, in which hints, including the performance models of tasks, can be given to guide the scheduling policies. Our work is orthogonal to prior efforts in that our solution exhibits excellent dynamic load balance. It also enables the GPU to exchange information with the CPU during execution, which enables these platforms to understand the runtime behavior of the underlying devices, and further improve the performance of the system.

3 CUDA Architecture

In this section we provide a brief introduction of the CUDA architecture and the programming model. More details are available on the CUDA website [20]. In the literature, GPUs and CPUs are usually referred to as the *devices* and the *hosts*, respectively. We follow the same terminology in this paper.

CUDA devices have one or multiple streaming multiprocessors (SMs), each of which consists of one instruction issue unit, eight scalar processor (SP) cores, two transcendental function units, and on-chip shared memory. For some high-end devices, the SM also has one double-precision floating point unit. CUDA architecture features both on-chip memory and off-chip memory. The on-chip memory consists of the register file, shared memory, constant cache and texture cache. The off-chip memory consists of the local memory and the global memory. Since there is no ordering guarantee of memory accesses on CUDA architectures, programmers may need to use memory fence instructions to explicitly enforce the ordering, and thus the correctness of the program. The host can only access the global memory of the device. On some devices, part of the host memory can be pinned and mapped into the device's memory space, and both the host and the device can access that memory region using normal memory load and store instructions.

A CUDA program consists of two parts. One part is the portions to be executed on the CUDA device, which are called *kernels*; another part is to be executed on the host, which we call the *host process*. The device executes one kernel at a time, while subsequent kernels are queued by the CUDA runtime. When launching a kernel, the host process specifies how many threads are required to execute the kernel, and how many *thread blocks* (TB) these threads should be equally divided into. On the device, all threads can access the global memory space, but only threads within a TB can access the associated shared memory, with very low latency. A thread can obtain its logic thread index within a TB and its logic TB index via built-in system variables. The hardware schedules and distributes TBs to SMs with available execution capacity. One or multiple TBs can reside concurrently on one SM, given sufficient hardware resources, i.e., register file, shared memory, etc. TBs do not migrate during the execution. As they terminate, the hardware launches new TBs on these vacated SMs, if there are still some TBs to be executed for this kernel. Each thread is mapped to one SP core. Moreover, the SM manages the threads in groups of 32 threads called *warps*, in the sense that all threads in a warp execute one common instruction at a time. Thread divergences occur when the threads within a wrap take different execution paths. The execution of those taken paths will be serialized, which can significantly degrade the performance. While CUDA provides a barrier function to synchronize threads within a TB, it does not provide any mechanism for communications across TBs. However, with the availability of the atomic instructions and memory fence functions, it is possible to achieve inter-TB communications.

4 System Design

In this section, we first describe the basic idea of our task queue scheme. Then we discuss the necessary mechanisms to perform host-device interactions correctly and efficiently, and then present the design of our task queue scheme in detail.

4.1 Basic Idea

With the current CUDA programming paradigm, to execute multiple tasks, the host process has to sequentially launch multiple, different kernels, and the hardware is responsible for arranging how kernels run on the device [21]. This paradigm is illustrated in Figure 1. On the

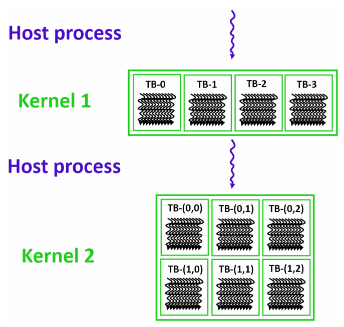

Figure 1: CUDA programming paradigm

other hand, in our task queue scheme, instead of launching multiple kernels for different tasks, we launch a *persistent* kernel with B TBs, where B can be as big as the maximum number of concurrently active TBs that a specific device can support. Since CUDA will not *swap out* TBs during their execution, after being launched, all TBs will stay active, and wait for executing tasks until the kernel terminates. When the kernel is running on the device, the host process enqueues both computation tasks and signalling tasks to one or more task queues associated with the device. The kernel dequeues tasks from the queues, and executes them according to the pre-defined task information. In other words, the host process dynamically controls the execution of the kernel by enqueuing tasks, which could be homogeneous or heterogeneous. This task queue idea is illustrated in Figure 2.

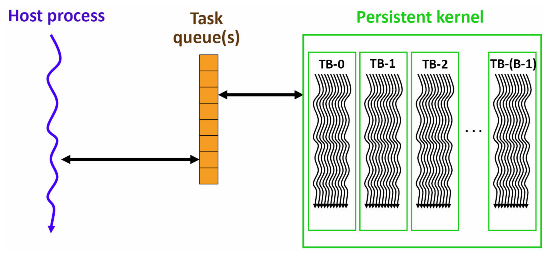

Figure 2: Task queue paradigm

4.2 Preliminary Considerations

Since task queues are usually generalized as producer-consumer problems, let us first consider the single-producer single-consumer case. Algorithm 1 shows the pseudo-code of *enqueue* and *dequeue* operations for such scenario on a shared memory system. *queue* is a shared buffer between the producer and the consumer. *start* and *end* are indexes of next location for dequeue and enqueue, respectively. At the beginning, both indexes are initialized as 0. By polling *start* and *end*, the producer/consumer can determine if it can enqueue/dequeue tasks.

Algorithm 1

```
Enqueue
Data: a task object $task$, a task queue $queue$ of a capacity of $size$
Result: $task$ is inserted into $queue$
1: repeat
2:     $l \leftarrow (end - start + size) \pmod{size}$
3: until $l < (size - 1)$
4: $queue[end] \leftarrow task$
5: $end \leftarrow (end + 1) \pmod{size}$

Dequeue
Data: a task queue $queue$ of a capacity of $size$
Result: a task object is removed from $queue$ into $task$
1: repeat
2:     $l \leftarrow (end - start + size) \pmod{size}$
3: until $l > 0$
4: $task \leftarrow queue[start]$
5: $start \leftarrow (start + 1) \pmod{size}$
```

If we want to establish a similar scheme for the host-device communication, where the host process is the producer, and the kernel is the consumer, the following issues have to be addressed properly.

The first issue is how to enable the host process to perform copies between the host memory and device memory without interrupting the kernel execution, which is of fundamental importance for our task queue scheme.

The second issue is where to keep the queue and associated index variables. For index variables, a naïve choice would be having *end* in the host's memory system, and having *start* in the device's memory system. In this case, all updates to index variables can be performed locally. However, this choice introduces serious performance issue, i.e., each queue polling action requires an access to a index variable in another memory system, which implies a transaction across the host-device interface. This incurs significant latency (as shown in Section 5 for a PCIe bus). On the other hand, having both *start* and *end* on one memory system will not help; either the host process or the kernel has to perform two transactions across the host-device interface, which actually aggravates the situation.

The third issue is how to guarantee the correctness of the queue operations in this host-device situation. Since each enqueue/dequeue operation consists of multiple memory updates, i.e., write/read to the queue and write to an index, it is crucial to ensure the correct ordering of these memory accesses while across the host-device interface. For example, for an enqueue operation described in Algorithm 1, by the updated value of *end* (Enqueue:line-5) is visible to the kernel, the insertion of *task* (Enqueue:line-4) should have completed. If the ordering of these two memory accesses were reversed by the hardware/software for some reason, the kernel will not be able to see the consistent queue state, and therefore the whole scheme will not work correctly.

The final issue is how to guarantee the correctness on accessing shared objects, if we allow dynamic load balance on the device. Lock is extensively used for this purpose. However, as presented in [6], a lock-based blocking method is very expensive on GPUs.

With evolving GPU technologies, now it is possible to address above issues by exploiting the new hardware and software features. More specifically, the current CUDA allows "asynchronous concurrent execution"[1]. This feature enables copies between pinned host memory and device memory concurrently with the kernel execution, which solves our first issue.

Since CUDA 2.2, a region of the host's memory can be mapped into the device's address space, and kernels can directly access this region of memory using normal load/store memory operations[2]. By duplicating index variables, and cleverly utilizing this feature, as we shall demonstrate in our design, the queue polling in the enqueue and dequeue operations only incurs local memory accesses, and at most one remote memory access to an index variable is needed for a successful enqueue/dequeue operation. This addresses part of the second issue, i.e., where to keep index variables.

The solution to the rest of the second issue and the third issue essentially requires mechanisms to enforce the ordering of memory access across the host-device interface. Memory fence functions are included in the new CUDA runtime. But they are only for memory accesses (made by the kernel) to the global and shared memory on the device. On the other hand, CUDA *event* can be used by the host process to asynchronously monitor the device's progress, e.g., memory accesses. Basically, an event can be inserted into a sequence of commands issued to a device. If this event is *recorded*, then all commands preceding it must have completed. Therefore, by inserting an event immediately after a memory access to the device's memory system and waiting for its being recorded, the host process can guarantee that such memory operation has completed on the device, before it proceeds. This is equivalent to a memory fence for the host process to

[1]. This feature is available on CUDA devices that support **deviceOverlap**.

[2]Provided that this CUDA device supports **canMapHostMemory**.

access the device's memory system. However, at this moment, there is no sufficient mechanism to ensure the ordering of memory accesses made by a kernel to the mapped host memory [17]. In this case, if we had the task queues residing on the host memory system, in the dequeue operation, the kernel cannot guarantee that the memory read on the task object has completed before updating the corresponding index variables. Therefore, the queue(s) can only reside in the global memory on the device. On the whole, by having the queue(s) on the device, and using both the event-based mechanism mentioned above and the device memory fence functions, we can develop correct enqueue and dequeue semantics that consist of memory accesses to both the host's memory system and the device's memory system.

With the advent of atomic functions on GPUs, such as *fetch-and-add* and *compare-and-swap*, it is possible to allow non-blocking synchronization mechanisms. This resolves the last issue.

Here we summary all necessary mechanisms to enable correct, efficient host-device interactions as follows.

1. Asynchronous concurrent execution: overlap the host-device data transfer with kernel execution.
2. Mapped host memory: enable the light-weight queue polling without generating host-device traffic.
3. Event: asynchronously monitor the device's progress.
4. Atomic instructions: enable the non-blocking synchronization.

4.3 Notations

Before we present the task queue scheme, we first introduce terminologies that will be used throughout the paper. On a device, threads have a unique *local_id* within a TB. *host_write_fence()* is the event-based mechanism described above, which guarantees the correct ordering of memory stores made by the host process to the device's memory system. *block_write_fence()* is a fence function that guarantees that all device memory stores (made by the calling thread on the device) prior to this function are visible to all thread in the TB. *block_barrier()* synchronizes all threads within a TB.

For variables that are referred by both the host process and the kernel, we prefix them with either *h_* or *d_* to denote their actual residence on the host or the device, respectively. For variables residing in the device's memory system, we also postfix them with *_sm* or *_gm* to denote whether they are in the shared memory or in the global memory. For those variables without any prefix or postfix just described, they are simply host/device local variables.

4.4 Task Queue Scheme

Here we present our novel task queue scheme, which allows automatic, dynamic load balancing on the device without using expensive locks. In this scheme, the host process sends tasks to the queue(s) without interrupting the execution of the kernel. On the device, all TBs concurrently retrieve tasks from these shared queue(s). Each task will be executed by a single TB.

A queue object is described by the following variables. At the beginning of a computation, all those variables are initialized to 0s.

- d_tasks_gm: an array of task objects.
- d_n_gm: the number of tasks ready to be dequeued from this queue.
- $h\_written$: the host's copy of the accumulated number of tasks written to this queue.
- $d\_written\_gm$: the device's copy of the accumulated number of tasks written to this queue.
- $h\_consumed$: the host's copy of the accumulated number of tasks dequeued by the device.
- $d\_consumed\_gm$: the device's copy of the accumulated number of tasks dequeued by the device.

Algorithm 2 Host Enqueue

```
Data: $n$ task objects $tasks$, $n\_queue$ task queues $q$ , each of a capacity of
      $size$, $i$ next queue to insert
Result: host process enqueues $tasks$ into $q$
 1: $n\_remaining \leftarrow n$
 2: if $n\_remaining > size$ then
 3:     $n\_to\_write \leftarrow size$
 4: else
 5:     $n\_to\_write \leftarrow n\_remaining$
 6: end if
 7: repeat
 8:     if $q[i].h\_consumed = q[i].h\_written$ then
 9:         $q[i].d\_tasks\_gm \leftarrow tasks[n - n\_remaining : n - n\_remaining + n\_to\_write - 1]$
10:         $q[i].d\_n\_gm \leftarrow n\_to\_write$
11:         $host\_write\_fence()$
12:         $q[i].h\_written \leftarrow q[i].h\_written + n\_to\_write$
13:         $q[i].d\_written\_gm \leftarrow q[i].h\_written$
14:         $i \leftarrow (i+1) \pmod{n\_queues}$
15:         $n\_remaining \leftarrow n\_remaining - n\_to\_write$
16:         if $n\_remaining > size$ then
17:             $n\_to\_write \leftarrow size$
18:         else
19:             $n\_to\_write \leftarrow n\_remaining$
20:         end if
21:     else
22:         $i \leftarrow (i+1) \pmod{n\_queues}$
23:     end if
24: until $n\_to\_write = 0$
```

The complete enqueue and dequeue procedures are described in Algorithm 2, and Algorithm 3, respectively. To enqueue tasks, the host first has to check whether a queue is ready. In our scheme, we require that a queue is ready when it is empty, which is identified by $h\_consumed = h\_written$. If a queue is not ready, the host either waits

Algorithm 3 Device Dequeue

```
Data: $n\_queue$ task queues $q$, $i$ next queue to work on
Result: TB fetches a task object from $q$ into $task\_sm$
 1: $done \leftarrow$ false
 2: if $local\_id = 0$ then
 3:     repeat
 4:         if $q[i].d\_consumed\_gm = q[i].d\_written\_gm$ then
 5:             $i \leftarrow (i+1) \pmod{n\_queues}$
 6:         else
 7:             $j \leftarrow fetch\_and\_add(q[i].d\_n\_gm, -1) - 1$
 8:             if $j \geq 0$ then
 9:                 $task\_sm \leftarrow q[i].d\_tasks\_gm[j]$
10:                 $block\_write\_fence()$
11:                 $done \leftarrow$ true
12:                 $jj \leftarrow fetch\_and\_add(q[i].d\_consumed\_gm, 1)$
13:                 if $jj = q[i].d\_written\_gm$ then
14:                     $q[i].h\_consumed \leftarrow q[i].d\_consumed\_gm$
15:                     $i \leftarrow (i+1) \pmod{n\_queues}$
16:                 end if
17:             else
18:                 $i \leftarrow (i+1) \pmod{n\_queues}$
19:             end if
20:         end if
21:     until $done$
22: end if
23: $block\_barrier()$
```

until this queue becomes ready (single-queue case), or checks other queues (multi-queue case). Otherwise, the host first places tasks in d_tasks_gm starting from the starting location, and update d_n_gm to the number of tasks enqueued. Then it waits on *host_write_fence()* to make sure that the previous writes have completed on the device. After that, the host process updates $h\_written$, and $d\_written\_gm$ to inform the kernel that new tasks are available in the queue.

On the device, each TB uses a single thread, i.e., the one of $local\_id = 0$, to dequeue tasks. It first determines whether a queue is empty by checking $d\_consumed\_gm$ and $d\_written\_gm$. If a queue is empty, then it keeps checking until this queue has tasks (single-queue case) or checks other queues (multi-queue case). Otherwise, it first applies *fetch-and-add* on d_n_gm with -1. If the return value of this atomic function is greater than 0, it means there is a valid task available in the queue, and this TB can use this value as the index to retrieve a task from d_tasks_gm. The thread waits on the memory fence until the retrieval of the task is finished. It then applies *fetch-and-add* on $d\_written\_gm$ with 1, and checks whether the task just retrieved is the last task in the queue by comparing the return value of the atomic function just called with $d\_written\_gm$. If yes, this thread is responsible for updating $h\_consumed$ to inform the host process that this queue is empty. Barrier (Algorithm 3:Line 23) is used to make sure that all threads in a TB will read the same task information from $task\_sm$ after the dequeue procedure exits. The dequeue procedure is a wait-free [14] approach, in the sense that in a fixed number of steps, a TB either retrieves a task from a queue, or finds out that queue is empty.

To avoid data race, this scheme does not allow the host process to enqueue tasks to an non-empty queue that is possibly being accessed by the kernel. This seems to be a shortcoming of this scheme because enqueue and dequeue operations cannot be carried out concurrently on a same queue. However, simply employing multiple queues can efficiently solve this issue by overlapping enqueue with the dequeue on different queues.

To determine when to signal the kernel to terminate, the host process has to check whether all queues are empty after computation tasks have been enqueued. If it is true, the host process enqueues B HALT tasks, one for each of the B concurrently active TBs on the device. TBs exit after getting HALT, and eventually the kernel terminates.

Although our task queue scheme is presented for a single device, we have extended it for multi-GPU systems (on a single node), where a higher level task queue is maintained on the host side for coordinating the multiple GPUs. Extension to multiple nodes can also be carried out smoothly, but it is out of the scope of this paper. Moreover, the task queue scheme can be extended to allow multiple host processes share the use of a single device, by sending heterogeneous tasks to the queue(s) associated with the device, given host processes are orchestrated by some synchronizations. This can potentially increase the utilization of the device if a host process cannot fully use the computation capacity of the device.

5 Implementation and Microbenchmarks

In this section, we first describe the platform used in our experiments and some implementation issues of task queue scheme. We then present the benchmarking results for operations used in our task queue scheme.

5.1 Implementation

We implemented our task queue scheme on a system equipped with 1 quad-core AMD Phenom II X4 940 processor and 4 NVIDIA Tesla C1060 GPUs. The system is running 64-bit Ubuntu version 8.10, with NVIDIA driver version 190.10. CUDA Toolkit version 2.3 ,CUDA SDK version 2.3, and GCC version 4.3.2 were used in the development. The above system provides all necessary features to implement our task queue scheme.

To utilize the asynchronous concurrent execution feature, CUDA requires using different, nonzero CUDA *streams*, where a stream is basically a sequence of commands that are performed in order on the device, and the zero stream is the default stream in CUDA. So, in our im-

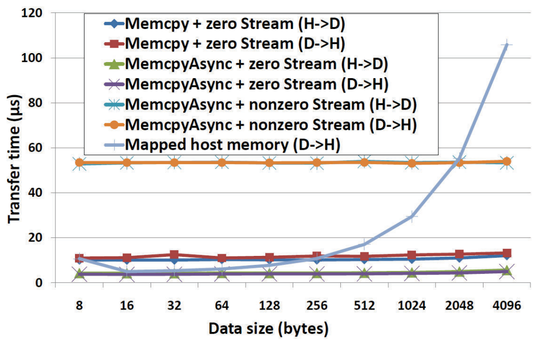

Figure 3: Data transfer time

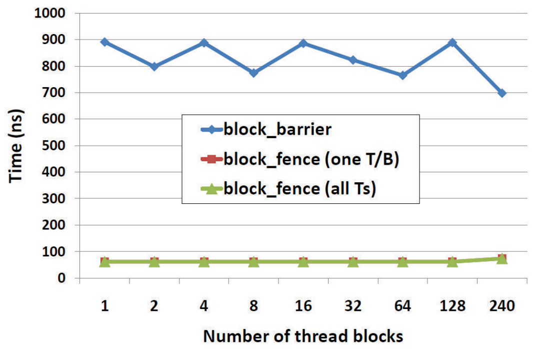

Figure 4: Barrier and fence functions (128Ts/B)

plementation, we use one stream for kernel launching, another stream for performing queue operations.

While CUDA does provide the required memory fence function, $\_\_threadfence\_block()$, it does not differentiate between stores and loads. In our implementations, it were used as a store fences. CUDA also provides a function to synchronize all threads in a TB, $\_\_syncthreads()$, which behaves as both a regular barrier and also a memory fence in a TB. Therefore, in our implementations, we took advantage of this and eliminated redundant operations.

5.2 Microbenchmarks

Here we report benchmarking results for major components performed in queue operations, such as, host-device data transfer, synchronizations, atomic instructions, and the complete enqueue/dequeue operations. Performance measurements of these individual components help us understand how our designs work with the real applications.

Host-device data transfer The host-device interface equipped in our system is PCIe 1.1 $\times$ 16. We measured the time to transfer contiguous data between the host and the device across this interface using the pinned memory. Since queue operations only update objects of small sizes, i.e., tasks and index variables, we conducted the test for sizes from 8 bytes to 4KB. Figure 3 shows measured transfer times for transfers initialized by the host process, i.e., using the regular synchronous copies (Memcpy) with the zero stream, asynchronous copies (MemcpyAsync) with the zero stream, and asynchronous copies with a nonzero stream, and the transfers initialized by the kernel, i.e., the mapped host memory, where H->D and D->H indicate the transfer from the host to the device, or the reverse, respectively. Note that one device thread was used to perform transfers from the device to the mapped host memory. From the figure, it is clear that using a nonzero stream to perform asynchronous copies is much expensive, compared to both synchronous copies and asynchronous copies performed with the zero stream, i.e., $5\times$-$10\times$ slower. Without exposing to the CUDA internal, we do not really understand why such operation is so costly. On the other hand, with the current CUDA programming environment, using nonzero streams is the only way to achieve the asynchronous concurrent execution.

For transfers initialized by the host process, the transfer time changes slowly in the above data range due to the high bandwidth, i.e., 4GB/s. So, if the host-device data transfer is inevitable, combining multiple data accesses into one single transaction is highly recommended. In fact, in our implementation of the enqueue operation, we actually update d_n_gm and d_tasks_gm with a single host-device transaction.

On the other hand, since currently there is no mechanism for a kernel to copy a contiguous memory region, such copy has to be performed with assignments on basic data types. Therefore, the transfer time is linearly proportional to the size of data.

Barrier and fence Barrier and memory fence functions are used in our task queue scheme to ensure the correctness of the operations. In this test, we made all threads (on the device) calling a specific barrier or fence function a large number of times, and measured the average completion time. For the fence function, we also measured the case that only one thread in each TB makes the call, which emulates the scenario in dequeue operations.

Figure 4 shows the results for these functions with a TB size of 128. We observed that the completion time of these functions tends to keep constant regardless the number of TBs launched. Especially, the fence functions are very efficient; it takes a same amount of time to complete for the case when called by a single thread in a TB (annotated with "one T/B" in the figure), and for the case when called by all threads in a TB (annotated with "all Ts/B"). Similar results were observed for various TB sizes.

Atomic instructions Atomic functions are used in our task queue scheme to guarantee correct dequeue operations on the device. In this benchmark, one thread in each TB performs a large number of *fetch-and-add* function on a device's global memory address. Experimental results show that atomic functions are being executed serially, and the average completion time is $327ns$. Experiments with other atomic functions show similar results.

Task queue operations We conducted experiments to show the average overhead of each enqueue and dequeue operation for our task queue scheme. For the enqueue operation, this was measured by calling an enqueue operation many times without running a kernel on the device. In experiments, each enqueue operation places 120 tasks in the queue. For the dequeue operations, we first preloaded queues with a large number of tasks, and then we launched a kernel that only retrieves tasks from queues, without performing any real work. The average enqueue time is $114.3\mu s$, and the average dequeue time is $0.4\mu s$ when the dequeue kernel was run with 120 TBs, each of 128 threads. Comparing these numbers with Figure 3, it is clear that host-device data transfers account for the major overhead in enqueue operations. For example, 2 PCIe transactions in enqueue operations need approximately $110\mu s$ to finish, which is about 95% of the overall enqueue time. While this seems a very high overhead, by overlapping enqueue operations with the computation on devices, our task queue scheme actually outperforms several alternatives, for a molecular dynamics application, as shown in Section 6,

We also conducted experiments for enqueue operations with varied number of tasks in each operation. We observed that inserting more tasks with one operation only incurs negligible extra overhead, when a single queue can hold these tasks. On the other hand, the average dequeue time is reduced when more TBs are used on the device. For example, when increasing the number of TBs from 16 to 120, the average dequeue time decreases from $0.7\mu s$ to $0.4\mu s$, which is about the time to complete an atomic function. This indicates that our dequeue algorithm actually enables concurrent accesses to the shared queue from all TBs, with very small overhead.

6 Case Study: Molecular Dynamics

In this section, we evaluate our task queue approach using a molecular dynamics application, which exhibits significant load imbalance. We compare the results with other load balance techniques based on the standard CUDA APIs.

6.1 Molecular Dynamics

Molecular Dynamics (MD) [11] is a simulation method of computing dynamic particle interactions on the molecular or atomic level. The method is based on knowing, at the beginning of the simulation, the mass, position, and velocity of each particle in the system (in general in a 3D space). Each particle interacts with other particles in the system and receives a net total force. This interaction is performed using a distance calculation, followed by a force calculation. Force calculations are usually composed of long range, short range and bonded forces. While bonded forces are usually among few atoms composing molecular bonds, the long range and short range forces are gated by a pre-determined *cutoff* radius, under the assumption that only particles which are sufficiently close actually impact their respective net forces. When the net force for each particle has been calculated, new positions and velocities are computed through a series of motion estimation equations. The process of net force calculation and position integration repeats for each time step of the simulation.

One of the common approaches used to parallelize MD simulations is atom-decomposition [24]. Atom-decomposition assigns the computation of a subgroup of atoms to each processing element (PE). Hereafter we assume that the N atom positions are stored in a linear array, A. We denote P as the number of PEs (GPUs in our specific case). A simple atom-decomposition strategy may consist in assigning N/P atoms to each PE. As simulated systems may have non-uniform densities, it is important to create balanced sub-group of atoms with similar number of forces to compute. Non-uniformity is found for instance in gas simulation at molecular level with local variation of temperature and pressure [3]. The computational reason of this load unbalancing is that there is not direct correspondence between the atom position in A and the spatial location in the 3D space. Two common approaches exist in literature to overcome this problem: *randomization* and *chunking*. They are both used in parallel implementations of state-of-the-art biological MD programs such as CHARMM [5] and GROMOS [8]. In randomization, elements in the array A are randomly permuted at the beginning of the simulation, or every certain amount of time steps in the simulation. The array A is then equally partitioned among PEs. In chunking, the array of atoms A is decomposed in more chunks than P, the number of available PEs. Then each PE performs the computation of a chunk and whenever it has finished, it starts the computation of the next unprocessed chunk.

We built a synthetic unbalanced system following a Gaussian distribution of helium atoms in a 3D box. The system has a higher density in the center than in periphery. The density decreases from the center to the periphery following a Gaussian curve. Therefore the force contributions for the atoms at the periphery are much less than those for the atoms close to the center. The force between atoms is calculated using both electrostatic potential and Lennard-Jones potential [11]. We used a synthetic example for two reasons: (1) real life examples are quite complex with many types of atoms and bonds (this would have required the development of a full MD simulator which is out of the scope of this paper) (2) it is very difficult to find real life examples where a particular atom distribution is constant as the simulated system size scales up (therefore it makes very hard to objectively evaluate different solutions with different system sizes).

6.2 Implementations

Using the standard CUDA APIs we implemented two solutions based on the randomization and chunking methods on the array of the atom positions A. As randomization of A may not be optimal for GPU computing (due to the presence of thread divergence, as it is shown later in the rest of the paper), we also implemented a re-ordering scheme based on the spatial information of the simulated system. We also implemented our Task Queue solution, where each task is the evaluation of 128 atoms stored contiguously in the array A. In the rest of this section we explain in more detail the four implementations used both in single and multi-GPU configurations.

The "Solution 1" is the one that is based on the reordering of A using the 3D spatial information of the simulated system, we call this technique *decomposition-sort*. The reordering is perform using the counting sort algorithm [9]. Specifically, the 3D space is decomposed in boxes of size equal to the *cutoff* radius. Then, these boxes are selected using the counting sort algorithm. In this way boxes with more atoms will be selected before boxes with less atoms. Atoms in the selected box are restored back in A starting from the beginning of the array. On each device, a kernel is invoked for simulating one region, where each TB is responsible for evaluating 128 atoms, and the number of TBs is determined by the size of the region. The computation of a time step finishes when all devices finishes their regions. This approach practically performs a partial sorting of atoms based on their interactions with the other atoms in the 3D space. This method reduces the thread divergence as atoms processed in a TB will follow most likely the same control path, which is the most efficient execution way on GPUs. Due to this feature this method is expected to be one of the fasted method for single GPU, however partitioning at multi-GPU level is very difficult. An uneven portioning has to be performed as the cost of distance calculation and force calculation has to be proportionally taken in account. This solution is designed to take advantage of single GPU computing sacrificing multi-GPU load balancing. We use it in a multi-GPU configuration equally dividing A into P contiguous regions, knowing in advance that it will have poor load balance behavior. The objective is to use it as a baseline to compare other load-balancing schemes in the multi-GPU experiments.

The "Solution 2" employs the randomization technique to ensure all atoms are re-distributed in the array A regardless their physical coordinates, therefore to eliminate the load unbalance in the original array. For the multi-GPU implementation the input array is equally divided into P contiguous regions and each device is responsible for computing one region. This solution ensures almost perfect load balance among multiple GPUs. However, it exposes the problem of thread divergence inside a warp, as now atoms with a lot of forces interactions are mixed with atoms with few force interaction.

The randomization procedure and counting sort are performed on the host, and we do not include their execution time into the overall computation time. Note that randomization and counting sort procedure have computational complexity $\Theta(N)$ and therefore can be fairly used in atom-decomposition MD computation which has complexity $\Theta(N^2)$.

The "Solution 3" uses the chunking technique and it is specifically designed to take advantage of both load balancing among multiple-GPUs and thread convergence in TBs. It basically invokes kernels with fine-grained workload on the array reordered with the *decomposition-sort* used in "Solution 1". The chunking technique is implemented as follows. The host process first decomposes the input array into many data chunks of equal atoms. Individual host control threads are then used to communicate with GPUs. Whenever a host control thread finds out that the corresponding device is free (nothing is running on the device), it assigns the computation of a data chunk by launching a kernel with the data chunk information. This device then starts the computation of this data chunk. The host control thread waits until a kernel exits and the device becomes free again, then it launches another kernel with a new data chunk. Since a device only receives a workload after it finishes the current one, this approach ensures a good dynamic load balance. The computation completes when all data chunks are computed.

The "Solution TQ" is the one based on our task queue scheme presented in section 4, where each task is the evaluation of 128 atoms stored contiguously in the array. To exploit the spatial locality in the system, we also perform the *decomposition-sort* procedure before the computation. To efficiently utilize the multiple-GPUs, we employ a simple and efficient load balance approach, based on our task queue scheme. For each time step, the host process first decomposes the computation to tasks and keeps them in a task pool. Then the host process spawns individual host control threads for communicating with each GPU. On each GPU, two queues are used to overlap the host enqueue with the device dequeue. Each queue holds up to 20 tasks. Whenever a task queue of a GPU becomes empty, the corresponding host control thread tries to fetch as much as 20 tasks from the task pool at a time, and sends them to the queue with a single enqueue operation. The kernel was run with 120 TBs, each of 128 threads. Note these configuration numbers used were determined empirically. Then host control threads send HALT tasks to devices to terminate the execution.

Note that in all 4 solutions the same GPU function is used to perform the force computation. Also, before timing, the position data (array A) are already available on GPUs. In this way we can ensure that all performance differences are only due to the load balancing mechanisms employed.

6.3 Results and Discussions

We evaluate the performance of all 4 implementations above with identical input data, for both single- and multiple-GPU scenario.

6.3.1 Single-GPU scenario

Figure 5 shows the normalized speedup of the average runtime per time step over Solution 1, with respect to different system sizes, when only 1 GPU is used in the computation.

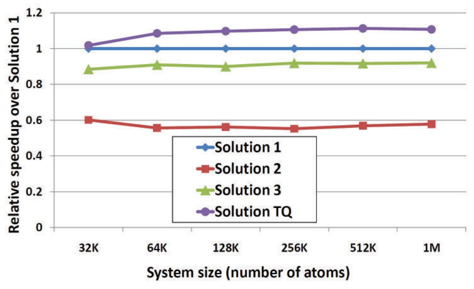

Figure 5: Relative speedup over Solution 1 versus system size (1 GPU)

As discussed in the previous sub-section, unlike other approaches, Solution 2 does not exploit the spatial locality in the system, and thus causes severe thread divergences on the TB. For example, for a 512K atoms system, the CUDA profiler reports that Solution 2 occurs 49% more thread divergences than Solution 1, and its average runtime per time step is 74% slower than Solution 1.

Due to the overhead of a large number of kernel invocations and subsequent synchronizations, Solution 3 cannot achieve better performance than Solution 1 on a single-GPU system, although evaluating a larger data chunk with each kernel invocation can alleviate such overhead.

Solution TQ outperforms other approaches even when running on a single GPU. In principle, for single GPU execution it should behave similarly to Solution 1 (same reordering scheme). However, for a 512K atoms system, the average runtime per time step is $93.6s$ and $84.1s$ for Solution 1 and Solution TQ, respectively. This is almost a 10% of difference.

Regarding this significant performance difference, our first guess was that Solution 1 has to launch much more TBs than our Solution TQ, therefore incurring in a large overhead. However, we experimentally measured that the extra overhead is relatively small. For example, when using a simple kernel, launching it with 4000 TBs only incurs extra $26\mu s$ overhead, compared to launching it with 120 TBs, which does not justify the performance difference between Solution 1 and Solution TQ. Therefore, the only reason lies in how efficient CUDA can schedule TBs of different workload.

To investigate this issue, we create several workload patterns to simulate unbalanced load. To do this, we set up a balanced MD system of 512K atom in which all atoms are uniformly distributed[3]. Since the computation for each atom now involves equal amount of work, TBs consisting of computation of same amount of atoms should also take a similar amount of time to finish. Based on this balanced system, we create several computations following the patterns illustrated in Figure 6. In the figure, P0, $\cdots$, P4, represent systems of specific workload patterns. All patterns consist of a same number of blocks. In Pattern 0, each block contains 128 atoms, which is the workload for a TB (Solution 1), or in a task (Solution TQ). Pattern P0 is actually the balanced system, and all blocks are of equal workload. For the rest of patterns, some blocks are labelled as *nullified*. Whenever a TB reads such a block, it either exits (Solution 1), or fetches another task immediately (Solution TQ). In Solution 1, the CUDA scheduler is notified that a TB has completed and another TB is scheduled. In Solution TQ, the persistent TB fetches another task from the execution queue.

Figure 7 shows the average run time per time step for

[3]We use the balanced system only to understand this behavior, we then return to the unbalanced Gaussian distributed system on the next section on multi-GPUs.

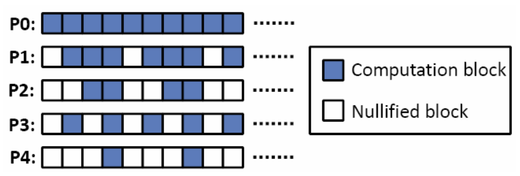

Figure 6: Workload patterns

Solution 1 and Solution TQ, for different workload patterns. To our surprise the CUDA TB scheduler does not handle properly unbalanced execution of TBs. When the workload is balanced among all data blocks, i.e., Pattern P0, Solution TQ is slightly worse than Solution 1 due to the overhead associated with queue operations. However, for Pattern P1, P3, and P4, while Solution TQ achieved reduced runtime, which is proportional to the reduction of the overall workload, Solution 1 failed to attain a similar reduction. For example, for Pattern P4, which implies a reduction of 75% workload over P0, Solution TQ and Solution 1 achieved runtime reduction of 74.5%, and 48.4%, respectively. To ensure that this observation is not only specific to our MD code, we conducted similar experiments with *matrixMul*, a NVIDIA's implementation of matrix multiplication included in CUDA SDK. The results also confirm our observation. This indicates that, when workload is unbalanced distributed among TBs, CUDA cannot schedule new TBs immediately when some TBs terminate, while our task queue scheme can utilize the hardware more efficiently.

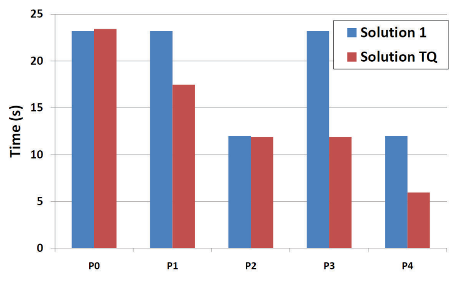

Figure 7: Runtime for different load patterns

6.3.2 Multi-GPU scenario

Figure 8 shows the normalized speedup of the average runtime per time step over Solution 1, with respect to different system sizes, when all 4 GPUs are used in the computation. When the system size is small (32K), Solution 2 achieves the best performance (slightly over Solution 1), while Solution 3 and Solution TQ incur relatively significant overhead associated multiple kernel launching (Solution 3), or queue operations (Solution TQ).

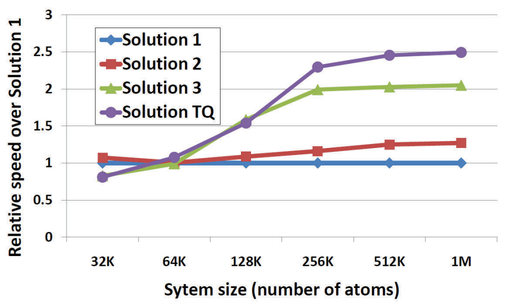

Figure 8: Relative speedup over Solution 1 versus system size (4 GPUs)

As expected, when the system size becomes larger, we observe that solutions incorporated with load balance mechanisms remarkably outperform Solution 1. For example, for a system size of 512K atoms, Solution 2, 3, and TQ are $1.24\times$, $2.02\times$, and $2.45\times$ faster than Solution 1, respectively. Especially, for these large system sizes, Solution TQ achieves the best performance among all solutions, i.e., it is constantly about $1.2\times$ faster than Solution 3, the second best approach.

Figure 9 shows the speedup with respect to the number of GPUs, for a 512K-atom system. From the plot, we observe that, except Solution 1, other 3 approaches achieve nearly linearly speedup when more GPUs are used (they are so close that virtually there is only one curve visible in the figure for Solution 2, 3, and TQ).

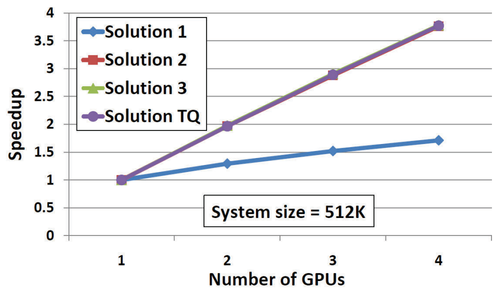

Figure 9: Speedup versus number of GPUs

This is well explained by Figure 10, which shows the actual "busy" time of individual GPUs, when 4 GPUs are used in the computation. As illustrated, for Solution 1, the load among GPUs is extremely unbalanced.

In contrast, other 3 approaches achieve good load balance. However, their absolute times vary. While Solution 2 is effective in terms of load balancing, it is $1.9\times$ slower than Solution TQ for large systems, i.e., 256K and up. As explained earlier, this is because it does not exploit the spatial locality, and therefore significantly increases the

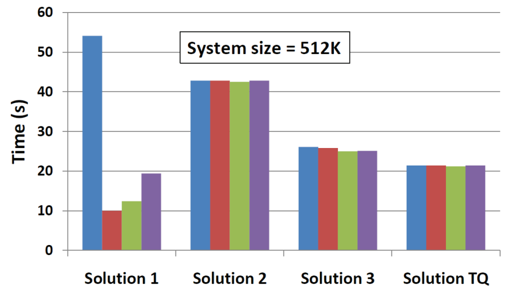

Figure 10: Dynamic load on GPUs

overall runtime.

Solution 3 balances the load among GPUs by assigning fine-grained data chunks with different kernel invocations. However, it does not solve the load imbalance issue within each data chunk; in a kernel invocation (for a data chunk), some TBs may need a longer time to finish than others, due to the unbalanced computation workload among them. Also it involves the overhead of kernel invocations and following synchronizations. One may argue that using larger data chunks can reduce such overhead. We investigated the effect of different sizes of data chunks, and discovered that using small data chunks, i.e., each of 15360 (120x128) atoms, actually achieved the best performance; using larger data chunks introduced load imbalance among devices, and using smaller data chunks simply underutilized the computation power of devices.

In contrast, Solution TQ both exploits the spatial locality, and achieves dynamic load balancing on individual devices, and among devices. Also, it is very easy to integrate the task queue solution with existing CUDA code. For example, given a molecular dynamics CUDA code and the task queue module, a first version of a task queue-enabled molecular dynamics code was obtained within 2 hours.

7 Conclusion and Future Work

We have presented the design of a task queue scheme, which can be employed to achieve dynamical load balance on single- and multi-GPU systems. In a case study of a molecular dynamics application, our task queue scheme achieved excellent speedup and performance improvement over other alternative approaches. We expect that our task queue scheme can be beneficial to other load imbalanced problems on GPU-enabled systems.

Acknowledgments

We want to thank Andrew Márquez for the comments on our early system design, and Joseph Manzano for his feedback on this paper.

References

[1] AMD. ATI Stream. `http://www.amd.com`.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009*, pages 863–874, Delft, Netherlands, 2009.

[3] G. Bird, editor. *Molecular gas dynamics and the direct simulation of gas flows : GA Bird Oxford engineering science series: 42.* Oxford University Press, 1995.

[4] M. Boyer, D. T., S. A., and K. S. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *IPDPS 2009*, pages 1–12, 2009.

[5] B. Brooks and H. M. Parallelization of Charmm for MIMD Machines. *Chemical Design Automation News*, 7(16):16–22, 1992.

[6] D. Cederman and P. T. On Dynamic Load Balancing on Graphics Processors. In *GH 2008*, pages 57–64, 2008.

[7] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *SPAA'95*, pages 74–83, New York, NY, USA, 1995. ACM.

[8] T. Clark, M. J.A., and S. L.R. Parallel Molecular Dynamics. In *SIAM PP'91*, pages 338–344, March 1991.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[10] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS'05*, pages 15–22, New York, NY, USA, 2005.

[11] D. Frenkel and B. Smit, editors. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, Inc., Orlando, FL, USA, 1996.

[12] M. Guevara, C. Gregg, and S. K. Enabling task parallelism in the cuda scheduler. In *PEMA 2009*, 2009.

[13] P. Harish and N. P.J. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[14] M. Herlihy. Wait-free synchronization. *ACM TPLS.*, 13(1):124–149, 1991.

[15] Khronos. OpenCL. `http://www.khronos.org`.

[16] M. D. Linderman, J. D. Collins, H. Wang, and T. H. M. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, 2008.

[17] T. Murray. Personal communication, June 2009.

[18] M. Mller, C.and Strengert and T. Ertl. Adaptive load balancing for raycasting of non-uniformly bricked volumes. *Parallel Computing*, 33(6):406 – 419, 2007. Parallel Graphics and Visualization.

[19] J. Nickolls, I. Buck, M. G., and K. S. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[20] Nvidia. CUDA. `http://www.nvidia.com`.

[21] Nvidia. NVIDIA CUDA Programming Guide 2.3, 2009.

[22] T. Okuyama, F. I., and K. H. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. In *ISPA'08*, pages 284–291, 2008.

[23] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. M. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP'08*, pages 73–82, 2008.

[24] W. Smith. Molecular dynamics on hypercube parallel computers. *Computer Physics Communications*, 62:229–248, 1991.

[25] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC 2008*, pages 1–11, 2008.