GPU-Based *n*-Body Simulation and the PCIe Bus: An Insurmountable Limitation?

David Uliana, Seth Hitefield, Thaddeus Czauski, Ben Shelton
ECE 5504 - Computer Architecture
Virginia Tech
Blacksburg, VA 24061 U.S.A.
Email: {duliana, hitefield, czauski, beshelto}@vt.edu

***Abstract*—We have measured CPU-GPU PCIe link performance, compared it to the theoretically calculated performance, and developed an empirical model to predict the speed of data transfer as a function of transfer size. In addition, we have built our own $O(n^2)$ n-body implementation and analyzed an existing $O(nlogn)$ oct-tree n-body implementation, and we have run both implementations on several cosmological data sets. Our results show that for n-body problems with fewer than 50 million particles, the bandwidth and latency of the PCIe bus have no significant detrimental effect on the performance of these algorithms.**

I. Introduction

The Peripheral Component Interconnect Express bus, commonly referred to as PCI Express or PCIe is a third generation I/O interconnect and has become the de-facto bus used for device communication within todays desktop and server computer architectures. The first PCIe specification was released in 2002 [1], and the PCIe bus aimed to address the shortcomings of the second generation busses, while also preserving their strengths. Additionally, PCIe was intended to create a bus that catered to modern application's needs such as faster transfer speeds and ways to enforce prioritized bus traffic [2].

With PCIe as the bus of choice for connecting devices within personal computers, today's high performance GPU equipment relies heavily on PCIe to communicate with the CPU and main memory. While studies have been performed ([3], [4]) to model GPU performance, none of them have taken an in-depth look at modeling PCIe and its impact on GPU performance and throughput. In our study, we create a PCIe model and describe the effects of PCIe data transfers on GPU kernel performance. We aim to answer the question of whether the GPU system would be constrained by the bandwidth and latency of the GPU- CPU link. We also strive to provide details on why most GPU manufacturers recommend minimizing use of PCIe transfers during kernel execution [5].

For this study we also are creating our own GPU kernel, which is an n-body problem solver. We aim to use our implementation to exercise the GPU and PCIe bus. Additionally, we use our experience designing GPU kernels to provide additional background on the GPU programming model and what role PCIe plays within the GPU programming model.

A. Related Work

With GPUs becoming a topic of greater interest many researchers have begun to analyze and document GPU performance aspects. Zhang and Owens [3] model the behavior of what is happening within the GPU at the hardware architecture level. They analyze GPU kernel performance by creating an instruction simulator and try and determine the types of instructions that dominate the kernels as they execute. Through this method they profile execution time of various instructions, and determine the hardware aspects factoring into a particular kernel's performance. They also look at the internal memory architecture of the GPU, and determine performance factors based on memory access patters and latencies within kernels. Their research does not attempt to model higher level system components such as CPU, system memory, or PCIe bus interactions and their performance on kernel execution.

Schaa and Kaeli [4] model the GPU as a piece of a larger system. In their work each component is treated as a timing element, where the sum of execution times for the CPU, main memory, GPU and bus operations are summed to create a performance factor. Schaa and Kaeli take measurements when using multiple GPUs with the PCIe bus, where they measure upper bounds on contention between multiple devices. Overall, though, they do not take an in-depth look at the factors governing PCIe issues such as overhead, but do acknowledge the effects of contention in their model.

II. Background

A. PCI Express

Because PCIe utilizes a point-to-point model the mechanics of data transfers are very similar to Ethernet. PCIe utilizes messages to send and receive data from other devices on the bus. Additionally, PCIe supports the concept of lanes. Each lane is a conduit through which data is transmitted serially to a receiver. Lanes can be used in a parallel manner, where messages can be striped across multiple lanes to increase transmission bandwidth from one point to another.

B. PCIe Messages

The message format of PCIe is fairly similar to the layered model used in Ethernet. PCIe has three layers [6], from top to bottom: the Transaction layer, the Data Link layer, and the Physical layer. As data is passed through each later, encapsulation data is added to the original message. Each layer also has a message type associated with it, and messages generated from higher layers in the model can pass through lower layers at the receiving end. We do not aim to detail all the messages, but want to highlight the message types used when devices make memory read and write requests. Memory requests and responses are represented by Transaction Layer Packets (TLPs). PCIe also manages flow control and receipt of messages between nodes on the PCIe bus using Data Link Layer Packets (DLLPs). DLLPs are generated in response to receiving TLP messages where the DLLP is used as an acknowledge (ACK) or no acknowledge (NAK) message informing the sender whether any given TLP was properly received.

DLLPs are also used for flow control, where DLLPs are used to give devices send credits. A receiving device periodically sends a DLLP message with credits to other devices, based on the number of free receive buffers the receiving device has. Senders decrement their credit count when sending messages, and will not send messages to a particular destination unless the sender holds enough credits issued by the destination device.

C. PCIe Overhead

PCIe message overhead can come from many sources. For this study, we choose to take a look at transfers between one CPU and one GPU. Thus, we do not take contention into consideration when a data transfer between the CPU and GPU is made. One of the largest overhead costs occurs at the physical layer where an 8b/10b coding scheme is used. Thus, every 8 bits of data are encoded as 10 bits, which results in a 20% reduction in throughput.

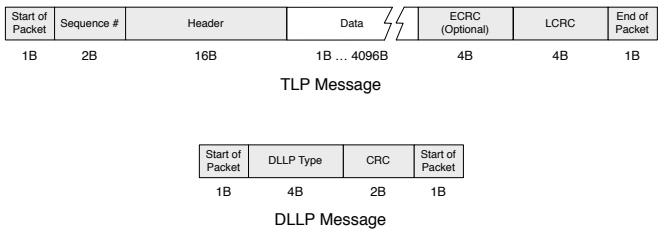

Fig. 1. TLP and DLLP message structures. Fields in grey are considered overhead. Also note that the TLP has an optional ECRC field which is not used in the overhead calculation.

Any data message sent has overhead data added as it traverses from the application down through the PCIe layers. For each TLP message, between 20 to 24 bytes of overhead is typically appended. The variance in the amount of overhead is due to the 32-bit or 64-bit addressing modes that PCIe supports. Due to the abundance of 64-bit computer architectures at the time of publication, the 24 byte overhead is used throughout the remainder of this study. A detailed description of the overhead within the TLP message is described in figure 1.

Since each TLP sent is acknowledged by the lower layers, every TLP received generates a corresponding DLLP ACK or NAK message to be sent. These messages are 8 bytes each, and are considered a part of the overhead for sending TLPs since these messages are automatically generated by the Data Link Layer upon successful receipt of a TLP. Flow control DLLPs are also essential in the transmission of TLP messages, but are not included as part of the overhead calculation. Much of the PCIe literature describes these messages as being sent 'periodically'. This implies that it is up to the hardware implementer to determine the best time to generate these messages. Due to the ambiguity of the frequency with which these messages could be sent, they are not considered in the overhead calculation.

A final source of overhead considered in PCIe data transmissions is the effects of latency as hardware devices fulfill the memory access request. Latency effects occur both in sends and receives from main memory. This access latency is the amount of time that the memory controller takes to fetch or write the data specified by a particular PCIe message. An example of this is demonstrated in the following example [7].

A memory controller receives a PCIe read request from a remote device. The remote device also has set the No Snoop (NS) bit to 0 within the request TLP, indicating that the CPU's cache should be checked for the latest value requested. In this scenario, the memory controller must first snoop on the cache and determine

whether any of the requested data is currently in cache and modified. If so, that data will be written back to system memory. If not, then the memory controller will fetch the requested values from RAM. Once the memory controller has the requested data, then it will be sent via PCIe TLP completion messages back to the requester. During the fetch from either CPU cache or RAM, multiple cache lines may need to be read if the data is not aligned on cache line boundaries, which increases the latency time. The entire time that is taken by the memory controller to fetch the requested values contributes to latency within the PCIe data read request.

D. N-Body Simulation

N-body simulation is the practice of iteratively computing the state of a system of N bodies in a vacuum. Its purpose is to solve the n-body problem–that is, the problem of predicting the motion of a system of celestial bodies in a vacuum, where the only forces at work are gravitational. At each simulation step, the position and velocity of each body is computed by calculating the sum of all gravitational forces acting it. These gravitational force vectors are dependent only on the masses and positions of all the other bodies in the system. The simple, direct-sum algorithm that implements a single iteration of an n-body simulation can be described by the pseudocode in Listing 1.

```
foreach body i in system
   mi = mass of body i
   pi = position of body i
   vi = velocity of body i
   force_sum = 0
   foreach body j != i in system
      mj = mass of body j
      r = distance between body j and body i
      forceSum += G * mi * mj / r
   vi_new = new velocity of body i
   pi_new = new position of body i
```

Listing 1: Pseudocode for the Direct-Sum, $O(N^2)$ N-Body Simulation Algorithm

The difficult task associated with solving this problem is that of accelerating the computation of system state. Typical real-world applications of n-body simulations involve systems made up of many millions or billions of bodies [8]. Systems of sizes such as these fare poorly on simulation implementations such as the direct-sum approach shown in Listing 1, which scales with the number of body-body interactions–N^2, where N is the system size. As a result, methods have been introduced to reduce the number of force calculations per iteration and, therefore, reduce total simulation time.

In [9], Barnes and Hut describe an algorithm that performs n-body simulation with order $O(N \log N)$, as

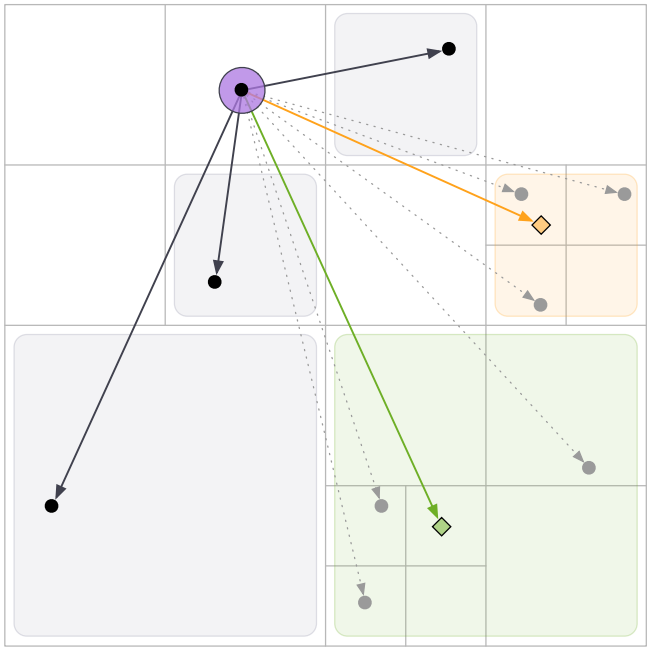

Fig. 2. Illustration of Barnes-Hut Octree Usage using a Quadtree in a 10-Body Planar System

opposed to the order $O(N^2)$ of the direct-sum approach. This is accomplished by storing the system in an *octree*, a tree-based data structure in which each node contains eight child nodes. In this application, the octree is used to recursively partition the three-dimensional system space into octants until each node contains at most a single body. When computing the force vectors acting on a body of interest, the masses and centers of mass of the other tree *nodes* are considered, rather than those of the other *bodies*. This is accomplished by walking down the tree and calculating the forces due to other nodes only when the algorithm reaches either a node containing a single body or a node with a center of mass far enough away from the body of interest. Figure 2 shows a system of ten bodies in a two-dimensional plane divided using a quad-tree–the octree's planar counterpart. Each small circle represents a body in the system, and each arrow represents a force vector. The body of interest is the one highlighted in purple. Each shaded square indicates a node used in the force calculation, and the dashed arrows indicate forces that have been calculated using a node mass and center of mass (indicated by the diamond shapes), rather than individual bodies. Note how, by using the Barnes-Hut approach, the number of force calculations for the total force on the body interest was cut from ten (system size) to five.

The one significant drawback of the Barnes-Hut algorithm is the approximation inherent in the calculation of the gravitational force due to a group of bodies. Treating

several bodies as a single, much larger body can result in enormous error when the group is a small distance from the body of interest. However, this error can be reduced to a negligible value by increasing the minimum distance threshold used to decide between computing a node containing multiple bodies or continuing to descend into the node. By increasing this distance threshold to infinity, the Barnes-Hut becomes essentially the direct-sum, $O(N^2)$ approach discussed earlier. Hence, by careful selection of this threshold, it is possible to maintain a reasonable balance between run-time performance and the accuracy of the results.

E. Burtscher's O(n log n) implementation

For an example Barnes-Hut algorithm implemented on the GPU, we looked at the implementation by Dr. Martin Burtscher, which is described as the first CUDA implementation of the classical Barnes Hut n-body algorithm that runs entirely on the GPU [10]. The code is publicly available at gpucomputing.net [11]. This site advertises that "Version 1.2 (for compute capability 1.2 and 1.3) takes 5.2 seconds to simulate one time step with 5,000,000 bodies on a 1.3 GHz Quadro FX 5800 GPU with 240 cores, which is 74 times faster than an optimized serial C implementation running on a 2.53 GHz Xeon E5540 CPU. Version 2.0 (for compute capability 2.x) is over twice as fast on a C2050" [11]. The Office of Technology Commercialization at UT Austin states that the algorithm is "eight times more energy efficient and six times more cost efficient than [the] CPU alternative" [12].

We chose to evaluate the original CUDA implementation of this algorithm. There is an OpenCL port by a third party, but it is incomplete and unverified, and there may be issues relating to the differences in behavior between the synchronization primitives available in OpenCL and CUDA [13].

The mass of each particle and its initial position and velocity are transferred to the GPU at the beginning of execution, after which no further data transfers occur until all iterations have finished. For each iteration, six separate kernels run in sequence, one at a time. Kernels 1 through 4 compute the bounding box around all the particles, build the oct-tree, compute the center-of-mass for each cell in the oct-tree, and sort the bodies by distance. Kernel 5 performs the actual force calculations. Finally, Kernel 6 computes the new position and velocity for each particle.

There are many GPU-specific optimizations present in this implementation. A few examples are as follows:

- If all the threads in a warp are accessing the same bank of memory, they can "coalesce" up to 16 accesses into a single access. For example, Thread 0 could access position 0, Thread 1 could access Position 1, Thread 2 could access position 2, and so forth, up to 16 threads, all in the same time that it would take one thread to access one memory location. This works well for regular data with little modification, but for irregular, tree-based data such as exists in the Barnes-Hut algorithm, the data structures must be designed to exploit this architectural feature. In this implementation, all trees are implemented in flat arrays, with references to nodes done in terms of array indices rather than pointers.
- All the threads in a warp operate in lock-step, and if any thread in a warp needs to do something that the other threads in the warp do not need to do, then the other threads must all wait until that thread has finished. This condition, termed "divergence" in the paper, is responsible for a huge performance hit. The authors saw a speedup of 5,000 times with a problem size of five million particles when they optimized the force-calculation kernel to avoid it [10].
- When determining whether the sub-cells in each cell are far enough away for their individual influence to be ignored, the GPUs voting function is used. If all the threads in a warp agree that the sub-cell each is working on is far enough away, then the center of gravity of the next-level cell in the tree is used rather than having to calculate for each sub-cell. This is done in a single instruction rather than having to perform a reduction through inter-thread communication.

Why go to such lengths to make the algorithm run entirely on the GPU? The authors themselves recognized the performance limitation inherent in applications that pass data over the PCIe bus between the CPU and the GPU, noting that their approach "avoids slow data transfers over the PCI bus during the simulation" [10]. If the CPU were to assist in building and updating the tree, there would be two possible ways to go about it. First would be to have the CPU move individual pointers and nodes within the tree. Based on our earlier analysis of the operation of the PCIe bus, we know that the worst-case overhead occurs for large numbers of small transactions, which is exactly what would happen in this case, leading to low performance. Second would be to have the CPU set up the tree structure and then copy the entire tree to the GPU. The GPU would then do the force calculations and copy the tree back to the CPU. The advantage here is that larger problem set sizes would be possible, since

the CPU could split the tree into several portions and send each one to the GPU for it to calculate; as it is now, the entire tree must fit into the GPU's memory. In addition, the PCIe memory copy would be done as a single operation, which is much more efficient. The drawback is that in this implementation, the entire tree would need to be passed over the PCIe bus twice per iteration. In addition, the CPU may not be as efficient at updating the tree as the GPU is, since the GPU can do it in a massively parallel fashion. For the problem set sizes Burscher examines, the GPU-only implementation clearly wins out.

F. Data Verification

In addition to analyzing other implementations, we attempted to verify the data for our N^2 application. Our initial goal was to compare outputted data from two implementations at specific time steps and determine the overall error between the results. However, this proved to be more challenging than expected. Unfortunately, unlike a problem such as the Laplace heat equation, no analytical solution exists for problems with more than 3 bodes. Because of this, we could not calculate the expected position for each of the individual bodies in a set, but instead had to rely on either visual or comparative data to validate our implementation.

The first step was to visually validate the implementation. Since it was originally written as a MATLAB script, we could easily plot the data for each step. We accomplished this by generating special cases that should result in an expected behavior. One such case was equally spacing n bodies in a circle around the center of mass with no overall velocity. As expected, the system would collapse to the center of mass, which is when it would break down. This was due to several factors such as the close proximity of the bodies to each other and their increasing velocities. Once they reached the center of mass, the simulation would divide by a very small distance causing them to quickly accelerate out of the system.

The next step was to compare our application to existing implementations. We chose to use two different implementations: the Burtscher Barnes-Hut algorithm, and a N^2 CUDA implementation written by a University of Illinois at Chicago student [14]. Since our implementation calculated the forces acting between each body using the gravitational constant of 6.674×10^{-11}, we needed to use a common dataset with known units ([15] [16]). However, even with this common data, we were unable to validate our results because both of the mentioned implementations do not use standard units or the gravitational constant. For example, the Burtscher application randomly generates its data sets, but does not state what units are being used. Without knowing these values, we could not compare our simulations results with these implementations.

III. Results

A. Modeling PCIe Performance

PCIe Message Efficiency

Knowing the messages used by PCIe, the overheads imposed by various messages, and the message protocol it is possible to put together a model detailing the efficiency of the PCIe messages which is shown in equation 2.

$$D_{total} = \left\lceil \frac{D_{msg}}{P_{max}} \right\rceil \times (D_{TLP} + D_{DLLP}) + D_{msg} \quad (1)$$

$$\frac{D_{msg}}{D_{total}} \quad (2)$$

This equation models the percentage of data within a send or receive message sequence. D_{msg} represents the number of bytes of data to be sent or received. P_{max} is the maximum allowable data payload per message. D_{TLP} and D_{DLLP} are the bytes of overhead imposed by the respective TLP and DLLP messages.

One of the most important terms that the model exposes is P_{max}. The PCIe specification allows the data size per TLP to be anywhere between 1 and 4096 bytes. From an implementation perspective, each device attached to the same PCIe bus has a maximum supported data size within a TLP message [7]. Modern Intel chipsets, such as the Intel 5520 [17] used in TACC's Lonestar cluster, support a P_{max} value of 256B. Other common P_{max} values include 64B and 128B. The maximum payload size is an important factor to consider, as this determines how many messages that D_{msg} must be split into when transferred over PCIe.

Figure 3 shows the message efficiency for D_{msg} values between 1 and 6400B transfers when P_{max} is 64B, 128B, and 256B. Notice how the same data transfer can have much higher performance with higher values of P_{max}. Based on the overheads imposed by the PCIe messages alone, systems which have low P_{max} values such as 64B only achieve 67% message efficiency, where as systems supporting a P_{max} of 256B achieve a higher message throughput with efficiency ratings above 88%. Also notice the sawtooth behavior of data transfers. This occurs since each message has a constant overhead, and the most efficient D_{msg} to P_{max} ratio occurs when a message of size $D_{msg} = P_{max}$ is sent, as this amortizes the fixed overhead penalty over the greatest number of

D_{msg} bytes. Thus, higher values of P_{max} combined with data transfers which send P_{max} bytes are preferred to maximize message efficiency when sending data across PCIe.

PCIe Transfer Time

The second metric we model is the time required to complete a data transfer of size D_{msg}, as shown in equation 3.

$$\frac{D_{total}}{S \times Lanes} + L \tag{3}$$

Equation 3 introduces S, $Lanes$, and L. S is the per PCIe lane speed. Taking into account the 8b/10b coding scheme, $S = 0.25$GB/s for PCIe revision 1.0 and $S = 0.5$GB/s for PCIe devices implementing revision 2.0. $Lanes$ is the number of PCIe lanes which data is being striped across. This is a value between 1 and 32 and must be a power of 2. Finally, L is the latency imposed by the device fulfilling the request, where the latency is the amount of time the device takes to read or write from the memory source addressed in the request. L will vary based on which devices are participating in a PCIe transaction, as well as the ordering in which PCIe requests are issued and fulfilled [18]. Equation 3 represents the ideal scenario where requests are batched together and the overhead imposed by latency is only observed on the initial PCIe data request. Equation 4 represents the effects of latency occurring during every transfer, due to a lack of PCIe message batching being used.

$$\frac{D_{total}}{S \times Lanes} + \left(\left\lceil \frac{D_{msg}}{P_{max}} \right\rceil \times L \right) \tag{4}$$

In order to measure the transfer performance over PCIe, the OCL Bandwidth Test application provided as part of NVIDIA's CUDA SDK [19] was used to send and receive data sets ranging from 1KB to 65,612KB. The test would then produce a bandwidth result, which was converted into transfer time in order to be compared to the model. The test was executed three times, and results were averaged together from two systems. The first system is the TACC Lonestar cluster, which utilizes the Intel 5520 chipset [17] in conjunction with two NVIDIA Tesla M2070 GPUs (Only one Tesla GPU was used in the test). The second system is a desktop machine using the AMD 880G chipset in conjunction with an NVIDIA GeForce GTS 450. The model used 256B for P_{max}, and 800ns for L. D_{msg} was the same set of message sizes as used in the OCL Bandwidth Test, and the overhead values were set to their expected values. The number of lanes used in the model was sixteen, and the per lane bandwidth used the PCIe revision 2.0 value of 0.5 GB/s.

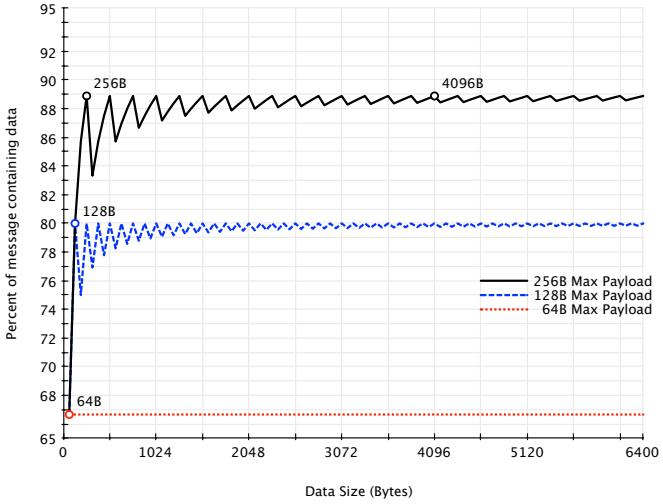

Fig. 3. The efficiency rating of PCIe memory read and write messages. As the maximum payload size of the message increases, the efficiency rating increases as the constant per message overhead is being amortized across more payload data.

The OCL Bandwidth Test also allowed PCIe transfer requests to be batched together, so Equation 3 was used to model the expected PCIe transfer time for the various data sets.

The results from Figure 4 indicate that additional factors besides the PCIe overheads represented in the model are at play. The model predicts almost a 2x faster transfer than the GTS 450 achieves. As Schaa and Kaeli [4] noted, the faster clocked GPUs in their experiment had higher PCIe transfer rates. This explains the small variance between the Tesla and GTS 450 transfer times, as the Tesla has a higher clock frequency than the GTS 450. Determining the source of the difference between the model and both GPUs is significantly more difficult. It is possible that the NVIDIA hardware is incapable of processing data at the rate which PCIe delivers or sends it, although this is difficult to verify due to the proprietary nature of NVIDIA's products.

B. N^2 Implementation

We began our search for data by building our own implementation of the direct-sum $O(N^2)$ n-body simulation algorithm. Though we originally planned to use Nvidia's CUDA programming platform, we settled on the OpenCL framework because of its support for both Nvidia and ATI (now AMD) GPUs. The implementation was designed with the intended use of reading a n-body system initial state from disk, transferring it to the GPU, and performing a specified number of iterations of system state calculation, writing back the intermediate system state from the GPU to the CPU after each

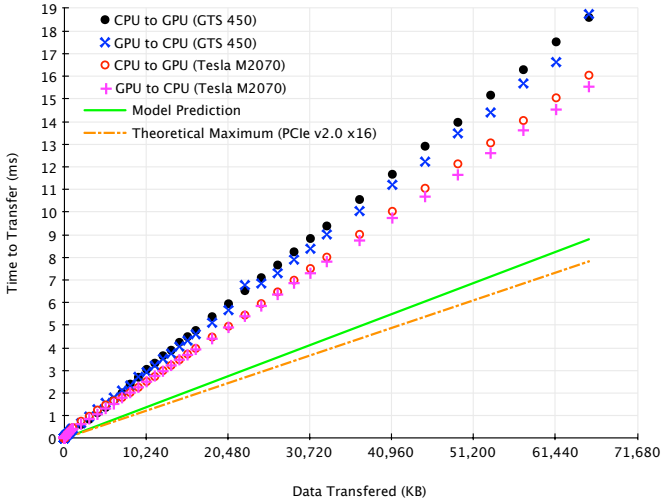

Fig. 4. The time it took to transfer data sets of various sizes via PCIe between the CPU and GPU.

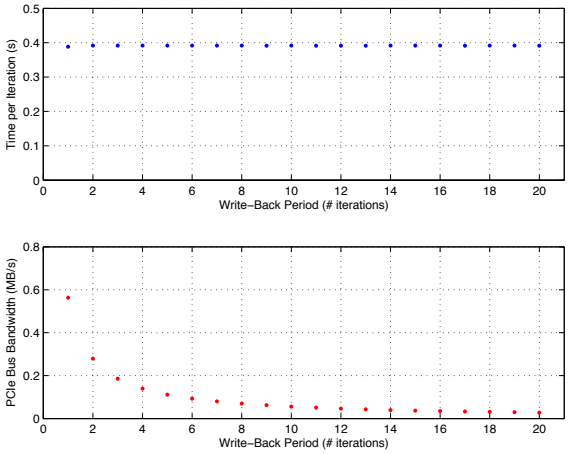

Fig. 6. Time Elapsed per Iteration and PCIe Bus Bandwidth vs. Write-Back Period ($N = 8192$)

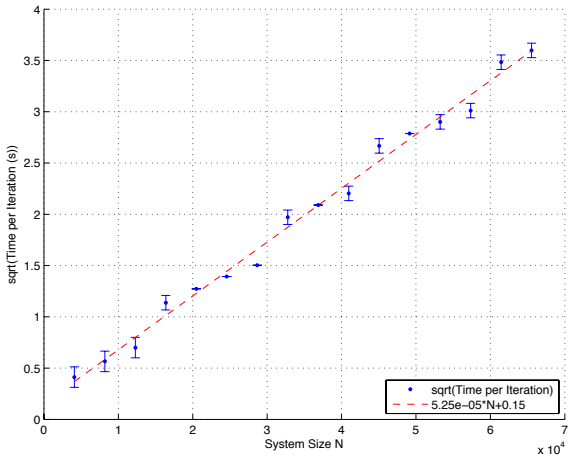

Fig. 5. Square Root of Time Elapsed per Simulation Iteration (sec) vs. System Size N

iteration.

To test the performance of our N^2 implementation, we generated systems of bodies with random positions, velocities, and masses using a MATLAB script. This approach, rather than using real-world systems (e.g. the Milky Way), provided precise control over system size, as well as a way of testing the implementation quickly and easily. Also, because the N^2 algorithm's execution time is dependent only on system size N–the algorithm computes N^2 body-body interactions each time step–and not the system state, the generated systems, despite their randomness, were found to be adequate to test run-time performance of the implementation.

The performance of the N^2 implementation was as expected. Figure 5 shows the square root of the time elapsed per simulation iteration versus the system size. Five runs were performed for each system size, and the error bars indicate the max, min, and median value of each run. Note that the system sizes and times per iteration were fairly small, ranging from 4k to 65k bodies, and 400 ms to 3.6 s, respectively. Regardless, it is clear that the square root of iteration time scales linearly with the system size. This lines up well with our understanding of the N^2 algorithm.

We ran some tests to understand the [possibly limiting] effect of PCIe bus bandwidth on the simulation performance. This was done by running the simulation and varying the frequency of the system state write-backs from once every iteration to once every 20 iterations. Figure 6 shows the results of these tests. It quickly became obvious that, even for a system as small as $N = 8192$, the simulation was computationally constrained, as the PCIe bus had more than enough bandwidth to sustain a write-back after every iteration. This can be explained by the poor scalability of the N^2 algorithm–even a small, 8192 body system requires the computation of 67,108,864 body-body interactions for each iteration.

C. $O(nlogn)$ implementation

In addition to testing our $O(N^2)$ application, we tested an existing implementation based upon the Barnes-Hut oct-tree algorithm [9]. As stated before, the Barnes-Hut algorithm has an order $O(NlogN)$, which reduces the required computation time for each iteration. Reducing the overall calculation time of the simulation allows for more frequent data transfers over the PCIe bus from the GPU to main memory. This allows us to more precisely test the bandwidth and latency of the PCIe bus.

The Burtscher $O(NlogN)$ implementation [10] was

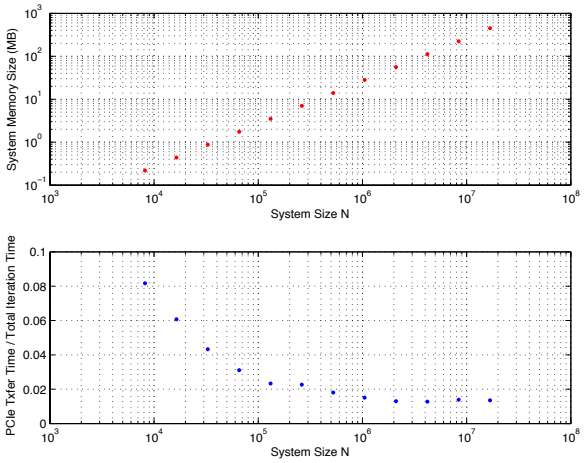

Fig. 7. $O(nlogn)$ Simulation Results

developed with CUDA for NVIDIA GPUs. All of our test runs were completed on the Lonestar cluster at the University of Texas's Advanced Computing Center. Each GPU node within the cluster has exclusive access to two NVIDIA Tesla M2070 GPGPUs. However, for our tests, we only utilized one of the available GPUs because the Burtscher implementation only supports a single GPU. This also helped us to avoid contention on the PCIe bus.

For the tests, we used 2 different methods of generating data. In the first test sets, we chose to use the same randomly generated systems as the $O(N^2)$ simulation since it was easy to generate the large system sizes using a simple MATLAB script. For the second set of tests, we decided to use the implementation's built-in random system generation. The number of bodies to generate was an argument for the implementation, which we provided in the job's batch file. This allowed us to test systems with over 15 million bodies. See the upper plot in Figure 7.

In order to generate a broad range of results, we tested systems sizes between 8192 and 16 million particles. Since each particle consists of 7 floating point values (mass, a position vector and a velocity vector), this amounted to a roughly 460 megabytes for the largest system. We also ran each configuration using two different time steps or iterations: 100 and 1000. To get our final data, we averaged all the results from each individual test.

Each run of the simulation returned timing data for 3 individual tests for that specific configuration. This data included the overall wall clock time and the execution times for each of the 6 kernels in milliseconds. Based upon the timing data, it was easy to see that the majority of execution time was consumed in Kernel 5, which performs the actual force calculations. See Table I. However, we were more concerned with the total time spent transferring data back over the PCIe bus. To measure this, we calculated the ratio of the average transfer time divided by the average wall clock time. The lower plot in Figure 7 shows the ratio versus the different system sizes. This graph shows that as the system size increases the ratio of time spent transferring data back across the bus decreases and levels out at approximately 0.1. This fits with our observations that larger data transfers are more efficient over the PCIe bus.

IV. Conclusion

The PCIe bus between the CPU and the GPU is a limiting factor in performance for many applications, but with careful design and implementation, it is possible to overcome this limitation for the n-body oct-tree problem.

PCI Express

Our model shows that there are many factors which go into determining the efficiency of any PCIe transfer. Designers must take these considerations into account, in an effort to maximize bus throughput otherwise their applications could suffer from paying large overheads from issuing very small PCIe requests. In addition, our model showed that modern GPUs have another, possibly hardware related, bottleneck as both GPUs showed up to 2x slower performance than what our transfer time model estimated. The inconsistent transfer times between different GPU products as well as not being able to saturate the PCIe bus are the most likely reasons that NVIDIA recommends minimizing PCIe transactions. However, it's possible that NVIDIA may revise their recommendations as they develop new products which are able to more effectively utilize the PCIe bus.

N-Body Simulation

Our results also show that even when sending the intermediate set of points/velocities back to the CPU after every iteration, the amount of time spent performing PCIe operations is very low compared to the amount of time spent performing useful computation. However, there are several significant penalties to pay.

The first penalty is time and ease of development. The Burtscher paper estimates that it took two man-months to port an existing C implementation to the GPU and to optimize it to achieve acceptable performance. As discussed earlier, they also state that some of the most significant performance gains came as a result of leveraging GPU-specific features and instruction types. These gains would not be available to the average

TABLE I
EXAMPLE OUTPUT O(N LOG N) (MS)

Wall Clock	6680
Kernel 1	50.5
Kernel 2	654.6
Kernel 3	203.3
Kernel 4	116.8
Kernel 5	4341.0
Kernel 6	50.5
PCIe Transfer	548.5

programmer, or even to one with prior experience in parallel programming, without a steep learning curve.

The second penalty is the limitation on problem set size. The Burtscher implementation only supports a single GPU, which limits the problem size to 50 million particles. A multiple GPU implementation, or a hybrid CPU-GPU implementation, would be necessary to move beyond this limitation, but this would re-introduce the PCI overhead and result in reduced performance.

REFERENCES

[1] PCI-SIG. Pci-sig board of directors approve pci express specifications for high-performance serial i/o.

[2] A. V. Bhatt, "Creating a pci express interconnect," Intel Corporation, Tech. Rep., May 2002.

[3] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, Feb 2011, pp. 382–393.

[4] D. Schaa and D. Kaeli, "Exploring the multiple-gpu design space," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 09)*, May 2009.

[5] *CUDA C Best Pratices Guide*, NVIDIA Corporation, Mar 2011.

[6] R. Budruk, D. Anderson, and T. Shanley, *PCI Express System Architecture*. Addison-Wesley Professional, 2003.

[7] J. Coleman and P. Taylor, "Hardware level io benchmarking of pci express*," Intel Corporation, Tech. Rep. 321071, December 2008.

[8] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulations of the formation, evolution and clustering of galaxies and quasars," *Nature*, vol. 435, no. 7042, pp. 629–636, 06 2005. [Online]. Available: http://dx.doi.org/10.1038/nature03597

[9] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, 12 1986. [Online]. Available: http://dx.doi.org/10.1038/324446a0

[10] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *GPU Computing Gems Emerald Edition*, 2011, ch. 6, pp. 75–92.

[11] M. Burtscher. (2011, Jul.) Cuda implementation of the tree-based barnes hut n-body algorithm. [Online]. Available: http://www.gpucomputing.net/?q=node/1314

[12] ——. A high-performance gpu implementation of the classic barnes hut n-body simulation algorithm. [Online]. Available: http://www.otc.utexas.edu/ATdisplay.jsp?id=872&m=Comp

[13] M. Arsenault. (2011, Aug.) Opencl vs. cuda gpu memory fences. [Online]. Available: http://www.whatmannerofburgeristhis.com/blog/posts/2011/08/04/224357.html

[14] S. J. Lee. (2008, Oct.) nbody simulation in cuda. [Online]. Available: http://www.evl.uic.edu/sjames/cs525/project2.html

[15] J. Dubinsky, J. C. Mihos, and L. Hernquist, "Using tidal tails to probe dark matter halos," *Astrophysical Journal*, vol. 462, p. 576, May 1996.

[16] J. Withagen, "On the collision between the milky way and the andromeda galaxy," Master's thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, 2008.

[17] *Intel 5520 Chipset and Intel 5500 Chipset*, Intel Corporation, March 2009.

[18] "Pci express high performance throughput reference design," Altera Corperation, Tech. Rep. AN-456-1.3, Aug 2010.

[19] *GPU Computing SDK*, NVIDIA Corporation, May 2011. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/4_0/sdk/gpucomputingsdk_4.0.17_linux.run

[20] M. Burtscher. (2011, Jan.) Irregular applications on gpu. Winter Advanced Computing Seminars, Universidade do Minho. Braga, Portugal. [Online]. Available: http://advcomp.di.uminho.pt/uta/wacs2011/IrregulApplGPU_MBslides.pdf

V. Appendix A: Individual Roles

A. David Uliana

Early in the project definition phase, David performed background research into the n-body simulation problem as a GPU workload that stresses the PCIe bus. He is responsible for the design and development of the OpenCL N^2 n-body simulation implementation, as well as the MATLAB script for the generation of random test data sets. He wrote the background section on the n-body simulation problem (section II-D) and the section on the N^2 implementation (section III-B).

B. Seth Hitefield

Initially, Seth was responsible for researching astronomical models and data sets to be used for simulation data and data verification of the N^2 implementation. This included searching for the specific data sets and the respective units for each value. However, due to the lack of large data sets available (most sets had a maximum of 80k bodies), we chose to use randomly generated sets.

In addition, he worked on trying to verify the N^2 implementation. This included researching other implementations that could possibly be used to validate our program. He also ran the simulations and collected data for the Burtscher Barnes-Hut algorithm using the Lonestar computing cluster.

He wrote the data verification section (Section II-F) and the results section for the Burtscher Barnes-Hut implementation (Section III-C).

C. Thaddeus Czauski

Thaddeus performed a literature review, where he gathered information about how the PCIe bus operates. The information gathered ranged from PCIe specifications to implementation notes from vendors who have implemented the PCI Express bus in their products. In addition, the literature review focused on identifying other academic papers and similar works that explore the topic of modeling GPU performance and behavior.

Thaddeus also collaborated with Ben to create the PCIe model. Specifically, Thaddeus derived the message overhead constants and some of the equations used in the model. He also analyzed the model in relation to the experimental feedback gathered from the GPUs.

Finally, he facilitated weekly group meetings from the proposal phase through completion of the project. He wrote the Introduction (section I), PCI Express Background (section II-A), and PCI Express Results (section III-A) sections.

D. Ben Shelton

Ben figured out how to run CUDA and OpenCL code on the Lonestar cluster and collected the initial PCIe bus performance data on this platform. He found the Intel whitepaper on PCIe bus transfer performance [7] and collaborated with Thaddeus to explain and model the PCIe bus behavior. Specifically, he contributed a majority of the background on memory alignment issues that could contribute to additional latency during PCIe reads and writes.

In addition, he got the Burtscher Barnes-Hut implementation up and running and modified it to work with externally-generated data and to send data back over the PCIe bus after every iteration.

He wrote the abstract, the Burtscher Barnes-Hut implementation analysis (section II-E), and the conclusion (section IV) for the final report and typeset the report in LaTeX.