Proceedings of the 11th IEEE Annual International
Conference on Nano/Micro Engineered and Molecular Systems (NEMS)
Matsushima Bay and Sendai MEMS City, Japan, 17-20 April, 2016

Using a Master and Slave approach for GPGPU Computing to Achieve Optimal Scaling in a 3D Real-Time Simulation

Gregory Gutmann[1], Daisuke Inoue[2], Akira Kakugo[2,3], and Akihiko Konagaya[1]

[1]Department of Computational Intelligence and Systems Science, Tokyo Institute of Technology, Yokohama, Japan
[2]Faculty of Science, Hokkaido University, Sapporo, Japan
[3]Graduate School of Chemical Sciences and Engineering, Hokkaido University, Sapporo, Japan

***Abstract*—With the ever increasing computational demand of scientific research and data analysis, there has been a migration towards GPU computing. GPU are now the primary source of compute power in most top supercomputers. But in order to make use of the power programs must utilize more than a single GPU. Within this paper we will explain various approaches we have taken to utilize multiple GPU, and attempt to reach close to perfect scaling on a multi-step simulation. The result of this is having developed our simulation to be computed on a master and slave setup of GPU. Our simulation mentioned is being developed for the purpose of simulating microtubule dynamics on a gliding assay.**

I. Introduction

Molecular biology poses many challenging questions and at times there seems to be no direct way of answering them. Due to this for many years computer simulations have been used to reproduce biological phenomenon with the aim of answering such questions. However, within complex biological systems the number of acting agents easily reaches into the millions, which require vast compute resources. With the recent movement towards the use of general purpose graphics processing units (GPGPU), which now have in the range of 3000 cores per card, it is possible to run such simulations efficiently on a local multi-GPU system.

This paper will look at two multi-GPU algorithms, as well as a single GPU algorithm for a comparison. The first multi-GPU algorithm takes a conventional approach. By computing the minor tasks on a single GPU, and distributing the primary work load to all of the GPUs for further acceleration. The second method uses a master and slave setup of GPUs, one master and two slaves. As well as using a two cycle pattern to enable further concurrency, by overlapping work across update iterations. We have named this algorithm master assist.

The master assist algorithm is an algorithm to organize and control execution of other computational algorithms. Where the computational algorithms are what make up the individual aspects of the model. Therefore, this paper will focus on the organization of algorithm exaction and not the individual computational algorithms.

A. Microtubule Background

A microtubule gliding assay experiment consists of molecular motors being placed on a glass surface, then microtubules added on top. Next the experiment is dosed with ATP and the molecular motors propel the microtubules in various directions. Our interest in microtubule gliding assay comes from the research field of bottom-up nano-scale molecular fabrication, known as molecular robotics[1], [2]. One of the goals in this research area is to form molecular actuators from molecular motors and microtubules. To do this we need to learn more about microtubule dynamics and swarm patterns, and microtubule gliding assay is a good tool for this purpose.

B. Simulation Background

Our work is creating a real-time 3D live-controlled microtubule gliding assay simulation, along with a graphical user interface for the setup of initial simulation parameters[3]. On simulation set up many values can be adjusted such as: microtubule length, speed (energy), simulation space size, various interaction potentials, custom placement of microtubules and hardware related setup. Then during the simulations run time the various interaction algorithms can be enabled and disabled along with having their parameters adjusted. Some of the forces involved have been depicted in figure 1. For the purpose of this paper we will only focus on the algorithm which computes the Lennard-Jones potential between all microtubule segments, as seen in equation 1 and figure 2. In our simulation microtubules are represented by a ball and chain model. Each microtubule consists of a head segment and a series of body segments connected by a spring force as seen in equation 2. **d** standing for the distance between segments and **e** standing for an adjustable equilibrium distance.

$$f(t) = \sum_{j=1}^{a} s * \left(\left(\frac{2}{d-e} \right)^{12} - \left(\frac{2}{d-e} \right)^{6} \right) \tag{1}$$

$$s(t) = (d-e) * c \tag{2}$$

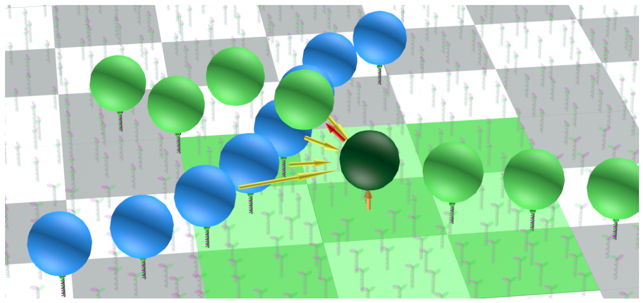


Fig. 1: Yellow arrows represent the Lennard-Jones Potential. Red arrows represent a spring force between segments. Orange arrow represent the force of motor protein holing up the microtubules

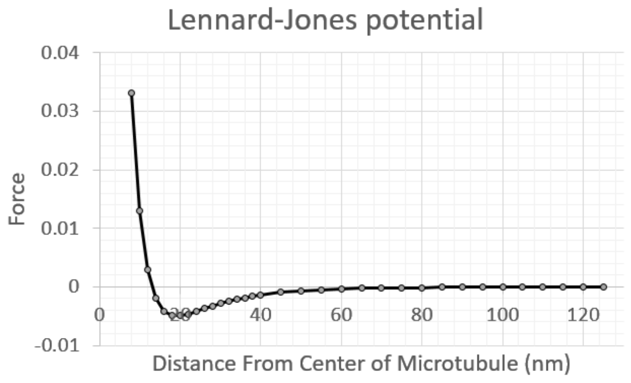


Fig. 2: Example graph of the Lennard-Jones Potential, behavior changes based on the set parameters

C. *Past Optimizations and Distributed Computing Challenges*

The main issues for multi-GPU computing in our simulation are: the separation of rendering and computational tasks, and the parallelization of the computational tasks.

The former technique allows us to use independent GPU for rendering and computation, so that each others performance dose not impact the other. Also, by doing this when looking at the computational performance it is almost as if it is purely a numerical simulation with no other bottlenecks.

The later technique is not so simple due to the spatial sort technique adopted in our simulation; a sort of the microtubule segments based on location[4]. This turns our update procedure into a multi-step process but, greatly reduces the amount of computational work that is needed. Changing the L-J potential algorithm from a $O(n^2)$ to a $O(n)$ limiting behavior, which is vital when simulating into the millions of objects; however using a multi-step process greatly complicates the conversion of the program from using a single GPU to multiple GPU.

This is due to many factors including the following. Data distribution and collection, which is required for distributed computing but very limiting as PCI-E speeds often range between 3-11GB/s in a single direction. A need to maintain an order of tasks; there are some data dependencies between tasks as well as limitations on dividing up the work. More specifically some of the work cannot be easily separated into smaller segments of work to hide memory transfers. Algorithm limitations, such as the radix sort that we chose to use which is only developed to run on a single GPU. However, it is rated as the fastest GPU key sort, CUB a production-quality library for CUDA architecture[6]. While doing a single computation on multiple GPU can be fairly straight forward, the work needed to be done is rarely ever that simple.

The last challenge is maintaining speeds to simulate in real-time, as well as enabling live control. In order to maintain real-time, at our simulation scale, the simulation must be updating at a rate of about 40 times per second. This gives us a time budget of 25ms per update. This constraint limits the number of applicable distributed computing algorithms. For example using larger workloads to better mask overhead for better scaling, as seen in part of Domnguez's paper[7]. Then to enable live control all of the parallel processes must be able to communicate or share parameters in some manner.

II. Traditional Approaches

This section will look at the single GPU algorithm and a basic approach to multi-GPU usage. Also, to better show the differences between the algorithms in this paper we have taken images of the Nvidia Visual Profiler results of each. They are shown below in figure 3, where in each case we are simulating around 100,000 microtubules made up of about 2,000,000 segments using a square simulation space with side lengths equal to the diameter of 2,033 segments. We will refer to the diameter of a segment as 1 unit in this paper. The simulation space and segment count above, results in a density of 0.5 segments/unit^2. In each of the images that we have taken from the visual profiler we have captured two or more updates to show how each update connects to the next in time.

A. *A look at the single GPU work pattern*

In figure 3a we have done visual profiling of the computational work within our simulation's single GPU algorithm. The purple in the image is the radix sort, next in aqua is the L-J Potential, followed by the segment location movement function in blue, and lastly an asynchronous memory copy from the GPU to the host during those. Having two sets of location data, sorted by segment ID, on the compute GPU allows the copy of the location data that has just been updated by the compute GPU to be sent to the host while the compute GPU creates the next version of the location data. This allows the program to hide the cost of the memory transfer to the host. This method is also used in both multi-GPU approaches. As seen in figure 3a the L-J potential is by far the greatest work load, which is one of the reasons why it is our focus for the multi-GPU algorithms.

B. *Straight Forward Multi-GPU*

Our first approach to use multiple GPU was to take our most computationally intensive and least data intensive algorithm, the L-J potential, and only port that over to multi-GPU computing. The L-J potential calculation runs in the range of 1.2 TFLOP/s, but only requires about 66GB/s of data. We have given this algorithm the name distributed L-J potential. After some additional work to organize data transfers, it appeared to work very well. When comparing figures 3a and 3b they follow a similar time line, but with figure 3b taking almost exactly 3x less time for the L-J potential. Figure 3b just includes a little extra data movement time, some copies were able to be overlapped with other operations though. However the sort, data reduction, and movement did not benefit from multiple GPUs. Therefore while the performance scaling of the L-J Potential is impressive, when looking at the simulation as a whole it is just an average performance gain.

Often the solution to this problem in multi-GPU programing is to break up the larger workloads to hide the memory transfers and insure continuous computation[5]. While this would work with our L-J potential kernel, there would be a small percent of added work for organizing buffered data between divisions. But more importantly this method would not hide the other operations. The CUB radix sort is constrained to one GPU, as mentioned. Then the movement kernel for our segments is a data bound operation, with numerous conditions to be checked for each microtubule segment, meaning requiring a great deal of memory in comparison to computation. Thus the cost of distributing all the data and recollecting it for the movement kernel would greatly outweigh the benefit of using multiple GPU for the computation. To give specific details, in the movement kernel the global memory bandwidth is about 400GB/s, whereas the computation is in the range of 90 GFLOPS/s. Very much so a data bound operation. Many of the challenges listed in section C of the introduction also apply here.

1) Differences in Rendering Process: The method of data movement for rendering is also different in our multi-GPU algorithms. As seen in figure 4, when using the single GPU algorithm the compute time is greater than the rendering process. However when using our multi-GPU algorithms the computation has become faster than the basic rendering operations. Due to this we switched to using DirectX interoperability with CUDA, to try to further minimize the rendering process time to better balance it with the CUDA compute time. The results of reducing the data movement time, by using DirectX interoperability instead of a parallel memory copy by the CPU, can be seen in figure 4.

This process works by having each computational update send a copy of updated data to the host. Then as the host rendering process reaches the point of updating the DirectX resources, the host copies the data to the render GPU and launches a kernel to move the memory into the DirectX buffers. When doing this the render thread calls **cudaSetDevice** to switch to the CUDA context to the render GPU. We have done lengthy testing and it seems having two simultaneous threads changing the selected CUDA device is not an issue. However due to there being an unknown time variation between computation updates and render updates, a circular triple buffer was used on the host to synchronize the two tasks use of the location data. The buffer is set up so the faster operation follows the slower process around the buffer, to prevent the processes passing over each other in the buffer. Also, having three buffers insures there are never any data collisions even with unknown task completion timings.

III. Master and Slave Two Cycle Approach

Our solution to the issue of incorporating single GPU algorithms and avoiding excess data movement in a multi-GPU system, is to isolate operations such as these to a single GPU. While sending the computationally intensive algorithm, the L-J potential, to be entirely computed on the two assist GPU. It is referred to as a two cycle approach because simulation updates are spread across two computational iterations, to allow for further concurrency. This algorithm also uses the same method of hiding GPU to host transfers as mentioned in the single GPU algorithm; and the same rendering process mentioned in the first multi-GPU section. The visual profiling of the algorithm can be seen in figure 3c.

A. Importance of the Master GPU

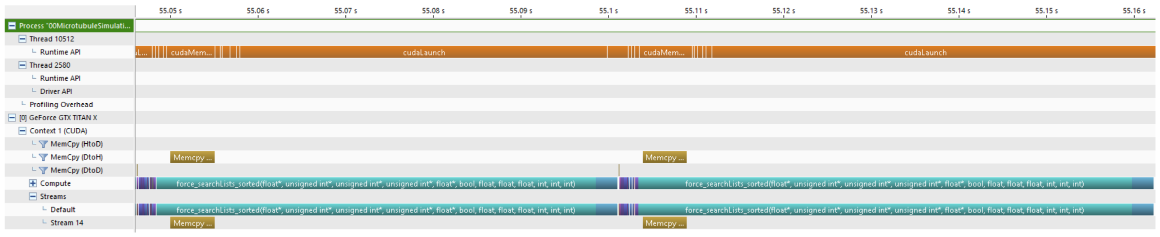
By designating one GPU to be a master GPU, the primary source of data distribution and collection. We are able to do computation that is better suited for a single GPU on the master GPU, while it conducts the data movement for distributed computation. This allows us to overlap work that is limited to a single GPU with work that can be done in a distributed manner. For example libraries or to work that is not suited to distributed computation such as algorithms with very high data requirements.

One of the operations that is done on the master GPU is the sorting of segments. By using the master GPU to sort the segments, we are able to create updated position data every iteration with minimal impact on the performance and zero impact on the performance from the perspective of the force calculations on the assist GPU. Previous works have attempted to minimize the effect of the sort time on performance by only sorting every kth update[5]. The down side of this is the speed of particles per iteration must be taken into consideration, to be sure the particle has not moved out of the cell it was sorted into before the next location sort is done. By sorting every iteration, the cells whose segments are sorted into can be set to the minimum required size for the search area. No buffer is needed for movement between sorts. This is very beneficial to performance because increasing the cell size, to have a buffer, has a squared relation in a 2D sort and a cubed relation in a 3D sort. Due to these relationships this means potentially drastically increasing the unnecessary segments found when looking for neighboring particles or segments.

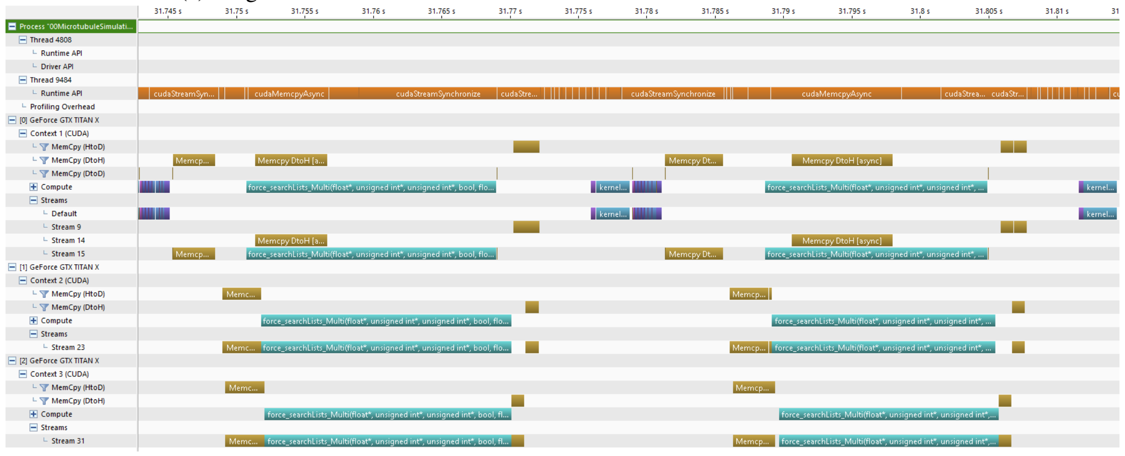
The second operation that we have used the master GPU for is the movement kernel. This is because it is purely a data bound operation as mentioned previously, and would be detrimental to performance to compute in a distributed manner.

B. Order of Executed Operations

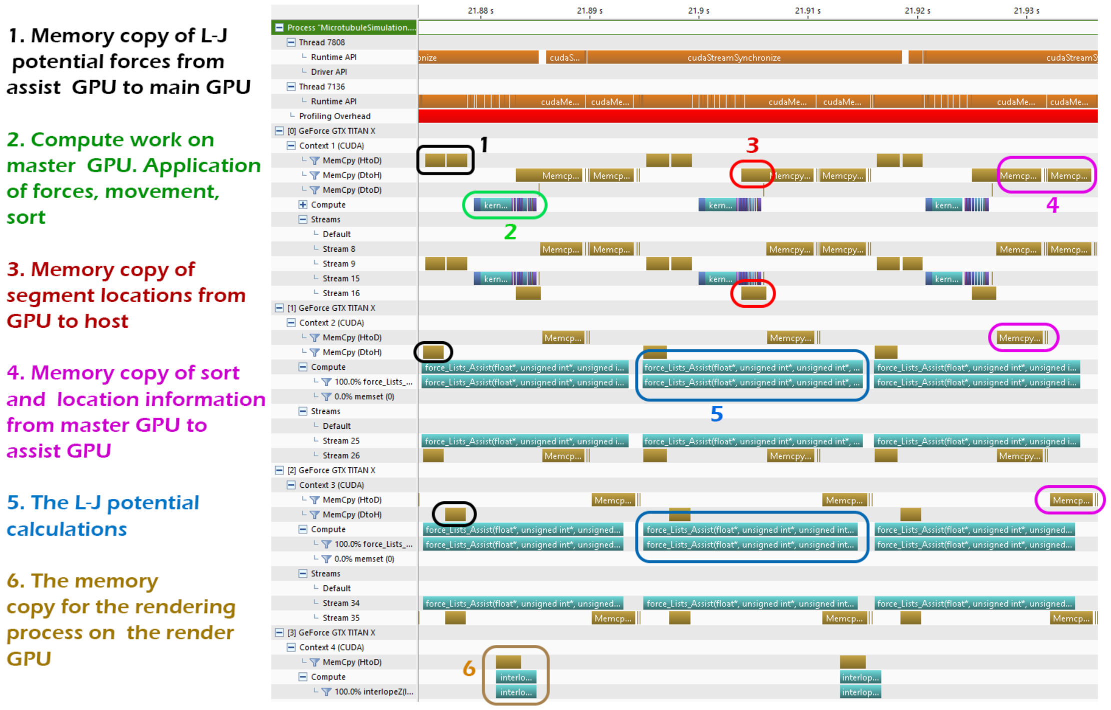
All three methods start off the same. The microtubules are placed in the manner specified by the user prior to running the



(a) Single GPGPU method. The L-J Potential calculations took about 47.8ms



(b) Distributed L-J potential using 3 GPGPU. Main limitations: single GPU work and the memory movement. The gap in work is caused by a synchronization to collect the data from each GPU. The L-J Potential calculations took about 15.7ms per GPU


1. Memory copy of L-J potential forces from assist GPU to main GPU
2. Compute work on master GPU. Application of forces, movement, sort
3. Memory copy of segment locations from GPU to host
4. Memory copy of sort and location information from master GPU to assist GPU
5. The L-J potential calculations
6. The memory copy for the rendering process on the render GPU

(c) Master assist algorithm using 3 GPGPU. The L-J Potential calculations took about 23.1ms per assist GPU

Fig. 3: Nvidia Visual Profiler images of the three algorithms. Time scales in each are slightly different, as they cant be manually set

simulation. Then when the user starts the movement initially no interactions are active by default. The logic of simulation starts by sorting the segments, running the segment movement kernel then sending the data to the CPU. When the various interactions are activated, using the single GPU or distributed L-J potential algorithms, they occur before the movement kernel.

The master assist algorithm also applies the interaction based changes before the movement kernel in the code; however, the interactions applied are from the perspective of the movement of the previous frame. This means that for the master and assist algorithm the application of the interactions are flipped from the perspective of the movement of the segments. The interactions are computed, segments moved, then the interactions are applied. This change has a minute amplifying affect for the Lennard-Jones Potential which was adjusted for.

C. *Memory Management*

On each assist GPU there are two sets of data, one active and one buffer. Each data sets consist of segment locations, sorted by grid location. As well as data which marks the start and end segment of each cell in the sorted list. The GPUs alternate memory sets with each update.

There are a couple reasons for using buffers. One is to overlap memory transfers and kernel execution. In the assist GPU's case this allows the L-J potential to run almost constantly with little idle time between updates. By simultaneously computing the L-J potential on one set of data, while transferring and receiving data on the other data set. The gap, runtime difference, for the master and slave workloads changes based on microtubule density. At very low densities the master GPU's work is the limiting factor, then at very high densities the L-J potential calculations are the limiting factor. At the densities of interest they are about even, which is why we chose the separation of tasks that we did. The work done by each group could be adjusted on a case by case basis.

The next reason is so that the memory transfers over the PCI-E bus can be more spread out, versus being bunched together when computations have finished. This is important because the PCI-E is limited to only one transfer at a time in a given direction. Therefore it is beneficial to keep the memory bus more active to avoid a backup or stall of computation due to memory movement. As a side note while profiling, we found the PCI-E transfers rates often fluctuate between about $3.5GB/s$ and $11GB/s$. This creates occasional performance fluctuations between some updates. The reason for this is currently unknown to us.

IV. Performance

As seen in figure 4 below using multiple GPU can be beneficial in either case, but a rework of the computational work flow was needed to reach closer to the theoretical scaling performance for our simulation. The difference between using a single GPU and three GPU with the distributed L-J potential algorithm resulted in performance gains averaging 1.7x faster. But when using our master assist algorithm there is an average performance gain of 2.9x compared to the single GPU algorithm. This is a difference of 1.7x between the master assist and the distributed L-J potential. For the master assist algorithm, the performance on the two assist GPU is able to maintain an almost constant 1.2 TFLOP/s per card across update frames.

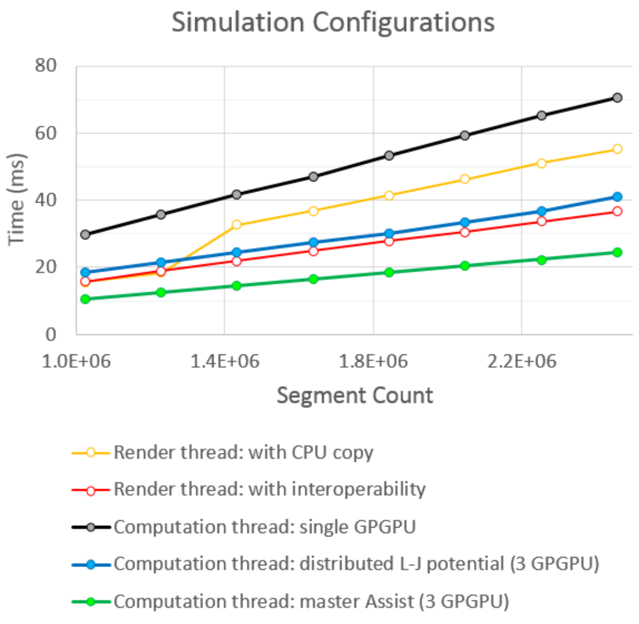


Fig. 4: Performance comparison to the three GPGPU approaches listed above. As well as the rendering thread time which, is executed independently. Density for this was fixed to 0.5 segments/unit

In table I the effect of density can be seen, by the number of segments which can compute the L-J potential, on all their neighboring segments, per second. Then below the table in equation 1, best fit cases can be seen to which estimate the performance for a given density x. Each segment is searching a 15 unit^2 area. For example, at a 0.5 segment/unit density this means about 135 distance checks. Then for this test the L-J potential cutoff distance was set to 7 units, which results in an average of 76 segments computed against[?]. Both search area and the L-J potential cutoff are adjustable. Below the table the cases for understanding density's effect on the master assist algorithm is also listed. A 0.3 segment/unit^2 density is roughly when the master GPU is the limiting factor.

TABLE I: Performance at various densities

Average Density	Segments/sec Computing L-J potential
0.3 segments/unit	133,210,000
0.5 segments/unit	101,233,000
0.7 segments/unit	78,173,000

$$f(x) = \begin{cases} \approx 130,000,000, & \text{if } x < 0.3 \\ 111x^2 - 249x + 197, & \text{if } x > 0.3 \end{cases} \qquad (3)$$

Next in figure 5 the performance effects of various densities can be seen for all three algorithms. The distributed L-J potential has a bit of overhead in data movement, and the performance of this algorithm in relation to the single GPU algorithm doesn't show as much of a gain until higher densities. This is because the larger the distributed work load is the greater the gains when distributing the work to more resources.

The master assist algorithm also has a minimum required compute load as seen there is a plateau from 0.15 to 0.3 segments/unit^2. This is the range at which the work on the master GPU is greater than the compute work on the assist GPU. By isolating management tasks and basic movement to the master GPU, the L-J computation is completely hidden at low densities. Then at the higher densities the reverse is true and the other operations are hidden. This means at the mid-range of densities the two groups of work are equal and overlap well. Currently the master GPU work primarily consists of memory movement; and even though it is saturated with memory transfers in our current model, there is still headroom for added computation on the master GPU.

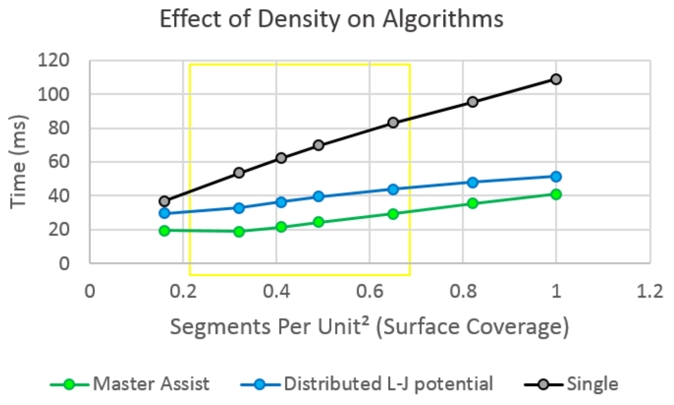


Fig. 5: Algorithm's behavior with varying densities. Yellow region is the area of interest for our microtubule simulation

V. Future Work

At this time there are still a few unknown artifacts within our master assist algorithm. Such as our issues faced when enabling the Tesla Compute Cluster (TCC) mode on our compute GPUs. When enabling it the L-J Potential kernel often took up to 10x as long to compute. Also, different manners of implicit and explicit synchronizations caused this performance drop at times. For example differences between synchronization statements in the code, synchronizations by CUDA based on memory dependency or PCI-E transfer queues, and synchronizations based on CUDA events. We found little to explain drastic changes between the similar operations. But if these can be solved TCC will enable faster memory movement.

Also, as previously mentioned there is still computational headroom on the master GPU. This will allow for the planned addition of more complex segment to segment interactions within microtubule chains without affecting the current performance results.

As for future technologies, Nvidia's upcoming release of NVLink will greatly aid in multi-GPU computation by offering faster GPU-GPU transfers. A greater memory transfer speed in our algorithm would mean the master GPU's limiting factor may not be memory movement. This would reduce the performance plateau caused by the master GPU at lower densities, or allow for adding additional memory movement for additional distributed work.

VI. Conclusion

We have found that when only porting the most intensive task within a simulation to multi-GPU computing the gains can be very small, even if that tasks performance scales perfectly across all GPU used. Within multi-step programs often a redesign is needed, which takes into consideration all of the limiting factors of each process, in order to reach closer to the expected gains from adding additional hardware. Our master assist algorithm has done this for our problem by constraining single GPU algorithms and memory managment to one GPU, and using the remaining GPU for distributed computation. By the use of our master assist algorithm we have achieved ideal scaling, about a 2.9x gain in simulation speed with 3 GPU. This has allowed for simulating 2.4 million particles within our time budget of 25ms.

Acknowledgment

This work was supported by a Grant-in-Aid for Scientific Research on Innovation Areas Molecular Robotics (No. 24104004) of The Ministry of Education, Culture, Sports, Science, and Technology, Japan.

References

[1] Murata, S., Konagaya, A., Kobayashi, S., Saito, H., & Hagiya, M. (2013). Molecular robotics: A new paradigm for artifacts. New Generation Computing, 31(1), 27-45.
[2] Hagiya, M., Konagaya, A., Kobayashi, S., Saito, H., & Murata, S. (2014). Molecular robots with sensors and intelligence. Accounts of chemical research, 47(6), 1681-1690.
[3] Gutmann, G., Inoue, D., Kakugo, A., & Konagaya, A. (2014). Real-Time 3D Microtubule Gliding Simulation. In Life System Modeling and Simulation (pp. 13-22). Springer Berlin Heidelberg.
[4] Green, S. (2010). Particle simulation using cuda. NVIDIA whitepaper.
[5] Rustico, E., Bilotta, G., Herault, A., Del Negro, C., & Gallo, G. (2014). Advances in multi-GPU smoothed particle hydrodynamics simulations. Parallel and Distributed Systems, IEEE Transactions on, 25(1), 43-52.
[6] Merrill, Duane. "CUB Documentation" CUB: Main Page. NVIDIA CORPORATION, 2011.
[7] Domnguez, J. M., Crespo, A. J., Valdez-Balderas, D., Rogers, B. D., & Gmez-Gesteira, M. (2013). New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. Computer Physics Communications, 184(8), 1848-1860.