GPU based Position Based Dynamics for Surgical Simulators

Doga Demirel[a*][0000-0002-8270-1163], Jason Smith[a], Sinan Kockara[b][0000-0002-5881-1653], Tansel Halic[c] [0000-0002-2558-4001]

[a]Department of Computer Science, Florida Polytechnic University, Lakeland, Florida
[b]Department of Computer Science, Lamar University, Beaumont, Texas
[c]Intuitive Surgical, Atlanta, Georgia
*ddemirel@floridapoly.edu
jason@smith.software, skockara@lamar.edu,
Tansel.halic@intusurg.com

Abstract. Position Based Dynamics is the most popular approach for simulating dynamic systems in computer graphics. However, volume rendering with linear deformation times is still a challenge in virtual scenes. In this work, we implemented Graphics Processing Unit (GPU)-based Position-Based Dynamics to iMSTK, an open-source toolkit for rapid prototyping interactive multi-modal surgical simulation. We utilized NVIDIA's CUDA toolkit for this implementation and carried out vector calculations on GPU kernels while ensuring that threads do not overwrite the data used in other calculations. We compared our results with an available GPU-based Position-Based Dynamics solver. We gathered results on two computers with different specifications using affordable GPUs. The vertex (959 vertices) and tetrahedral mesh element (2591 elements) counts were kept the same for all calculations. Our implementation was able to speed up physics calculations by nearly 10x. For the size of 128x128, the CPU implementation carried out physics calculations in 7900ms while our implementation carried out the same physics calculations in 820ms.

Keywords: Position Based Dynamics, GPU

1 Introduction

Position Based Dynamics (PBD) is the most popular approach for simulating dynamic systems in computer graphics [1]. Utilizing collision constraints to manipulate the positions of points in 3D space allows for simple physics calculations on large models [2]. Attempts to parallelize PBD saw specialized graphs and graph coloring to divide sets of constraints into independent sections, which can provide calculation time decreases of several orders of magnitude [3]. However, large and complex models are still computationally expensive, even with graph coloring. Reduced models can somewhat solve this problem by providing low dimensional copies of the model on which the PBD calculations are run and then projected back to the higher quality, displayed model [4].

As complex PBD models allow for complicated constraints, calculations can be slow on large models. In addition, specialized solvers can cause objects that would otherwise interact with one another to pass through. Utilizing a unified solver function on particle-based simulations allows many different substance types to interact [5].

Additionally, utilizing PBD provides excellent advantages when using volumetric models, such as VEG files. These files already contain information used by the PBD model, which allows modelers to define parameters for how the physics will affect their models, such as hardness and bend constraints [6].

We implemented GPU-based Position Based Dynamics (PBD) to iMSTK, an open-source toolkit for rapid prototyping interactive multi-modal surgical simulation, and compared it with available GPU-based PBD solvers. As iMSTK was formerly only able to do PBD calculations utilizing the CPU, it could not handle deforming large models in real-time [4]. Using the vast multithreading ability of the GPU allows us to efficiently manage significantly larger models with little to no performance impact.

Pan et al. [7] proposed a PBD-based Virtual Reality simulation framework for cholecystectomy in the paper. The authors used graph coloring to solve PBD constraints in parallel to satisfy organ deformations. For 22,650 tetrahedrons, they improved the time cost per step from 28.51ms-29.10ms (PBD on CPU) to 12.20ms-12.35ms (parallel PBD on GPU). Berndt et al. [8] proposed a PBD approach to simulate electrosurgery and interactive cutting. This study uses PBD to model the objects and their dynamics used in the surgery. Berndt et al. compared their PBD results with Pan et al. [9], where Pan et al. used extended PBD to simulate soft bodies. For 2,385 and 4,079 elements, simulation in [8] resulted in 4.2ms and 5.3ms, while [9] resulted in 8.1ms and 19.3ms, respectively.

In [3], a CUDA-based PBD is implemented using graph coloring for interactive deformable bodies. In [3], for 16,000 particles (only quantitative result with stretch, bend, and tetrahedral constraints), the average FPS for a single-core CPU was 15, while for a multi-core CPU and GPU, the reported results were 45 and 326, respectively. However, using graph coloring and solving with Gauss-Seidel [10] causes an imbalanced amount of work for each kernel due to the number of constraints for each color being different. Another work that utilizes Gauss-Seidel iteration with PBD is [12]. The study noted that a model with ≈3,000 vertices would run with 7-8FPS on the CPU while the FPS would increase to 42-43 on the GPU. In [11], authors introduced Vivace, a CUDA-based PBD is implemented using graph coloring for interactive deformable bodies. For 30,000 vertices, 52,000 elements, and 150,000 constraints, the solver could run at 15ms per frame. Due to the limited number of iterations, Vivace provides approximate results, which leads to artifacts.

2 Methods

Utilizing NVIDIA's CUDA toolkit [13], we took the physics calculations that would otherwise have been done on the CPU and moved these calculations to the GPU. As the CUDA library does not support the standard library vectors, these values must be copied into individual arrays before being copied to the GPU. Once the data is on the GPU, the vector calculations can be carried out utilizing GPU kernels which behave similarly to the CPU calculations. However, special care was required to ensure that another thread did not overwrite the data used in one calculation. Graph coloring cannot be utilized because the GPU has much higher parallelization than the CPU. This was accomplished by allocating a second array and writing all the results into the second array.

Tasks sent to the GPU for calculation are split into blocks of threads. By adjusting how many threads we want per block, we can control how the tasks are divided on the GPU. As we are given access to the thread (T) and block number (B) in our kernels (functions that run on the GPU), we can treat these values similarly to indexing using 2D coordinates ($T + B * width$), where $width$ is the block dimension. In addition, we compare the mass of each point with DBL_{min} (the smallest number a double can represent that is greater than 0) to determine if we need to do any calculations.

In our velocity kernel (as seen in Table 1), we calculate the velocity by taking the current position ($\boldsymbol{P_i}$) and subtracting the previous position ($\boldsymbol{P_{i-1}}$), then dividing that vector by the delta time (Δt).

In the integration kernel (as seen in Table 2), we start by adding the gravity constant and acceleration multiplied by Δt. Then, we copy our coordinates for the next frame's velocity calculation ($\boldsymbol{V_i}$). Finally, we update the position based on the velocity, delta time, and viscous damping coefficient *(VDC).*

Table 1. GPU Velocity Kernel

```
$i \leftarrow T+B*width$                          ▶Initialization
if $|M_i| > DBL_{min}$ then
    for all index $i \in M$ do
        $V_i \leftarrow (P_i - P_{i-1}) / \Delta t$        ▶Calculate Velocity
```

Table 2. GPU Integration Kernel

```
$i \leftarrow T+B*width$                          ▶Initialization
if $|M_i| > DBL_{min}$ then
    for all index $i \in M$ do
        $V_i \leftarrow V_i + (A_i + g) * \Delta t$          ▶Calculate Velocity using Acceleration
        $P_{i-1} \leftarrow P_i$                          ▶Calculate Previous Position
        $P_i \leftarrow P_i + (1 - VDC) * V_i * \Delta t$    ▶Update Position
```

3 Results

Our PBD implementation utilized NVIDIA’s CUDA, a General-Purpose Graphics Processing Unit (GPGPU) library, which allows us to utilize graphical processing hardware to do general-purpose computing tasks, in this case, PBD physics calculations.

Utilizing two different computers, we tested each of the different settings with different-sized models to determine which were most optimal and which number of threads per block was optimal (as seen in Table 3 for hardware specifications).

Table 3. Hardware specifications for each computer

CPU: Intel i7 4790s RAM: 16GB DDR3 GPU: NVIDIA GTX 750	a. PC #1
CPU: Xeon E-2144G RAM: 16GB DDR4 GPU: NVIDIA Quadro P2000	b. PC #2

In addition to utilizing iMSTK, we tested using an open-source example project, setting the number of particles to the same number as the iMSTK simulation. We utilized Macklin et al. 's work [5] as a baseline for how fast we should expect our project to run, quickly identifying if the hardware or software is the cause for any perceived stutter or slowdown.

Time computation tests were performed first on PC #1, running CPU and GPU-based tests (as seen in Fig. 1) while comparing the computation time in microseconds (μs) with different sizes, ranging from 8x8 to 128x128. Tests utilizing the GPU were split into two categories, GPU with copy and GPU without a copy.

The results with copy included copying the data from the CPU to the GPU, doing the calculations, and then copying the results back from the GPU to the CPU. This shows the overhead of copying data through the PCIe interface to which the GPU is connected.

The results showed that the GPU without copy is faster than the CPU for larger sizes (32x32, 64x64, and 128x128) but is slower for smaller sizes (8x8 and 16x16). The GPU with copy is slower than the GPU without copy but still faster than the CPU for larger sizes (64x64 and 128x128). However, the computation time on the CPU increases much faster than GPU with data copy as the size increases.

For all sizes, our implementation on GPU with copy and GPU without copy showed at least 2x improvement over the performance in the Macklin et al. example. At 128x128, our GPU implementation with data copy showed a 2.7x improvement over Macklin et al.’s model and a 1.98x improvement over the CPU. For the same size, our GPU implementation without data copy showed a 4.66x improvement over Macklin et al.’s example and a 3.41x improvement over the CPU. When comparing GPU with and without data copy, data copy overhead decreased the GPU performance by 0.57x to 0.62x.

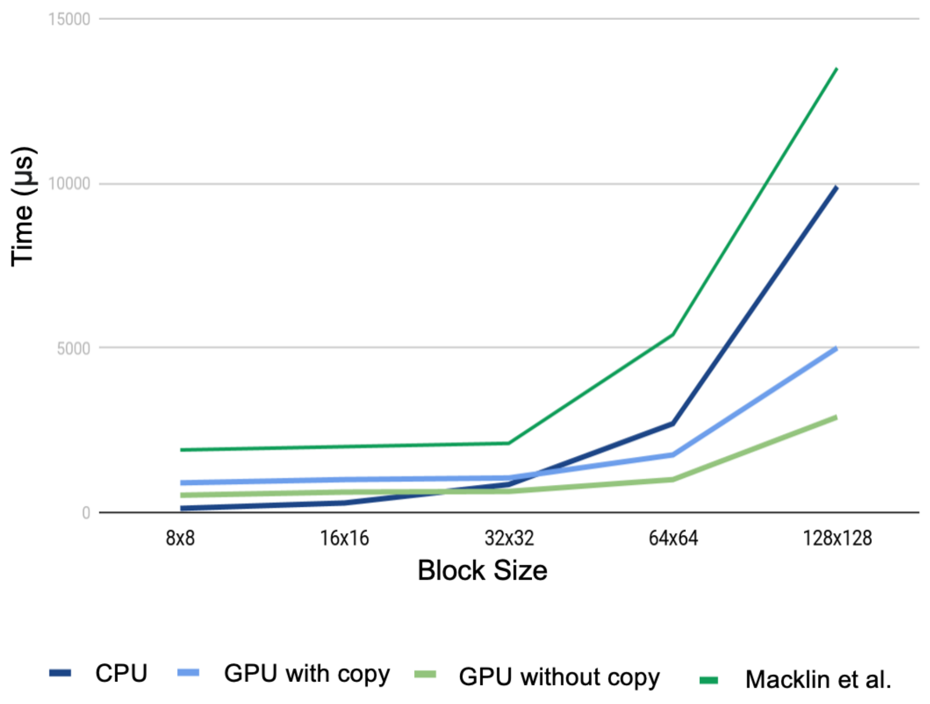

Fig. 1. Timing results from PC #1

Next, the same suite of time computation tests was performed on PC #2 (as seen in Fig 2). Similar to the results from PC#1, for PC #2, smaller sizes (8x8 and 16x16) CPU outperformed the GPU with and without data copy, and Macklin et al.'s example. Comparable to PC #1, in PC#2, the CPU had the most significant increase in computation time from 64x64 to 128x128 with 3.16x.

For Macklin et al. and GPU with data copy, the increase was 2.6x, while for GPU without data copy, the increase was 2.65x. Unlike PC #1, in PC #2, Macklin et al.'s example outperformed CPU at 128x128 by 17.7%, while at 64x64, the time computation results were equal at 2500 microseconds. For PC #1, at 64x64, CPU computation results were 50% less than Macklin et al.'s example.

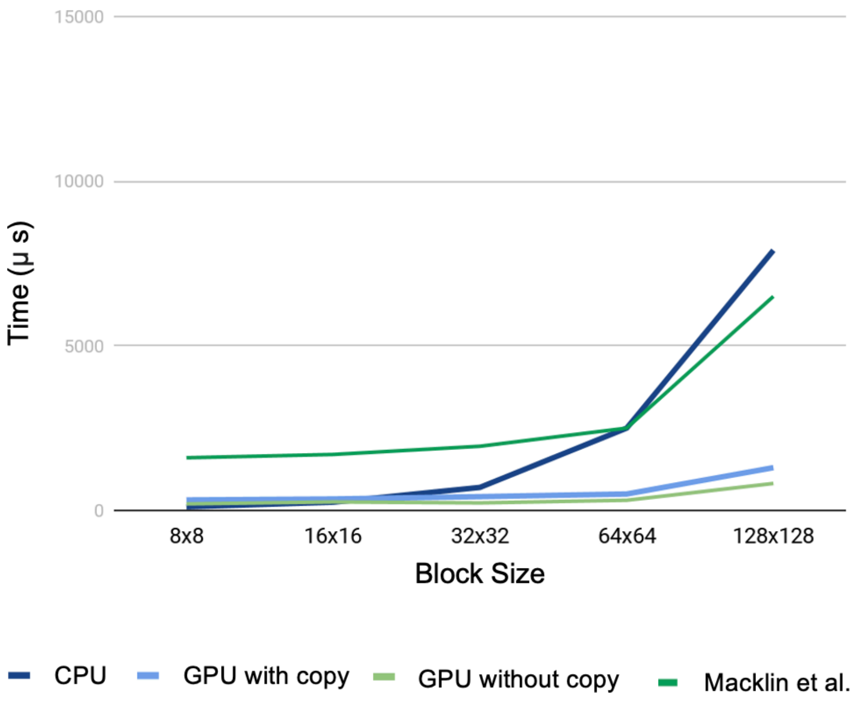

Fig. 2. Timing results from PC #2

We tested our implementation (GPU without copy) with different block sizes and threads per block (TPB). The results showed that, generally, the time taken to perform the operation decreases as the block size and TPB increase. However, the differences between 32 and 64 TPB were all less than 10%. The most considerable difference between 32 and 64 TPB was recorded for 8x8 block size at 8.5%. For the 32x32 block size, 32 and 64 TPB performance was 240µs.

For comparison of 8 and 16 TPB, block sizes 64x64 and 128x128 had a performance difference of 31.1% and 41.43%, respectively. The performance results for the smallest block size of 8x8 were 210µs for 8TPB, 200µs for 16TPB, 70µs for 32TPB, and 64µs for 64TPB. The performance results for the largest block size of 128x128 were 1400µs for 8TPB, 820µs for 16TPB, 570µs for 32TPB, and 560µs for 64TPB.

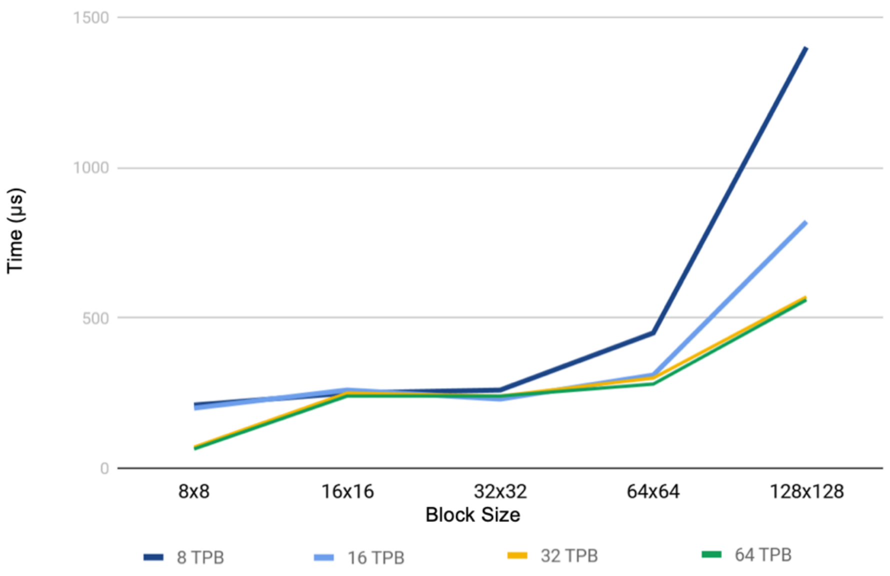

Fig. 3. PC #2 performance for different block sizes and number of threads per block

Finally, tests were run on PC #2 to determine the effects of changing the number of TPB, as seen in Fig 4. These tests utilized a volumetric dragon model of various sizes and a source VEG (3D volumetric mesh) file [6]. The model had 959 vertices and 2,591 tetrahedral mesh elements, the counts were kept the same for all calculations.

The results indicate that the processing time decreases as the number of TPB increases. One exception is the 128 TPB case (except for VEG File), where the processing time increases. This is due to the overhead of managing a large number of threads. For the VEG file, 128 TPB had the fastest performance at 16,000µs, while 64 TPB was the slowest at 30,500µs.

It is also worth noting that the processing time increases as the block size increases. This is likely because larger blocks require more processing power to handle a more significant number of data points. For the model block size of 70x42x42, the fastest TPB was 32 with 26,100µs, while the slowest was the CPU with 253,000µs.

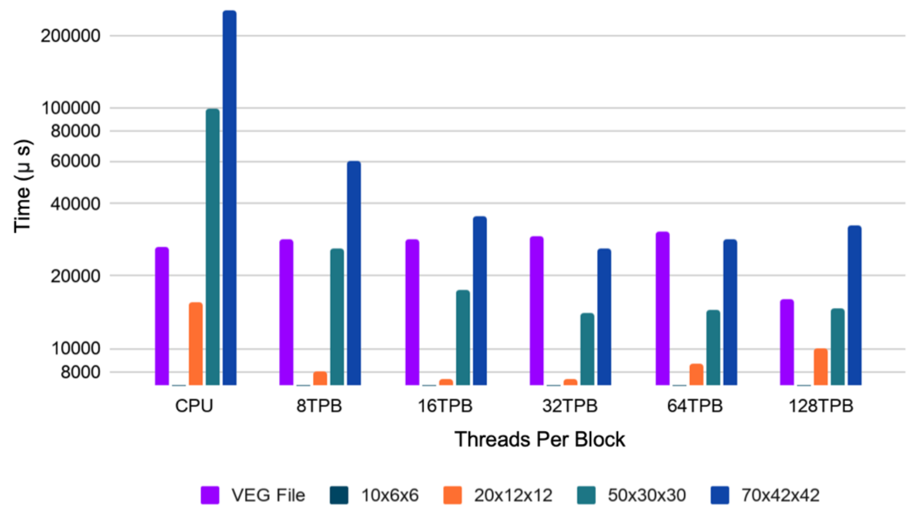

Fig. 4. Timing results (PC #2) from testing showing increased performance when using CUDA.

Acknowledgments. This project was supported by grants from the National Institutes of Health (NIH)/ NIBIB 1R01EB033674-01A1, 5R01EB025241-04, 3R01EB005807-09A1S1, and 5R01EB005807-10.

4 Conclusion

Volume rendering with linear deformation times is another challenge in virtual scenes. PBD is the most popular approach for simulating dynamic systems in computer graphics. We implemented GPU-based PBD to iMSTK, an open-source toolkit for rapid prototyping interactive multi-modal surgical simulation, and compared it with available GPU-based PBD solvers. We successfully showed that utilizing the GPU is 10x faster than using the CPU for the PBD calculations, as the GPU is optimized for highly multithreaded workloads.

References

1. J. Bender, M. Müller, and M. Macklin, “A survey on position-based dynamics, 2017,” Proc. Eur. Assoc. Comput. Graph. Tutor., pp. 1–31, 2017.
2. M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, “Position based dynamics,” J. Vis. Commun. Image Represent., vol. 18, no. 2, pp. 109–118, 2007.
3. M. Fratarcangeli and F. Pellacini, “A gpu-based implementation of position based dynamics for interactive deformable bodies,” J. Graph. Tools, vol. 17, no. 3, pp. 59–66, 2013.

4. J. Yan, S. Arikatla, and A. Wilson, "Fast deformation dynamics using model order reduction in iMSTK September 9, 2020".
5. M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, "Unified particle physics for real-time applications," ACM Trans. Graph. TOG, vol. 33, no. 4, pp. 1–12, 2014.
6. F. S. Sin, D. Schroeder, and J. Barbič, "Vega: non-linear FEM deformable object simulator," in Computer Graphics Forum, 2013, vol. 32, no. 1, pp. 36–48.
7. J. Pan et al., "Real-time VR simulation of laparoscopic cholecystectomy based on parallel position-based dynamics in GPU," in 2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 2020, pp. 548–556.
8. I. Berndt, R. Torchelsen, and A. Maciel, "Efficient surgical cutting with position-based dynamics," IEEE Comput. Graph. Appl., vol. 37, no. 3, pp. 24–31, 2017.
9. J. Pan, J. Bai, X. Zhao, A. Hao, and H. Qin, "Real-time haptic manipulation and cutting of hybrid soft tissue models by extended position-based dynamics," Comput. Animat. Virtual Worlds, vol. 26, no. 3–4, pp. 321–335, 2015.
10. J. P. Milaszewicz, "Improving jacobi and gauss-seidel iterations," Linear Algebra Its Appl., vol. 93, pp. 161–170, 1987.
11. M. Fratarcangeli, V. Tibaldo, and F. Pellacini, "Vivace: A practical gauss-seidel method for stable soft body dynamics," ACM Trans. Graph. TOG, vol. 35, no. 6, pp. 1–9, 2016.
12. O. Cetinaslan, "Position-Based Simulation of Elastic Models on the GPU with Energy Aware Gauss-Seidel Algorithm," in Computer Graphics Forum, 2019, vol. 38, no. 8, pp. 41–52.
13. M. Fatica, "CUDA toolkit and libraries," in 2008 IEEE hot chips 20 symposium (HCS), 2008, pp. 1–22.