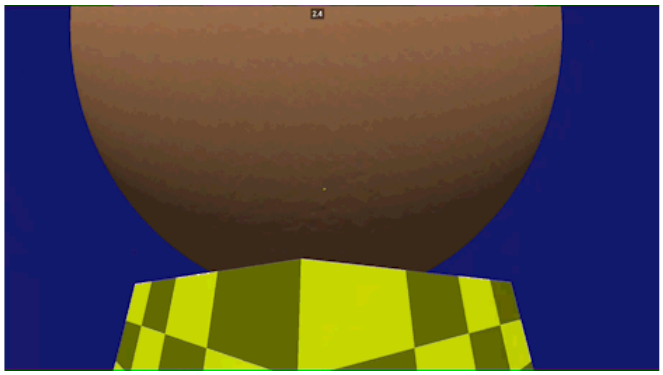
Jolt Physics Engine Optimizations Using CUDA Parallel Programming

Authors: David Khachatryan, Ege Yilmaz



Explore this project on GitHub

Introduction

Physics Engines are simulation tools that are used in a myriad of industries to emulate physical behavior of and between objects. In game engines, they are commonly run as a parallel simulation to the game simulation, though with a lesser frequency of cycles compared to that of the game. In Robotics, physics engines such as Gazebo, approximate real life conditions for testing and development in ROS2, as well as allow for the training of Reinforcement Learning agents that progressively become better at completing physical tasks, such as doing a backflip or learning how to climb. In Virtual Reality/Augmented Reality Physics Engines are crucial tools used to emulate real-time environments and are utilized in training professionals in medical and military fields. Recently, even those in the animation and film industries utilize game engines as a user interface for interacting with the physics engine, where many custom visual effects are created by utilizing particle physics.

PROF

Many of the required calculations for such simulations, however, can be quite expensive, which is a big bottleneck for such systems. As simulations of real life scenarios, it is crucial that physics engines maintain the illusion of continuity, and slow CPU cycles can cause breaks and pauses in the graphical representation of the simulation. Broad-phase collision detection, constraint solvers, fluid and particle simulations, and soft body and cloth simulations are among the most computationally expensive operations within these systems, and are commonly the culprits for slowdowns in the simulation. In the background section we will examine these sources further to explain why we chose soft-body systems as our optimization target.

Jolt, more specifically, is a very popular open source engine that is in fact used within the Godot game engine to conduct physics simulations [1]. Online, it has a reputation for being well-documented and actively supported by its developer, Jorrit Rouwe, whom we had the opportunity of questioning throughout our time working on this project. Our discussions were instrumental in leading us in the right

direction, especially during the first phase of development, and as such, will be further elaborated on during the next section.

Background

GPU optimizations for physics engines is not a new avenue for research. Many companies have dedicated valuable research time and resources to get their simulations to run more smoothly, even when they're constrained by the high computational requirements of certain subsystems. NVIDIA made the biggest push to switch computations onto the GPU. Just yesterday, on March 18 2025, they announced Newton, an open-source Physics Engine for robotics simulation which is built on their GPU acceleration library. Before, there were efforts to build a full engine on OpenGL by researchers in California Polytechnic State University - CalPoly - and an array of attempts at accelerating Molecular Dynamics and Particle Physics calculations through the GPU. More references to such research can be found in our Related Works section [2].

As such, when we messaged Jorrit Rouwe on GitHub, our intention was to focus on collision detection, and not simulation. Jorrit, however, led us in an entirely new direction. He said that the current architecture of the Jolt engine runs broad phase and narrow phase collision detection in parallel and what he meant by this was the following: at an arbitrary point in the collision detection step of the simulation, a given object can be either within the broad phase collision stage or the narrow phase collision stage, as although these stages run sequentially for a given object, they run in parallel within the view of the program. If we were to kernelize broad phase collision detection, then broad and narrow phase would need to be done fully sequentially; this might end up hurting the systems performance more than the gains we might get from parallel processing. Instead, Jorrit suggested that we focus on the simulation step of the soft-body subsystem, which is the task that is done for a soft-body after collision detection is resolved for them and it is detected that they are colliding. All soft body related jobs already run in parallel using CPU threads via the sophisticated Job Management system utilized by the Jolt Engine, which we will explain in more detail in the next section. The soft body subsystem is even able to run vertices which belong to the same body in parallel as computations that are done per vertex do not require any data about the simulation of the other vertices. We agreed soft bodies were the perfect candidate for GPU optimization and got to work.

PROF

Technique Description and Methodology

Our process of integrating the optimized CUDA code into the existing architecture of the program was threefold. First, we had to understand how the codebase operated. The Jolt Engine is an incredibly well isolated, yet still sophisticated and complex system. It utilizes a Job Management abstraction to allocate the various tasks that are required for each subsystem, be it Broad Phase Collision Detection, Rig Constraints, Force Vector Calculations or anything else. These jobs often have prerequisite jobs which they have to wait upon before they can begin their execution. The thread pool is managed to go between these jobs and eventually execute all that were listed as the simulation progressed.

Within the Soft Body subsystem, there were four jobs that were relevant to our project:

```
void PhysicsSystem::JobSoftBodyPrepare(PhysicsUpdateContext *ioContext,
PhysicsUpdateContext::Step *ioStep)
```

```
void PhysicsSystem::JobSoftBodyCollide(PhysicsUpdateContext *ioContext)
const
```

```
void PhysicsSystem::JobSoftBodySimulate(PhysicsUpdateContext *ioContext,
uint inThreadIndex) const
```

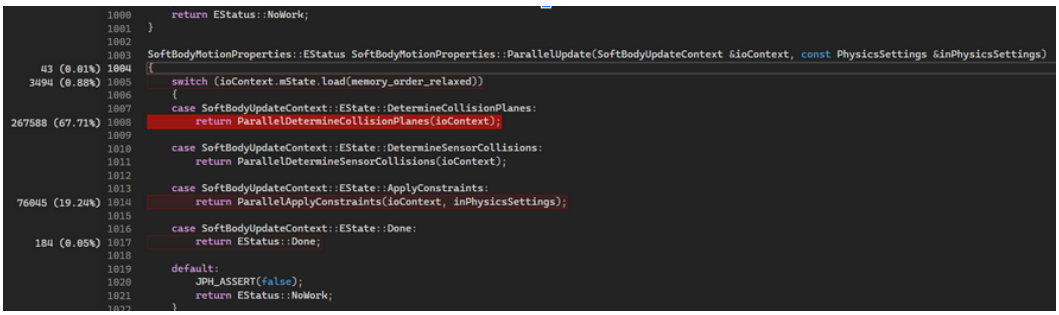
```
void PhysicsSystem::JobSoftBodyFinalize(PhysicsUpdateContext *ioContext)
```

We meticulously read through the code and inserted breakpoints to examine the state of the program when these jobs were being executed. Visual Studio, our code editor, allowed us to even view the state of the different CPU threads that were being run at the time a breakpoint was hit. Through our study, we noticed that JobSoftBodySimulate() actually ran the simulation computations on a batch of vertices each time it was called, and the subsystem enforced that these vertices belong to the same soft body, a fact that would be important in later stages of our project.

Having a better understanding of the code's architecture, we moved onto the second step of our work-flow: profiling. In this stage we primarily relied on three valuable tools; first was the Jolt Engine Debug Renderer, which allowed us to to visualize our test scenes and even detect hiccups in performance during the initial stages of the project when we were still looking for candidate subsystems for optimization. Below are the visualizations for our two most commonly used test scenes; they were able to capture the performance of the CPU and GPU implementations of the soft body system in two edge scenarios, creating a gradient view of the engines performance as the number of soft bodies and vertices in a scene change.

PROF

The next tool we utilized during the profiling stage was the Visual Studio profiler. We were advised by Professor Nikos Hardavellas to use a profiler to determine the bottlenecks in our program, which was exactly what this tool allowed us to do. As it can be seen in the profiler screenshot below, it was able to show us what percentage of the CPU computation time was spent within a given function call. As soon as we profiled the Soft Body Stress Test in Joly, it became clear that Jorrit was right; the soft body subsystem, even the CPU parallel implementation, had a lot of room for improvement. 93% of execution time was being spent on SoftBodyMotionProperties::ParallelUpdate(), which corresponds to the SoftBodySimulate job from the function snippets above.



67.71% of the execution time was spent within ParallelDetermineCollisionPlanes()

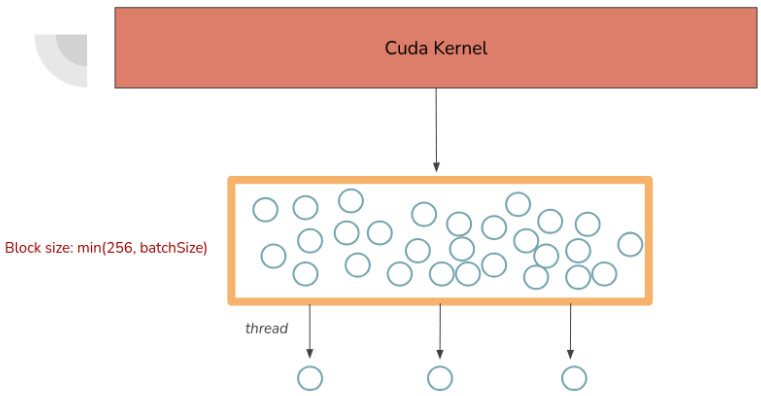
Additionally, among the tasks that were performed by the SoftBodySimulate job, ParallelDetermineCollisionPlanes() dominated the rest, with 67.7% of the execution time being spent within that function as opposed to 19.24% for the next most costly task of ParallelApplyConstraints().

Looking at these results we determined that our CUDA optimization would be focused on the computations that are being done within ParallelDetermineCollisionPlanes(), as opposed to the entire system. In hindsight, this would restrict our prospect of achieving substantial speedups due to how it restricted our handling of bodies per kernel call, but more on that later.

The last tool we used within the profiling step was another already existing module written within the CUDA codebase: PerformanceTest. This program takes in an array of options, such as the maximum CPU threads that will be used during the test, the motion quality setting of the objects, or the number of iterations to run, and spits out the average simulation steps that were performed per second. At the very end of our project, this tool allowed us to get a concrete measure of the speedup we had achieved.

Finally, the last stage of our project was Execution. During this step, we had all the information we needed and were ready to create a model for how we would integrate the CUDA code into the Jolt Engine. As we had decided that we would optimize ParallelDetermineCollisionPlanes(), this would mean that our implementation would have to suffer from a restriction that was put on it by the rest of the code architecture. At this point in the code, the SoftBodySimulate job is already allocated, and as such the code is executing on a single soft body. To be more precise, the thread that is working this job is given a number of vertices within that soft body which is determined by the batchSize variable within the SoftBodyUpdateContext object. Thus, our implementation would have to launch serial kernel calls at least as many times as we have a number of soft bodies.

We wrote the code for ParallelDetermineCollisionPlanes in CUDA, replaced the old computations with the call to the kernel, setup for this call by moving the important data from their relevant data structures into input arrays which we would pass into the kernel call and copied the output data back into the relevant data structures so that the rest of the system could function as it did before. Next, we performed the optimizations that we learned in class such as minimizing bank conflicts, and reducing access to global memory, though doing so was difficult given how much data we had to work with.



Experimental Results

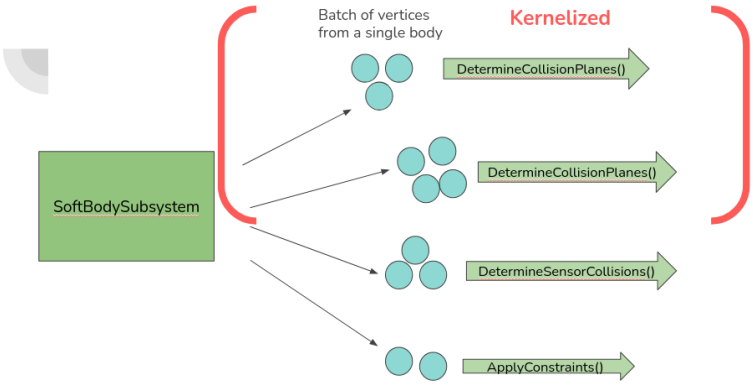
Due to the restriction imposed upon us by the rest of the system with how we have to make a kernel call per soft body, our CUDA integrated implementation of the Jolt engine performed best when it simulated a single soft body with a very large number of vertices, which would correspond to a Soft Body Stress Test with only one soft body and our tests supported this conclusion. We ran the PerformanceTest program that was included in the codebase with both of the above scenes on both the old CPU implementation and our new GPU implementation at 150 iterations and a single CPU thread and saw that while the scene with multiple soft bodies had a 8x slowdown when we went from the original to the CUDA implementation, the scene with only one soft body performed roughly the same as we achieved 6 steps/second on the CPU implementation and 4.9 steps/second on the CUDA optimized version.

Related Work

GPU acceleration via CUDA is ubiquitous in the world of physics engines. Brax, developed by google, mainly focused on reinforcement learning of robots, is built on top of JAX, which supports both GPU and TPU accelerations [3]. An example on the Brax engine displays a humanoid with soft links learning how to walk. Notably, NVIDIA's PhysX engine has also incorporated GPU-accelerated soft body support since version 5.1, utilizing CUDA only [4]. NVIDIA omniverse documentation describes a method for the generation of tetrahedral meshes. In the PhysX engine, the createConformingTetrahedronMesh function is used to produce a collision mesh. This matches the surface of the resulting tetrahedral mesh with the input triangle mesh's surface. The vertex list of the collision mesh replicates all vertices from the render mesh, with additional vertices appended as necessary to maintain a valid tetrahedral structure.
Titan, a simulation engine that uses CUDA for high-performance simulations, allows dynamic updates to multiple bodies during runtime [5]. It supports the creation, deletion, and modification of components, making it ideal for applications like topology and shape optimization. Titan's ability to update the GPU at controlled intervals enables real-time changes to mass-spring meshes, which is essential for simulating soft robotics and adaptable materials. This dynamic approach offers potential for simulating programmable matter and adaptive materials in the future.

Conclusion

In hindsight, we have realized that our implementation could have achieved much higher levels of performance had we made our kernel call one layer higher within job abstraction. As demonstrated by the figure below, this would mean that we could have our Kernel operate on a batch of vertices independent of which soft body they belong to. Consequently, the amount of kernel invocations our system would have to make would depend on limitations imposed on us by our GPU and not our CPU architecture, and would most likely be within the 1-2 range depending on how many vertices are within the scene. With much less overhead per simulation cycle, we believe that this implementation would actually be able to achieve significant speedups in simulating soft bodies within the Jolt Physics Engine.



References

This project was done in collaboration with Ege Yilmaz

[1] Jolt. (n.d.). Jolt Physics Engine. GitHub. Retrieved from https://github.com/jrouwe/JoltPhysics

[2] NVIDIA. (2025, March 18). Announcing Newton: An open-source physics engine for robotics simulation. NVIDIA Developer Blog. Retrieved from https://developer.nvidia.com/blog/announcing-newton-an-open-source-physics-engine-for-robotics-simulation

[3] Brax. (n.d.). Brax: A high-performance physics simulation library. GitHub. Retrieved from https://github.com/google/brax

[4] NVIDIA. (2025). NVIDIA PhysX 5.1.0 Documentation. Retrieved from https://nvidia-omniverse.github.io/PhysX/physx/5.1.0/

[5] Titan: A Parallel Asynchronous Library for Multi-Agent and Soft-Body Robotics using NVIDIA CUDA. (n.d.). Retrieved from [https://arxiv.org/pdf/1911.10274]