GPGPU Acceleration for Video Game Physics Engines

Falco Girgis, Earl Wells
Electrical and Computer Engineering Department
University of Alabama in Huntsville
Huntsville, AL 35899 USA
wells@ece.uah.edu

Abstract-- Physics engines are becoming utilized more and more in modern video games. As the complexity of these games scale, so does the computational intensity of the scenarios these physics engines must simulate. With the introduction of GPGPU computing through libraries such as OpenCL and CUDA, the immense parallel floating-point power of the GPU may be harnessed for more than just graphical applications. In this project, a software-based 2D rigid body physics engine was written in C then ported to the GPU with the OpenCL library in an attempt to exploit data-level parallelism within the engine. As the number of objects in the simulations increase, so does the benefit from the parallel-execution of a physics pipeline on the GPU.

keywords: opencl, phyics engine, rigid body, gpgpu gaming

I. INTRODUCTION

Since the introduction of 3D gaming, physics engines have played increasingly important roles in video games. As the power of the PC and gaming machines grew, so did the levels of realism portrayed by physics in video games. Not surprisingly, updating a physics simulation is often one of the most computationally demanding tasks involved with updating a frame in a video game.

Historically, physics engines in video games had always been CPU-bound tasks. With a fixed-function pipeline, the CPU is responsible for everything that is not directly involved with the rendering pipeline. As the computational intensity of physics processing rose, developers became more motivated to hardware accelerate the demanding floating point mathematics that they required. One such approach was to introduce a third coprocessor (along with the CPU and GPU) to offload these intense calculations: the physics processing unit (PPU).

As time progressed, and the programmable pipeline was introduced with shader programming, the GPU finally became available as a highly-parallelized general-purpose mathematics coprocessor. The universal appeal of the GPU over the PPU stemmed from the fact that no additional hardware would be necessary to hardware accelerate physics calculations. The calculations involved in physics simulations are generally independent computations performed in a software pipeline on a per-object basis. This approach is data-level parallel by nature, and can easily be scaled so that each stage of the pipeline executes on different sets of data in parallel on the GPU.

This paper presents an overview of designing a pipeline for a 2D rigid body physics engine capable of operating on large sets of data in parallel on the GPU using OpenCL. The following section surveys existing commercial solutions and the approaches that they have taken. The next two sections outline the method chosen for this project and the overall results from implementing such a method. The final section presents future goals and considerations for the project.

II. EXISTING SOLUTIONS

Due to the high demand of in-game physics engines, and the complexity of creating such an engine, most modern development studios license prebuilt physics engines for use in their games. There are a variety of different commercial physics engines targeted at various platforms with their own individual trade-offs between performance and realism.

PhysX is one of the most popular physics solutions for modern video games. The engine is developed and licensed by NVidia. It uses their own

GPGPU technology to hardware accelerate physics calculations through the use of CUDA. Physics calculations are performed in software when this kind of hardware acceleration is not available (through either a GPU or PPU), resulting in poorer CPU-bound performance. Today, the PhysX engine is supported by Microsoft Windows, Mac OSX, various flavors of Linux, Nintendo Wii, Sony Playstation 3, and Microsoft Xbox 360.

Another extremely popular independent physics solution is the Havok engine, developed by Irish company Havok. The Havok engine is a software-based solution, allowing it to support a wide array of platforms including the Playstation 3, Wii, Xbox 360, and previous generation gaming machines. Since the launch of its SDK, it has been used in over 150 commercial video game titles.

III. IMPLEMENTATION

The physics engine developed for this project was originally written in the C programming language on the CPU, then ported to OpenCL-C to be run on the GPU. When developing software for SIMD-based parallel execution, certain considerations have to be made with regard to both the algorithms and data structures implemented. The data structure chosen to store all collision geometry and rigid body data in this engine is a simple pool implemented with a static array. This allows us to keep all required data packed contiguously in memory. This greatly simplifies the task of moving this data to and from the GPU in the future, provides great cache coherency, and allows us to develop algorithms to simultaneously manipulate data in various regions in the pool.

Three different pools were required to hold the data used in this engine: 1) collidable 2) rigid body 3) attribute. The collidable pool houses the collision geometry of each object. This is kept separate from the rigid body, so that the physics pipeline may be used for collision checks between bodies that are non-interactable. It contains the size, position, and orientation of each body. The rigid body pool is used to contain motion-related attributes that are required for objects that are interactable. These kinetic attributes include both linear and rotational quantities such as velocity, angular velocity, acceleration, angular acceleration, and forces and torques applied to each body. Finally, the attribute pool is used to house attributes that are static or are shared between instances of the same rigid bodies such as mass, moment of inertia, and coefficient of restitution.

Various stages of the physics pipeline operate on different components of this data. Because of this, each RigidBody contains an index to its corresponding Collidable and Attribute data in their respective pools. This ultimately allows collidables to exist without rigid bodies (used purely for collision detection) and two rigid bodies of the same type to share the same attributes (two basketballs have the same mass, moments of inertia, and coefficients of restitution) to reduce overall memory usage.

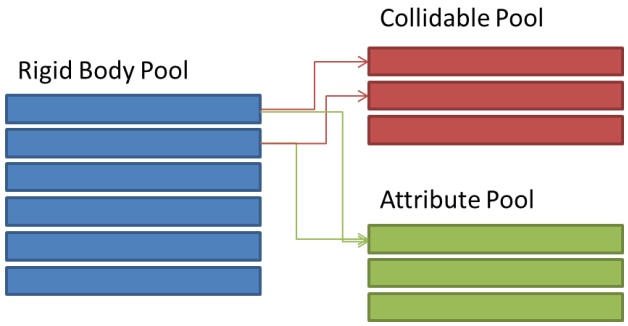

Figure 1. Data Pools indexing into each other

From an algorithmic standpoint, the physics engine is implemented as a multi-stage pipeline where each stage is a separate function operating on a specific region in the data pool. This will not only allow for each stage of the pipeline to be executed in parallel on the GPU, but it also allows the entire simulation process to be performed as a single GPU operation. Data is transferred to the GPU, sent through the pipeline, then transferred back with no intermediate processing required from the CPU.

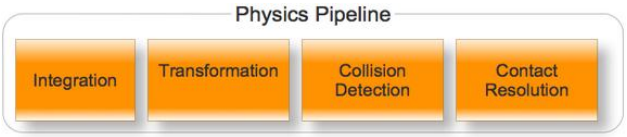

Figure 2. The Physics Engine Pipeline

The tasks of each stage in the pipeline are as follows: 1) Integration-- updates the state of each

rigid body. The integration method chosen for this paper is based on simple Newtonian physics. Each frame, the integrator accepts any outside forces and transforms them into acceleration and torque. Acceleration is added to velocity, and torque is added to angular velocity. Velocity is added to position, and angular velocity is added to orientation. Finally, force/torque accumulators are cleared, and any damping is applied to the object's velocities. 2) Transformation-- creates a an instance matrix based on an object's size, orientation, and position then translates the object's local coordinates into world coordinates and stores them. 3) Collision Detection/Contact Generation-- Checks for collision between each collidable's geometry with every other collidable's geometry and generates a "contact" object with relevant data for every confirmed collision. The algorithm employed in this project is based upon the separating axis theorem and works with any convex polygon [1]. 4) Contact Resolution-- Resolves both the interpenetration and velocities of both objects involved in every contact. The resolution scheme used by this engine is impulse-based. Impulse-based resolution methods sacrifice mathematical accuracy for performance by greatly simplifying the micro-scale effects of a collision. This accuracy loss is perfectly believable for video games and allows interpenetration and velocity to be resolved completely independently of one another [2].

Due to the manner in which our pipeline was constructed, each "stage" in the pipeline maps effectively to a parallel-executing kernel in OpenCL. The first stage in the pipeline and kernel that was written was the integration kernel. This was able to utilize OpenCL's built-in vector data types and operators for updating the values of each rigid body. Integration ultimately updates the position and orientation of the body's collidable geometry. The Integration kernel received pointers to the GPU-side RigidBody pool and Collidable pool. It used each rigid body's collidable index to index into its corresponding collision geometry in the Collidable Pool.

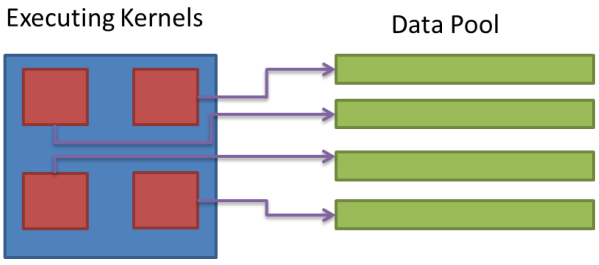

Figure 3. Kernels operating on different regions of data.

The next stage of the pipeline that was ported to the GPU was the transformation stage. This stage was also able to utilize OpenCL's vector types along with its matrix types and operators. It was later decided that this stage should also store two normalized vectors representing the two directions of the perpendicular edges of each quadrilateral in the simulation. This data is necessary in the collision detection phase and can be calculated once rather than for each collision the object is involved with when calculated during this phase (saving quite a few normalization operations requiring a floating point square root). The Transformation Kernel operates on each object's collision geometry, so it took only a pointer to the GPU-side Collidable Pool.

The third and final stage that was ported to the GPU in this project was the Collision Detection/Contact Generation phase. It was able to utilize many built-in vector operations offered by OpenCL for dot products, vector projections, and various other operations required for separating axis-based collision detection. This stage also introduced an additional level of complexity into the pipeline. It must also generate a list of contacts for every pair of bodies that have confirmed collisions. This list of contacts was implemented as a GPU-side pool accessed by the collision detection kernel. An issue presented with this approach is regulating access to this pool. Since there are potentially hundreds of instances of the collision detection kernel executing in parallel that may be generating contacts in this pool, some kind of mechanism must be utilized to prevent them from stepping on one another in memory. The solution chosen was to use OpenCL's "atomic operations." Every collision kernel with a confirmed collision performs an atomic read/increment operation on a global variable storing the number of contacts [3]. This retrieves a contact

index for the current kernel to access the Contact Pool, and also increments it for the next kernel with a positive collision. In the end, this kernel required a pointer to the GPU-side Collidable Pool and Contact Pool as arguments.

The contact resolution stage of the pipeline was unable to be ported to the GPU (addressed in part IV. Results and Conclusions).

Once each kernel had been rewritten for GPU-side execution, the entire pipeline was ready to be executed on the GPU. At the beginning of each frame, the Collidable, RigidBody, and Attribute Pools were all transferred to the GPU. A command queue was then executed containing each kernel in the pipeline. While the execution of each stage in the pipeline was an independent task performed on a per-object basis, each stage was still dependent upon the completion of the previous stage. This meant that clEnqueueBarrier() had to be called between the queueing of each kernel to ensure that the previous stage had completely finished its execution before the next stage was allowed to begin. After the entire pipeline has executed, the three pools were transferred back to the host machine along with the GPU-generated Contact Pool.

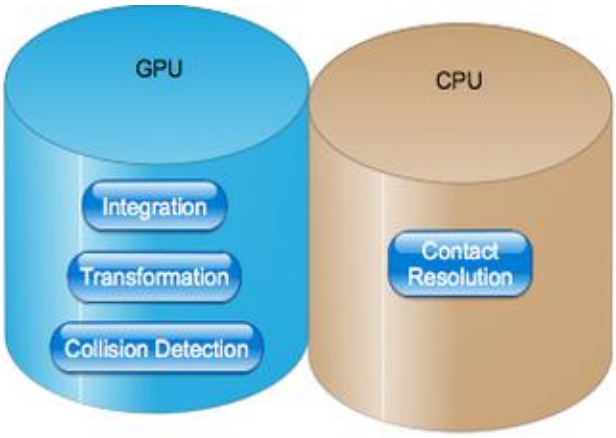

Figure 4. The final pipeline implementation.

IV. RESULTS AND CONCLUSIONS

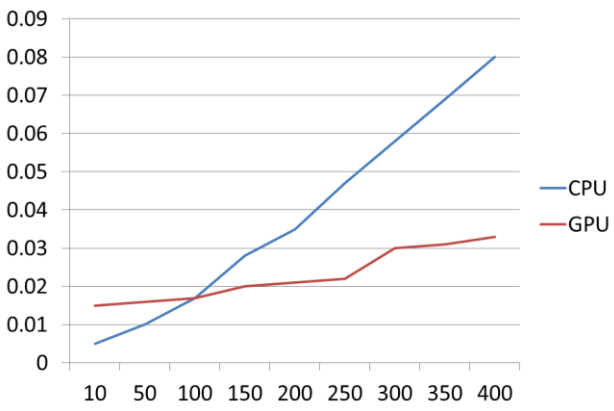

Figure 5. Overall performance of GPU and CPU-based engines. X-Axis: Number of objects in simulation. Y-Axis: Processing time.

The initial time required to transfer each pool of data to and from the PCI-E bus is relatively large. For small amounts of contacts, this initial time requirement saturates out any performance gain from GPU acceleration, and the software-based approach is better suited. However, as the number of objects in the simulation scales, the performance of a GPU accelerated approach becomes significantly greater than the performance of the same engine bound to the CPU. When there are hundreds of objects involved in a scene, the framerate of the CPU-bound calculations drops significantly even on modern PCs while the GPU-bound engine produces next to no noticeable framerate drop.

Another observation that can be made about the GPU's overall performance is regarding the number of available work-items in a workgroup. The time taken to process a kernel generally does not scale until it requires another work-group to finish. This is apparent in figure 5. The processing time jumps between 250 and 300 bodies, because the number of available work-items was 256.

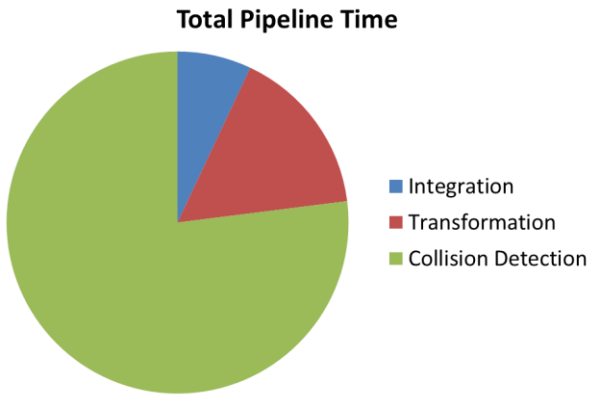

Figure 6. Time spent in each stage of the pipeline on the GPU at 100 objects.

The most time-consuming stage of the pipeline (under normal conditions) was collision detection. This is the most computation-heavy, and required the most iterations/simultaneous kernel executions as a function of the number of objects in the scene. On the CPU-bound implementation, this quickly became a bottleneck. On the GPU-bound implementation, this was far less of a problem. However, since the contact resolution stage was not ported to the GPU, when many contacts were generated within a scene using either pipeline (such is the case with a large number of stacked bodies), the bottleneck became bound to contact resolution.

One observation that was made during the porting of this physics engine to the GPU was that while an impulse-based collision resolution approach generally offers the greatest performance on a CPU, this approach does not scale well for a GPU-based implementation. Each contact in the contact list is resolved sequentially (based on most severe to least severe contact). Since a single object may be involved with multiple contacts, resolving one contact's penetration may potentially affect other contacts' penetration. For each contact resolved, the algorithm must traverse the list of contacts updating any other adjacent contacts. This means that each contact is not resolved independently of one another, so parallel execution cannot work with this method. For this reason, the contact resolution stage of the physics pipeline remained CPU-bound.

V. FUTURE WORK

Due to time constraints, there is still quite a bit of future work to potentially improve the proposed pipeline. The most pressing matter is to look into alternate contact resolution algorithms that lend themselves better to a data-level parallel paradigm. Generally speaking, impulse-based collision response algorithms tend to run the fastest. However, the extra processing time required for more realistic collision resolution may be masked by the overall performance gain from parallelizing this stage in the pipeline.

Another potential optimization would be to look more closely into aligning data sizes and storage for optimal cache mapping on the GPU. This generally tends to be hardware-specific, so it was outside of this project's scope. However, caching issues tend to be some of the biggest factors for determining performance on the GPU [4].

Some of the linear algebra routines used within this project and paper (especially for the collision kernel) probably did not take full advantage of OpenCL's built-in vector/matrix types or operators. Performing less of these calculations in software and relying more on OpenCL for them would most likely yield another performance gain [5].

Another consideration for this project was the fact that OpenGL was used for rendering independently of OpenCL. The two operated independently of one another and utilized the PCI-E bus completely obliviously of each other. This meant that world vertices that were calculated on the GPU had to be sent back to the host to be resubmitted to the GPU through OpenGL for rendering. OpenCL offers many auxiliary functions for sharing data, buffers, and objects between OpenCL and OpenGL, so the two could definitely be integrated much more gracefully than they were in this project. This would result in better utilization of bandwidth available on the PCI-E bus and ultimately greater performance [6].

REFERENCES

[1] D. Kodicek, *Mathematics and Physics for Programmers*, 2005, pp. 188-190.

[2] I. Millington, *Game Physics Engine Development*, 2007, pp. 103-124

[3] D. Andrade, "OpenCL C99 Atomics," http://www.cmsoft.com.br/index.php?option=com_content&view=category&layout=blog&id=113&Itemid=168

[4] Apple Inc, "OpenCL Programming Guide for Mac OSX," May 2009, http://developer.apple.com/library/mac/#documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html

[5] Khronos Group, "OpenCL API 1.0 Quick Reference Card," 2009, khronos.org/files/opencl-quick-reference-card.pdf

[6] A. Munshi, B. Gaster, T. Mattson, J. Fung, D. Ginsburg, *OpenCL Programming Guide*, May 2011, Ch. 10 "Interoperability with OpenGL."