Optimization of CUDA-based n-Body Simulation

Tyler Allen

***Abstract*—**

I. Introduction

II. Methodology

A. Implementation

The n-body simulation was written in C++11 and CUDA. Bodies are implemented with the following elements: a 3-D position vector, a 3-D velocity vector, a 3-D acceleration vector, and mass. These elements are required portions of the n-body algorithm[`?meyer`]. The initial implementation represents bodies using C++ objects. The calculation for force applied to body i by all other bodies in one unit of time is presented below[`?meyer`]:

$$F_i = \sum_{j=1, i \neq j}^{N} \frac{G m_i m_j (p_j - p_i)}{||p_j - p_i||^3}$$

Fig. 1: Equation for force applied to body i by all other bodies.

This formulation includes position vectors represented by p, mass represented by m, and a gravitational constant represented by G. The output of this equation is force, which is trivially converted to acceleration since mass is known. For each timestep, acceleration can be used to find velocity and then position. The value used for for parameter G is 6.674×10^{-11}[`?meyer`]. An additional factor, ϵ, has been included as an additive constant to the magnitude of the distance between two bodies seen in figure 1[`?nvidia`]. This factor is to prevent the force calculation from approaching infinity when objects overlap. The value of ϵ used in our simulation is $\frac{1}{16}$. This value is based on the results of the visualization of the n-body algorithm, discussed later in this section.

The n-body algorithm was implemented as a brute-force solution derived from the equation in figure 1. This is primarily because the n-body problem is highly parallelizable. This paper does not focus on more advanced solutions to the n-body problem, such as the Barnes-Hut algorithm[`?barneshut`], since this paper is focused on CUDA optimizations to an existing algorithm. The algorithm is implemented as two seperate CUDA kernels. The first kernel, `calculate_force`, calculates the force (acceleration) applied to each body during one time step. The second kernel, `update_pos` updates the velocity and position values of each body according to their new acceleration. The kernels are split because `calculate_force` has a data dependency on the current position of each body. Allowing a position update before all new acceleration values have been calculated would cause some small computation errors. The `calculate_force` and `update_pos` functions can be seen in appendix **??** and **??**.

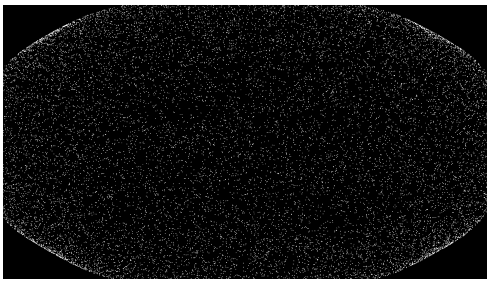

Fig. 2: A screenshot of the n-body visualization rendering 16384 bodies.

A visualization for the n-body problem is contained within this implementation. The visualization is written using OpenGL[`?opengl`] and using the window utility library GLFW[1]. The visualization shows each body as a white particle on the screen. Interactions are not limited in any way, and will render as fast as the computation will allow them to. The visualization is disabled for all calculations and tests in the remainder of this paper because it eventually becomes a bottleneck for the simulation. Figure 2 contains a screenshot of the visualization.

B. Testing Conditions

The testing conditions and performance metrics will be described here. We will rate the performance of an individual run of the simulation using "frames". A frame is one full iteration of the main loop. The main loop contains the two CUDA kernels mentioned previously, as well as a memory transfer of position data to main memory from the device. Frames are used because this optimization will be focused around generating data for a specific use case, such as rendering. Therefore, each "frame" requires the transfer of data back to main memory. As previously mentioned, the rendering is not part of the test. To account for this, we will procede under the assumption that visualization is able to be rendered "infinitely fast," and that the rendering does not require any computational resources provided the position data is present in main memory after each iteration.

Other metrics may also be used to describe performance where relevant, such as locality or occupancy. These metrics are provided by profiling tools described in the next section. Since these profiling tools affect the performance of the simulation, these metrics will not be taken from the same simulation runs as the frames metric is taken. Instead, frames will have several non-profiled runs from which the average frame count will be derived, and then profiled runs will be performed.

Every simulation will be 60 seconds long. Time is kept on an internal timer within the main loop. Therefore time may add bias against implementations with greater performance by slowing the main loop down for each iteration. We will show this bias is minimal. An external clock could have been used but may have been unreliable for stopping the program at the appropriate time. This would have also introduced a degree of bias.

The number of bodies used in the experiment will be stated for each test case. The number of bodies changes because the initial implementation was too slow to render any larger number of bodies. In order to maintain the frame of reference, a comparison will be made between the same implementation at higher number of bodies when the number of bodies increases.

C. Performance Optimization

In this section we will described the tools used for optimization, as well as the optimization techniques used. These optimization techniques are general techniques for improving performance. The next section will focus on optimizations targeting specific hardware and CUDA versions for improving performance across different GPU hardware. We focus soley on the time spent in the main loop of the simulation, so time spent in initialization and finalization is not a concern of this paper. The areas of optimization that received focus are GPU parameters, memory organization and allocation, numerical and branch operations within CUDA kernels, type optimizations, and optimization flags.

In order to gauge the performance of the program and identify possible areas for improvement, several profiling tools were used. These profiling tools include `gperftools`[2], NVIDIA `nvprof`[`?nvprof`], and NVIDIA NSight[`?nsight`]. `gperftools` is a CPU profiler. For the purpose of our simulation, its purpose is to identify CPU-based bottlenecks within the main program segment. We use this tool to help mitigate the CPU's impact on the CUDA-based code to improve the accuracy of performance data. NVIDIA NSight is a visual profiling tool for profiling GPU usage. It includes information such as the amount of program time spent executing each kernel, locality information, and device/host memory transfer time. It also provides performance analysis information to guide the user towards areas that can be optimized. We use this tool to acquire metrics on our CUDA kernels that we can attempt to improve. `nvprof` provides the same primary functions as NSight, but it is a command line based tool.

D. GPU-based Optimization

In this section we will describe the CUDA parameters used to optimize the simulation for different hardware, as well as the hardware used in our test cases. The primary tool available for this optimization is the `__launch_bounds__`[`?bounds`]. This function uses the arguments `maxThreadsPerBlock` and `minBlocksPerMultiprocessor` to determine the appropriate number of registers to allocate per thread[`?bounds`]. The maximum values for these two parameters are dependant on the CUDA version being used. In the test studies, the number of blocks argument passed to each CUDA kernel is always the same value as `maxThreadsPerBlock`, except in the one case where we do not use