Indian Institute of Technology, Kanpur

Association for Computing Activities

N-Body Simulation

End-term Evaluation

Project Members

Hunar Preet Singh

Project Mentor

Akash Kumar Dutta

Contents

1. Introduction

An N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. A familiar example is an astrophysical simulation in which each body represents a galaxy or an individual star, and the bodies attract each other through the gravitational force.
N-body simulation arises in many other computational science problems as well. For example, protein folding is studied using N-body simulation to calculate electrostatic and van der Waals forces. Turbulent fluid flow simulation and global illumination computation in computer graphics are other examples of problems that use N-body simulation.
The aim of this project is to fasten these simulations.In other words parallelize our computions using GPUs. Towards the end tabulated results are provided that compare the times, for fixed number of bodies, for serial implementation vs parallel implementation

2. Software Used

- CUDA Toolkit 8.0
- OpenCV (version compatible with cuda toolkit 8.0)
- Cmake

3. CUDA and GPU

- CUDA Architecture
    1. Expose GPU parallelism for general-purpose computing
    2. Retain performance
- CUDA C/C++
    1. Based on industry-standard C/C++
    2. Small set of extensions to enable heterogeneous programming
    3. straightforward APIs to manage devices, memory etc.
- This project uses Nvidia Geforce GTX 960m GPU

4. Algorithm

Given any two bodies of mass m_1 and m_2 gravitational force between them is given by:

$$f = G\frac{m_1 m_2}{r^2} \tag{1}$$

Now given n bodies using this simple concept we can simulate their motion in space using some approximations. At a particular, for a particle i say we are given by velocity $\vec{v_i}$ and position $\vec{r_i}$. We need to calculate force on it due to other particles, given by $\vec{F} = \sum_{i \neq j} \vec{F_{ij}}$ where F_{ij} is force between i^{th} and j^{th} particles. Given $\vec{F}$ we can assume acceleration vector of that particle to be constant for a small time Δt and calculate the final velocity and position vectors after this time using Newton's equations of motion.

$$\vec{v_i^f} = \vec{v_i} + \vec{a_i}\Delta t \tag{2}$$

$$\vec{r_i^f} = \vec{r_i} + \frac{\vec{a_i}(\Delta t)^2}{2} \tag{3}$$

After getting these new states for a particular particle repeat the steps to get updates for other particles too. Now we have a whole new state for all the particles we repeat the same procedure to get next updates and keep displaying these coordinates using openCV, we get the required simulation. Clearly in the naive serial implementation time complexity is $O(n^2)$ but parallel model takes much less time than serial implementation.

4.1. Serial Implementation

Link to serial implementation can be found here

4.2. Parallel Implementation

Link to parallel implementation can be found here

4.2.1. CUDA Memory

There are many types of memory available for use in a CUDA program, choosing the memory most suited to the task is critical for the performance of the program

- Global Memory
  Global Memory is the main GPU memory with the largest size but also slowest access time. A typical Global memory access takes 200-400 ms. CUDA can improve the time for memory transactions using other times of memories wherever they can be used. This is the memory accessible to all the threads.
- Shared Memory

Shared Memory is visible only at the block level. Threads in the same block and read and write to this mem- ory to share data. This type of memory is much faster than global memory but cannot be used to store the values of the ield as it cannot be accessed by the host code. However, it can be used to effectively share data to reduce the number of global memory transactions required.

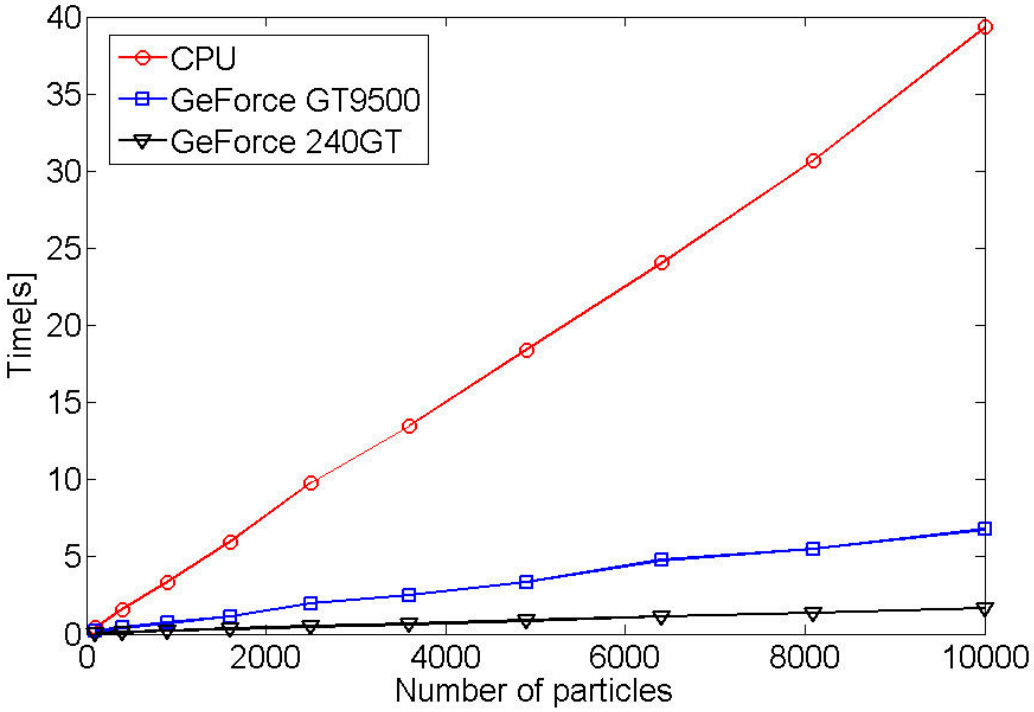
- Local Memory

  Each thread has a memory which only it can access. It is the most quickly accessible memory.

4.2.2. Calculation

Tile Calculation Tile calculation makes use of shared memory to increase the performance of the simulation. The threads are allocated into blocks of p particles with a total of N/ p blocks. Each block will process one tile of N/ p particles at a time using shared memory to reduce global memory transactions. Each thread calculates the force on its particle by performing the following algorithm:
1. Load one particle's data into shared memory.
2. Synchronise with the other threads in the block.
3. Calculate the force each of the particles stored in shared memory exerts on the thread's particle.
4. Sychronise with the other threads in the block.
5. Repeat 1-4 until all particles have been processed.
6. From the force on the particle, calculate the total change in position and velocity.
7. Write the results back to the output memory.

5. Results



It is clear from the graph that as the GPU(both in terms of ram and processing units) gets better and better, the time complexity for calculating force on one particle due to other particles approaches to constant where as in case of CPU it is linear.