Work 1 Report

João Pacheco
56982
Faculdade de Ciências e Tecnologia

***Abstract*—This document will provide the analysis of a series of tests in order to parallelize the NBody All Pairs algorithm utilizing an Nvidia GPU and their CUDA programming language.**

***Index Terms*—Nbody, all-pairs, CUDA, parallelization, GPU**

I. NBody Problem

The NBody problem is a physics focused problem applied to N celestial bodies/particles with the goal to try and predict where each particle will be after interacting with the others. To apply the motion to the particles, in this case, we utilize the gravitational force.

A. All Pairs Approach

For this work we used a more specific approach called the **All Pairs Approach**. This approach considers in every instance of time, for each particle, the result of their interaction with all the others in the system.

II. Solution

My solution revolves around in utilizing the GPU to achieve parallelization over the *Compute Forces* and the *Move All Pairs* methods, utilizing the CUDA language, to analyze and achieve the highest performance over this HCP problem and understand which implementation resulted in the biggest performance boost and understand why.

III. Implementation Details

There were multiple steps/implementations to be done and tested. What I can advance right now that was common in almost every implementation made in CUDA was the following:

- Before doing anything it was required to copy the particles to the GPU.
- At the end was always needed to copy the particles back to the host/CPU.

A. Only Outer Loop

The simplest of all the loops to parallelize. To do so it was only needed to create one kernel.

During the kernel execution, each thread was going to read from the memory, on the position of their global index, a particle. For their particle, the thread would set their force on both X and Y axis to 0. Then they would run through each particle in the system and update their own particle forces with the effect the others had on them.

B. Both Loops

For both loops being paralleled, there were 2 kernels.

The first one is the one relative to the outer loop. It had the same implementation as before but now it would only set every particle's X and Y forces to 0.

Next the host would iterate over each particle. For each particle (going to call it the *reference particle*) it would launch the kernel where each thread would:

- Go to global memory fetch the particle that corresponded to their index.
- Calculate the impact their particle had on the *reference particle* on both the X and Y axis.
- Add the impact to the *reference particle* force fields.

Since this was done over Global Memory, it was necessary to utilize Atomic Operations. Since there was no Atomic Add implementation over 2 double-precision floating points, I grabbed an implementation of it in *Nvidia's CUDA Toolkit Documentation* which utilizes **atomicCAS()**;

C. All Pairs

To parallelize the All Pairs loop, first thing I did was remove the **Memcpy** that would copy the particles from the GPU to the CPU at the end of *Calculate Forces* since I was going to need it on this step and moved it to the end of this method instead.

I needed to create 2 pairs of 1D array variables, 1 for the speeds (host and device) and 1 for the accuracy (host and device).

The **kernel** I made was more or less a direct copy of the sequential one with some slight changes such as:

- Instead of a loop over each particle, each thread took care of their own respective particle by coinciding their index in the particles array to their global index.
- At the end of it, each thread would write their respective particle's speed and accuracy.

After the execution of the kernel, we are left with the 2 lists filled with each particle's speed and accuracy values

Since to calculate the next step of the simulation we need the MAX value of the speeds and of the accuracy, I iterated over both lists to get them. To do this, I first copied those values from the device variable over to the host one since the operation was going to be made on the CPU.

After doing so, the function was going to return both of those values. This required some changes along the project, such as needing to change the return type of the sequential version and on the headers.

D. Streams

This version was built on top of the Outer Loop only implementation as it had shown the best results so far. This implementation optimizes the latter by creating independent parallel streams which were put on a list. Then for each stream it will copy asynchronously a part of the particles to the GPU, computes the **kernel** created for the Outer Loop part and asynchronously copies the results back to the host.

IV. Results

Before entering with results a little disclaimer must be said: These results were obtained with DEBUG mode on. Without this mode on, the results should be faster but their relative comparisons should remain the same (i.e if A $>$ B with debug on then with debug off it should still be A $>$ B).

I made a copy of the teacher's Google Sheets table so it was easier to keep record of the scores.

The left results are tests with 1000 particles on 50 iterations and on the right it's 1500 particles on 50 iterations.

Note: In section VI, the whole table of results can be seen.

CUDA Outer Loop (parameter: thread block size)	32	5938	9601.53	17699	16195,00
	64	5984	9692.4	17540	16071.3
	128	5940	9597.46	17660	16178.5
	256	5936	9585.44	17673	16179.6
	512	6681	9749.13	19350	16509,00
	1024	8411	10079.7	23336	17350.3

Fig. 1. Outer Loop

CUDA Both Loops (parameter: thread block size)	32	453129			
	64				
	128				
	256				
	512	474485			
	1024				

Fig. 2. Both Loops

CUDA Outer Loop All Pairs(parameter: thread block size)	32	6027	9704.41	17767	16195.4
	64	6045	9720.22	17750	16215.5
	128	6041	9702.93	17753	16189.5
	256	6035	9700.62	17737	16203.4
	512	6792	9847.38	19468	16547.8
	1024	8509	10122.9	23450	17392.9

Fig. 3. Outer Loop w/ All Pairs

CUDA Outer Loop STREAMS (parameter: thread block size)	32	2158	23450	2628	736.101
	64	936	972.115	2213	1215.33
	128	937	976.267	3025	1064.31
	256	963	978.24	1081	1321.84
	512	1242	558.623	3335	966.176
	1024	2021	3421.54	2444	1788.36

Fig. 4. Outer Loop w/ Streams

V. Analysis

As an overall remark that I can already say without diving deep into the details of each result is that every implementation (except for the both loops one) was **faster** than the sequential one.

CUDA Outer Loop STREAMS 3000 100 (parameter:number of streams)	2	3560	2487.75
	3	2447	5186.57
	5	2853	7858.37

Fig. 5. Streams with 3000 particles done 100 times

A. Outer Loop

The outer loop implementation was the one that had the best results (without optimizations) and table 1 shows that. Since this implementation makes it so one thread treats their own particle, it results in there not being any concurrency issues.

Comparing these results with the OpenMP version, it all around manages to compare with it, but without any optimization (which we'll see later on), it loses against the vectorized version.

B. Both Loops

By analyzing the table of results in figure 2, this version was the one with the worst results and it was predictable such was going to be the case. Although each thread is computing the result of their particle with the *reference one*, they are all trying to write at the same time in the same memory spot. To avoid that I used an Atomic operation, which tanks the performance really hard and makes the whole process really slow.

I tested both the 32 Thread Block Size version and the 512 one and didn't bother with the rest since it was obvious it was going to be as bad or worse than these 2.

C. All Pairs Outer Loop

This version built on the Outer Loop version and parallelized the all pairs loop as well. This resulted in a slight worse performance across the board in every parameter. This is most likely due to their being 2 MAX checks now: one after getting the list of speeds and accuracies to find the MAX of both, and then in the CPU to check if the MAX that was received from the CUDA implementation was higher than the current MAX in the nbody_all_pairs global variables.

Since the outer loop version was equal or worse than the OpenMP versions and this version made it slower we can deduce without looking at the complete table that this version is still inferior to the OpenMP one.

D. Streams

This initial version shown in 4, shows a gigantic leap in performance numbers. It showed a 21% increase in performance over the 1000 particles, and 13% on the 1500 particle version. This is due to the transfer of data being asynchronous and the streams being parallel and independent, one could be accessing the kernel, while the others could be reading or writing on the memory. The possibility to overlap data transfers speeds up the whole program's execution.

To illustrate this, I'm taking an image from the professor's slides.

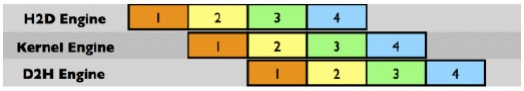

Fig. 6. Data Overlapping

1) Best Number of Streams: After finding out the grand difference that the streams made, I decided to figure out what was the sweet-spot in terms of number of streams.

To do this, I tested with 3000 particles and over 100 iterations so the results were more telling and more precise.

Analyzing the results in figure 5, we can see that the best number of streams was 3.

VI. Annex

		1000		1500	
Version	Parameters	Average	Std Dev	Average	Std Dev
Sequential		30786	72.63	94532	1059.05
Vectorized		5026	29.04	15325	240.13
OpenMP (parameter: number of threads)	1	30365	46.12	92838	1036.52
	2	16941	31.81	48937	484.22
	4	8691	45.00	25122	273.63
	8	4791	37.64	13277	90.00
	16	3000	118.44	7766	194.56
	24	4344	35.43	11431	149.48
	32	4262	27.06	10553	209.50
OpenMP Vectorized (parameter: number of threads)	1	7517	17.75	23007	259.97
	2	8154	590.83	18156	119.60
	4	4689	77.99	9145	48.68
	8	2684	89.51	5294	22.82
	16	2122	232.05	3519	81.52
	24	2161	59.50	4715	24.67
	32	2116	71.30	4668	60.56
CUDA Outer Loop (parameter: thread block size)	32	5938	9601.53	17699	16195
	64	5984	2846063	17540	16071.3
	128	5940	9597.46	17660	16178.5
	256	5936	9585.44	17673	16179.6
	512	6681	9749.13	19350	16509
	1024	8411	10079.7	23336	17350.3
CUDA Both Loops (parameter: thread block size)	32	453129			
	64				
	128				
	256				
	512	474485			
	1024				
CUDA Outer Loop All Pairs(parameter: thread block size)	32	6027	9704.41	17767	16195.4
	64	6045	9720.22	17750	16215.5
	128	6041	9702.93	17753	16189.5
	256	6035	9700.62	17737	16203.4
	512	6792	9847.38	19468	16547.8
	1024	8509	10122.9	23450	17392.9
CUDA Outer Loop STREAMS (parameter: thread block size)	32	2158	23450	2628	736.101
	64	936	972.115	2213	1215.33
	128	937	976.267	3025	1064.31
	256	963	978.24	1081	1321.84
	512	1242	558.623	3335	966.176
	1024	2021	3421.54	2444	1788.36

Fig. 7. Whole Results