Report

Assignment 4

Ioannis Koutoulas and Alexandros Tsagkaropoulos

February 17, 2025

Introduction

The objective of the Exercise 4 is to parallelize the n-body solver for particles influenced by the universal gravitation [4]. Additionally, it aims to implement the parallelized version using CUDA. An n-body solver is a program that provides the solution to the corresponding n-body problem by simulating the behavior of the particles. The inputs to the solver determine the initial configuration of the system by specifying the mass, position and velocity of each particle. The output of the solver provides the position and velocity of each particle at a sequence of user-specified intervals.

The universal gravitation and Newton's laws of motion are integrated into the solver's calculations. The universal gravitation states that the force $\mathbf{f}_{qk}(t)$ on particle q exerted by the particle k at time t is given by the formula:

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{|\mathbf{s}_q(t) - \mathbf{s}_k(t)|^3}\,[\mathbf{s}_q(t) - \mathbf{s}_k(t)],$$

where $G = 6.6743 \times 10^{-11}\,\mathrm{m}^{-3}\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$ is the gravitational constant, m_q and m_k are the masses and $\mathbf{s}_q(t)$ and $\mathbf{s}_k(t)$ are the positions of particles q and k at time t, respectively. Therefore, in an n-body system, the total force $\mathbf{F}_q(t)$ on particle q at time t is given by:

$$\mathbf{F}_q(t) = \sum_{\substack{k=0\\k\neq q}}^{n-1} \mathbf{f}_{qk}(t) = -Gm_q \sum_{\substack{k=0\\k\neq q}}^{n-1} \frac{m_k}{|\mathbf{s}_q(t) - \mathbf{s}_k(t)|^3}\,[\mathbf{s}_q(t) - \mathbf{s}_k(t)].$$

The position $\mathbf{s}_q$ and the velocity $\mathbf{v}_q$ of particle q can be found by utilizing the second law of motion [5]:

$$\mathbf{F}_q = m_q \mathbf{a}_q \Leftrightarrow \mathbf{F}_q = m_q \frac{\mathrm{d}\mathbf{v}_q}{\mathrm{d}t} \Leftrightarrow \mathbf{F}_q = m_q \frac{\mathrm{d}^2\mathbf{s}_q}{\mathrm{d}t^2},$$

where $\mathbf{a}_q$ is the acceleration of the particle q. To solve the aforementioned differential equations the Euler method [6] is employed. Specifically, given the velocity $\mathbf{v}_q$ and position $\mathbf{s}_q$ of a particle q at time t, their values at time displacement of Δt can be computed with the following formulas:

$$\mathbf{v}_q(t + \Delta t) \approx \mathbf{v}_q(t) + \Delta t\,\frac{1}{m_q}\,\mathbf{F}_q, \quad \mathbf{s}_q(t + \Delta t) \approx \mathbf{s}_q(t) + \Delta t\,\mathbf{v}_q.$$

For the sake of simplicity, the n-body solver assumes that the Gravitational constant and the masses of all particles have unitary values.

Methods

The starting point of the development was the n-body solver host-only code, provided in the final section of the NVIDIA course *Getting started with Accelerated Computing in CUDA C/C++*. The code file is included in the submitted files of this assignment, as `reference_code.cu`. This implementation of the n-body solver is a CPU only application, that initiates no kernels or memory transfers in the GPU device. The acceleration of the solver will be achieved by refactoring this reference code.

The body of the code is the `main` function in which the system size is passed as a command line argument. Then follows the initialization of hidden values in order to check the correctness of the code by scripts from the course. This section should not be fiddled with. The memory allocation of the number of bodies of the system is implemented with the `malloc` function. The central section of the `main` function contains a `for` loop that simulates the interactions of the n-body system over `nIter = 10` iterations. Each iteration is timed and from the average of the timings over all the iterations the final average execution time is calculated. In this section of the code *number of interactions per second* are also calculated, which is a metric inversely proportional to the average time. The `for` loop that models the system interactions contains a call to the provided `bodyForce` function, which calculates the force felt by all the bodies on the system and their resulting velocity. After this function call, there is another `for` loop over all the bodies of the system that calculates in each iteration the renewed position of each body, based on its previously calculated velocity. In essence the outer `for` loop calculates unit displacements and velocity change of all the bodies of the system over 10 iterations.

In this work some steps where followed in order to accelerate the execution of the aforementioned `for` loop of interactions, since this is the part of the code that it is timed. First of all it was crucial to target parts of the application that could be off-loaded to the GPU device to run there instead of the CPU. One such code section was the entire `bodyForce` function, which was refactored into a `__global__` kernel. Inside the `bodyForce` definition there are two nested `for` loops, that perform various calculations on attributes of the bodies of the system. Each iteration of the outer loop has no internal dependencies with the other iterations, therefore it is possible to perform *loop unrolling* assigning the execution of each iteration to different threads. Each thread will execute a number of iterations independent from the other threads. Another section of the code was wrapped into a `__global__` kernel as well. This was the `for` loop over all the bodies of the system that updates their positions based on their new velocities. This kernel was named `integratePosition` and contains this `for` loop over all the bodies, which, again, is unrolled, for each thread to perform a group of its initial iterations.

These two kernels are called one after the other without any need for synchronization, since they both execute in the same `(default) stream` and thus are executed in a sequential fashion. The need for synchronization arises after their consecutive calls, because there is the need to take timing snapshots of the execution. Therefore `cudaDeviceSynchronize` is used. To achieve optimal execution, the body data on which the two kernels act on, have to be available in the device beforehand, otherwise a large number of page faults will occur that will result in memory transfer from host to device, especially in the first iterations. This is because the data are initialized on the host. To avoid these data transfers during the simulation calculations `cudaPrefetchAsync` was used before the system simulation `for` loop, in order to transfer the body data to the GPU device, where they will be needed. This CUDA API was also used to transfer the resulting data back to the CPU after the simulation to produce the timing results, even thought this has no impact on the acceleration of the solver.

The execution configuration of the kernels was chosen based on device specific properties and general good practices in CUDA programming. More specifically, it was deemed important to have threads per block be a multiple of the warp size. It was found that for the GPU under test, which has a warp size of 32, the optimal results were for 128 threads per block (four times the warp size). Then the number of blocks aims at having work distributed evenly among them. This is achieved by dividing the size of the problem by the threads per block number. If they do not divide exactly then one additional thread block is employed.

Results

The application was tested on the NVIDIA A10G GPU device, which was available by the remote environment set up by the NVIDIA course. This GPU has 80 Streaming Multiprocessors (SMs), each one capable of launching up to 1534 threads. CUDA version 11.6 was used for the development of this application and it was compiled using the `nvcc` compiler on the virtual environment of the course. The reports in the submitted code files were generated by the `nsys` profiler. The application was also debugged on the NVIDIA Nsight Systems graphical profiling tool, using the generated reports from the `nsys` profiler.

The goal of the application was to accelerate its execution to run in *less than 0.9 seconds for for 4096 bodies* and *less than 1.3 seconds for 65536 bodies*. Of course the program must maintain its correctness. The way the NVIDIA course was set up it uses scripts that automate the validation of the submitted code; both in terms of acceleration and correctness.

The submitted code achieved the aforementioned requirements. For a problem size of 4096 bodies the execution configuration had 32 blocks, with 128 threads each, and resulted in **execution time of 0.000280 seconds** (280 microseconds) and 59.940 billion interactions per second. For a problem size of 65536 bodies the execution configuration had 512 blocks, again, with 128 threads each, and achieved **execution time of 0.009373 seconds** (9.373 milliseconds) and 458.223 billion interactions per second. The assessment scripts provided by the course validated the correctness of the execution, which led to the issuance of our certificates (see Figure 1).

Discussions

The configuration of the kernels was optimized to maximize the SM occupancy and facilitate load balancing. The occupancy of a SM is defined as the ratio of active warps to the maximum amount of warps that can reside on it. The question we addressed was whether using fewer and larger blocks was preferable to using more and smaller blocks. Namely, we investigated wether assigning finer-grained or coarser-grained work to SMs resulted in improved performance for the particular application. It is of utmost importance to note that the problem size in combination with our methods does not exceed the GPU's capacity. That is, the total number of threads needed by the n-body solver does not surpass the total number of threads the GPU can manage. Therefore, there is no need for strided loops or other techniques to compensate for the lack of computational resources. Furthermore, each SM in high-end NVIDIA GPUs[1] issues multiple instructions from different warps at the same time. This is achieved with multiple Warp Schedulers per SM or Warp Schedulers that issue more than one instructions per clock cycle. Some architectures, particularly those designed for cloud computing, combine both techniques.

[1]There are no resources found to specify the exact number of parallel SIMD instructions from each SM in NVIDIA A10G.

NVIDIA

Certificate of Competency

This certificate is awarded to

Alexandros Tsagkaropoulos

for demonstrating competence in the completion of

Getting Started with Accelerated Computing in CUDA C/C++

Greg Estes
Vice President, NVIDIA

Issue Date: : February 13, 2025
Certification ID: W0B3-OD0Qq-pzZ6Fity4Lg

NVIDIA

Certificate of Competency

This certificate is awarded to

Yannos Koutoulas

for demonstrating competence in the completion of

Getting Started with Accelerated Computing in CUDA C/C++

Greg Estes
Vice President, NVIDIA

Issue Date: : February 14, 2025
Certification ID: v-xuZE6ERdqGnlF9FKTpVw

Figure 1. Certificates from the course: *Getting Started with Accelerated Computing in CUDA C/C++*.

On the coarse side, if the number of threads per block reaches its maximum value, which is 1024, then no SM could achieve occupancy greater than 67%. This is due to the fact that the maximum number of threads each SM could support is 1536. In addition, a number of SMs will be stale since no work would be assigned to them. Since there is no intra-kernel synchronization points the coarser-grained approach should be avoided, due to low SM occupancy and poor load balancing properties.

On the other side, the threads per block will be minimal and each SM would hold many blocks. In this scenario, the least subscribed SM's load differs from the most subscribed SM's load by no more than the block size. Therefore, better load balancing is achieved by the Giga Thread Engine. Specifically, a smaller unit of assignment (block) for the Giga Thread Engine means that the SMs get close to the same work assigned to them. The smaller number of threads per block that makes sense would be equal to 32; the warp size[2]. Consequentially, the blocks that will be needed to fill an SM in the case of 65536 particles are 48, that might[3] not be supported by it. Despite the fine-grained approach being more optimized than the coarser-grained approach, it does not exploit every hardware feature available. Specifically, the parallel execution of multiple warp instructions cannot be utilized to the fullest, resulting in underutilization of the SMs. Therefore, the optimal configuration resides in the middle.

The choices in the middle should preserve the positive aspects of the finer-grained policy while addressing its weakness. This can be achieved by ensuring that the threads per block are a multiple of the warp size, multiplied by the number of parallel instructions supported. Also, in order to maintain the load balancing advantages of the fine-grained approach, the threads per block should be the minimum possible. Since we could not determine the exact number of parallel executing instructions we picked 128 as the number of thread per block, considering its slightly better performance compared to the other alternatives.

[2]Warps are executed in SIMD fashion by the SM. Consequently, to optimize the utilization of the SM's cores, all threads within a warp should perform meaningful calculations. To archive this, the threads per block should be a multiple of the warp size.

[3]We could not spot any resource to support that claim. The manuals are obscure and do not go into implementation details.

References

[1] Vasileios Karakostas. *Assignment 4 (Optional) – Programming with CUDA*.

[2] Peter Pacheco, Matthew Malensek. *An Introduction to Parallel Programming*. 2nd Edition.

[3] John Hennessy, David Patterson. *Computer Architecture; A Quantitative Approach*. 6th Edition.

[4] *Newton's law of universal gravitation*. https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation.

[5] *Newton's laws of motion*. https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion.

[6] *Euler Method*. https://en.wikipedia.org/wiki/Euler_method.