A Simple Many Body Code

T. Hater (t.hater@fz-juelich.de)

January 15, 2016

This project guides you through the process of porting a very simple program from a CPU-only implementation to an optimised CUDA code, focusing on central optimisation techniques.

Keep a log of changes and the corresponding performance data. You should use the GitLab repository to track your progress. Collect interesting results for your presentation.

Introduction

Before we start working with the GPU, we are going to start with the original program which will be ported to CUDA. The example is a simple N-body simulation of gravitational interaction. A set of N particles of mass m is uniformly distributed in the unit box $[0,1)^3$. The evolution of the system is given by the equations of motion

$$\ddot{\mathbf{r}}_i^{(n)} = \sum_{j \neq i} G \cdot m \cdot \frac{\mathbf{r}_i^{(n)} - \mathbf{r}_j^{(n)}}{\left| \mathbf{r}_i^{(n)} - \mathbf{r}_j^{(n)} \right|^3}$$

For handling the temporal integration, we are going to use a simple Euler scheme to compute

$$\begin{aligned} \dot{\mathbf{r}}_i^{(n+1)} &= \dot{\mathbf{r}}_i^{(n)} + \Delta t \cdot \ddot{\mathbf{r}}_i^{(n)} \\ \mathbf{r}_i^{(n+1)} &= \mathbf{r}_i^{(n)} + \Delta t \cdot \dot{\mathbf{r}}_i^{(n+1)} \end{aligned}$$

Analysis

Familiarise yourselves with the source code, which is quite short and straightforward. Next, develop an understanding of the critical paths.

1. Profile execution on a single core and identify important routines.
2. Identify the performance limiters by analysing the algorithm. Verify the hypothesis using performance counters. Do you expect a different limiter on the GPU?
3. Design and implement a measure for correctness. Check this criterion for all tasks!

This extends to all tasks. Document what is working and what not. Performance is only relevant if the code is correct.

Porting and Basic Optimisations

Your first task is to provide a working implementation for all the kernels in CUDA. Insert the proper data movements and kernel invocations; eliminate superfluous data movements.

Document the expected bottlenecks for both kernels on the CPU and the GPU. Take into account vector execution units, instruction throughput, bandwidths and latencies (cache and memory) etc.

Storage Format

To improve memory access replace the storage of the particle data from a single array holding structures of vectors. The new storage format should use two arrays of *four element vectors*, where the last element in each vector is left untouched.

Obviously, this is a trade-off between space and (possibly) speed. What effect will this have on the maximum simulation size?

For GPUs, this change allows for better *coalescing*, i.e. the combining of memory requests from different threads into larger chunks. This in turn reduces the pressure on the memory system.

Shared Memory

Next, we are going to utilize *shared memory* to further speed up memory access. Both this and coalescing are central ideas for improving the performance of CUDA programs, which will come up frequently.

Shared memory is essentially like a cache on a CPU: small, fast and shared between cooperating threads. However, we have to manage this memory explicitly.

1. In the kernel add a shared memory array for the particle positions so that every thread in a block can store a position.
2. Split the loop over all N particles interacting with the local one into small loops processing a tile of particles at a time.
3. Before processing each chunk, copy the local particle's position into the shared memory array and synchronize.

By having each thread copy the local element, all threads will fill the chunk together. As all threads need to read this data, the greater access speed to shared memory will increase the performance. Do not forget the synchronization!

Tuning

In this task, we consider micro-optimisations that can potentially make the program faster. Most parameters require empirically searching for the best value. Therefore, these are finishing touches on a porting project.

1. Using faster primitives for the inverse square root
2. Eliminating unnecessary conditionals
3. `#pragma unroll <n>` for relevant loops.

4. Tuning the launch parameters of the kernels.

Back to the CPU Code

After the previous changes, the CUDA code is likely many times faster than the CPU code. This is a common pattern, when code is ported and optimised. Often, speed-up numbers of accelerated applications are overestimated since the first serious attempts at optimisation went into the GPU code.

Go through the list of optimisations above, decide if they make sense for the CPU and if so implement them. Measure the utilisation of the most constraining resource. Make sure that the generated code is vectorised and uses the memory hierarchy efficiently.

Next, add shared memory parallelisation using OpenMP and tune the OpenMP parameters for thread placement and memory locality. Find the optimal number of threads. Can more parallelism be extracted? Document the final product using hardware performance counters and compare its performance against the GPU implementation.

Choose your measure of 'optimal' as either efficiency or time to solution.

Optional: MPI Parallelisation

If you have time left, you can consider doing the following task. Design a method for parallelising this problem to multiple GPUs over MPI. Discuss and explore scalability and options for improvement. You might want to consider a different algorithm and give a quick analysis.