Advanced Topics in Numerical Analysis, High Performance Computing: Assignment 4

Mariya Savinov, mariyasavinov@nyu.edu

Repo location: `https://github.com/mariyasavinov/HPC_Spring_22.git`

1 Matrix-vector operations on a GPU

After successfully implementing a parallelized element-wise product between two-vectors (which could then be summed to achieve the inner product of the two vectors), I generalized this code to a matrix-vector multiplication (both on the CPU with OpenMP and on the GPU with CUDA). For the CPU part, the CPU parallel section is completed with the 32 threads on both of the following machines.

On `cuda2`, which has an `NVIDIA GeForce RTX 2080 Ti` GPU and `Intel Xeon E5-2660 v3` CPU (2.60 Hz), we attain the following CPU and GPU timings and bandwidth estimates:

```
CPU 0.066537 s
CPU Bandwidth: 6.051574 GB/s

GPU 0.029899 s, 0.003756 s
GPU Bandwidth: 13.467070 GB/s

Error = 0.000000
```

On `cuda3`, which has an `NVIDIA TITAN V` GPU and `Intel Zeon Gold 5118` CPU (2.30 Hz):

```
CPU 0.116728 s
CPU Bandwidth: 3.449496 GB/s

GPU 0.033467 s, 0.002017 s
GPU Bandwidth: 12.031516 GB/s

Error = 0.000000
```

Note that the first number for the GPU is total time including memory transfer to/from CPU, while second excludes the memory transfer. It is clear that the implementation on the GPU is significantly faster, with time dominated by the communication between the CPU and GPU rather than the actual computation.

2 2D Jacobi method on a GPU

The CPU parallelized version of the Jacobi method in 2D is retained from assignment 2, which has fairly large runtime for even small $(N+2) \times (N+2)$ 2D grid sizes because we made use of very inefficient `if` statements. However, since we know that the originally submitted CPU code converges (the residual reduces at each step appropriately), we retain it in full without changes in order to assess the accuracy of the GPU implementation. As before, the CPU parallelization is done with 6 threads.

We test the CPU results against the GPU results on `cuda2`, which has an `NVIDIA GeForce RTX 2080 Ti` GPU and `Intel Xeon E5-2660 v3` CPU (2.60 Hz). Testing for $N = 32$, $N = 64$, and $N = 128$ for 100 iterations of Jacobi, we find that:

```
N=32
--------
Using CPU with OpenMP:
CPU 0.251168 s

Using GPU with cuda:
GPU 0.003281 s, 0.003195 s
Error = 0.000000
--------
N=64
--------
Using CPU with OpenMP:
CPU 7.328198 s

Using GPU with cuda:
GPU 0.005350 s, 0.005180 s
Error = 0.000000
--------
N=128
--------
Using CPU with OpenMP:
CPU 78.416327 s

Using GPU with cuda:
GPU 0.008669 s, 0.008427 s
Error = 0.000000
```

It is clear that the GPU implementation is correct, as compared to the same computation with the CPU implementation from assignment 2. Notably, the GPU implementation is significantly faster, especially for large grid sizes. We expect that even if the CPU implementation was modified to eliminate inefficient aspects, that the GPU performance would be faster for sufficiently large N values (at small N values, it is expected that the memory transfer would hurt the GPU performance in comparison to the CPU). Notice though that unlike for the first problem, memory transfer to/from the CPU takes up significantly less of the total time of the GPU computation since much smaller sized matrices/arrays are being used here.

3 Update on final project

Thus far, Paul Beckman and I have implemented and are in the process of debugging a serial version of the Barnes-Hut multipole algorithm built on a tree data structure. It has become throughout this process that it may be difficult to parallelize the tree-based implementation, due to its reliance on recursive function calls and pointer linkages. Our plans are to complete a basic OpenMP parallelization on the tree-based implementation, but then to experiment with developing an alternative array data structure implementation for more efficient parallelization.