HW 5: GP-GPU Programming with CUDA
Issued: April 29, 2019
Due Date: May 13, 2019 10:00am - Mandatory Part
May 27, 2019 10:00am - Optional Part

Warm-Up & Information
This homework requires the use of CUDA-enabled nodes of CSCS Piz Daint. Along with this pdf, you should download the code containing:

- task1/ The skeleton code for Task 1.
- task2/ The skeleton code for Task 2.

Grading
- All students: Choose one of either tasks in this homework. Submitting a correct implementation of the chosen task along with a report answering every question equates to a 6.0 grade. Submit your solution for the chosen task within 2 weeks (May 13th).

- Bonus (1/2): Correctly solving the other task within 4 weeks (May 27th) will grant additional 1.0 point to improve the grade of this homework or any other homework.

- Bonus (2/2): Each one of these tasks have an optional item which, solved correctly, will grant each 0.5 additional bonus points.

Guidelines for Moodle submission:
- Either (heat2d_gpu.cu and heat2d_gpu.hpp) or (nbody_opt.cu). You can include both solutions if you choose to also submit the optional task.
- Your report in PDF format. Do not forget to answer every question.

---

1At our discretion, we may subtract points for missing/incomplete answers or erroneous code.
2If you deliver both tasks and their optional items correctly, you will receive 2 bonus points to use in this or other homework, not to exceed 6 points.
Task 1: Heat2D on CUDA

Just when you thought you could sit back and enjoy your success as the head of operations in the car chassis factory, you receive an urgent call from the CEO: «I have received news from our corporate informants that our main competitor is improving their processes by adopting new computational technologies. Some say they are using graphical processing units to evaluate their models! Aren’t those just for videogames!? Anyway... you have two weeks to find out whether we can catch up with them. I will call my friends at CSCS to grant you access to use their computers.»

Fearing that you would lose your status as an up-to-date professional, you set your goal to optimize the Heat2D solver you used for the n-Candle problem to work with CUDA. First, develop the CUDA version of Heat2D by modifying the heat2d_gpu.cu and heat2d_gpu.hpp files.

**Engineers:** Parallelize the `applyJacobi`, `calculateResidual`, and `calculateL2Norm` operations using CUDA. **CSE:** Parallelize every kernel in the model.

**Important:** For correctness, make sure the L2Norm is equal to the one produced by heat2d_cpu.

Second, answer the following questions in your report:

1. Explain the criteria you used for allocating and copying memory back and forth between device and host.

   Provided that your GPU has enough memory, you should allocate permanent memory for all the necessary fields on all grid levels at initialization time, in exactly the same manner that you treat the CPU memory:

   ```c
   gridLevel* g = (gridLevel*) malloc(sizeof(gridLevel) * gridCount);
   for (size_t i = 0; i < gridCount; i++)
   {
     g[i].N = pow(2, N0-i) + 1;
     g[i].h = 1.0/(g[i].N-1);
     
     cudaMalloc((void **) &g[i].dU, sizeof(double) * g[i].N * g[i].N);
     cudaMalloc((void **) &g[i].dUn, sizeof(double) * g[i].N * g[i].N);
     cudaMalloc((void **) &g[i].dRes, sizeof(double) * g[i].N * g[i].N);
     cudaMalloc((void **) &g[i].df, sizeof(double) * g[i].N * g[i].N);
   }
   ```

   If you do not have enough memory on the GPU, you can allocate only the largest grid level (and associated fields); however, this would require copying

---

back and forth from the host to the device anytime you have to work on a different grid, therefore increasing the memory copy operations.

Regarding memory copying: the GPU memory is permanent, so if you perform all operations on the GPU (you wrote every kernel in CUDA) you can reduce the number of copy operations to two: copying the initial guess to the GPU and, once you reached convergence, copy the results back to the CPU for saving them to file (you could even generate the initial guess directly on the GPU, to eliminate one more operation). If you implemented in CUDA only some of the kernels, then you need to copy back and forth between GPU and CPU anytime you need updated data on one or the other device.

2. Explain what challenges did you find when adapting calculateL2Norm. How did you solve them?

calculateL2Norm consists of two operations: point-wise squaring the residual and computing the sum of all its point values, reducing to a single scalar. The first operation is rather straightforward and can be implemented in a standalone kernel:

```c
__global__ void squareKernel(double* res, size_t N)
{
    size_t myRow = blockIdx.y*blockDim.y+threadIdx.y;
    size_t myCol = blockIdx.x*blockDim.x+threadIdx.x;

    size_t index = (myRow*N + myCol);

    if (myRow >= N || myCol >= N) return;

    res[index] = res[index]*res[index];
}
```

The second operation requires to perform a reduction, which can benefit from the use of shared memory. The solution code implements an unoptimized reduction: each thread loads one element from global to shared memory, where the reduction is then performed. Note that the reduction operation is implemented on a one-dimensional array, not on a two-dimensional one. This not only simplifies the kernel, but also allows for a reduced memory usage as compared to a two-dimensional version (can you explain why?). The only drawback is that care needs to be taken when calling the kernel from the solver.

3. CSE only: Explain what challenges did you find when adapting applyProlongation and applyRestriction. How did you solve them?

The main issue in these kernels is the use of different grid levels that need to correctly be used in the kernels. Additional issues when parallelizing these functions arise in the correct use of shared memory as well as in the correct
use of the the array indexes when their size is not a multiple of the number of threads per block. One way to solve this is to load boundary elements in threads assigned to elements next to the array bounds.

4. Analyze the performance you obtained with your GPU implementation on each individual kernel (not the entire model) and compare it with the CPU performance. What can you say about using the GPU instead of the CPU for each kernel?

Overall you should observe a speedup across all kernels implemented on the GPU. If you test different block sizes you should also find that a configuration with $16 \times 16$ threads per block is the best performing on a grid of $N = 2^{10} + 1$ elements. As can be expected, the greatest speedup is achieved in the kernels with the highest amount of computations, i.e. smoothing and prolongation, while other kernels only marginally benefit from a CUDA implementation, at least for this grid size. Moreover, higher speedup is achieved on the grids with more elements. The overall speedup is over $10 \times$.

5. **Optional: 0.5 Points** Optimize your applyJacobi kernel using shared memory. What challenges did you find when dealing with boundary cells? How did you solve them? Did you observe any speedup in your solution? Explain.

As for applyProlongation and applyRestriction, threads next to boundaries need to load extra elements into the shared memory. Overall, the use of shared memory does not improve the computational time, at least for this grid size, since the threads have to perform the additional copy operation, and extra care needs to be taken with conditional statements.
Task 2: Optimizing the N-body Problem

An N-body simulation approximates the motion of particles, often specifically particles that interact with one another through some type of physical forces. Using this broad definition, the types of particles that can be simulated using n-body methods are quite significant, ranging from celestial bodies to individual atoms in a gas cloud.\(^4\)

Besides its importance in scientific simulations, the N-Body method is also an interesting motif for optimization on GPUs. To calculate the force on a particular body (e.g., Planet), an evaluation of its force with the other n-1 bodies is required. Hence, this method typically has an $O(n^2)$ complexity upper bound. Furthermore, each of these evaluations contains a set of expensive operations and memory access patterns, making this a great basis for optimization techniques.

In this task, we will apply the optimization techniques we have seen in lectures and exercise sessions to improve the performance of an unoptimized CUDA version of this code (nbody_naive.cu), which only calculates the forces for a given N bodies system.

- Optimize the N-Body solver by working on nbody_opt.cu. For correctness, make sure your code passes the verification and produces the same forces as the base naive version. You are allowed to make all the transformations you consider necessary, except changing the problem size and the solving method. If in doubt, ask us on Piazza.
- Write about all the optimizations you have applied in your report annotating their time-to-solution (TTS), speedup compared to the base code, and what difficulties you faced in implementing them.

**Grading:** We will post on Piazza a table with the timings you need to reach for:

- Baseline TTS for Pass (Engineers)
- Baseline TTS for Pass (CSE)\(^5\)
- Baseline TTS for Bonus 0.5 Points (Engineers)
- Baseline TTS for Bonus 0.5 Points (CSE)\(^6\)
- Baseline TTS for Honor Degree of Ultimate Master of GPU Computing\(^7\)

\(^4\)http://physics.princeton.edu/~fpretori/Nbody/intro.htm  
\(^5\)The optimizations seen in class should suffice to reach this time.  
\(^6\)This time may require further research. Recommended read: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html  
\(^7\)And we will post time on Piazza for everyone to bask in your awesomeness. Reaching this time may need some serious researching.
The optimized version of the N-Body solver can be found in `solution_codes/task2/mpi_opt.cu`. The list of optimizations applied and the speed-up achieved, based on time-to solution, for every optimization are listed below:

- **Naive implementation**: Time $\approx 17.0$ s.
- **Add shared memory**: Shared block "s" can be allocated statically or dynamically, difference in timing shouldn’t be noticeable. Time $\approx 13.3$ s.
- **Register Memory for temporal result storage**: Time $\approx 9.5$ s. In the above code snippet, introduce constants `myForX`, `myForY`, `myForZ`, update those in the for loop for every block. Update the sum of forces for every particle in the end of the kernel.
- **Constant / Arithmetic Optimizations**: Time $\approx 4.40$ s. In the above code snippet, introduce constant variables `myPosX`, `myPosY`, `myPosZ`, `myMass` for the position and mass of each particle $m$, so that they are used in computations in the loops. Furthermore, compute $r$ as $1/r$ and introduce the constant variable `mr3` in the second for loop to avoid calling the same variables three times.

```c
__global__ void forceKernel(double* xPos, double* yPos, double* zPos, double* mass, double* xFor, double* yFor, double* zFor, size_t N)
{
    size_t m = blockIdx.x*blockDim.x+threadIdx.x;
    size_t t = threadIdx.x;
    extern __shared__ double s[];
    for (size_t b = 0; b < N; b += blockDim.x)
    {
        s[4*t + 0] = xPos[b+t];
        s[4*t + 1] = yPos[b+t];
        s[4*t + 2] = zPos[b+t];
        s[4*t + 3] = mass[b+t];
        __syncthreads();
        for (size_t i = 0; i < blockDim.x; i++) if (b + i != m)
        {
            double xDist = xPos[m] - s[4*i + 0];
            double yDist = yPos[m] - s[4*i + 1];
            double zDist = zPos[m] - s[4*i + 2];
            double r = sqrt(xDist*xDist + yDist*yDist + zDist*zDist);
            xFor[m] += xDist*mass[m]*s[4*i + 3] / (r*r*r);
            yFor[m] += yDist*mass[m]*s[4*i + 3] / (r*r*r);
            zFor[m] += zDist*mass[m]*s[4*i + 3] / (r*r*r);
        }
        __syncthreads();
    }
}
```
• Use better alternative to 1/sqrt(): Time ≈ 3.15 s. In the above code snippet: Use the math operator rsqrt() instead of 1/sqrt().

• Avoid divergence (avoid if clauses): Time ≈ 2.73 s. In the above code snippet. Introduce tolerance YPSILON, in order for the variable r never to diverge, remove if (b + i != m).

• Create structure information to minimize communication overhead. Unroll for loops: Time ≈ 2.5 s. See solution in solution_codes/task2/mpi_opt.cu.