CS 758
Fall 2019
Homework 0
Suggested Completion: Thursday, September 19th, 2019

This assignment is purely intended as practice, it will not be graded. Thus, I encourage you to do this assignment on your own, if you feel you need additional practice writing CUDA code. You are also encouraged to talk with classmates in person or on Piazza about any issues you may have encountered. I will also post solutions to this assignment on Canvas.

What to Hand In

Since this is a practice assignment, you do not need to turn anything in. However, you are encouraged to measure the performance of your solution compared to the sequential CPU version, as well as measure the performance of any GPU optimizations you may apply to the code.

Total Points: 0

Description

The purpose of this assignment is to familiarize yourself with GPU programming using CUDA, different CUDA programming features and optimizations to improve performance. To practice these programming skills, we will implement an iterative, parallel method to solve systems of linear equations using **Jacobi's Method**. Given a matrix *A*, and vectors *x* and *b*, let $Ax = b$ be a square system of *n* linear equations. Jacobi's method exploits the fact that the matrix A can be decomposed into a diagonal component (*D*) and the remainder (*R*). The algorithm first makes a random guess at what the solution vector $x^{(k)}$ is. Then, Jacobi's method iterates on this, progressively obtaining a more refined solution $x^{(k+1)}$ by solving for each individual element of *x*, assuming that all other elements of *x* are fixed. The derivation of Jacobi's method is relatively simple; for example, consider the calculation for the i^{th} equation in the system:

$$\sum_{j=1}^{n} a_{ij} * x_j = b_i$$

If we solve for x_i while keeping the other values of *x* fixed, the method becomes:

$$x_i = \frac{b_i - \sum_{j \neq i} a_{ij} * x_j}{a_{ii}}$$

Effectively, this calculates the value of one solution cell based on the current value of all the other cells in the solution. Written in the iterative form (we are solving for the kth iteration based on the values from the (k-1)st iteration) yields:

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{ij} * x_j^{(k-1)}}{a_{ii}}$$

Previously, the Jacobi Method had limited applicability and was less popular than other solutions like Gauss-Seidel, which converges more quickly but is not easy to parallelize. However, the ease of parallelizing Jacobi and the rise of parallel computing have made the Jacobi Method more attractive. The serial version of the algorithm is:

```
                Algorithm 1: Jacobi Method
1: procedure Jacobi
2:    Input: nxn matrix A, vector b, initial guess solution
x(0)
3:    Output: Solution in x vector
4:    k ← 0
5:    while convergence not reached do
6:         for i:= 1 to n do
7:              σ ← 0
8:              for j := 1 to n do
9:                   if i != j then
10:                       σ ← σ + aij*xj(k)
11:             xi(k+1) ← (bi – σ)/aii
12:        k ← k + 1
```

Your job is to exploit the parallelism in this algorithm on the GPU, given multiple inputs.

Coding Process

1. **Serial Version**: create a sequential version of Algorithm 1.
    a. **Verification**: This version should be able to run on a CPU (e.g., a CPU from the CSL). To verify that your code gets the correct answer, simply multiple *Ax*, where *x* is your computed results vector. Compare this result to the input *b* vector – if your solution *Ax* is within 1.0 for each value in the input *b* vector, you can consider this result correct.
    b. **Convergence Criterion**: Note that since this is an iterative algorithm, and based on an initial guess, the "until convergence" criteria should just be to run for a fixed number of iterations. Each input file as a specific number of *k* iterations you should run for embedded in the file.
    c. **Requirements & Assumptions**: Your program should be able to parse the input file, allocate memory as appropriate, and then proceed to do the computation, verification, etc. You can assume that the initial guess for the *x* vector is all zeros.
2. **GPU Version**: create a GPU version of your serial code in Step 1.
    a. Use the i-loop-level parallelism in the algorithm to create a simple GPU kernel that solves the problem using discrete GPU memory transfers (i.e., explicitly copy the data from the CPU to the GPU).

    a. Hint: cudaMalloc and cudaMemcpy are the right things to use for copying memory between the kernels.
    b. To judge how your algorithm performs for a variety of inputs, you should run it for all 3 inputs in the attached tarball (256.txt, 1024.txt, 5120.txt). I have also included a very small input (8.txt) to use for testing and debugging.

b. (Optional) **Optimizations**: the code you wrote in Step 2a is likely highly unoptimized GPU code (although it is likely still much faster than the CPU code!). There are many optimizations you could consider to improve performance. For example:
    a. You may consider optimizing the register allocation.
    b. You can fetch multiple words from memory simultaneously using data types like float2 or float4.
    c. You can take advantage of GPU-specific memories like shared memory, constant memory, or texture memory.
    d. You can use keywords like __restrict__ to help the compiler optimize your code.
    e. Make sure to tell the compiler to generate code for the appropriate GPU architecture.
    f. You can adjust the number of threads per thread block to find a "sweet spot" in terms of performance.
    g. You can tile your accesses to improve cache reuse.
c. (Optional) **Unified Memory**: Recent GPUs have also added support for unified memory. Compare the performance of your solution that explicitly copies the memory from the CPU to the GPU vs. one that uses unified memory. Note that memory management is included in kernel times when using unified memory (but can be broken out separately when you use explicit copies).
    a. Hint: cudaMallocManaged or cudaMallocHost are a good places to start.

Where to Run

By enrolling in CS 758, you all should already have accounts on the CSL lab machines. To run your sequential version, you can run it on any machine in the lab. You may also run it on a personal laptop, if desired. If you do not have a CSL account, please let me know.

To run the GPU version, you should have followed the directions on Piazza to create an account on the euler cluster. If you have not received your login, please let me know. Keep in mind that Professor Dan Negrut has graciously allowed us to use this cluster for 758. So please be mindful of the shared nature of this cluster. *I will update this part of the assignment with additional information from Professor Negrut about cluster use and rules once I receive it.*

The euler cluster uses the SLURM *sbatch* submission mechanism to submit/run jobs. If you need help writing a SLURM submission script, WACC has a (beta) tool to help with this.

You shouldn't have any problems as long as your code finishes quickly and you don't leave cuda-gdb open for long periods of time (they have come across a few bugs where cuda-gdb sometimes blocks access to all other GPUs).

Information on the hardware provided in the Euler cluster is available here. You may use any free GPU in the cluster, although I personally suggest using the GTX 1080 (Pascal-class GPUs), as there are large number of them (so contention should be less than other, newer GPUs in the cluster) and they have a lot of CUDA cores (2560).

You may also use any version of CUDA you want – as of last month, the default version installed on euler was CUDA 9.2, which I recommend using instead of installing your own version.