CPEN 512 Assignment 3

Parallel Gaussian Elimination using CUDA

Muchen He, 44638154, Oct 20, 2020

1. Introduction

As discussed in previous assignments, are are implementing the parallel version of the serial forward-elimination (FE) process during the Gauss-Jordan elimination operation of a matrix. In this assignment, we explore the potential speedups provided by the parallel SIMD or SPMD model provided by NVIDIA's CUDA; and evaluate the performance against the serial case. Because `nvcc`, NVIDIA's C compiler, as well as `nvprof`, NVIDIA's profiler tool does not appear to be available on the cpen512.ece.ubc.ca computer, these programs are compiled and run on my personal computer using with NVIDIA's GeForce GTX 1070 GPU.

2. Background

The same assumptions outlined in the previous assignment reports still apply: we assume that the matrix we are operating on is invertible and all elements start off with random non-zero values. Furthermore, the matrix has number of rows less than or equal to number of columns.

CUDA [1] is a parallel computing platform that provides API to write code, or *kernels* for the GPU hardware to execute — utilizing its architecture to exploit data-level parallelism. Each kernel runs on a different thread processing a single set of data. These threads can be organized into blocks, and the blocks can be further grouped into grids. CUDA provides methods to configure these grid and blocks in one, two, or three dimensions to further fine-tune parallelism but I won't experiment with this. More details on implementation can be found in Section 3.

3. Implementation

As discussed in Section 2, we use CUDA to exploit the data-parallelism in FE algorithm. We can extract two part of the FE process that can be data-parallelized: (1) scaling the pivot row such that the leading coefficient is unity, and (2) subtracting rows below the pivot row such that the element sharing the same column as the pivot is 0.

To perform (1), the *host*, the CPU that's running the main program code, issues a kernel call to the GPU such that each thread divides the value of the leading coefficient. To perform (2) naïvely, we can make a for-loop to iterate each of the remaining rows and subtract the pivot row multiplied by a factor. However, NVIDIA profiler shows that excessive kernel calls that resulted from this for-loop takes up 98% of the execution time, so the next step is to combine all of the subtraction step for all rows below the pivot into a single kernel call. For example if there are 16 rows remaining, the host issue kernel for (2) once with 16 times the elements instead of issuing kernel for (2) 16 times.

This optimization requires the kernel to be modified to compute the correct matrix indices, but the speed up is significant. Step (1) and (2) can be further optimized by ignoring the values in the matrix. For example, during (1), there is no point scaling elements in the row that's before the leading coefficient — since assuming step (2) for the previous row is correct, these values would be 0. Same logic applies for subtracting rows in step (2).

Thread synchronization in the threads as well as device synchronization in the host code are required. Thread synchronization using `__syncthreads()` are used so that each thread can obtain the required leading coefficient value, before mutating the matrix. Device synchronization using `cudaDeviceSynchronize()` are used since step (2) depends on step (1), and step (1) depends on step (2) of the previous row. These synchronizations makes up the majority (88-95%) of the CUDA execution time overhead.

Threads Per Block	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Execution time	1.351	1.353	1.353	1.355	1.357	1.360	1.358	1.359

Table 1. Execution time in seconds using different CUDA block size

4. Performance

The performance of the CUDA implementation is compared against the serial case using a single core on the CPU (Intel i9 9900K). First, I test the evaluation time of the CUDA program based on number of threads-per-block and keeping the matrix size constant ($M = 4096$). Table 1 shows that execution time varies very little. Next, we measure the speedup of the CUDA implementation against the serial implementation. The code is compiled using nvcc with optimized compiler flags[1]. Note that certain optimization flags such as tree-vectorize or loop-unroll could not be used as it was not available with nvcc. Figure 1 shows the speedups with matrix sizes varying from 16 up to 4096. The best speedup of **5.2×** was observed with largest matrix ($M = 4096$) as expected. Figure 2 is the log-log execution times versus matrix size cubed and shows the comparison more clearly — that the CUDA version starts to outperform the CPU only version for $M > 512$.

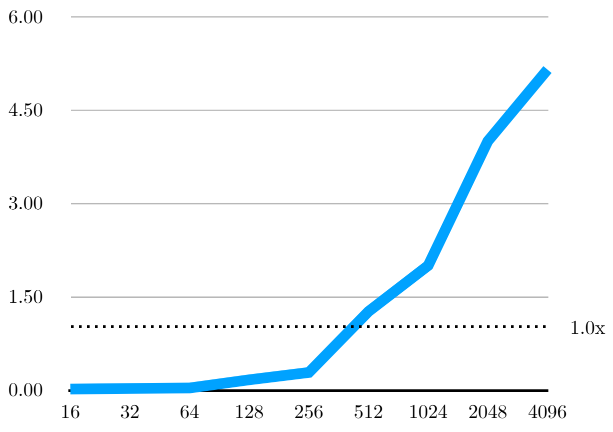

Figure 1. Speedups with different matrix sizes M

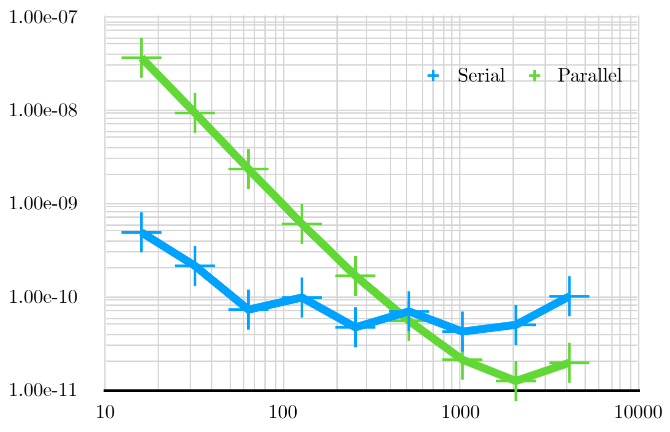

Figure 2. Normalized execution time of serial and parallel executions

[1] Custom compiler flags: —`O2 -m64`