Simplex Algorithm - CUDA Implementation

By Cody Cziesler and Praneeth Pulusani

Abstract:

Linear optimization, or linear programming (LP), is model used to determine the maximum or minimum of an objective function with constraints. There are many different algorithms that are used to solve LP problems; the Simplex Method is one of the simpler algorithms and contains steps in the algorithm that are parallelizable. In this investigation, the Simplex Algorithm was initially implemented to solve LP problems in a sequential model to run using the CPU. After verifying that results were accurate, the same algorithm was parallelized on the CUDA programming platform to run on the GPU in two slightly different variants. In the first variant, global memory was used. In the second variant, page-locked zero copy memory was used. Then, the results were compared with the sequential version to observe the difference in performance in terms of response time and speed up. All three implementations correctly solved linear optimization problems. Unexpectedly, the GPU versions of the program performed worse than the CPU implementation. It was speculated that this phenomenon occurred due to a high communication to computation ratio. The size of the matrices created in the lab adversely affected the speed up and it was likely because while the size of the matrix got bigger, the laboratory born matrices were not computation intensive and therefore, the time for the large memory transfers outweighed the benefits of parallel computation. In all, this exercise was completed successfully because LP solvers using the simplex method were created both sequentially and in parallel.

Procedure/Design:

The input matrices were loaded into the program in two different ways. In the first method, the functions, constraints, and slack variables of textbook Linear Optimization problems were converted into a one dimensional array formatted manually. The solutions were known and they were used to check the accuracy of the CPU and GPU implementations while developing. In the second method, MPS-formatted files were obtained from the Linear Programming Datasets[1] directory. MPS-formatted files are column-oriented text files with rows, variables, and constraints having additional aliases. The format originated during the time of punch card machines and continues to this day as the same files are used in benchmarks for current applications. MPS-formatted files were converted into *new line value* formatted files (NLV). The NLV file was then parsed into a one dimensional array format that included all the constraints and slack variables. The conversion from MPS to NLV was done via the LPSolve IDE 5.5 program with help from a custom Java program. The conversion from NLV to the one-dimensional array was done through a helper C++ function. To compare the performances, timing of each implementation was measured and displayed on the screen.

For timing measurements, the current time, obtained by the ctime library, is recorded before the reduction is executed. After the reduction executes a predefined number of times, another time is recorded. The time is calculated by dividing the total time with number of iterations. For the GPU, the same process is used, but a warm-up is done before the time is obtained as starting the CUDA kernel to be executed takes some time.

In the algorithm, there are four general steps -- convert problem to initial tableau, choose a pivot column and pivot row, perform row reduction, and repeat until the indicator row is all positive. Listing 1 shows the Simplex Algorithm pseudocode.

```
SIMPLEX:
WHILE (bottom_row_is_all_positive? and max_iterations_not_reached?)
 get_pivot_column_index
 get_pivot_row_index
 modified_gaussian_elimination:
   normalize pivot row                          // This can be parallelized
   row reduction on rows other than pivot row   // This can be parallelized
ENDWHILE
```

Listing 1: Simplex Algorithm on a CPU

In the sequential (CPU) implementation, the high level simplex algorithm was organized into one function, and the lower-leveled row reduction was organized into a second function. In addition, many other helper functions were written to keep the code organized. In the outer simplex level, the cycle was repeated as long as the last row, the indicator row, had all positive values or the maximum number of iterations was reached. Inside the loop, row reduction was performed. In this row reduction, the pivot row and pivot column were chosen according to the Simplex Algorithm. More specifically, the pivot column was selected by choosing the column that has least positive number in the indicator row, excluding the solution column. Then, the pivot row was selected by comparing the ratios of all the elements in the solution column over the element in the corresponding pivot column. The row with the lowest ratio was picked and again, with the indicator row excluded. Once the pivot row index and pivot column index were obtained, the standard elementary row operations were performed for that column. If the maximum iterations was reached in the outer loop, it meant that a solution was not found. Otherwise, a solution was found and it was displayed on the terminal.

In the GPU implementation, two different kernels were used; one for normalization and one for row reduction. For normalization, one-dimensional blocks were used with 512 threads and the number of blocks depended on the size of the problem matrix. For row reduction, two-dimensional square blocks were used with 32 threads on each side, resulting in 1024 threads per block. Again, the number of blocks depended on the size of the problem matrix. In the GPU, the outer loop, or the high level Simplex, could not be parallelized as the indicator row needed to be checked for positivity once all the threads in all the blocks were completed. Threads from multiple blocks can only be synchronized outside the kernel, so the kernel calls needed to be sequential. However, elementary row operations and normalization were parallelizable and they were parallelized.

In addition to the normal GPU implementation, a zero copy page-locked memory version was implemented. In this version, instead of transferring memory back and forth between GPU and Host, the memory was allocated on the host using a special `CUDAHostAlloc`() function and the allocated memory was accessed using its pointer from the device as well as the host. It was thought that the page-locked memory version would reduce the amount of communication between memories, reducing the execution time.

Results/Analysis:

Table 1: Simplex Algorithm Timing Results

Number of Columns (size)	CPU Time (ms)	GPU Time (ms)	GPU (Page-locked) Time (ms)	GPU Speedup	GPU (Page-locked) Speedup
100	0.15	0.67	0.39	0.224	0.385
200	0.54	1.04	0.95	0.519	0.568
500	3.16	9.42	9.77	0.335	0.323
750	7.1	38.3	39.9	0.185	0.178
1000	12.7	113.1	117.5	0.112	0.108

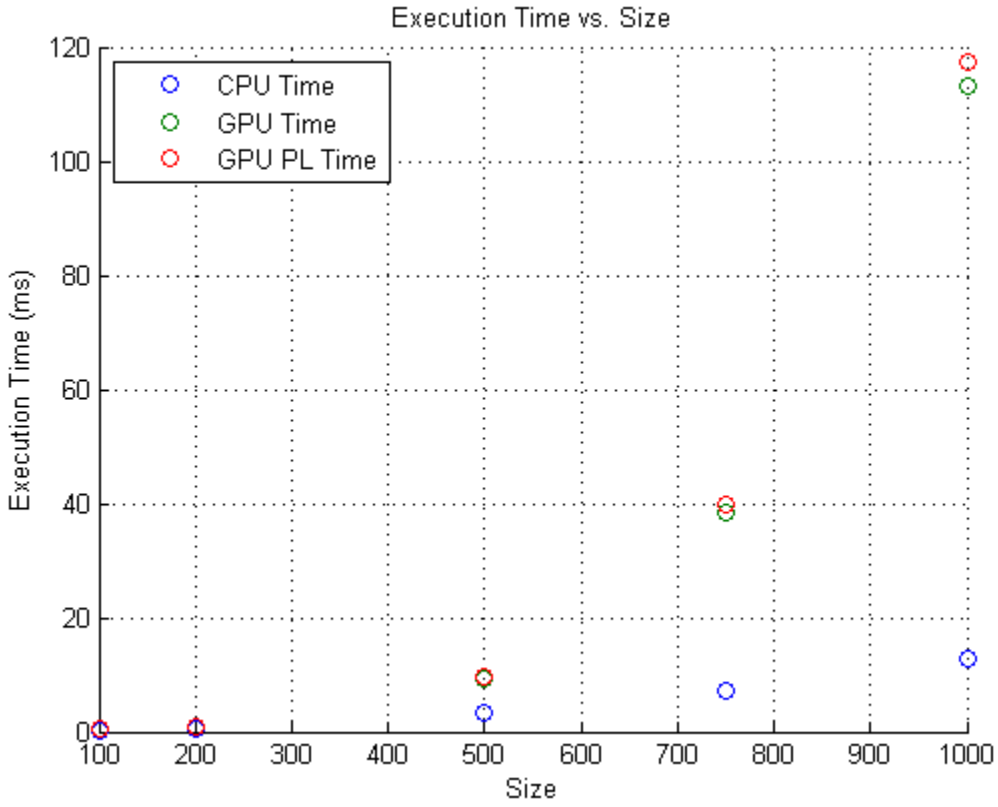

Figure 1: Execution Time vs. Size

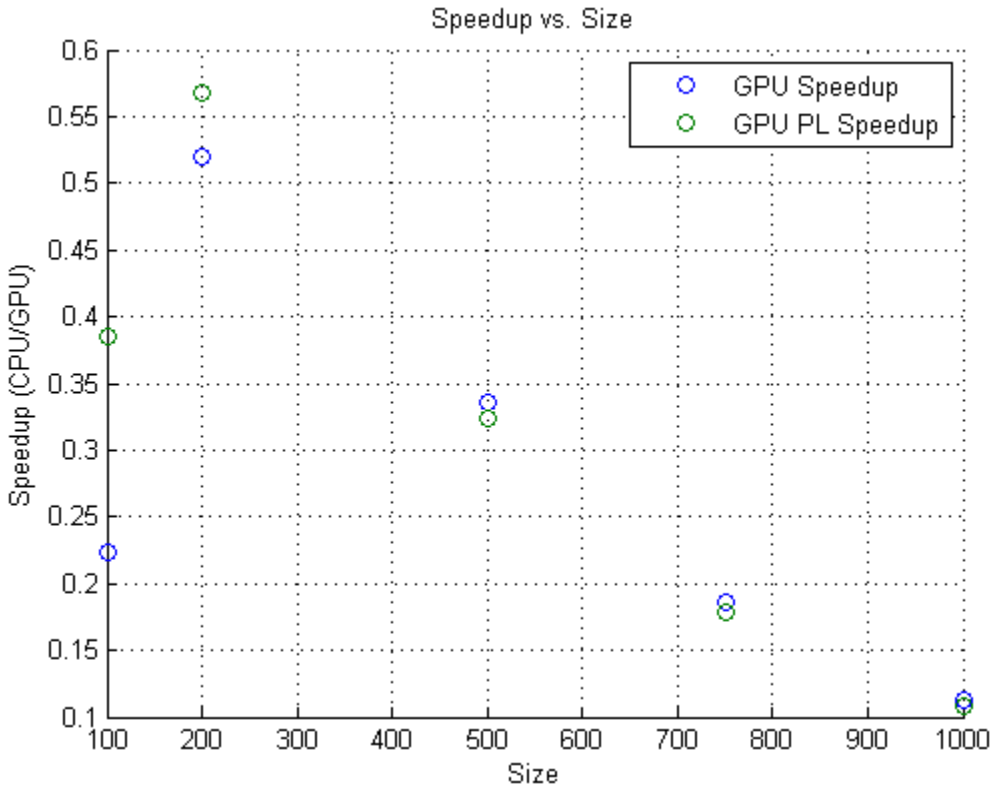

Figure 2: Speedup vs. Size

From Table 1, the Simplex timing results and the corresponding figures of timing and speedup, and Figures 1 and 2, it can be seen that the GPU speedup increases as the problem size increases up to 200 variables, then the performance of the GPU worsens as the problem size is increased further. However, these results cannot be used to form any strong conclusion on the Simplex Algorithm on the GPU in the general sense because only non-computation intensive matrices were used for the test cases. In addition, for simple and smaller sized matrices, it can be seen that zero copy page-locked memory provides a slightly higher performance than the standard GPU implementation (Figure 2). As the problem size increases, the relative performance of zero copy implementation decreases compared to standard GPU implementation. Observing the execution time graph (Figure 1), it can be seen that GPU took longer to solve the same problem in both variants. It was speculated that this phenomenon occurred due to high communication to computation ratio. The size of the matrices created in the lab adversely affected the speed up and it was likely because as the size of the matrix got bigger, the laboratory born matrices were not computationally intensive and therefore, the time for the large memory transfers outweighed the benefits of parallel computation.

Hurdles:

In completing this exercise, several hurdles were found that needed to be overcome. The first was the creation of the linear optimization problems. It was hypothesized at first that the matrices could be generated using a random number generator, and the solution would always be found. However, it was determined that randomly generated matrices worked well for small matrix sizes, but larger sizes resulted in a smaller percentage of solvable problems. This was because the larger sizes had a higher probability that the constraints would counteract each other. For example, one constraint could be "x1 > 4", and a second could be "x1 < 4". Obviously, a solution could not be found because x1 could not be larger than 4

and smaller than 4 at the same time.

Because the randomly generated matrices did not give solutions for large problem sizes, additional linear optimization problems were researched. Several websites exist that contain problem sets with solutions so that linear optimization problem solving programs, such as this one, can be tested. However, another problem existed where the problems available were not given in standard form. Standard form means that the constraints are given such that the variables are non-negative. Non-standard form is where the variables are given such that the variables are not related to 0, but some other number. An example of non-standard form would be where "x1 > 4, such that x1 < 12". In this example, the constraint is "x1 > 4", however there is an additional constraint where x1 must also be less than 12. Of course, this example is simplified and could be expanded to a large number of variables and a large number of constraints. Had the linear optimization problems found been in standard form, they could have been inputted into the parser program and been solved. However, the problems were in non-standard form, and the math required to convert them to standard form was outside the scope of the project.

Therefore, due to the lack of large standard form linear optimization problems, a new problem was constructed. This problem was relatively simple: it contained five variables with four rigid constraints. Additional constraints were added such that they did not modify the solution to the problem. In other words, the additional constraints were less constraining than the other constraints, making the original constraints the restraining factor to the problem. For example, the original constraint was "x1 < 40". The added constraints were in the form "x1 < 40 + p", where p was a positive integer. Thus, the constraint "x1 < 41" was not effective in changing the solution because "x1 < 40" is a tighter constraint. This problem was good in terms of increasing the problem size, but not in increasing the amount of work that must be done per iteration. A script was written in Perl to create the input matrix of any number of extra constraints, and several sizes were tested.

After testing the created linear optimization problem, it became clear that the parallelized version of the Simplex Algorithm did not perform as well as the serialized code. Several tests were run to determine if a different sized matrix would result in better, or worse, speedup. As can be seen in the Results section, the best speedup was obtained around the 200 size, but that is of course with the single created matrix. To try to increase performance of the CUDA code, a few techniques were tried. The first was the use of page-locked host memory. Page-locked host memory is when the host's memory is allocated such that the device (GPU) can read and write from it. This feature was written and verified to work properly, yet gave no better timing results. Next, the code was benchmarked by use of the "ctime" library to see where the code was spending most of its time executing. It turned out that the Row Reduction kernel was taking the longest. To help reduce the time spent in the kernel, the memory calls were modified so that the kernel was not reading and writing from the same array. This attempt proved futile, since performance was not increased with the changes. Given more time and an advanced knowledge of NVIDIA's Nsight would have been helpful in determining more about the program, and how the performance could have been increased.

Future Work:

One future addition to this work would be an enhancement to solve integer problems in non-standard form. There are many integer problems freely available on the Internet, and elsewhere. However, very few problems are given in standard form, meaning the program written in this project is not able to solve the problems easily. To fix this, the program would need to be modified so that non-standard form

problems may be inputted to the program, which would then convert it to standard-form and solve it regularly.

Another addition would be to run the program through a benchmarking tool, such as NVIDIA's Nsight. This tool is able to determine where bottlenecks occur in the code, and gives real time profiling on the kernels currently executing. It even shows a summary showing how much time is spent on each function in the system, showing where the most execution time is being used. For example, if there is a lot of time spent copying memory, Nsight would notify that. By using Nsight to examine the code, the program's performance could be improved immensely.

Conclusion:

In this project, the Simplex Algorithm, which is used to solve linear optimization problems, was implemented both sequentially and in parallel. Both versions of the code correctly solved linear optimization problems that were in standard form. However, the execution time of the parallel implementations did not perform better than the execution time of the sequential implementation. Likely, this is due to the lack of computations in the chosen matrices. If test matrices were used that required many iterations, each with a lot of computations, the GPU implementations would have performed much better. This assignment was completed successfully, yet the performance results of the parallelized code were not as high as expected.

References

1. http://people.sc.fsu.edu/~jburkardt/datasets/mps/mps.html