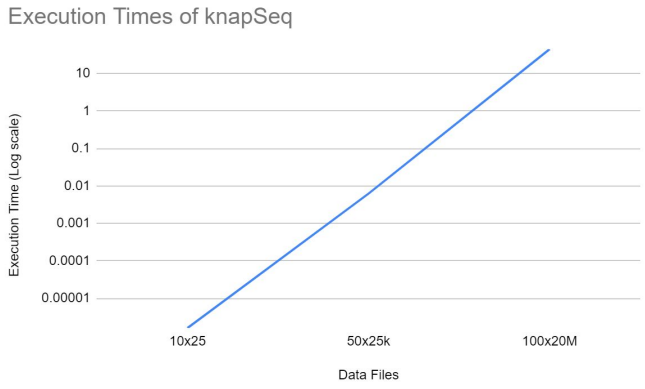
Adam Valdes

12/13/2020

CS475 Term Project Report

For my term project I did a CUDA implementation of the knapsack problem from PA3 of the Fall 2018 section of this course. I thought it would be interesting to see how applying CUDA to this problem would work out and see how much speedup I could achieve using it. I ran my project on the planet machines of the computer science department during development, and I ran my final performance testing on the jupiter machine.

For the performance measurement for this project I used input files containing the relevant data, such as the number of objects, the capacity of the knapsack and the weight and profit of each object. The test files that were provided in the 2018 assignment were k10x25.txt, k50x25k.txt and k100x20M.txt. For each filename such as k10x25.txt, the first number is the number of objects and the second number is the capacity. So for the file k10x25.txt there are 10 objects and the knapsack has a capacity of 25. In my submission the input files are stored in the sample directory, and can be used with a call like this: knapSeq sample/k100x20M.txt. This is a graph of the execution times for running the base knapSeq for the original input files of 10x25, 50x25k and 100x20M.



The algorithm used for this version of the knapsack problem takes an input file with the parameters as mentioned previously. The algorithm then constructs a table, with each entry in the table being the optimal profit for a certain set of objects and a certain capacity. Once the table is constructed the algorithm then backtracks through the table to construct the solution vector, which consists of which objects were and were not chosen for the solution with the optimal profit. For each program in this project, adding a 1 at the end of the arguments for the program will print the solution vector, such as: knapSeq sample/k100x20M.txt 1.

For my optimization approach I began by writing a CUDA kernel, FirstRowKernel, to compute the first row of the table. The reason this kernel is separate from the other is that the first row of the table is calculated differently from the others. In this kernel (Capacity/32)+1 thread blocks are launched, each with 32 threads. Each thread calculates one entry of the first row of the table.

After this I wrote another CUDA kernel, FullTableKernel, which calculates the most recent row of the table (row i) using the previously calculated row, i-1. This kernel is then called in a loop for each row of the table in order to build the table. FullTableKernel is launched with the same configuration of blocks and threads as FirstRowKernel, with each thread calculating one index of the i^{th} row using the data from $i\text{-}1^{th}$ row. At this point when I was testing my code I found that the cudaMalloc call to allocate device memory for the table was giving an out of memory error when using the file k100x20M.txt. After some testing I found that this was because the 100x20,000,000 array was too big to fit in the GPU memory.

Therefore I had to modify the program somehow to be able to process the table in the GPU. As such, I decided that since each row of the table was being calculated based on the

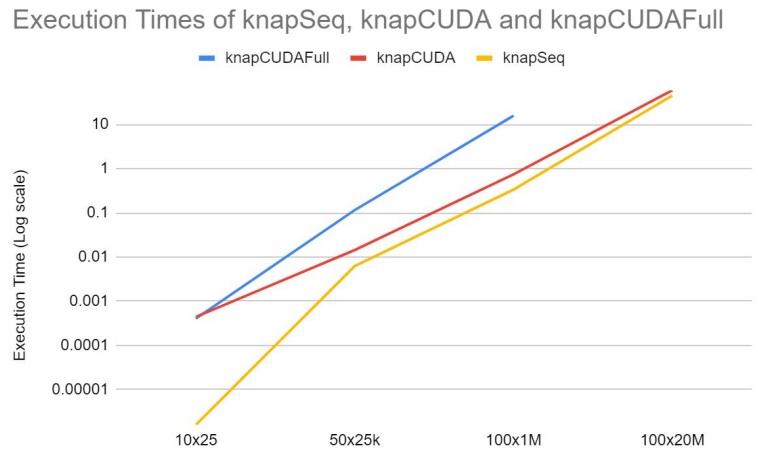
previous row, I only needed to pass 2 rows to the GPU at a time: the row that is currently being calculated, the i^{th} row, and the row that is being used to calculate the i^{th} row, which is the $i\text{-}1^{th}$ row. That way I would only need to allocate a 2x20,000,000 table in the GPU for the 100x20M file instead of a 100x20,000,000 table. As such, I created two different versions of the CUDA version of the program: knapCUDAFull, which calls the FullTableKernel with the entire table, and knapCUDA, which call the FullTableKernel with only the i^{th} and $i\text{-}1^{th}$ rows.

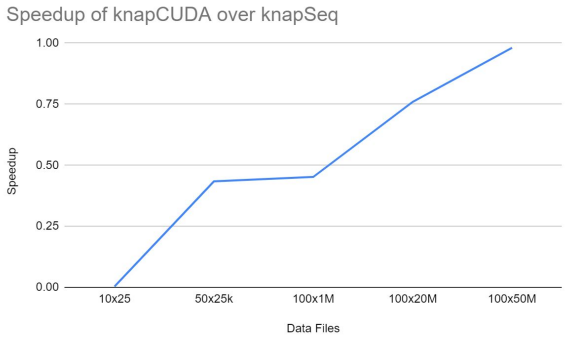
After this I also looked into creating a CUDA kernel for building the final solution vector. The final solution vector is built after the table and is not timed. I thought it would be interesting to parallelize that as well, however I do not think it can be effectively parallelized with CUDA, as it relies on decrementing a global value of c, which I do not think is parallelizable in CUDA. Additionally, computing the solution vector only takes N=(number of objects) iterations, which is an extremely minimal portion of the work that the program performs.

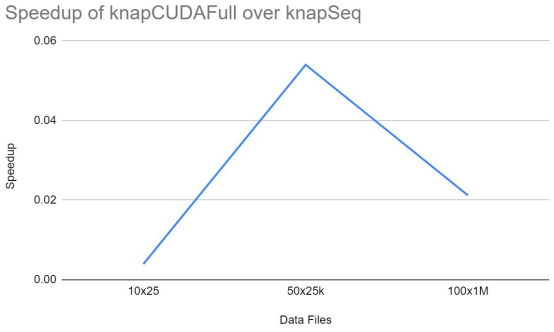
Since FullTableKernel does not work for the 100x20M input file I created my own input file: k100x1M.txt, which has 100 objects and a capacity of 1 million. This file is meant to be used to give data for knapCUDAFull for an input with capacity close to the 100x20M file while also having a capacity low enough that the entire table can still fit into GPU memory. The weights and profits for this file were generated using a program with a bounded random number generator. I also generated another input file: k100x50M.txt. This file is meant to test knapCUDA with a table size that is close to the maximum the GPU would allow.

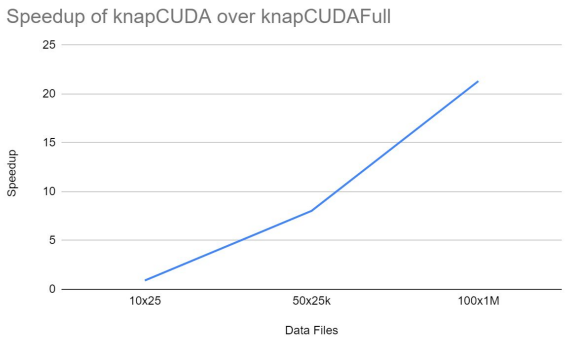
I tested knapSeq and knapCUDA with the 5 input files: k10x25.txt, k50x25k.txt, k100x1M.txt, k100x20M.txt and k100x50M.txt. I tested knapCUDAFull with the 3 input files:

k10x25.txt, k50x25k.txt and k100x1M.txt. These are the results for the execution times and speedups:









Overall I observed very unsatisfactory performance for the CUDA programs. Although knapCUDA performed similarly to knapCUDAFull for the small data file and much better than

knapCUDAFull for the medium-size and large-size data files, it still performed worse than knapSeq for every data file. Even though I observed increased speedup for knapCUDA over knapSeq as the size of the data files grew, it still did not even reach a speedup of 1.0 even for the largest data file.

In conclusion, the speedup I achieved with CUDA for this version of the knapsack problem was very unsatisfactory. Although the performance of knapCUDA relative to knapSeq increased as the size of the data file increased, knapSeq still performed better for every data file size. Even for the 100x50M data file I created the speedup was 0.98, meaning that knapSeq was still faster than knapCUDA. Since this data file was meant to test a table size close to the maximum of what the GPU would allocate and even this file did not yield good speedup, I think it is conclusive that parallelizing this problem with CUDA in this way will not achieve good performance.