Algorithm Description:
Using parallelism to do a matrix multiplication scan of a list of matrices. We are going to achieve this by programming in CUDA and using an Nvidia graphics card.

Description of parallel approach:
We are going to split up the tasks into three phases. But before we do that we are going to determine a block size for dividing our initial array of matrices, and determine how many threads we want to use per block to find which yields the most efficiency. The first phase is going to be taking each block and finding the scan of that subsection of matrices. We are then going to use that output from phase 1 and calculate the scan of it's whole array, so that it outputs a smaller array. For phase 3 we are again going to find the scan of each block of our initial array but this time using the output from phase 2 as a starting point for the scan, which will yield our final result.

We are going to complete these phases by using single pointer arrays for all of our variables so that CUDA can more quickly copy them to shared memory. We are also going to be using three temprary variables to do our matrix multiplication and read in values from our input and write them to our output. This will ensure we don't overwrite any data when we are doing our calculations.

Machine description:
- **Name**: earth
- **Number of cores**: 8
- **Cache sizes**:
  - L1d cache: 32K
  - L1i cache: 32K
  - L2 cache: 256K
  - L3 cache: 15360K
- **Graphics card**: Titan V

Experimental results:
The best results I got for N = 500,000, B = 32, was 1.531 seconds using G = 625, S = 16 with sequential being 11.736 seconds. The best results I got for N = 100,000, B = 64, was 0.766 seconds using G = 625, S = 32 with sequential being 20.715 seconds. As G grows larger the parallel program generally gets faster, but only if it doesn't get too large. As S increases execution time decreases pretty significantly, unless S = B then execution time jumps by almost double.

Conclusion:
I found that always using the maximum value for S seems to return the best results, but only if S =/= B. When S = B the execution time seems to double so using S = B / 2 seems to be the best value. As long as G is not super small or big I don't see a significant decrease in time for G values. There is some small fluctuation though and I found 625 to be a good number, but using, say, 10,000 for N = 100,000 only increases total time by ~10%.