EECS587 Assignment 4: Parallel Computing with Cuda

Tianhao Gu
Xueqing Su

Technique Analysis:

(1) Matrix and Memory Initialization:
For original matrix A, I flatten it to a 1-D array (which makes calculation of indices in later parts easier) with length = n * n, and initial values filled. Then allocate memory to d_A which is copies from host to device, as well as d_B for converting values with d_A back and forth.
Regarding dimensions of the kernel, I choose block dimension to be (1024, 1, 1) which is the maximum limit of threads within a block. And for grid dimension, I let it to be (n * n / 1024 + 1, 1, 1) to allocate enough blocks for updating values of A.

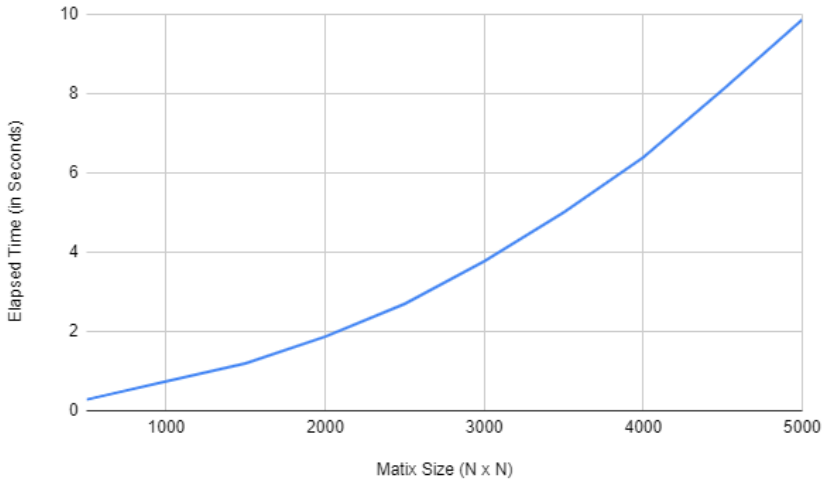
(2) Matrix Update:
Within the 10 iterations, for odd number of turns, update values from d_A to d_B, and vice versa for even number of turns. 2-nd smallest values are calculated by in-place comparison. The reason for exchanging values in turn is to avoid copying values each time. Indices are calculated according to that of 1D grid of 1D blocks.

(3) Calculating sum and specific value:
Initializing another memory for summation. The core idea is to reduce blocks in each iteration. That is for the first iteration, sum each two values and report to indices of even numbers. For the second iteration, sum each four values (sum each two values from indices of even numbers) and report to indices having idx % 4 == 0, so on and so forth. Finally, we have the first index of d_C to have the sum of the whole matrix (flatter to an array). Finally, initialize another memory d_F with size of two doubles. Then assign the sum to d_F[0], and A[37, 47] to d_F[1], and then copy memory from device to host. Therefore, we can print the corresponding values.

Verification and Performance Analysis:

The verifications and elapsed time for different sizes of matrix A show in the table below.

N x N Matrix Size	500	1000	2000
Sum	1.21E+08	4.92E+08	1.98E+09
A[37, 47]	541.703	541.703	541.703
Elapsed Time (Sec.)	0.279328	0.741472	1.87043



Run the same program for more N values to generate the plot above showing how the elapsed time change for matrix size increases. It is easy to note that as N increases the speed gets faster. The reason could be that when increasing N, more blocks will have to be spawned and since each thread still runs in parallel, the block interaction gets smaller.

Since there is one thread per element of the matrix and each thread performs a constant amount of work no matter what the value of N is, the complexity of the stencil kernel should be O(1). According to the way how we split the matrix to get the sum, the sum kernel should be O(logN). This means that the overall complexity of the algorithm should be O(logN).

Output via Great Lakes:

```
Results using 500*500 matrix are:

Sum of the matrix A is: 1.2147e+08
Value of the matrix at A(37, 47): 541.703
Elapsed time is: 0.279328

Results using 1000*1000 matrix are:

Sum of the matrix A is: 4.91725e+08
Value of the matrix at A(37, 47): 541.703
Elapsed time is: 0.741472

Results using 2000*2000 matrix are:

Sum of the matrix A is: 1.97807e+09
Value of the matrix at A(37, 47): 541.703
Elapsed time is: 1.87043
```