IIT DELHI

COL380: Introduction to Parallel andDistributed Programming

Assignment-4 Report

Prof. Subodh Kumar

ARYAN SANTOSH SAHOO

2019CS10335

ALGORITHM:

I implemented block wise matrix multiplication the following way.
First, I read both the input files and loaded them into two arrays. These arrays hold the nonzero blocks and their elements for both the matrices were of the size ‘k1*m*m’ and ‘k2*m*m’.
Additionally, I also loaded two more arrays of size ‘n/m*n/m’ which contained the address corresponding to the block if it is present in those nonzero blocks and -1 otherwise.
Now I am sending these four arrays to the gpu to compute the matrix multiplication. I am defining the block size to be ‘m*m’ in this case ‘m’ can be 4 or 8 and grid size to be ‘n/m*n/m’ so I can cover the entire input matrix with the gpu thread matrix. I am also allocating a shared memory per block of 1kB which is more than enough for our computations.

For my code each block, and their thread grid are responsible for each block in the output matrix multiplication computation. Therefore, I am running a loop over the total number of blocks in a row i.e., ‘n/m’ and for each iteration in that loop the threads for that block multiply the corresponding blocks from input matrices and store it in their local variables after the loop is complete each thread now has the correct value corresponding to each cell for the output matrix. For the multiplication part I had previously initialized outside the loop two shared memory 2D matrices of size ‘m*m’ to hold the temporary matrices during multiplication. Inside the loop all the threads first load these two shared matrices and then after they have been synced, they all proceed to calculate the block multiplication by accessing these shared matrices. All these block matrix multiplications occur after fetching their address from the input matrices and their address matrices so if a block is all zeroes in either of the input matrices it won’t be computed by the threads.

After all the threads have calculated the final output value pertaining to that cell in the output matrix, I am checking its value and if its nonzero adding to a flag which is private for each block. If the flag value is zero after all the threads in the block have computed their final output value and exited that means that all the cells in that output block are zero and hence this information is stored in another array so that we can use it while writing out the nonzero blocks in the output file.

ANALYSIS:

1.Profiling

Given below is the profiling for **n** = 20000, **m** = 8, **k1** = 625000, **k2** = 1250000

```
-bash-4.2$ python gen-input.py -n 20000 -m 8 -z 625000 -o inputFile1.bin
Namespace(m=8, n=20000, non_zero=625000, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 20000 -m 8 -z 1250000 -o inputFile2.bin
Namespace(m=8, n=20000, non_zero=1250000, output='inputFile2.bin')
-bash-4.2$ nvprof --unified-memory-profiling off ./exec inputFile1.bin inputFile2.bin outFile.bin
==1324== NVPROF is profiling process 1324, command: ./exec inputFile1.bin inputFile2.bin outFile.bin
==1324== Warning: Profiling results might be incorrect with current version of nvcc compiler used to compile cuda app. Compile with nvcc compiler 9.0
 later version to get correct profiling results. Ignore this warning if code is already compiled with the recommended nvcc version
Time taken for gpu: 39832ms
==1324== Profiling application: ./exec inputFile1.bin inputFile2.bin outFile.bin
==1324== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   95.45%  39.8318s         1  39.8318s  39.8318s  39.8318s  blockwise_matrix_multiply(int*, int*, int*, int*, int*, int, int, int*)
                    3.63%  1.51372s         2  756.86ms  21.686ms  1.49203s  [CUDA memcpy DtoH]
                    0.92%  383.35ms         6  63.892ms  2.9390ms  165.80ms  [CUDA memcpy HtoD]
      API calls:   94.27%  39.8319s         1  39.8319s  39.8319s  39.8319s  cudaDeviceSynchronize
                    4.50%  1.90288s         8  237.86ms  3.1850ms  1.49369s  cudaMemcpy
                    0.73%  308.03ms         6  51.338ms  156.51us  304.08ms  cudaMalloc
                    0.49%  206.83ms         6  34.471ms  627.71us  152.67ms  cudaFree
                    0.00%  667.60us         1  667.60us  667.60us  667.60us  cuDeviceTotalMem
                    0.00%  341.00us         1  341.00us  341.00us  341.00us  cudaLaunchKernel
                    0.00%  181.83us       101  1.8000us     170ns  79.986us  cuDeviceGetAttribute
                    0.00%  23.484us         1  23.484us  23.484us  23.484us  cuDeviceGetName
                    0.00%  17.257us         1  17.257us  17.257us  17.257us  cuDeviceGetPCIBusId
                    0.00%  3.1670us         2  1.5830us     279ns  2.8880us  cuDeviceGet
                    0.00%  2.3160us         2  1.1580us     509ns  1.8070us  cuDeviceGetCount
                    0.00%     361ns         1     361ns     361ns     361ns  cuDeviceGetUuid
-bash-4.2$
```

Time taken for gpu = 39832 ms

We can now break this down into components based on the amount of time taken for execution.

Breakdown:
Matrix multiplication: 39.8s
CudaMemcpy: 1.5s (device to host) and 0.4s (host to device) a total of 1.9s
CudaMalloc: 0.3s
CudaFree: 0.2s

As we can see most of the computation is going on in the kernel which is around 95% for large matrices. But when dealing with smaller matrices the kernel computation time is negligible and the efficiency instead depends on faster memory allocation and memory copy. We can use streams and asynchronous execution to achieve that.

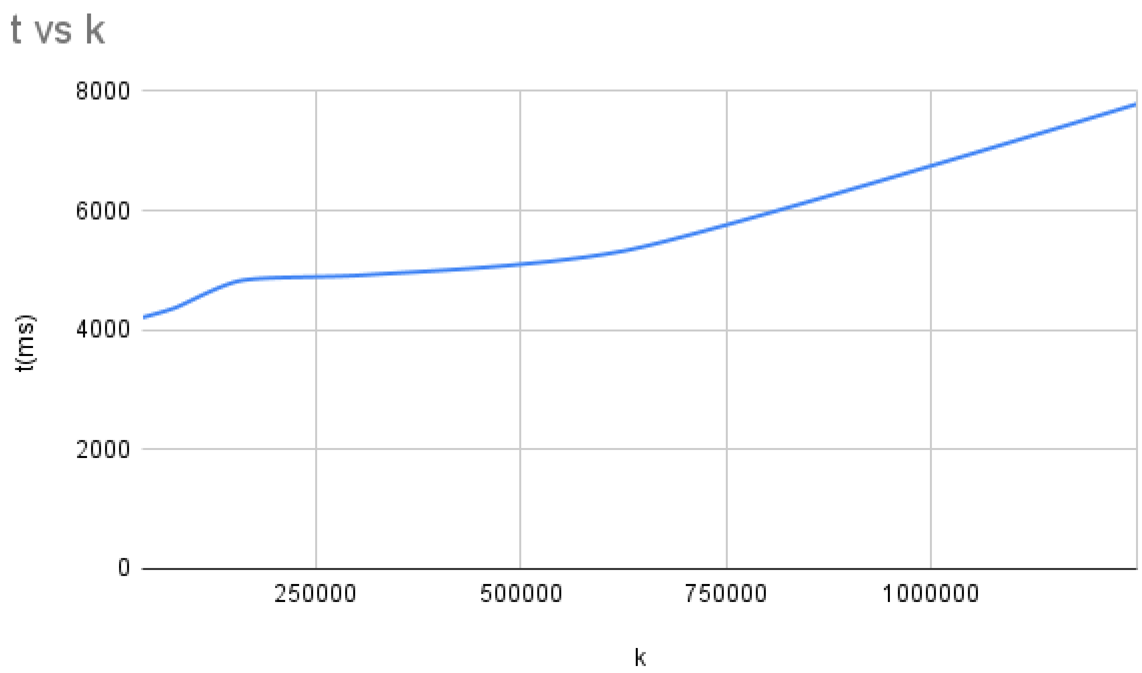
Now we will make a table and vary the values of k1, k2 and n to see the relation in the runtimes.

2.Plotting

Varying k

n = 10000, m = 8, k1 = 625000, k2 = 1250000	time = 7786ms
n = 10000, m = 8, k1 = 312500, k2 = 625000	time = 5323ms
n = 10000, m = 8, k1 = 156250, k2 = 312500	time = 4921ms
n = 10000, m = 8, k1 = 78125, k2 = 156250	time = 4817ms
n = 10000, m = 8, k1 = 39063, k2 = 78125	time = 4363ms
n = 10000, m = 8, k1 = 19531, k2 = 39063	time = 4207ms

Plotting k_{max} = max (k1, k2) with runtime of gpu:



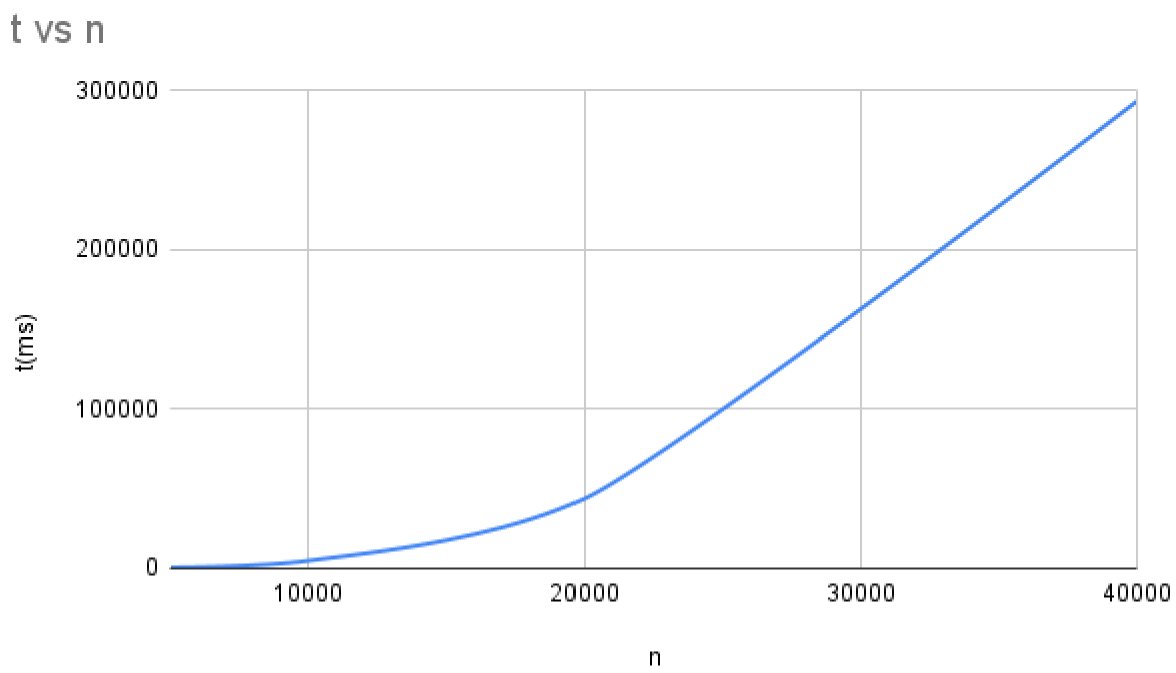
Using curve fitting we can find a, b, c, d values:

$$t = ak_{max}^3 + bk_{max}^2 + ck_{max} + d;$$

since it has to be a polynomial of max degree 3.

Varying n

n = 40000, k1 = 78125, k2 = 156250 time = 293429ms
n = 20000, k1 = 78125, k2 = 156250 time = 43962ms
n = 10000, k1 = 78125, k2 = 156250 time = 4817ms
n = 5000, k1 = 78125, k2 = 156250 time = 598ms



Similarly here also we can use curve fitting to find a’, b’, c’, d’ values:

$$t = a' n_{max}^3 + b' n_{max}^2 + c' n_{max} + d';$$

since it has to be a polynomial of max degree 3.

3. Bandwidth and Throughput

For calculating effective bandwidth, I am using the following formulae:

$$BW_{Effective} = (R_B + W_B) / (t * 10^9)$$

Using the above formula, we put the values we obtained from our nvidia profiling and get the bandwidth to be around 1TB/s.

Total time for data transfer = 1.5137 (HtoD) + 0.38335 (DtoH) = 1.8970s
Total bytes transferred = k1*m*m*4 + k2*m*m*4 + n*n*4 + 3*n/m*n/m*4 (RB) + n*n*4 + n/m*n/m*4 (WB)

Total Bytes = 16*10^7 + 32*10^7 + 8*10^8 + 7.5*10^7 + 8*10^8 + 2.5*10^7
= 2.18*10^9

BW = 2.18*10^9/1.8970*10^9 = **1.15 GB/s**

For calculating throughput, we can use the following formula:

$$GFLOP/s_{\ Effective} = 2N / (t * 10^9)$$

N is the total number of multiply-add operations taking inside the kernel

Thus throughput (T_P) is:

N = k1*k2*m*m*m
t = 39.8318s

T_p = 2*625000*1250000*8*8*8/39.8318*10^9

T_p = **20,000** approx

Screenshots:

t vs k:

```
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 625000 -o inputFile1.bin
Namespace(m=8, n=10000, non_zero=625000, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 1250000 -o inputFile2.bin
Namespace(m=8, n=10000, non_zero=1250000, output='inputFile2.bin')
-bash-4.2$ ./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 7786ms
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 312500 -o inputFile1.bin
Namespace(m=8, n=10000, non_zero=312500, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 625000 -o inputFile2.bin
Namespace(m=8, n=10000, non_zero=625000, output='inputFile2.bin')
-bash-4.2$ ./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 5323ms
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 156250 -o inputFile1.bin
Namespace(m=8, n=10000, non_zero=156250, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 312500 -o inputFile2.bin
Namespace(m=8, n=10000, non_zero=312500, output='inputFile2.bin')
-bash-4.2$ ./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 4921ms
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 78125 -o inputFile1.bin
Namespace(m=8, n=10000, non_zero=78125, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 156250 -o inputFile2.bin
Namespace(m=8, n=10000, non_zero=156250, output='inputFile2.bin')
-bash-4.2$ make run
./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 4817ms
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 39063 -o inputFile1.bin
Namespace(m=8, n=10000, non_zero=39063, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 78125 -o inputFile2.bin
Namespace(m=8, n=10000, non_zero=78125, output='inputFile2.bin')
-bash-4.2$ make run
./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 4363ms
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 19531 -o inputFile1.bin
Namespace(m=8, n=10000, non_zero=19531, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 39063 -o inputFile2.bin
Namespace(m=8, n=10000, non_zero=39063, output='inputFile2.bin')
-bash-4.2$ make run
./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 4207ms
```

t vs n:

```
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 78125 -o inputFile1.bin
Namespace(m=8, n=10000, non_zero=78125, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 10000 -m 8 -z 156250 -o inputFile2.bin
Namespace(m=8, n=10000, non_zero=156250, output='inputFile2.bin')
-bash-4.2$ ./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 4974ms
-bash-4.2$ python gen-input.py -n 5000 -m 8 -z 78125 -o inputFile1.bin
Namespace(m=8, n=5000, non_zero=78125, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 5000 -m 8 -z 156250 -o inputFile2.bin
Namespace(m=8, n=5000, non_zero=156250, output='inputFile2.bin')
-bash-4.2$ ./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 598ms
-bash-4.2$ python gen-input.py -n 20000 -m 8 -z 78125 -o inputFile1.bin
Namespace(m=8, n=20000, non_zero=78125, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 20000 -m 8 -z 156250 -o inputFile2.bin
Namespace(m=8, n=20000, non_zero=156250, output='inputFile2.bin')
-bash-4.2$ ./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 43962ms
-bash-4.2$
-bash-4.2$ python gen-input.py -n 40000 -m 8 -z 78125 -o inputFile1.bin
Namespace(m=8, n=40000, non_zero=78125, output='inputFile1.bin')
-bash-4.2$ python gen-input.py -n 40000 -m 8 -z 156250 -o inputFile2.bin
Namespace(m=8, n=40000, non_zero=156250, output='inputFile2.bin')
-bash-4.2$ make run
./exec inputFile1.bin inputFile2.bin outFile.bin
Time taken for gpu: 293429ms
```