CECS 625 Parallel Programming Homework Assignment #2

September 10, 2019 (100 points)

Due: September 23 (Monday) midnight

(Submit your project report and VS 2015 projects to the Blackboard. You may be asked to present your project in class.)

Assignment Description

Use VS 2015 (VS 2017 does not work for CUDA 9.2 for now) to do your project and do the performance timing tests on your computer or the Alienware Aurora R7 computers in DC 119.

1(30 points) Download the timerDemos VS 2015 project to your computer. This project  
 demonstrates how to use three different timers to measure and compare performance of   
 sequential and C++multithreading dot product.

(a) List the specification of the CPU in your computer used for this project and calculate  
 the peak Gflops of the CPU.

My computer is MSI GS-75 Stealth with an RTX 2070 GPU.

As far as the CPU goes here are the specs:

Type: Intel Core i7-8750H Processor

Cores: 6

Threads: 12

Processor Base Frequency: 2.20 GHz

Max Turbo Frequency: 4.10 GHz

Making the peak Gflops at: (Peak Frequency)\*(# of Cores)\*(Number of Floating Point Operations in a Cycle) = (4.10)\*(6)\*(16) = 393.6 GFlop/s

I was not running it on the turbo setting so during my analysis on this project my GFlops were at (2.20)\*(6)(16) = 211.2 GFlop/s

(b) Run the project application for vectors size = 14000000, 16000000 and 17000000, and   
 for number of therads = 4, 6, 12, and 16, respevtively. Collect the results and   
 assemble them into a table like the following (or design your own table as you wish):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Crono Timer - Dot Result | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 |
| 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 |
| 1.70E+07 | 1.68E+07 | 1.70E+07 | 1.70E+07 | 1.70E+07 | 1.70E+07 |
|  |  |  |  |  |  |
| Windows Timer - Dot Result | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 |
| 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 |
| 1.70E+07 | 1.68E+07 | 1.70E+07 | 1.70E+07 | 1.70E+07 | 1.70E+07 |
|  |  |  |  |  |  |
| chTimer Timer - Dot Result | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 | 1.40E+07 |
| 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 | 1.60E+07 |
| 1.70E+07 | 1.68E+07 | 1.70E+07 | 1.70E+07 | 1.70E+07 | 1.70E+07 |
|  |  |  |  |  |  |
| Crono Timer - Runtime | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 0.016772 | 0.009454 | 0.007155 | 0.008799 | 0.010913 |
| 1.60E+07 | 0.019188 | 0.00971 | 0.008548 | 0.009276 | 0.010946 |
| 1.70E+07 | 0.020424 | 0.010348 | 0.008192 | 0.010222 | 0.011402 |
|  |  |  |  |  |  |
| Windows Timer - Runtime | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 0.0167844 | 0.0069447 | 0.006808 | 0.0088952 | 0.0112485 |
| 1.60E+07 | 0.019153 | 0.0079291 | 0.007593 | 0.0098043 | 0.0112323 |
| 1.70E+07 | 0.0203481 | 0.0078977 | 0.007791 | 0.0098113 | 0.0117244 |
|  |  |  |  |  |  |
| chTimer Timer - Runtime | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 0.016721 | 0.0068859 | 0.006933 | 0.009043 | 0.0105945 |
| 1.60E+07 | 0.0191425 | 0.007713 | 0.007561 | 0.0103274 | 0.0108664 |
| 1.70E+07 | 0.0211787 | 0.008352 | 0.007979 | 0.0104274 | 0.0117874 |
|  |  |  |  |  |  |
| Crono Timer - Speedup | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 1 | 1.77406 | 2.34256 | 1.89737 | 1.53844 |
| 1.60E+07 | 1 | 1.97611 | 2.25351 | 2.07072 | 1.74785 |
| 1.70E+07 | 1 | 1.97371 | 2.47168 | 1.98112 | 1.78179 |
|  |  |  |  |  |  |
| Windows Timer - Speedup | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 1 | 2.41686 | 2.4584 | 1.89341 | 1.48374 |
| 1.60E+07 | 1 | 2.41553 | 2.52645 | 1.94429 | 1.71064 |
| 1.70E+07 | 1 | 2.57646 | 2.63313 | 2.07027 | 1.78522 |
|  |  |  |  |  |  |
| chTimer Timer - Speedup | 1 | 4 | 6 | 12 | 16 |
| 1.40E+07 | 1 | 2.4283 | 2.41334 | 1.86 | 1.57408 |
| 1.60E+07 | 1 | 2.48185 | 2.54546 | 1.85383 | 1.76014 |
| 1.70E+07 | 1 | 2.53576 | 2.55621 | 1.94552 | 1.72067 |

Discuuss the results.

For the timer analysis done here we ran dot product operations of various vector sizes for various threads. Analysis included Runtime, Speedup and the Dot Product Result. As you can see in the Runtime and Speedup analysis the optimal performance metrics were under the parallelism of 6 threads. As the threads increased to 12 and 16 accordingly, performance weakened. A graph is shown below for visual analysis on peak performance.

Another thing to note is the difference in the output from the timers. They are all relatively the same but none produced the same results due to them being based on a different system.

(c) For the vectors size, 1.7e7 floats, the sequential computing produced a less accuate   
 result whereas all the multithreadings produced exact results. Exaplin why (hint: study  
 the SequentialDot2 function in the source file, Dot.cpp.)

The dot product operation is memory bound and floating-point computation is not exact. For increasingly large amounts of vector sizes the results of the dot product can prove to be inaccurate if done sequentially. When the dot product is calculated in parallel like SequentialDot2 does in the Dot.cpp file it performs better. This is all due to the dot product being memory bound meaning it it relies on accessing cache to track the operation. If cache misses occur when done sequentially the result will be less accurate but if we can make it to where we optimize cache access by chunking the process it will produce a more accurate result due to less cache misses.

2(40 points) Download the MatrixMultiTing VS2015 project to your computer. Study the following C++ source files and functions contained in them:  
  
MatrixMuli.cpp – C CPU implementation of matrx multiplication including  
void matrixMul1(float\* C, float\* A, float\* B, int RA, int CA, int CB);

void matrixMul2(float\* C, float\* A, float\* B, int RA, int CA, int CB);

void matrixMul3(float\* C, float\* A, float\* B, int RA, int CA, int CB);

void matrixMul4(float\* C, float\* A, float\* B, int RA, int CA, int CB);

sgemm.cpp – a slightly modified function of the CLAPACK library’s sgemm function  
void sgemm(char transa, char transb, int m, int n, int k,

float alpha, float a[], int lda, float b[], int ldb, float beta,

float c[], int ldc);

(a) Perform timing tests of matrix multiplication for matrix sizes of 320, 640, 960, 1600,   
 and 3200 using the CPU functions matrixMul1, matrixMul2, and matrixMul4 as   
 given in the project. Assemble your test results into an easy-to-read table. Discuss   
 your results and explain why the matrixMul4 gives the best timing results.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Size | 320 | 640 | 960 | 1600 | 3200 |
|  |  |  |  |  |  |
| A |  |  |  |  |  |
| Runtime (Allocation) | 0.034556 | 0.114033 | 0.255287 | 0.709308 | 2.84397 |
| Runtime (MatrixMul1) | 0.030795 | 0.262131 | 0.842422 | 8.71123 | 121.026 |
| C[2] (MatrixMul1) | 85.9865 | 162.8 | 242.359 | 418.266 | 794.871 |
| Runtime (MatrixMul2) | 0.034004 | 0.279742 | 0.929492 | 8.70957 | 121.597 |
| C[2] (MatrixMul2) | 85.9865 | 162.8 | 242.359 | 418.266 | 794.871 |
| Runtime (MatrixMul4) | 0.03526 | 0.281038 | 0.947798 | 4.38852 | 35.1073 |
| C[2] (MatrixMul4) | 90.1749 | 163.768 | 238.618 | 415.479 | 805.498 |

As you can see the actual allocation of the matricies plays a big role in runtime of matrix multiplication. Performance between Mul1 and Mul2 are roughly the same for all matrix sizes but when the size goes to 1600 Mul4 performs a lot bettwe.

MatrixMul4 gives the best results when the number of matricies is large. It performs better under these conditions because it transposes the B matrix initially. This allows for less memory access when running through the multiplication.

(b) Perform timing tests similar to (a) to compare the performance of matrixMul3 and   
 sgemm functions and write a function to verify these two functions produce the same   
 result within a specified tolerance value. Discuss your results.

SGEMM performs much better than Mul3 and the ratio of the difference in performance increases as the number of matricies become large. Meaning it does moderately better for small matrix sizes but does substantially better when the matrix sizes become very large. You can see thedifference in runtimes from the size of 320 vsthe size of 3200.

When verifying that they produce the same amount I found that due to floating point operations not being exact, the operations might not give back the same result. I tested the tolerance levels a lot and found that the two methods did produce the same result for all tests I ran. You can see the results in the green comments below the source.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Size | 320 | 640 | 960 | 1600 | 3200 |
| B |  |  |  |  |  |
| Runtime (Allocation) | 0.034881 | 0.1141 | 0.255773 | 0.708347 | 2.83489 |
| Runtime (MatrixMul3) | 0.034804 | 0.279414 | 0.933267 | 8.82632 | 118.876 |
| C[2] (MatrixMul3) | 84.3811 | 160.209 | 240.378 | 405.927 | 809.296 |
| Runtime SGEMM | 0.018365 | 0.146185 | 0.492774 | 2.32173 | 18.6209 |
| Ref\_C[2] SGEMM | 84.3811 | 160.209 | 240.378 | 405.927 | 809.296 |

Verification Method

void Problem3(int min, int max)

{

float result1 = 0.0;

float result2 = 0.0;

for (int i = min; i <= max; i++)

{

min = i;

int RA = min;

int CA = min;

int CB = min;

int RB = CA, RC = RA, CC = CB;

float \*A, \*B, \*C, \*Ref\_C;

printf("single precision %dx%d matrix times %dx%d matrix:\n\n", RA, CA, RB, CB);

INIT\_TIMER;

START\_TIMER;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dist(0, 1);

A = (float \*)malloc(RA \* CA \* sizeof(float)); assert(A);

B = (float \*)malloc(RB \* CB \* sizeof(float)); assert(B);

C = (float \*)malloc(RC \* CC \* sizeof(float)); assert(C);

Ref\_C = (float \*)malloc(RC \* CC \* sizeof(float)); assert(Ref\_C);

for (int i = 0; i < RA \* CA; ++i)

A[i] = (float)(dist(rd));

for (int i = 0; i < RB \* CB; ++i)

B[i] = (float)(dist(rd));

for (int i = 0; i < RC \* CC; ++i)

C[i] = Ref\_C[i] = 0;

STOP\_TIMER("allocating and initilizing matrices using uniform distribution(0,1)");

std::cout << "\n\n";

START\_TIMER

matrixMul3(C, A, B, RA, CA, CB);

STOP\_TIMER("MatrixMul3", "")

std::cout << "C[2] = " << C[2] << "\n\n";

result1 = C[2] + result1;

START\_TIMER;

// Ref\_C := alpha\*op( A )\*op( B ) + beta\*C

sgemm('n', 'n', RA, CB, CA, 1.0, A, CA, B, RB, 1.0, Ref\_C, CB);

STOP\_TIMER("sgemm", "");

std::cout << "Ref\_C[2] = " << Ref\_C[2] << "\n\n";

result2 = Ref\_C[2] + result2;

}

std::cout << "\nResult of all Mul3 operations: " << std::fixed << std::setprecision(25) << result1 << "\n";

std::cout << "\nResult of all sgemm operations: " << std::fixed << std::setprecision(25) << result2 << "\n";

}

Results:

Enter Min Matrix Size:

2500

Enter Max Matrix Size:

2520

Result of all Mul3 operations: 13248.7167968750000000000000000

Result of all sgemm operations: 13248.7167968750000000000000000

// Took about 10 min to compute, I can probably optimize

Enter Min Matrix Size:

600

Enter Max Matrix Size:

700

Result of all Mul3 operations: 16400.9296875000000000000000000

Result of all sgemm operations: 16400.9296875000000000000000000

Enter Min Matrix Size:

1

Enter Max Matrix Size:

1000

Result of all Mul3 operations: 125211.1640625000000000000000000

Result of all sgemm operations: 125211.1640625000000000000000000

// Performed a variety of tests and could not see a tolerance range where matrixmul3 and sgemm functions produced different results.

(c) Explain why sgemm timing results are better than those of matrixMul4. In the future we   
 will try to beat the timing performance of sgemm by using AVX and multithreading.

Sgemm has optimal performance due to conditional optimization prior to computation. The function sets the matrix up to be computed optimally by transposing the give matricies and does checks to see how it should operate on the given operation before computation starts. You can see the checks in the comments like so:

Form C := alpha\*A\*B + beta\*C.

Form C := alpha\*A'\*B + beta\*C

Form C := alpha\*A\*B' + beta\*C

Form C := alpha\*A'\*B' + beta\*C

MatrixMul4 does something similar but only transposes the B matrix, this is still a good optimization proactice due to decreasing the amount of memory access when running the multiplication process.

3(30 points) Do problem 5 on pp. 73-74 of the textbook. In this problem, do your own   
 timing tests using your computer and include your timing results in the report.

The reason for the plain\_max\_unroll\_2 and plain\_max\_unroll\_4 functions improved performance is due to the parallel construct in the for loops. Instead of searching the entire array all at once the plain\_max\_unroll\_2 has two elements searching in the for loop where ones index is set at i and the other set at i+1, both indexes are incremented by 2 in each run. This means that from the start and onward the indexes look like so:

Index(max\_0, max\_1) = (0,1), (2,3), (4,5), (6,7) etc

After the max is found between the split indexes it is compared again between the two in order to see who had the largest value between the two parallel searches.

The same concept is applied in the plain\_max\_unroll\_4 function it just uses 4 elements instead of two. The only difference is at the end when it needs to compare the values of the four max elements. It compares max2 and max3 then the result of that is compares with max1 and the result of that compared with max0. Whichever had the largest will end up being the result.

Running my own results, I utilized the chronoTimer on array sizes atrting at 1000000 and going to 1000000000.

Here is the output:

/\*

Testing Array of size: 1000000

Runtime of plain\_max = 0.0005 seconds

Max for plain\_max: 999

Runtime of plain\_max\_unroll\_2 = 0.000536 seconds

Max for plain\_max\_unroll\_2: 999

Runtime of plain\_max\_unroll\_2 = 0.000562 seconds

Max for plain\_max\_unroll\_2: 999

Testing Array of size: 10000000

Runtime of plain\_max = 0.005294 seconds

Max for plain\_max: 999

Runtime of plain\_max\_unroll\_2 = 0.005232 seconds

Max for plain\_max\_unroll\_2: 999

Runtime of plain\_max\_unroll\_2 = 0.005325 seconds

Max for plain\_max\_unroll\_2: 999

Testing Array of size: 100000000

Runtime of plain\_max = 0.051716 seconds

Max for plain\_max: 999

Runtime of plain\_max\_unroll\_2 = 0.052309 seconds

Max for plain\_max\_unroll\_2: 999

Runtime of plain\_max\_unroll\_2 = 0.053357 seconds

Max for plain\_max\_unroll\_2: 999

Testing Array of size: 1000000000

Runtime of plain\_max = 0.528571 seconds

Max for plain\_max: 999

Runtime of plain\_max\_unroll\_2 = 0.529172 seconds

Max for plain\_max\_unroll\_2: 999

Runtime of plain\_max\_unroll\_2 = 0.538039 seconds

Max for plain\_max\_unroll\_2: 999

\*/