Rutgers University
Parallel and Distributed Computing
Project 3 - Group 3 and 6

Allocations:
(Group 3)
Khalid Akash (Problem 2, part 1 and 2)
Brandon Smith (Problem 5)
Suva Shahria (Problem 3, Part BFS)
Ryan Morey (Problem 1)

(Group 6)
Bhargav Gokalgandhi (Problem 2, Part 3 and 4)
Dan Yates (Problem 4)
Jon Garner (Problem 3, Part Prims)
Krithika (Problem 5)

Problem 1

For this problem, we wrote programs to calculate min, mean, max, and standard deviation for an large array of floats. We wrote a single-threaded version using only the CPU, as well as a version using CUDA. The system we tested this on has 8 Nvidia 1080Ti GPU's, for a total of 28672 CUDA cores.

For all of our results, we show the average time for CPU execution across a number of runs, and we show our best GPU time - we varied the number of blocks and number of threads per block, in order to determine the best configuration. In general, we saw that using the most threads per block available (1024) lead to best performance, with number of blocks being slightly less significant. While these results are accurate, and do represent the real speedup due to parallelization, they may skew slightly in favor of the GPU, it is still a representative comparison - we are showing the GPU time for the optimal configuration for each problem, which is what one would use in a real world setting. Our full results are available with our submission.

Minimum and Maximum

Minimum and maximum are trivial to calculate. For both of these, the CPU version simply iterates through the array, keeping track of the smallest or largest value seen so far. The GPU enabled version is also straightforward, except now we divide the array amongst our CUDA threads, and each thread finds the min or max for its portion of the array.

There was **no discrepancy** between the min and max value calculated by the CPU and GPU versions; as long as every array element is processed, the exact min or max will be found.

Minimum

N	**CPU time (ms)**	**GPU time (ms)**	**Speedup**
50M	199.310	110.396	1.805
100M	385.956	213.388	1.809
150M	578.445	318.636	1.815
200M	787.774	426.897	1.845

Maximum

N	CPU time (ms)	GPU time (ms)	Speedup
50M	117.320	101.689	1.154
100M	228.894	235.121	0.974
150M	341.511	349.735	0.976
200M	457.219	459.307	0.995

Note that $speedup < 1$ indicates a worse performance - we find this result highly unusual: why would finding the max have such different performance from finding the min? We were unable to discover the cause of this.

Mean

For both the CPU and the GPU, the mean here is calculated by summing up the elements of the array, and dividing by the size of the array. In the GPU, the array is split up and this is done in parallel, accumulating the results in the CPU at the end.

We observed very little discrepancy between calculating the mean between the CPU and the GPU - at the very most 0.03%

N	CPU time (ms)	GPU time (ms)	Speedup
50M	236.771	114.589	2.066
100M	468.146	233.481	2.005
150M	702.395	329.584	2.131
200M	946.006	481.708	1.964

We achieved fairly good speedup by parallelizing here.

Standard Deviation

The naive approach to calculating standard deviation is to sum up the squares of the delta's between each element and the mean. This approach, however, is not numerically stable - for floating point arithmetic the results are particularly bad. For this reason, in both the CPU and the GPU version we use Welford's Online algorithm for calculating variance. This allows us to process each element exactly once, and update our estimate for the standard deviation element by element. Additionally, this algorithm parallelizes well, and allows the GPU-enabled version to calculate the standard deviation in parallel for chunks of the GPU, and then combine them in the CPU at the end.

Similar to the mean, we see very little discrepancy between the CPU and GPU in calculating the standard deviation - less than 0.03%

N	CPU time (ms)	GPU time (ms)	Speedup
50M	621.7	122.0	5.09
100M	1226.4	242.4	5.06
150M	1847.3	355.7	5.19
200M	2465.6	488.2	5.05

We achieve great speedup here: ~5x over the CPU

All stats

Here we calculate all statistics concurrently. In both the CPU and GPU, we take only one pass through the data, keeping track of the min, max, standard deviation, and mean, in the same manner as before.

N	CPU time (ms)	GPU time (ms)	Speedup
50M	627.3	116.3	5.39
100M	1234.1	231.2	5.34
150M	1853.9	339.4	5.46
200M	2477.9	455.1	5.44

Here, we see the best speedup results so far. We gain ~5.4x speedups over the CPU. This is likely due to the CUDA-induced overhead being mitigated, by our calculating all results at once.

Conclusion

We believe future work could achieve far greater speedups here. We are likely limited by the accumulation of results in the CPU - which in principle could also occur within the GPU, but at the cost of additional complexity and memory requirements within the GPU.

Problem 2

Part 1 - Naive Matrix Multiplication

```
NVidia - GTX 965M
1024 Cuda Cores
[CUBLAS] Matrix multiplication of 128x128 Matrices for 1000 iterations...
elapsed time 7 microsecs
[CUBLAS] Matrix multiplication of 256x256 Matrices for 1000 iterations...
elapsed time 7 microsecs
[CUBLAS] Matrix multiplication of 512x512 Matrices for 1000 iterations...
elapsed time 378 microsecs
[CUBLAS] Matrix multiplication of 1024x1024 Matrices for 1000 iterations...
elapsed time 540 microsecs
[CUBLAS] Matrix multiplication of 2048x2048 Matrices for 1000 iterations...
elapsed time 1938 microsecs
[CUDA] Matrix multiplication of 128x128 Matrices for 1000 iterations...
elapsed time 89 microsecs
[CUDA] Matrix multiplication of 256x256 Matrices for 1000 iterations...
elapsed time 6 microsecs
[CUDA] Matrix multiplication of 512x512 Matrices for 1000 iterations...
elapsed time 363 microsecs
[CUDA] Matrix multiplication of 1024x1024 Matrices for 1000 iterations...
elapsed time 727 microsecs
[CUDA] Matrix multiplication of 2048x2048 Matrices for 1000 iterations...
elapsed time 2182 microsecs
[NAIVE CPU] Matrix multiplication of 256x256 Matrices for 500 iterations...
elapsed time 19140955 microsecs
[NAIVE CPU] Matrix multiplication of 512x512 Matrices for 500 iterations...
elapsed time 236002370 microsecs
```

The above is some sample data taken on a run of the naive matrix multiplication algorithm. For this algorithm, we use the standard iteration method of surveying every element, and doing a dot product of the row of the left hand matrix, and the column of the right hand matrix. Clearly, this took an enormous amount of time via the CPU. Implementing this in CUDA and CUBLAS posed little challenge, but gave an extremely interesting result. When the matrices reached a square size of 256 x 256 dimensions, there was a significant dip in timing (increased performance!) in both CUBLAS and CUDA implementation. This is due to the number of threads able to be executed at once, and in this particular case, 65K threads would have to have been executed which is the exact number of threads the 965M is capable of. Another thing that was noticed was that CUBLAS is much better at handling smaller sized matrices but matched/failed in performance on bigger matrices. This was found on multiple runs of the program.

Part 2 - Matrix Multiplication via Tiling

```
NVidia - GTX 965M
1024 Cuda Cores
[CUDA_BLOCK] Matrix multiplication of 128x128 Matrices for 1000 iterations...
elapsed time 83 microsecs
[CUDA_BLOCK] Matrix multiplication of 256x256 Matrices for 1000 iterations...
elapsed time 26 microsecs
[CUDA_BLOCK] Matrix multiplication of 512x512 Matrices for 1000 iterations...
elapsed time 158 microsecs
[CUDA_BLOCK] Matrix multiplication of 1024x1024 Matrices for 1000 iterations...
elapsed time 466 microsecs
[CUDA_BLOCK] Matrix multiplication of 2048x2048 Matrices for 1000 iterations...
elapsed time 1920 microsecs
```

The tiling algorithm again experiences a similar trend of having a huge performance increase at 256x256 matrix. In this part, we decided to use a tile size of 32 threads, corresponding to the warp size to load into shared memory. We created two shared memory regions of size 32 that corresponded to the two matrices that were to be multiplied. We then grabbed the local values of both to do the dot product and aggregate them into an intermediate value. After syncing the threads, we then placed the intermediate value into the final matrix product. This lead to a minor, but still a performance increase against the bare CUDA implementation.

Part 3 - Matrix Multiplication with loop unrolling

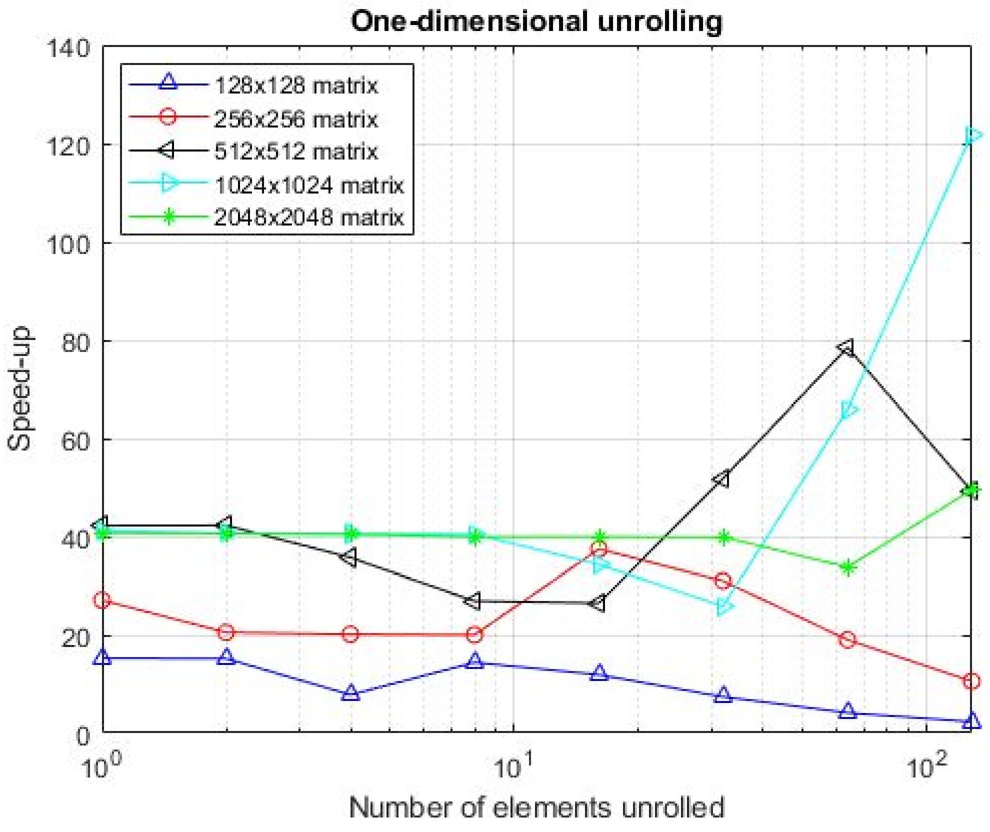
For matrix multiplication with loop unrolling, the outermost loops are unrolled within each thread. This can be performed in two ways, by unrolling only one of the two loops, or by unrolling both loops. In this implementation, both are performed. Firstly, both loops are unrolled. So, each thread in the GPU will perform matrix multiplication for a small ‘nxn’ part of the final matrix. Consider matrix C which is MxP which is the result of matrix multiplication of A which is MxN, and B which is NxP. For factor 2 unroll, each thread performs matrix multiplication of 2x2 part of the matrix C. So, each thread takes 2x2 elements of A and 2x2 elements of B and performs matrix multiplication to give the output of 2x2 part of C. For single loop unroll each thread takes multiple elements from one of the rows of A and one of the columns of B and performs matrix multiplication for multiple elements of one of the rows of C.

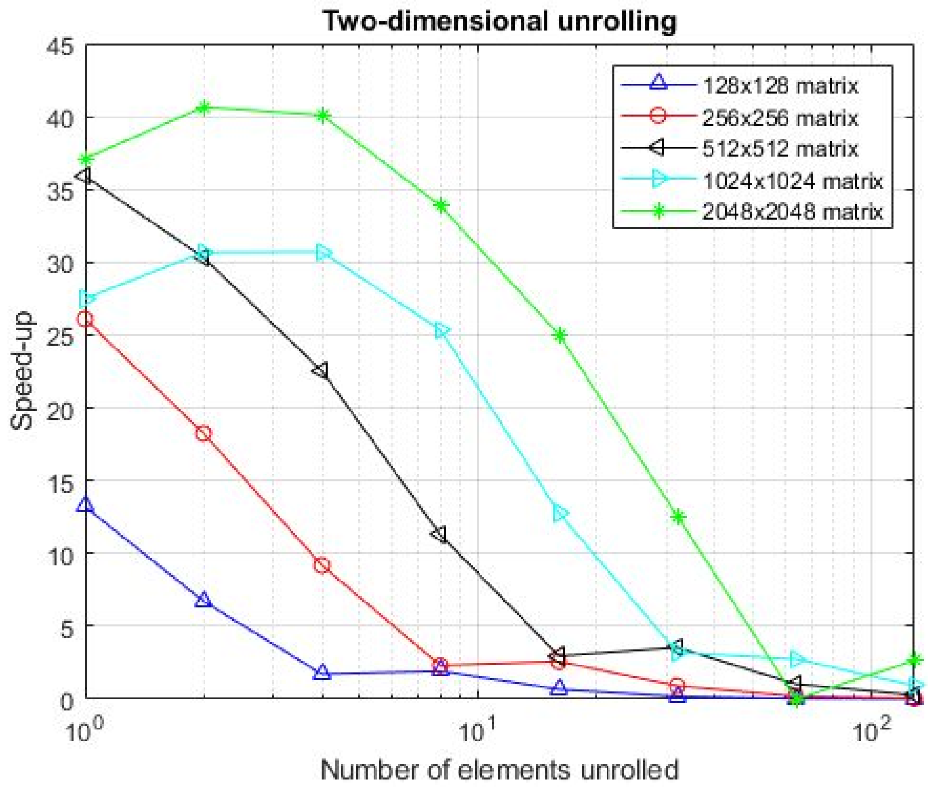
Part 4 - Strassen - Winograd Algorithm

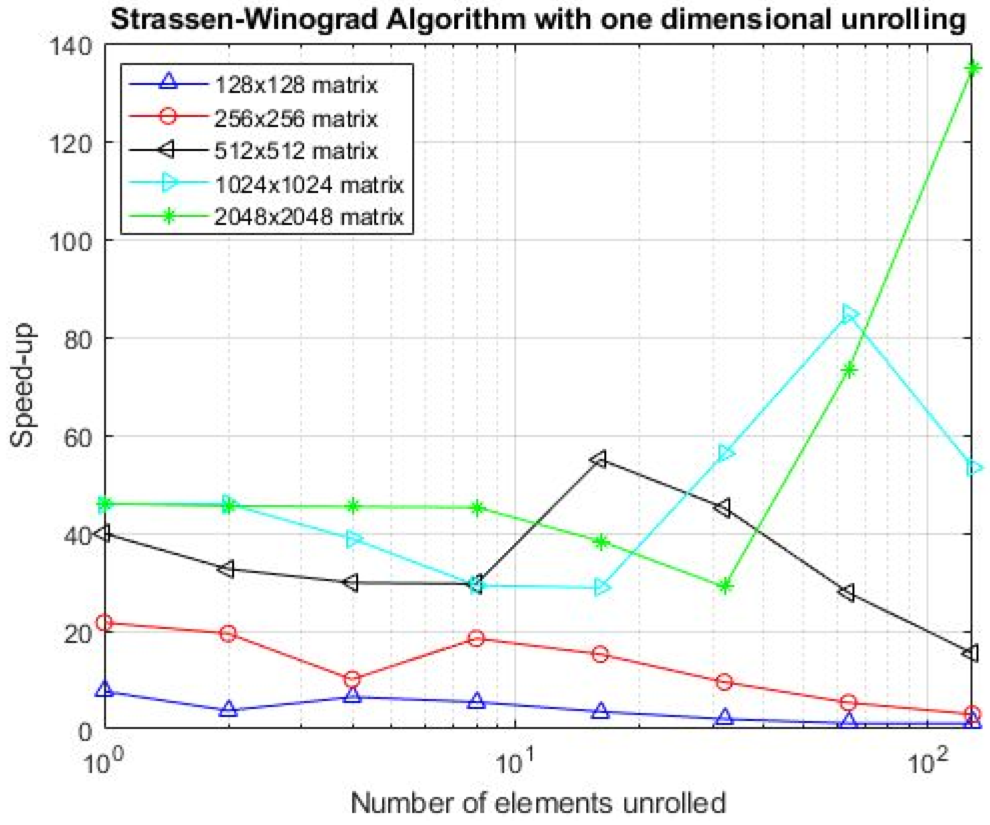
Strassens Algorithm performs the matrix multiplication in O(n^2.81) instead of O(n^3). It does that by replacing one multiplication by a number of additions/subtractions. So, for two square NxN matrices, Strassens algorithm divides the matrices in 4 blocks of N/2 x N/2 each, and then performs 7 multiplications and 18 additions/subtractions. Usually such block based multiplication will require 8 multiplications.

Now, Winograds variant of Strassens algorithm reduces the number of additions/subtractions to 15, which is a small but important part in speeding up the matrix multiplication process. This Winograd variant of Strassens algorithm is performed in the GPU and it is shown that it gives speed-up for large scale matrix multiplication only. Also. this algorithm can only be performed for square matrices in this case. The reference for this implementation is taken from https://www.cise.ufl.edu/~sahni/papers/strassen.pdf.

Speed-up of part 3 and 4 w.r.t. Naive matrix multiplication in the CPU. The speed-up on node21-1, which has a Tesla k40m GPU with CUDA 3.5 compute capability, is shown. The speed-up values for node21-2 are also present, which are stored in the ‘part 3 and 4’ folder of the problem_2. To create the graphs below, we execute the application for multiple iterations and we increase the size of the matrices from 128x128 doubling till 2048x2048. The number of elements to be unrolled are doubled from 1 to 128. The speed-up is saved for all the implementations.




Two-dimensional unrolling
Speed-up
Number of elements unrolled
128x128 matrix
256x256 matrix
512x512 matrix
1024x1024 matrix
2048x2048 matrix


Strassen-Winograd Algorithm with one dimensional unrolling
Speed-up
Number of elements unrolled
128x128 matrix
256x256 matrix
512x512 matrix
1024x1024 matrix
2048x2048 matrix

It can be seen that as the number of elements that are unrolled increase, the performance is better with one-dimensional unrolling than two-dimensional unrolling. Now, the matrices are stored in row major format. So, each thread in the two-dimensional unrolling part has to access non-contiguous memory location i.e. non-contiguous access for both input matrices as well as output matrix. This means that memory coalescing cannot occur and that increases the memory access latency. This access is much less in single dimensional loop unroll where only the outer loop is unrolled. So, the non-contiguous memory access is only for one of the two input matrices. Also, it can be seen that non loop unrolled i.e. naive matrix multiplication performs better than most two dimensional loop unrolled executions and some one dimensional loop unroll executions.

For Strassen-Winograd algorithm, the execution is better only for very large size matrices i.e. 1024x1024 and 2048x2048 have better speed-up than only GPU unrolled. One reason being that this algorithm gives best speed-up for a recursive implementation. Since the implementation here is only of one-level i.e. non-recursive, the speed-up due to reduction of single multiplication is going to be limited and the increase in the number of additions and subtractions increases the execution time. But, this algorithm helps with speeding the execution of large size matrices. For matrices of size 2048x2048, the speed-up goes more than a 100x when using Strassen-Winograd with 128 element unroll.

Problem 3

Part 1 - BFS - Suva Shahria

My computer has a GeForce GT 730 with a compute capacity of 3.5.

Based on the starting index chosen the code creates the bfs tree from that index.

Pictures of results and the graphs are in the results folder under part 3.

My create_graph.cpp makes a random amount of edges but i decided to make 7000 edges for each graph so we would have large bfs trees and eliminate the possibilities of very small amount of edges. You can comment in and out lines 46 and 47 to switch back and forth.

Time: in microseconds	Loop	Recur	Cuda
Graph 1	142	436	42.9

Graph 2	141	283	48.2
Graph 3	196	403	45.7

I ran my cuda code in the ilabs and got much better results. It seems my gpu is weak. I have a 7 year old computer.

I created new graphs for the ilab
Graph 1 - 5837 edges 100 vertices
Graph 2 - 7038 edges 100 vertices

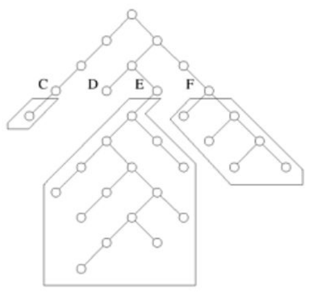
Time: in microseconds	Loop	Recur	Cuda
Graph 1	83	201	1.9
Graph 2	106	901	2.2

The recursive version was slower than the looped version because as it calls itself the stack needs to grow bigger taking up time.

Bfs finds all unvisited vertexes than push them to a queue. These vertices can be processes independently which lends itself well to parallelization. Each processor or thread has an evenly distributed workload.

If dfs the vertices are visited one by one along a line. Some vertices can't be processed until a chain of other vertices are processed. Not idle for parallelization. If the answer is far from the root dfs can outperform bfs because bfs has to slowly expand.

There can be major work load balance issues with dfs. Some processes will have to make a deep run into the tree while other processes will have an easier time.



In this picture, node e will need to be processed for longer than node c. The processor or thread that is working on c will become idle as e is finishing.

Part 2 - Prim's Algorithm - Jonathan Garner

I ran the compiled code on my personal computer using a GeForce GTX 970 graphics card with a compute capacity of 5.2. To run the code, a graph must first be generated using create_graph_weighted, which will create an NxN matrix representation of a binary tree, where each column/row represents a vertex and each nonzero value represents the weight of the path between the vertices that intersect at that column/row. This code was based off of Suva's create_graph, but modified to create weighted edges for Prim's algorithm.

By running the sequential version or cuda version of the algorithm code with inputs indicating the graph size and starting node, each vertex will be scanned and its nearest neighbor will be found in order to find the shortest path throughout the tree. Various randomly generated graphs of different sizes were used for testing, and some examples of these graphs and their outputs can be found in the results folder.

3) Overall, the cuda version of the algorithm was always faster than the CPU sequential execution, as expected. All time values displayed below are in microseconds. The console outputs of the following are also available as screenshots in the results folder.

Graph	**Loop**	**Recursive**	**Cuda**
Graph1.txt (50x50)	7730	7832	5701
Graph2.txt (80x80)	14663	14838	7785
Graph3.txt (200x200)	36716	38734	20585

4) After researching, I found that Kruskal's algorithm is very similar to Prim's, however the differences between them lead to the conclusion that Prim's algorithm is much more effective when parallelized. Kruskal's algorithm depends on a constantly updated heap while iterating through the vertices of a graph, which would require each parallel thread to access shared memory for every calculation, which would negatively impact the performance of the program. For Prim's algorithm, each thread is able to access a single vertex and find its nearest neighbor without having to reference or update a shared heap, which vastly improves performance as a program is scaled.

Further research also showed that the advantage of using Prim's algorithm is more noticeable in graphs with many edges, and in the case of this project the graphs I had used for testing were very edge-heavy. In a graph with V vertices and E edges, the runtimes of Kruskal's algorithm and Prim's algorithm are $O(E\log V)$ and $O(E+V\log V)$ respectively. This means that Prim's algorithm is much faster in the case of a graph with many edges.

5) The Boruvka algorithm is similar to both Prim's and Kruskal's algorithms, except it was designed with parallelization in mind and structured so that it must be parallelized for maximum efficiency. Instead of growing the tree sequentially as each vertex is found like Prim's algorithm,

the Boruvka algorithm adds all of the neighbors of the current vertex to the tree in a group and then operates on each of these neighbors simultaneously. In order to parallelize this, each neighbor to the starting vertex would have its own thread assigned to find its neighbors (and each of those would be assigned their own thread etc) until all vertices were accounted for.

Problem 4: (Dan Yates)

Problem 4 asks us to implement a parallel version of the function find_repeats using GPU cards, given an array of one million integers. The first step is to build the array of random integers. These integers could not go above 100, so I simply used the built in rand() function and modded it with 101. This was first done in C using Rutgers ilabs for simpler testing. Once the array is generated the code will look for repeating integers. These will be marked and removed while building a new array with no repeating integers. In order to accurately place these integers in the new array the function exclusive_scan needed to be passed several times. To implement exclusive_scan I followed the example from a given link to NVIDIA's developer forums. First I coded a simple version in C, once again using ilabs, and then followed their example.

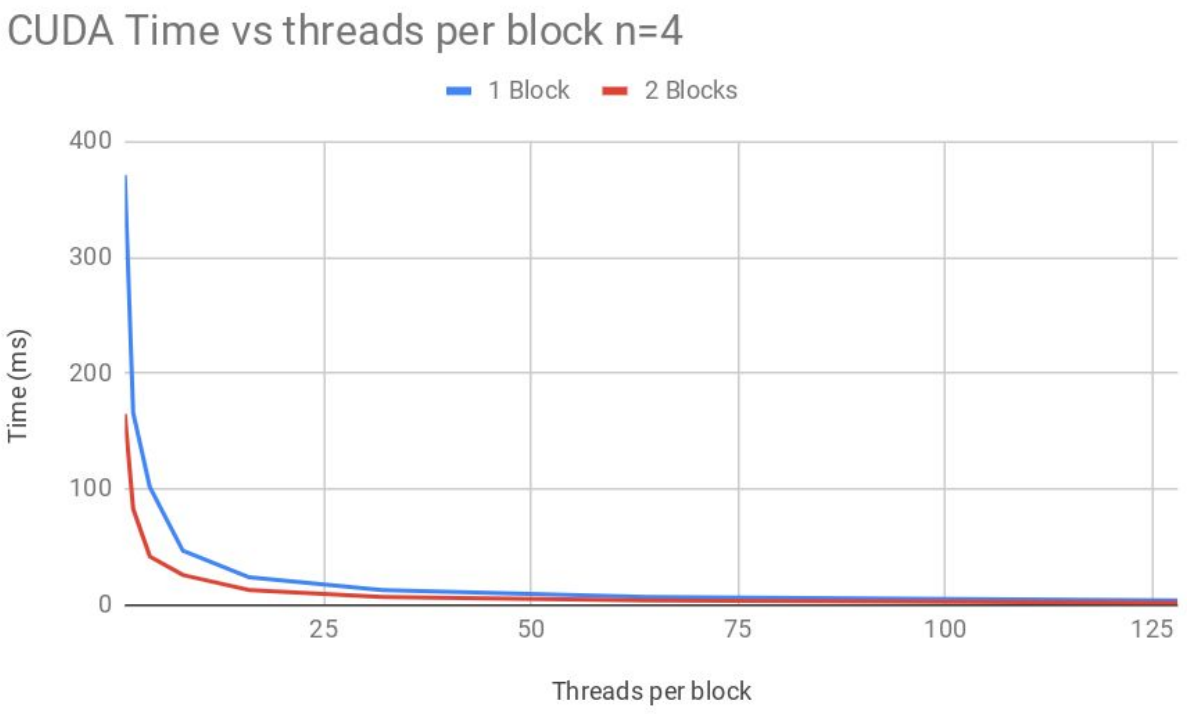
For implementing find_repeats I once again wrote a much simpler version in C to help me better understand the problem. I needed to find all repeating integers and mark their index. Afterwards the code needed to remove these repeating integers and create a new array without any repeats.

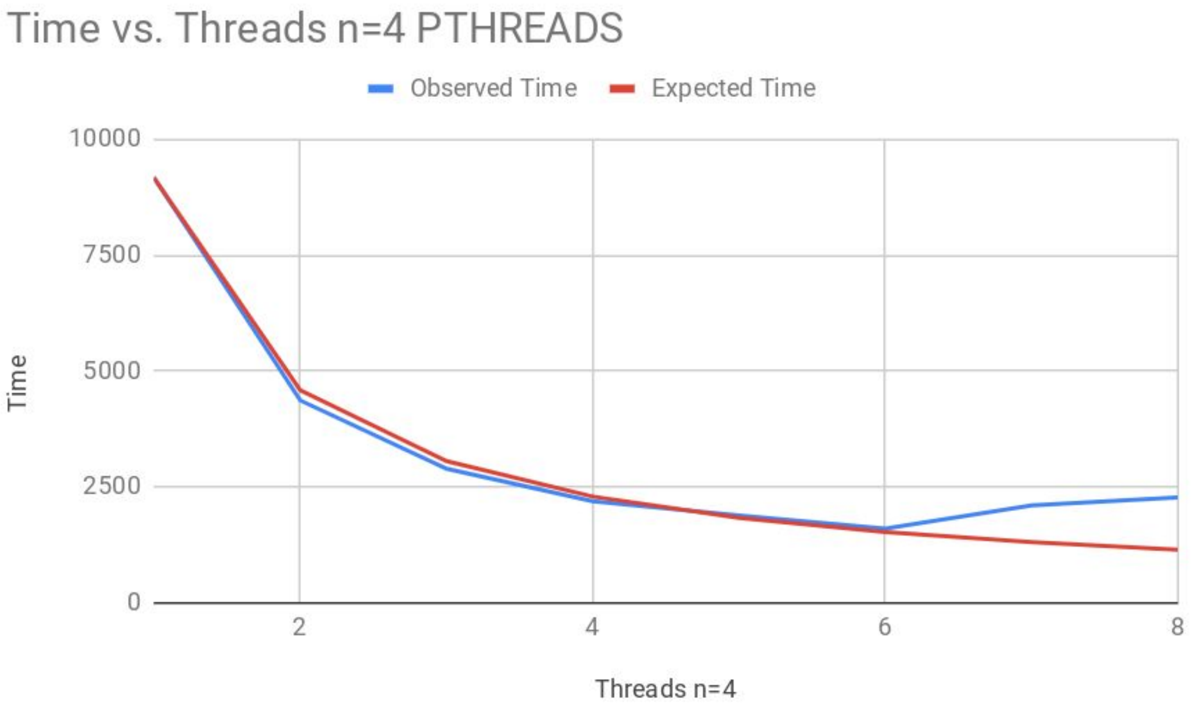
The idea behind this code was easier to follow after writing simpler versions but I struggled with the syntax of CUDA and making my code more dynamic. Using flags to mark repeating integers and removing them later is a better way to find repeating integers. Breaking up the code to find these repeats and to remove them simplified the problem but I still struggled with getting it to run.

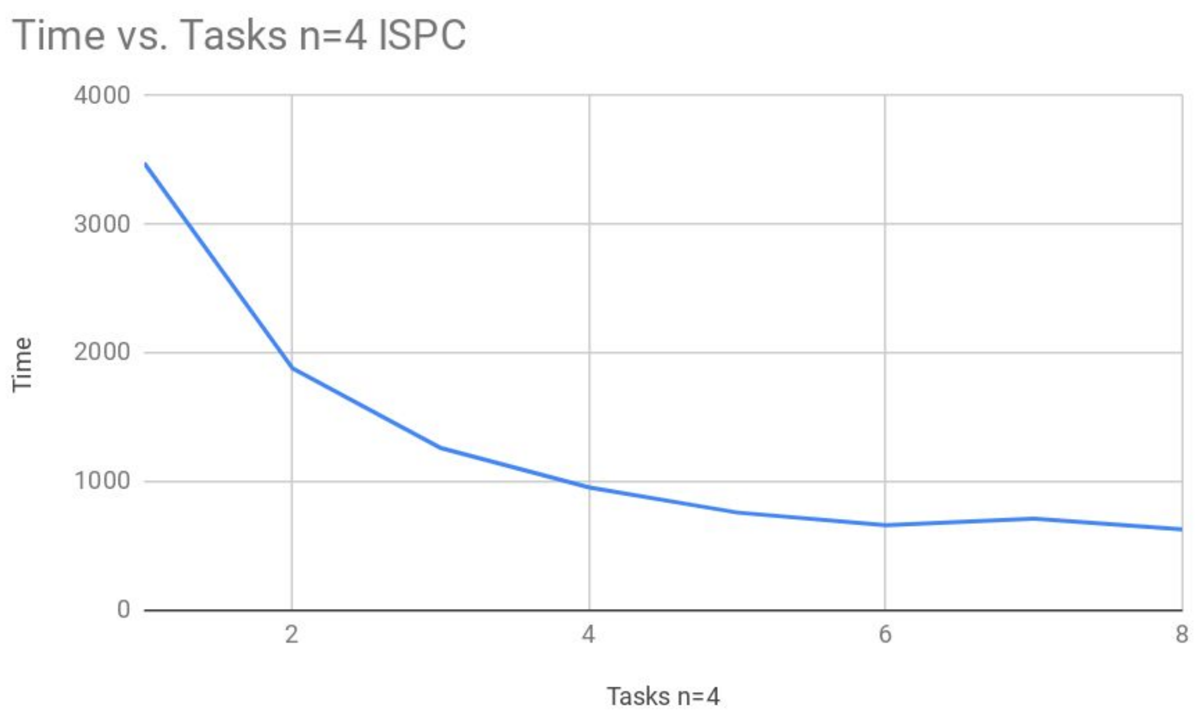
Problem 5(Group 3)

The implementation of my password cracker was similar to my implementation for pthreads, except modified to take advantage of cuda. Similar to my pthread implementation, first I took in a 32 length MD5 hash and converted it to a 128 bit digest. This time I made a change and instead of having 2 64-bit variables, I put it into a 4 length array of 32 bits each. This helped make the comparison with md5 a little easier.

I then had to make some global variables for the current block word, the cracked word, and the charset. The purpose of the current block word is because with CUDA we split the work into blocks, and the current_block word specifies the starting word for each block. The specific word that we attempt to hash is then based on its thread idx value. This way we have a specific thread working on its own md5 hash to take advantage of the huge parallelism of GPUs and CUDA. I did make one small change because as we increase the length of the password, the number of passwords we need to crack exponentially increases. I gave each thread a specific amount of passwords to hash. In my case I found 128 to be a good number per thread. With each thread working on more than 1 hash, we have a good balance between thread workload and number of threads. If each thread only had 1 job then there would be a lot of overhead with creating a new thread for each password we iterate through. The pthread version had a thread that split each length equally, and this is similar in a way except we have many many more threads to work with. This implementation is also a little similar to ISPC because we have warps which does perform SIMD. This makes CUDA seem like a combination of both ISPC and pthreads but optimized for heavy parallelization.
The function which compares the hash is called “compare_hashes” and this will iterate over the chosen number of hashes starting from its thread idx value, so in my case 128 hashes per thread. If a hash matches the given hash we copy it to the device_cracked variable. I try to copy as many variables as possible to local device memory to increase speed. Attempting to read from host memory is much slower than the device memory, and even faster shared local memory. Once all the blocks are done we copy the value back from device memory and see if we got a result. If we do we break from the loop and print the result. I use cudaDeviceSynchronize(); to make sure no extra work is done before we check for a solution. This is similar to a barrier in pthreads, except that it is a barrier for device code. When we launch a kernel, it returns immediately and I use this to prevent the loop from running before we check the answer.
In terms of even workload, each block should have an even workload since they are only finding hashes for the range 0 to TOTAL_THREADS * HASH_ITER * TOTAL_BLOCKS each iteration of the loop. So the blocks get progressively harder workloads every iteration.
The performance of this password cracker is much much higher than my previous iterations. Looking at the table below with the password we’re trying to crack being “zzzz” cuda performs much better than both ISPC and pthreads by more than a factor of 10. This is probably because I used many more threads in CUDA vs pthreads or ISPC. In the later I was limited to 12 threads on my PC, however with CUDA I have a GTX 1080 ti which has 28SM and each can run 2048 threads concurrently which is much more than my CPU.


CUDA Time vs threads per block n=4
1 Block
2 Blocks
400
300
200
100
0
Time (ms)
25
50
75
100
125
Threads per block


Time vs. Threads n=4 PTHREADS
Observed Time
Expected Time
10000
7500
5000
2500
0
Time
2
4
6
8
Threads n=4



Next, I experimented with a different number of passwords. For this I used 8192 total blocks with 128 threads each. Each thread runs for 64 iterations. As you can see, the time was 24ms for the first 4 lengths. This is because every password from length 1-4 can fit in the amount of threads and run the first iteration. Once we start getting to 5 and 6 the time rises, and with 6 we can see a trent in the password length vs time. The time is about 70x more than the previous for length of 5 to 6. This is because the charset length is about 70, so we see an increase in 70^n time for cracking these passwords.

Password length	Password	Time (ms)
1	z	24
2	zz	24
3	zzz	24
4	zzzz	24
5	zzzzz	447
6	zzzzzz	31654

Problem 5: (Group 6)

Password Cracker:

The approach for password cracker that we have used till is:

The work is split among the threads by the number of characters at the top level. By this approach, the peak performance was when we spawned 23 threads. This saturation is achieved as the number of cores was less (it was 4 for machine in which the code was run). So here, each thread iterated over 4 characters in the top level and iterator over all the other 95 characters in the subsequent levels.

But in a GPU, we have 100's of cores. The maximum parallelization the above approach will obtain is 1 character at the top level for each thread, so at the maximum 95 threads. Even this type of work decomposition will not utilize the capabilities of the GPU.

We did our experiments on the Sandbox 4. The GPU present in Sandbox 4 is GeForce GT 640 GDDR5. The specifications for this GPU is, it has 384 cores. The SIMD width is 32. Hence if we have 8 cores per ALU, we can perform 98304 operations in one clock cycle.

We can make two types of improvements here. Either we can perform the hash of a password using GPU and employ the GPU to use that. But since it is not the motivation here and calculating hash for 8 letter word takes on an average 3.3 ms (this is for 100 observations of length 8).

So, we have precomputed all the possible combinations and their hash values for different lengths for this project. (Similar to rainbow table). Each thread compares an element of the hash to the element of given hash. If they match, the corresponding string combination is the password. Each CUDA thread, gets the id from the block id and the thread id. Then each thread compares the two hashes. If they are the same, then it sets the answer string to the correct combination. Then the corresponding cracked password is passed to the host system.

The precomputation for strings of length 4 by itself takes 357 MB and their hashed passwords takes 4.5 GB of space.

The precomputation of string of length 5 takes 46 GB. The computation of this alone took 4.5 hrs in local machine. (2 GHz Intel Core i5).

The Sha computation for each thread was done in SB4 again, in batches as I could not copy the compressed 13 GB file to sb4. In three batches. The Cuda SHA 256 implementation (the open ssl one could not be used because of it being a host function). https://github.com/Horkyze/CudaSHA256 was used to generate the SHA 256 values. Over all the running time to compute all the hashes was about 2.5 hrs.

We couldn’t generate all the string combinations for letter 6. But that is because of the inefficiency of the local machine. I took a sample of 20% combinations (23 GB) for 6 letter combinations.

We did not generate for 7 and 8 letter because of the huge processing time in local machines. This process is sequential and it takes a lot of times and memory of more than 320 GB, which is not even available in the hard drive. But it can also be parallel batch processed, but due to time constraints we did not do that. We can see the improvements of the approach in GPU even for 3 and 4 character lengths.

Results:

The average time to crack a password of size 3 characters using pthreads is

Password length	P threads	GPU
1	37.52ms	0.764ms
2	43.12ms	0.794ms
3	47.89 ms	3.09 ms
4	129 ms	9.54 ms
5	~22 mins	89.42 seconds
6(with 20% accuracy)	Did not give an output for some time	132.52 seconds

The values are average, taken over 100 random sample passwords.

Of course, the precomputation helps CUDA output a lot, but for the same precomputed table, in pthreads, (it was done only for 10 samplespasswords = 23 threads, password size of 3), it took around 31.8 ms.

This is because all the 23 threads are still not run parallel like in CUDA, and SIMD is also not utilised, where as here, it becomes inherent because of warps.

The sudden spike in run time for passwords from 3 to 4 and from 4 to 5 maybe because of the huge memory that the GPU now handles, and memory is always a bottle-neck for GPU.
The precomputed files for size 5 and 6 characters is not included as the size if huge and is not possible to attach.