Project 3

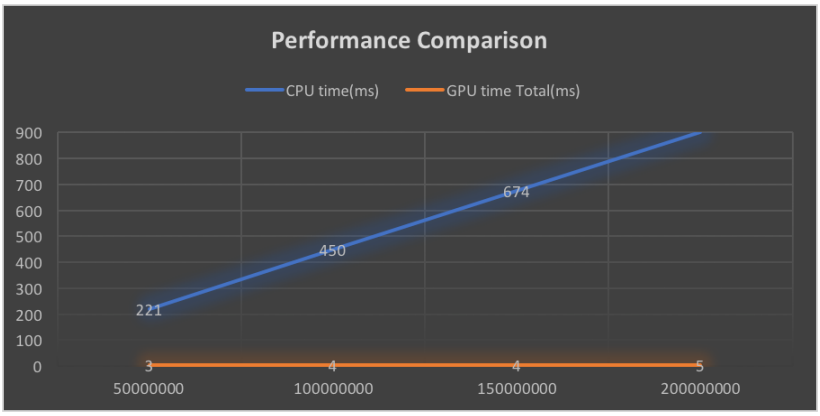
Yuwei Huang
Shuo Fan
Careena Braganza
Nirali Shah

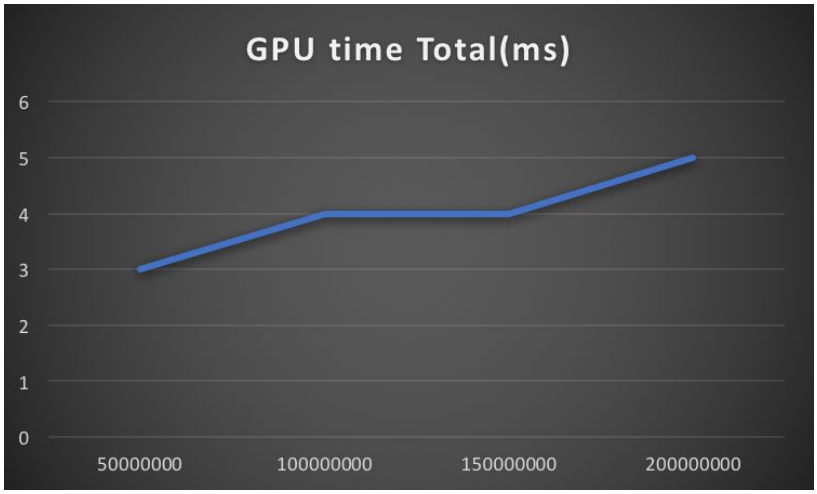
Part 1:

In this part, we implementation the CPU version and GPU version to find the max element in a vector. We record the running time of CPU and GPU, for the GPU, we record and compare the total running time and the execution time of the Kernel function The experiment result is shown as the following table. For the input data, we set the range of data from 0 to 100.

Size of Vector	CPU time(ms)	GPU time Total(ms)	GPU time Kernel Execution(ms)	CPU find max element	GPU find max element	Speedup GPU to CPU
50000000	221	1287	3	100	100	73.6
100000000	450	1662	4	100	100	112.5
150000000	674	2043	4	100	100	168.5
200000000	901	2614	5	100	100	180.2

Based on the result, we can see the output max value of CPU and GPU is the same.
For the comparison of performance, we find that the GPU has much better performance than the CPU implementation. When the size of vector become larger, the advantage of GPU also become larger. The maximum speedup of GPU comparing to CPU can be up to 180 times. But the total time of GPU is larger than the CPU time. We know that the program need to copy the data to the GPU memory from the host memory before calculation. When the calculation finished, the data also needed be copied from the GPU memory to the host memory. This result indicates that the copy of data cost much of the time. It means if we need to use GPU for calculation, the bandwidth of memory will be very important. If the bandwidth is not enough, the copy of data will cause very serious performance loss.





Extra ：Calculate max, min, mean, standard deviation concurrently

In this part, we computed the Max value, min value, mean and std concurrently with CPU and GPU, and compare the output results. The input data set is the same as the last experiment. Data ranges from 0 to 100.

Size of Vector	Max CPU	MAX GPU	Min CPU	Min GPU	Mean CPU	Mean GPU	STD CPU	STD GPU
50000000	100	100	0	0	50.01	51.01	28.87	28.88
100000000	100	100	0	0	49.99	50.99	28.86	28.88
150000000	100	100	0	0	50.006	50.12	28.87	28.88
200000000	100	100	0	0	50.003	50.99	28.87	28.88

Based on the result, we found that the mean value found by CPU and GPU has a little difference for the large size of vector. The reason may be when computing the sum of the vector for the mean, the number of digits remained by CPU and GPU is different, the accuracy of means computed by CPU and GPU is also different.
There is a same result to the standard deviation. This may due to the same reason because there is also a sum process when computing the standard deviation.

The performance comparison of CPU and GPU is shown as the following table.

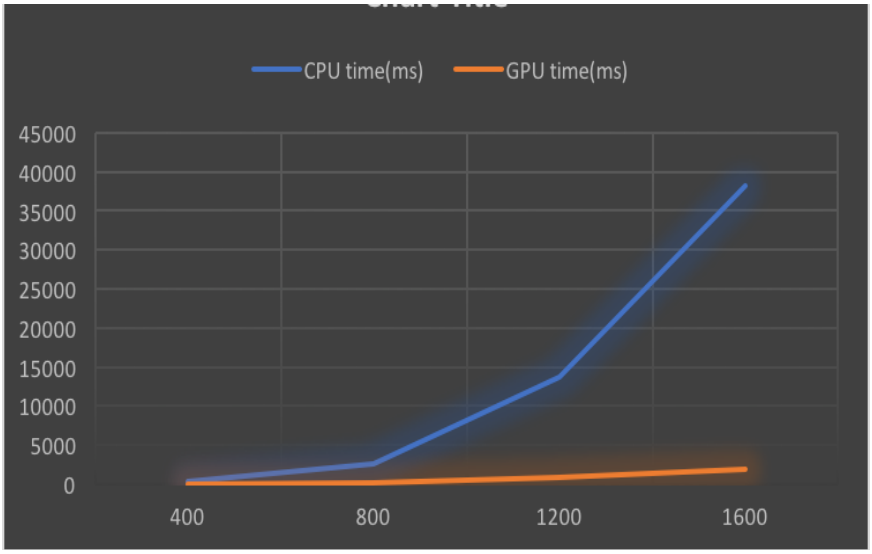
Size of Vector	CPU time(ms)	GPU time Total(ms)	GPU time Kernel Execution(ms)	Speedup
50000000	435	2235	25.23	17.18
100000000	877	3025	35.08	25
150000000	1313	4231	44.38	29.58
200000000	1654	5568	55.68	29.70

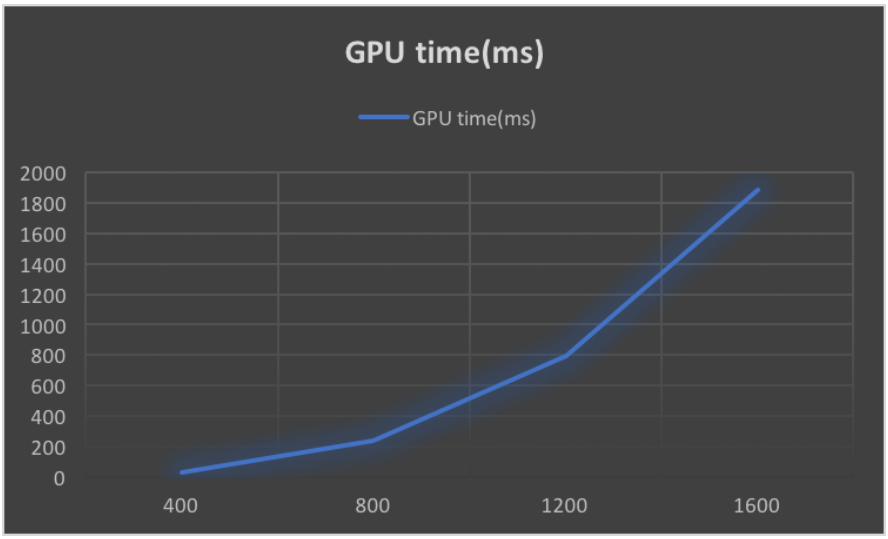
Based on the result, we can see a similar result as the last experiment. GPU has obviously performance advantage than the CPU. WHen the size of vector becomes larger, the advantage of GPU also becomes larger.

Part 2: MATRIX MULTIPLICATION

Problem 1: Compare the time of CPU implementation and GPU implementation
In this part, we implement the matrix multiplication both on CPU and GPU. For the CPU version, we used three loops to calculate the product of the two matrix. The time complexity of this implementation should be O(N^3). For the GPU version, we used each thread of GPU to calculate one element of the product matrix. That means for every thread, we firstly calculate the the row and column it needs to calculate, then use a loop to calculate the value of element for this row and column. The test result is shown in the following table.

Dimension **of matrix**	**CPU time(ms)**	**GPU time(ms)**
400	298	31
800	2601	237
1200	13765	796
1600	38219	1884





We compare the result of CPU and GPU, obviously, the performance of GPU is much better than the CPU.

Problem2: Calculate the estimated GFLOPS

Dimension of matrix	**Total FLOP**	**CPU FLOPS**	**GPU FLOPS**	**Speedup**
400	128M	429M	4129 M	9.62X
800	1024M	393M	4320 M	10.99X

1200	3456M	251M	4341M	17.29X
1600	8192M	214M	4357M	20.36X

From the test result, we can find the FLOPS of GPU is much larger than the CPU. When the dimension of matrix increases, the FLOPS of CPU decreases, when the dimension of matrix change from 400 to 1600, the FLOPS of CPU decreased about 50%. It is a very big performance lost. The FLOPS of GPU is stable as the increase of the dimension of matrix, keep at more than 4000 MFLOPS. So, when the input matrix becomes larger, the speedup of GPU comparing to CPU also becomes larger.

Problem 3: Implementation using cuBLAS library and compare with CPU and GPU implementation

In this part, we implement the matrix multiplication with the cuBLAS library and verify the result of our implementation using the cuBLAS library with mean square error (MSE). Also, we compared the performance of CPU and our implementation with cuBLAS library.

Compare of CPU and cuBLAS

Dimension of matrix	CPU time(ms)	cuBLAS time(ms)	Mean Square Error
400	298	1	2.58*10^-11
800	2601	2	1.03*10^-10
1200	13765	5	2.82*10^-10
1600	38219	10	4.13*10^-10

Compare of GPU and cuBLAS

Dimension of matrix	GPU time(ms)	cuBLAS time(ms)	Mean Square Error
400	31	1	2.58*10^-11
800	237	2	1.03*10^-10
1200	796	5	2.82*10^-10
1600	1884	10	4.13*10^-10

From the result, we find that the mean square error(MSE) of our result is very close to zero, that means the result of our implementation is close to the result of cuBLAS library. So, based on the result, we can concluded that our implementation is correct. We also find that the performance of cuBLAS library is much better than our GPU implementation. That means our implementation

can have many methods to be optimized and cuBLAS library is a very efficient implementation that has been optimized very well.

Problem 4: Optimizations of GPU implementation

Optimization 1: Loop unrolling

Used five times loop unrolling, that means in every loop, we add the products of corresponding elements for five times, this process is shown in the following figure.By this method, the total times execution of loop will decreased.

```
for (int ii = 0; ii < n; ii=ii+5)
{
    t = t + aa[row*n + ii] * bb[ii*n + column];
    t = t + aa[row*n + ii + 1] * bb[(ii + 1)*n + column];
    t = t + aa[row*n + ii + 2] * bb[(ii + 2)*n + column];
    t = t + aa[row*n + ii + 3] * bb[(ii + 3)*n + column];
    t = t + aa[row*n + ii + 4] * bb[(ii + 4)*n + column];
```

Figure: The implementation of 5 times loop unrolling

The experiment of loop unrolling is shown as the following table.

Dimension of matrix	**CPU time(ms)**	**GPU time(ms)**	**GPU time with loop unrolling(ms)**	**Speedup**
400	298	31	26	16%
800	2601	237	205	13%
1200	13765	796	657	17.4%
1600	38219	1884	1523	19.1%

From the result, we can find the efficient of loop unrolling is not very good. When used 5 times loop unrolling, the execution time of CPU only decreased a little. The max speedup is about 20%.So, we still need to find more efficient optimization method.

Optimization 2: Shared memory

In this optimization, we copied each row of the matrix into the shared memory, used the shared memory for high speed access to achieve better performance. But in this case, the data in the shared memory can only be shared by threads in the same block. So, we used the thread within the same block to compute each row we copied to the shared memory. The experiment result is shown in the following table.

Dimension of matrix	**CPU time(ms)**	**GPU time(ms)**	**GPU time with shared memory)**	**Speedup**

400	298	31	15	51.6%
800	2601	237	103	56.54%
1200	13765	796	353	55.65%
1600	38219	1884	946	49.78%

Based on the result, we find the optimization of shared memory is better than the loop unrolling. The speedup can be more than 50%. That means with the shared memory, the GPU run time can be just the half comparing to the naive implementation. But the efficient of shared memory is still worse than the cuBLAS library. So better optimization methods can still be found.

Part 3: Breadth First Search

In the breadth first search algorithm, we begin exploring all the vertices of the graph in each level before moving to the next. The idea is - start at all vertices that are 1 edge away from source, the move to 2 edges away and so on.

Pseudo code for sequential BFS:

```
1.  let G <- adjacency matrix
2.  let Q <- queue
3.        Q.enqueue(source)
4.        label source as discovered
5.        source.value <- 0
6.        while Q not empty
7.          v ← Q.dequeue()
8.          for (vertices v to w in adjacentEdges(v)):
9.              if w not discovered
10.                 Q.enqueue(w)
11.                 label w as discovered
12.                   w.value <- v.value + 1
13.             end for
14.    end while
15. return result
```

Parallelizing BFS

One thread is assigned to each vertex. We make sure that once a level is complete, it is not visited again. To ensure this, Barrier Synchronization is added. Two boolean arrays are maintained - one for the current level vertices *(Frontier array)* and one for the visited vertices.

In each iteration - each thread updates its cost for the neighbours as 1 + its own cost. The vertex **removes its entry** from the *frontier array* and it is added to the visited array and adds its neighbours to frontier array, if not already visited. This is repeated until the frontier array is empty.

Pseudocode for Parallel BFD:

```
1.  let Va <- visited array, Fa <- frontier array, C <- cost array.
2.  initialize Fa , Va <- false, Ca <- infinity
3.  Fa[source] <- true
4.  Ca[source] <- 0
5.  while ( Fa not empty):
6.          for (v in parallel):
7.                  call to CUDA kernel
8.          end for
9.  end while
10. copy result host to device
```

BFS Kernel

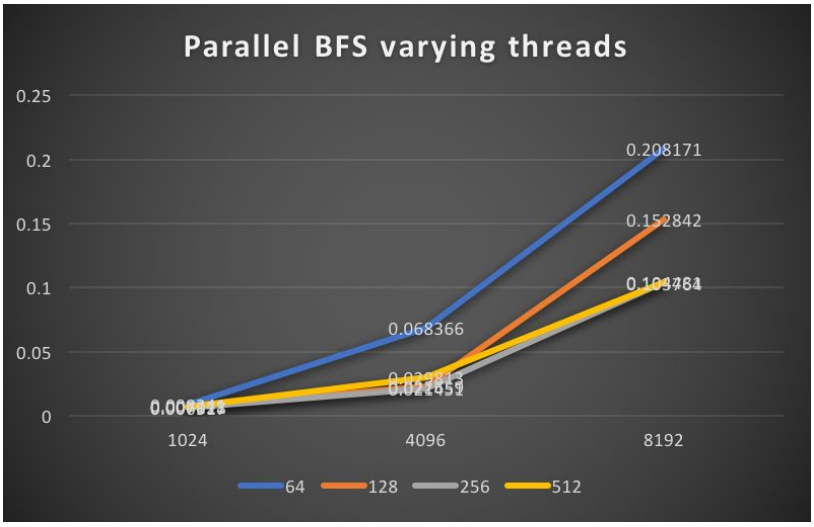
```
1.  tid ← getThreadID
2.  if Fa[tid] == true
3.      Fa[tid] ← false
4.      for (neighbors nid of tid):
5.          if NOT Va[n]:
6.              Ca[n] ← Ca[tid]+1
7.              Fa[n] ← true
8.          end if
9.      end for
10. end if
```

Experimental Results

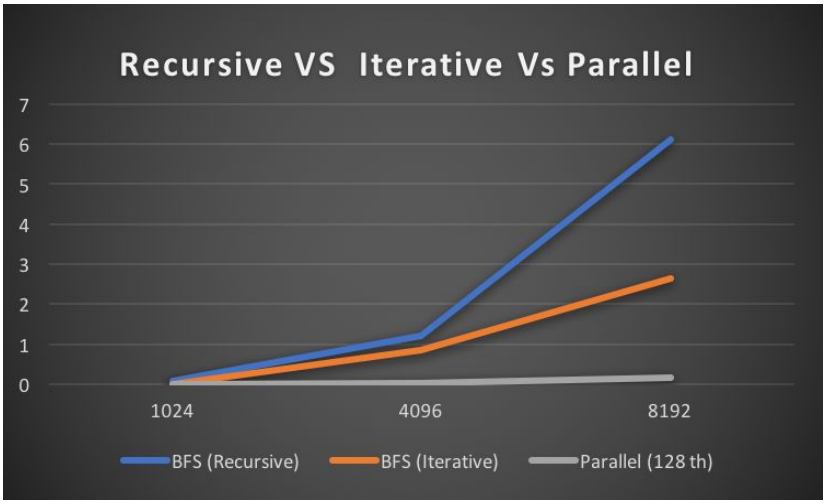
Input size	BFS (Recursive)	BFS (Iterative)	Parallel BFS (varying threads)			
			64	128	256	512
1024	0.08105	0.00812	0.008241	0.007018	0.006517	0.007121
4096	1.22308	0.86329	0.068366	0.022659	0.021451	0.029813

8192	6.10332	2.63451	0.208171	0.152842	0.104481	0.103764

Comparison for varying number of threads



Performance Improvement for Parallel BFS (128 threads)



Part 4:

We use an input array A stored in Input.txt. The test input is defined. The first device function is exclusive scan, there are two phases up sweep and down sweep,which takes an array A and produces a new array output that has, at each index i, the sum of all elements up to but not including A[i]. We use shared block memory to store the range of variable to be used in one block. Using find_repeats and its helper to detect replicate. Output is in the ArrayA.txt ArrayB.txt ArrayC.txt .
On the display, we output the last element of the find_repeat results.