Programming 2

In this programming homework, I will choose the version of cuda-toolkit 8.0.44 as the CUDA compiler environment. For hardware environment, I will choose NVDIA Tesla P100 as the main GPU for calculation. The specific computing environment shown in Figure 1.

```
[wufangm@login001 ~]$ qsub -I -l select=1:ncpus=16:ngpus=1:gpu_model=p100,walltime=0:30:00
qsub (Warning): Interactive jobs will be treated as not rerunnable
qsub: waiting for job 4338489.pbs02 to start
qsub: job 4338489.pbs02 ready

[wufangm@node0082 ~]$ module purge
[wufangm@node0082 ~]$ module load cuda-toolkit/8.0.44
[wufangm@node0082 ~]$
```

Figure 1: Calculation environment for programming 2

Part1:

In this part, I have changed the matrix size for 4 times. In these four changes, the size of matrix A and B will increase in turn. The matrix sizes of A are (640, 480), (768, 576), (1024, 768), (1280, 960). The matrix sizes of B are (480, 320), (576, 384), (768, 512), (960, 640). Correspondingly, the sizes of matrix C calculated by matrix multiplication are (640, 320), (768, 384), (1024, 512) and (1280, 640). Among these changes of matrix sizes, I found that the time spent of computational as increase as the matrix size(shown in Figure 2).

```
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 2521.79 GFlop/s, Time= 0.078 msec, Size= 196608000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
[wufangm@node0602 ~]$
```

Figure 2(1): The final performance when the size of A is (640,480), the size of B is (480,320), and the size of C is (640, 320)

```
[wufangm@node0602 ~]$ ./CUDA/mmCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(768,576), MatrixB(576,384), MatrixC(768,384)
Computing result using CUBLAS...done.
Performance= 2754.64 GFlop/s, Time= 0.123 msec, Size= 339738624 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
[wufangm@node0602 ~]$
```

Figure 2(2): The final performance when the size of A is (768,576), the size of B is (576,384), and the size of C is (768, 384)

```
[wufangm@node0082 ~]$ ./CUDA/mmCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(1024,768), MatrixB(768,512), MatrixC(1024,512)
Computing result using CUBLAS...done.
Performance= 4224.82 GFlop/s, Time= 0.191 msec, Size= 805306368 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
[wufangm@node0082 ~]$
```

Figure 2(3): The final performance when the size of A is (1024, 768), the size of B is (768, 512), and the size of C is (1024, 512)

```
[wufangm@node0082 ~]$ ./CUDA/mmCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(1280,960), MatrixB(960,640), MatrixC(1280,640)
Computing result using CUBLAS...done.
Performance= 5806.77 GFlop/s, Time= 0.271 msec, Size= 1572864000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
[wufangm@node0082 ~]$
```

Figure 2(4): The final performance when the size of A is (1280, 960), the size of B is (960, 640), and the size of C is (1280, 640)

In my opinion, the reason why the execution time as increase as the size of the matrix is that the amount of computation of GPU for the matrix multiplication increase under the same condition, which will take more time for calculation.

Part2:

In this part, I try to use the basic CUDA to implicate the matrix multiplication, which replaces to the function of cublasSgemm() in the program of part1(the code shown in Figure 3).

```
30
31 // Multiple Matrix C = A * B by using GPU
32 __global__
33 void matrixMulKernal(float *C, const float *A, const float *B, unsigned int hA, unsigned int wA, unsigned int wB){
34     /*
35     *  A[hA][wA]
36     *  B[wA][wB]
37     *  C[hA][wB]
38     */
39
40     int threadId = (blockIdx.y * blockDim.y + threadIdx.y) * gridDim.x * blockDim.x
41                     + blockIdx.x * blockDim.x + threadIdx.x;
42
43
44     if (threadId < hA * wB) {
45
46         int row = threadId / wB;
47         int column = threadId % wB;
48
49         C[threadId] = 0;
50
51         for(int i = 0; i < wA; i++)
52         {
53             C[threadId] += A[row * wA + i] * B[i * wB + column];
54         }
55
56     }
57     else
58     {
59         printf("The current threadId has exceeded the range of matrix C\n");
60     }
61
62
63     // Synchronize (ensure all the data is available)
64     // __syncthreads();
65
66 }
67
```

Figure 3(1): Implicate the matrix multiplication by using basic CUDA

```
235     // Send Matrix to Device
236     printf("Sending Matrics to GPU....\n");
237     checkCudaErrors(cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice));
238     checkCudaErrors(cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice));
239     printf("done.\n");
240     //Calculate the number of blocks and threads
241     dim3 threads(block_size,block_size);
242     dim3 grid(matrix_size.uiWC / threads.x, matrix_size.uiHC / threads.y);
243     // dim3 grid(matrix_size.uiWB/block_size, matrix_size.uiHA/block_size);
```

Figure 3(2): Sending matrix A and B to GPU and calculate the total number of blocks and the amount of threads in each block

```
for (int j = 0; j < nIter; j++)
{
    //note cublas is column primary!
    //need to transpose the order
    matrixMulKernal<<<grid,threads>>>(d_C,d_A,d_B,matrix_size.uiHA,matrix_size.uiWA,matrix_size.uiWB);
    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();


}
```

Figure 3(3): Call the matrix multiplication function I have already defined to perform the operation

```
298         // Get the result from Device
299         printf("Receiving the result from GPU....\n");
300         checkCudaErrors(cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost));
301         printf("done.\n");
```

Figure 3(4): Receive the final result from GPU

After observing the final input of the program implicated by using the basic CUDA, I found that (shown in Figure 4).

```
[wufangm@node0082 ~]$ ./CUDA/CUDA_matrixMultiple
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Create Matrix A.....
done.
Create Matrix B.....
done.
Create Matrix C.....
done.
Sending Matrics to GPU....
done.
Time recording begin....
Time recording done.


Performance= 252.30 GFlop/s, Time= 0.779 msec, Size= 196608000 Ops

Receiving the result from GPU....
done.
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
Clean up Host Memory.....
done.
Clean up Deivce Memory.....
done.
Program Completed!!!
[wufangm@node0082 ~]$
```

Figure 4(1): The final performance when the size of A is (640,480), the size of B is (480,320), and the size of C is (640, 320)

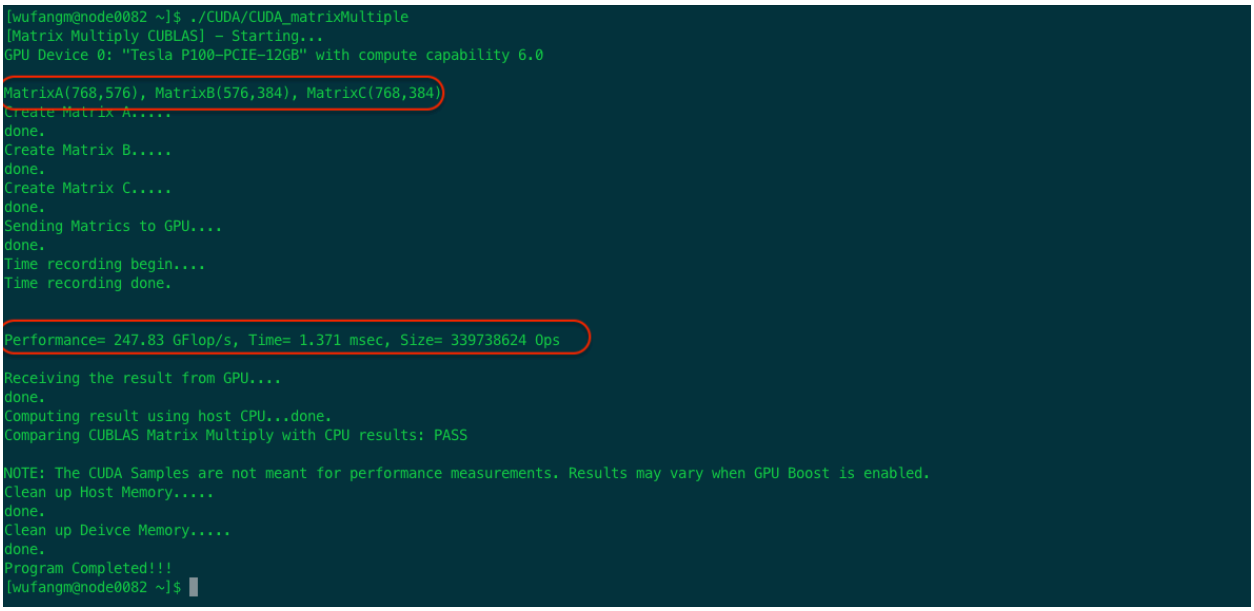

Figure 4(2): The final performance when the size of A is (768,576), the size of B is (576,384), and the size of C is (768, 384)

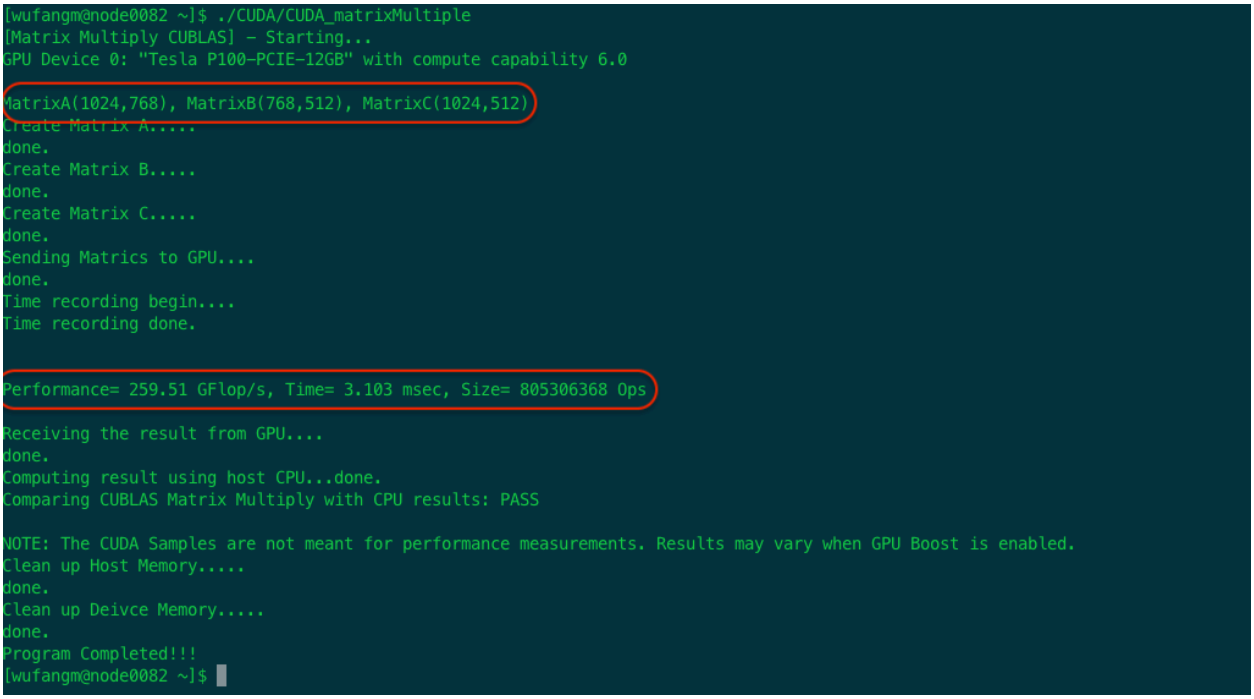

Figure 4(3): The final performance when the size of A is (1024, 768), the size of B is (768, 512), and the size of C is (1024, 512)

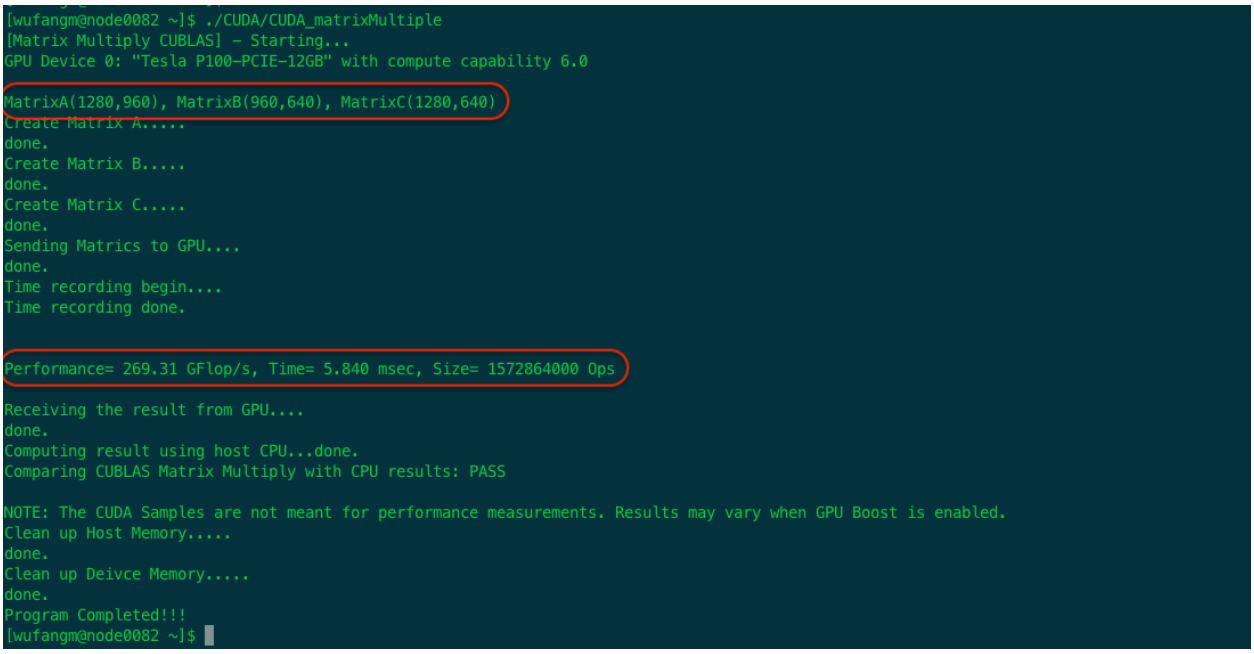

Figure 4(4): The final performance when the size of A is (1280, 960), the size of B is (960, 640), and the size of C is (1280, 640)

I found that my implication for matrix multiplication by using global memory is much slower than the function of cublasSgemm(), where the execution time cost experience a substantial increase but the performance degradation significantly. Take the matrix multiplication when matrix size of A is (1024, 768), B is (768, 512) and C is (1024, 512) as an example, the time cost in part 1(shown in Figure 2(3)) is 0.191 msec and performance is 4224.82 GFlop/s, but the time cost is 3.103 msec and the performance decrease to 259.51 GFlop/s in my program(shown in Figure 4(3)).

For my part, the reason for this performance and time gaps is that we use the global memory in GPU to calculate the matrix multiplication. Compared with the normal calculation in GPU that thread and thread do not communicate with each other, all the threads can read and write the device global memory in our program, which will provide some extra overheads. Also, it takes extra time from thread to global memory when a thread reads data from global memory. Thus, A large delay occurs when all data is read from global memory.

To increase the performance of my program, I think that we should add a warp in each block for my program. The reason is that threads in the warp can communicate with each other with low latency and will not be stall, which will reduce a lot of unnecessary overhead and save time in transferring data to improve computing performance.

Part3:

In this part, I have implemented the matrix multiplication with UVM(Unified Memory). The only different between Unified Memory and basic CUDA programming in the programming level is the way to request RAM. In basic CUDA, we use the function of “cudaMalloc()” to get the space from GPU. In Unified Memory, however, we just change the function of cudaMallocManaged() for requesting the space(code shown in Figure 5). The benefit of Unified Memory is the space allocated by Unified Memory can be accessed by both of CPU and GPU. Generally speaking, CPU allocates the data is pageable by default, which makes GPU unable to read directly. Each time when GPU want to read the data from CPU, it need to transfer this pageable data. In Unified Memory, however, GPU can read the data directly without the need for data transfer.

```
205     // Initialzie Matrix A, B, C
206     float *d_A, *d_B, *d_C;
207     //Matrix A
208     printf("Create Matrix A.....\n");
209     unsigned int size_A = matrix_size.uiHA * matrix_size.uiWA;
210     unsigned int mem_size_A = sizeof(float) * size_A;
211     float *h_A = (float *)malloc(mem_size_A);
212     checkCudaErrors(cudaMallocManaged((void **) &d_A, mem_size_A));
213     randomInit(h_A, size_A);
214     printf("done.\n");
215
216     //Matrix B
217     printf("Create Matrix B.....\n");
218     unsigned int size_B = matrix_size.uiHB * matrix_size.uiHB;
219     unsigned int mem_size_B = sizeof(float) * size_B;
220     float *h_B = (float *)malloc(mem_size_B);
221     checkCudaErrors(cudaMallocManaged((void **) &d_B, mem_size_B));
222     randomInit(h_B,size_B);
223     printf("done.\n");
224
225     //Matrix C
226     printf("Create Matrix C.....\n");
227     unsigned int size_C = matrix_size.uiHC * matrix_size.uiWC;
228     unsigned int mem_size_C = sizeof(float) * size_C;
229     float *h_C = (float *)malloc(mem_size_C);
230     checkCudaErrors(cudaMallocManaged((void **) &d_C, mem_size_C));
231     printf("done.\n");
232
233
```

Figure 5: Request the space by using the function of cudaMallocManaged()

```
[wufangm@node0082 ~]$ ./CUDA/CUDA_withUVM_matrixMultiple
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Create Matrix A.....
done.
Create Matrix B.....
done.
Create Matrix C.....
done.
Sending Matrics to GPU....
done.
Time recording begin....
Time recording done.

Performance= 228.79 GFlop/s, Time= 0.859 msec, Size= 196608000 Ops

Receiving the result from GPU....
done.
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
Clean up Host Memory.....
done.
Clean up Deivce Memory.....
done.
Program Completed!!!
```

Figure 6(1): The final performance when the size of A is (640,480), the size of B is (480,320), and the size of C is (640, 320)

```
[wufangm@node0082 ~]$ ./CUDA/CUDA_withUVM_matrixMultiple
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(768,576), MatrixB(576,384), MatrixC(768,384)
Create Matrix A.....
done.
Create Matrix B.....
done.
Create Matrix C.....
done.
Sending Matrics to GPU....
done.
Time recording begin....
Time recording done.

Performance= 236.38 GFlop/s, Time= 1.437 msec, Size= 339738624 Ops

Receiving the result from GPU....
done.
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
Clean up Host Memory.....
done.
Clean up Deivce Memory.....
done.
Program Completed!!!
[wufangm@node0082 ~]$
```

Figure 6(2): The final performance when the size of A is (768,576), the size of B is (576,384), and the size of C is (768, 384)

```
[wufangm@node0082 ~]$ ./CUDA/CUDA_withUVM_matrixMultiple
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(1024,768), MatrixB(768,512), MatrixC(1024,512)
Create Matrix A.....
done.
Create Matrix B.....
done.
Create Matrix C.....
done.
Sending Matrics to GPU....
done.
Time recording begin....
Time recording done.


Performance= 254.27 GFlop/s, Time= 3.167 msec, Size= 805306368 Ops

Receiving the result from GPU....
done.
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
Clean up Host Memory.....
done.
Clean up Deivce Memory.....
done.
Program Completed!!!
[wufangm@node0082 ~]$
```

Figure 6(3): The final performance when the size of A is (1024, 768), the size of B is (768, 512), and the size of C is (1024, 512)

```
[wufangm@node0082 ~]$ ./CUDA/CUDA_withUVM_matrixMultiple
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(1280,960), MatrixB(960,640), MatrixC(1280,640)
Create Matrix A.....
done.
Create Matrix B.....
done.
Create Matrix C.....
done.
Sending Matrics to GPU....
done.
Time recording begin....
Time recording done.


Performance= 266.47 GFlop/s, Time= 5.902 msec, Size= 1572864000 Ops

Receiving the result from GPU....
done.
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
Clean up Host Memory.....
done.
Clean up Deivce Memory.....
done.
Program Completed!!!
[wufangm@node0082 ~]$
```

Figure 6(4): The final performance when the size of A is (1280, 960), the size of B is (960, 640), and the size of C is (1280, 640)

In the Figure 6, we can see that despite the time cost and performance between basic CUDA(shown in Figure 4) and Unified Memory(shown in Figure 6) are pretty close, the basic CUDA is better than Unified memory whatever it is running time and performace. The reason is that, in my opinion, the page migration engine in Unified Memory creates extra overheads compared with the normal CUDA and this overheads may cause his performance to be inferior to the normal CUDA.

Part4:

In this part, I will try to implicate the matrix multiplication by using OpenMP(the code part of OpenMP shown in Figure 7). After compile and run my OpenMP code, I found that the performance in my OpenMP program has shown the worst performance in all the programs I wrote in this assignment(the performance shown in Figure 8). The one of reason, I think, is that I am not proficient in OpenMP and unfamiliar with the resource allocation in OpenMP.

```
// Multiple Matrix C = A * B by using GPU
void
openMP_matrixMulGPU(float *C, const float *A, const float *B, unsigned int hA, unsigned int wA, unsigned int wB)
{
    #pragma omp target
    {

        #pragma omp parallel for
        for (unsigned int i = 0; i < hA; ++i)
            #pragma omp parallel for
            for (unsigned int j = 0; j < wB; ++j)
            {
                double sum = 0;
                #pragma omp parallel for
                for (unsigned int k = 0; k < wA; ++k)
                {
                    double a = A[i * wA + k];
                    double b = B[k * wB + j];
                    sum += a * b;
                }

                C[i * wB + j] = (float)sum;
            }
    }

}
```

Figure 7

```
[wufangm@node1287 CUDA]$ ./openMP_matrixMultiple
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Create Matrix A.....
done.
Create Matrix B.....
done.
Create Matrix C.....
done.
Time recording begin....
Time recording done.

Performance= 2.58 GFlop/s, Time= 76.289 msec, Size= 196608000 Ops

Receiving the result from GPU....
done.
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
Clean up Host Memory.....
done.
Clean up Deivce Memory.....
done.
Program Completed!!!
[wufangm@node1287 CUDA]$
```

Figure 8(1): The final performance when the size of A is (640,480), the size of B is (480,320), and the size of C is (640, 320)

```
[wufangm@node1287 CUDA]$ ./openMP_matrixMultiple
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-12GB" with compute capability 6.0

MatrixA(768,576), MatrixB(576,384), MatrixC(768,384)
Create Matrix A.....
done.
Create Matrix B.....
done.
Create Matrix C.....
done.
Time recording begin....
Time recording done.

Performance= 2.08 GFlop/s, Time= 163.666 msec, Size= 339738624 Ops

Receiving the result from GPU....
done.
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
Clean up Host Memory.....
done.
Clean up Deivce Memory.....
done.
Program Completed!!!
[wufangm@node1287 CUDA]$
```

Figure 8(2): The final performance when the size of A is (768,576), the size of B is (576,384), and the size of C is (768, 384)

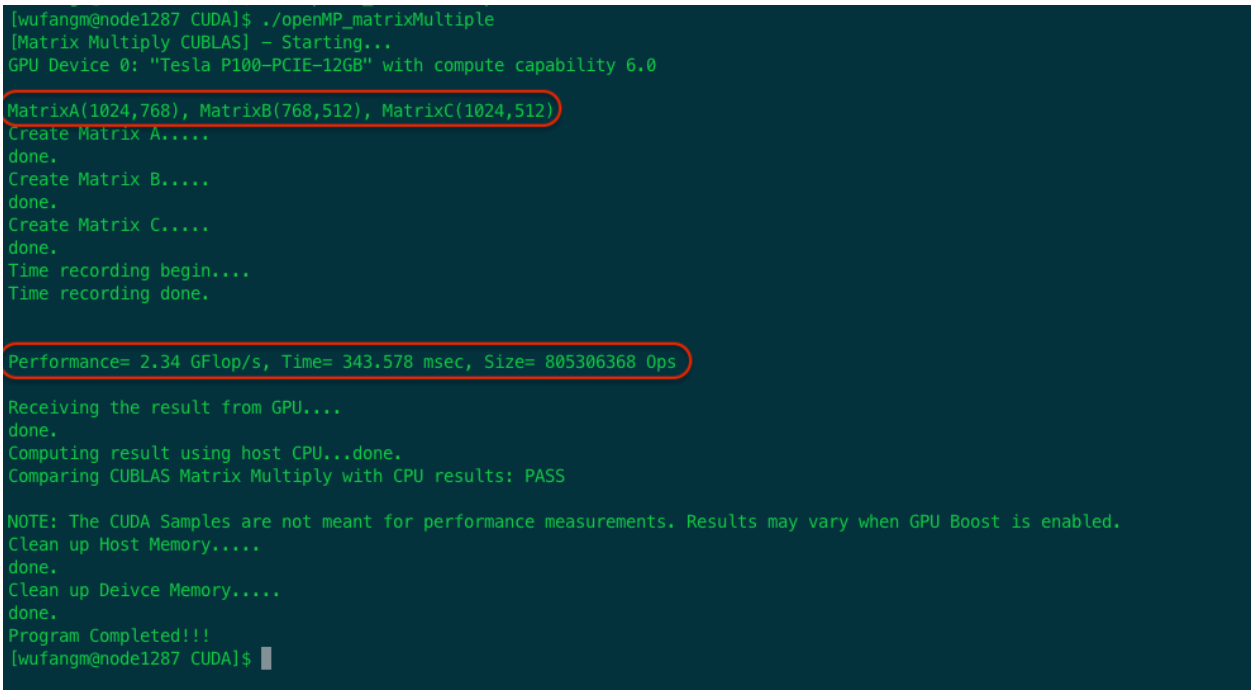

Figure 8(3): The final performance when the size of A is (1024, 768), the size of B is (768, 512), and the size of C is (1024, 512)

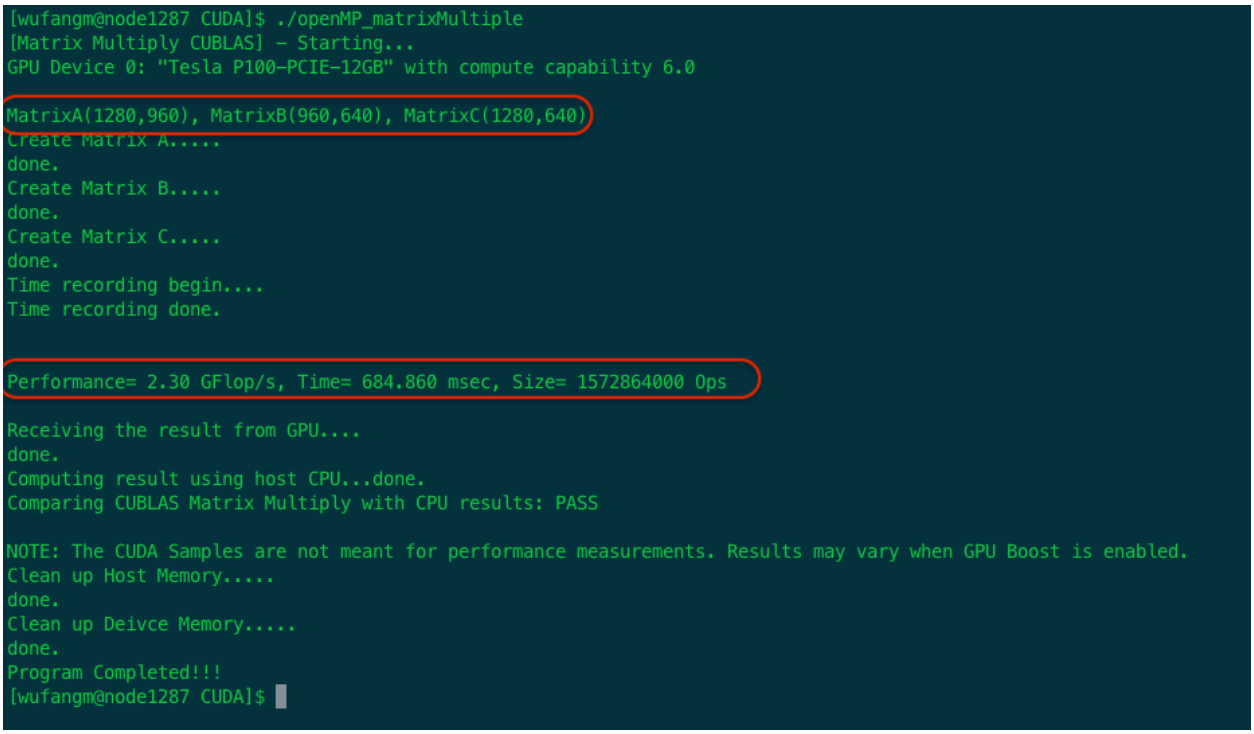

Figure 8(4): The final performance when the size of A is (1280, 960), the size of B is (960, 640), and the size of C is (1280, 640)

According to the time consumption and performance based on the program I wrote in this assignment and my understanding of all programming architecture and their principles in this assignment, I think that OpenMP is a programming architecture which easiest to think. The reason is that we don't need to think too much in the OpenMP compared with CUDA. In other words, we don't need to distribute the threads and block in the OpenMP, all the things we need to do is just allocate the variable to GPU. For the second question, I think CUDA using the function of cublasSgemm() have shown the best performance to us for the problem of matrix multiplication.

If encounter the same problem as matrix multiplication in my future work, I would like to choose the first one (That is mmCUBLAS). The reason is that the first one have shown the best performance to me. In the real work, a small performance boost may have a profound and positive impact on the entire project. Moreover, less computing time and higher performance may help the entire team and company to save much more money, which is also a very important factor for the company.