# REPORT

## 보고서 작성 서약서

1. 나는 타학생의 보고서를 복사(Copy)하지 않았습니다.
2. 나는 타학생의 보고서를 인터넷에서 다운로드 하여 대체하지 않았습니다.
3. 나는 타인에게 보고서 제출 전에 보고서를 보여주지 않았습니다.
4. 보고서 제출 기한을 준수하였습니다.

나는 보고서 작성시 위법 행위를 하지 않고,
성.균.인으로서 나의 명예를 지킬 것을 약속합니다.

**Class Name : 임베디드시스템설계**
**Subject : HW2**
**Professor : 오 윤 호**
**Department : 전자전기공학부**
**Student ID : 2019310649**
**Name : 임 용 성**
**Submission Date : 2021-10-13**

# 1. Profiling and Optimizing Matrix Concatenation

## 1) Figure out inefficiencies of your CUDA code

Fig. 1.은 HW1에서 프로그래밍한 gpu code이다.

```
__global__
void device_matrix_concat(int * C, int * A, int * B, int numOps, int block, int thread, int m, int p, int q)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int offset = gridDim.x * blockDim.x;


    for (int i = 0 ; i < numOps; i++)
    {
        if( index < m * (p + q))
        {
            if(index % (p + q) < p)
            {
                C[index] = A[p * (index / (p + q)) + index % (p + q)];
                //printf("HEllo  index : %d, A[%d] :)\n", index, p * (index / (p + q)) + index % (p + q) );
            }
            else
            {
                C[index] = B[q * (index / (p + q)) + index % (p + q) - p];
                //printf("HEllo  index : %d, B[%d] :)\n", index, q * (index / (p + q)) + index % (p + q) - p);
            }
            index += offset;
        }
    }
}
```

Fig. 1. HW1 gpu code

Fig. 1의 kernel code에서 어떤 비효율성이 있는지를 확인해보자.

어떤 것이 GPU 사용의 비효율성을 초래했는지를 확인할 수 있다. 가장 먼저, branch inefficiency 문제가 있는지를 확인해보자. %를 활용한 분기를 사용했으므로, if문을 잘 확인해보아야 한다. 32개의 쓰레드가 같이 동작하는 1개의 warp단위로 같은 명령어를 처리해야만 control divergence문제를 피할 수 있다. 우리의 code에서는 정해진동작( 주어진 Matrix의 크기를 사용하고, 32의 배수의 threads를 사용할 때) branch inefficiency가 발생하지 않는다. 왜냐하면 행렬의 가로에 해당되는 부분이 32의 배수이므로(1024와 512는 32의 배수이다) 분기에서 항상 같은 명령어를 처리하게된다.

```
cudaMalloc((void **)&device_matrixA, sizeA);
cudaMalloc((void **)&device_matrixB, sizeB);
cudaMalloc((void **)&device_matrixC, sizeC);
```

Fig. 2. HW1 gpu code_cudaMalloc

다음은 memory부분에서 inefficiency가 있는지 확인해본다. Fig.2에서 볼 수 있듯이, device에 메모리를 할당할 때, cudaMalloc을 사용하여 메모리를 device의 global memory영역에 할당하였는데, 커널함수에서 글로벌 메모리에 접근할 때, 많은 cycle이 필요하다. 그런데 이 concat 프로그램에서는 global 메모리에 있는 데이터 값을 한번만 로드하므로 오히려 shared

memory를 거쳐서 output으로 옮기기 보다는 shared memory를 사용하지 않는 것이 더 효율성이 좋다. 따라서 shared memory는 사용하지 않는다.

```
//concat in GPU
device_matrix_concat<<<block,thread>>>(device_matrixC, device_matrixA, device_matrixB, numOps, block, thread, M, P, Q);
```

위의 사진에서 볼 수 있듯이, numOps, m, p, q를 인자로 받아서 사용하게 되는데, 이때, 스레드별로 앞의 인자를 레지스터에 저장해야하므로 레지스터의 사용을 절약할 수 있음에도 사용하였으므로 효율이 상대적으로 떨어진다.(레지스터의 사용을 절약하면 액티브 스레드의 개수가 늘어나 프로그램의 효율성이 증가한다) 이것은 상수메모리를 통해 읽기 전용으로 반복해서 사용함으로써 효율성을 증가시킬 수 있다.

이제 nvprof를 통해 확인해보자. 그것은 다음과 같다.

```
==23534== Profiling application: ./gpu_time 512 512 1 512 1024 512
==23534== Profiling result:
==23534== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*, int, int, int, int, int, int)
          1                  warp_execution_efficiency                 Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                      achieved_occupancy                      Achieved Occupancy    0.855297    0.855297    0.855297
          1                       branch_efficiency                       Branch Efficiency      100.00%     100.00%     100.00%
          1                          gld_throughput                   Global Load Throughput  389.65MB/s  389.65MB/s  389.65MB/s
          1                        gld_transactions                 Global Load Transactions      393218      393218      393218
          1                          gst_throughput                  Global Store Throughput  389.65MB/s  389.65MB/s  389.65MB/s
          1                        gst_transactions                 Global Store Transactions       98304       98304       98304
```

우선, 예상한대로 branch_effieiency와 warp_execution_effieiency는 100%로 최고의 효율을 띄고있음을 확인할 수 있다. 이것은 앞서 말했듯이, 32개 thread단위로 warp가 작동하는데 우리가 사용하는 문제에서는 32의 배수의 thread를 사용하고, 코드가 32의 배수 단위로 if가 나눠지므로 각 warp가 하나의 명령어를 실행하므로 100%의 효율을 띄는 것은 당연하다.
Achieved occupancy에 대해서는 다음과 같다.
우리는 cuda 코어의 수가 128개이고, 1개의 SM안에 2048개의 thread가 가능하므로, thread는 512일 때, block은 512개를 생성했을 때, 가장 효율적으로 모든 SM을 사용하는 것이고, SM안에 모든 warp를 활성화시키는 것이다. 그런데 그 값이 0.856224가 나왔음을 확인할 수 있다.

### 2) Optimize your CUDA code based on what you studied until week 6

상수 메모리를 할당하여, kernel 함수의 인자로 받지 않고, 데이터를 사용하였고, global memory에 있는 data를 align하게 추출하고, 한 cycle에서 최대한 같은 명령어를 수행하게 설정하기 위해 다음과 같이 코드를 작성하였다.

```
__constant__ int cdata[5];
```

__constant__를이용하여 인자 값 전달하여 효율성을 향상시켰으며, global 메모리에 접근하는 방식을 다르게 하여 memory coalescing을 maximize하였다.

```
int index2 = blockDim.x * blockIdx.x + threadIdx.x;
int offset = gridDim.x * blockDim.x;

for (int i = 0 ; i < cdata[0]; i++)
{
    if(index2 < cdata[4])
    {
        C[(index2 / cdata[2]) * (cdata[2] + cdata[3]) + (index2 % cdata[2])] = A[index2];
    }
    else
    {
        C[((index2 - cdata[4]) / cdata[3]) * (cdata[2] + cdata[3]) + (cdata[2]+ (index2 - cdata[4]) % cdata[3])] = B[index2-cdata[4]];
    }
    index2 += offset;
}
```

Nvprof의 결과는 다음과 같이 나왔다. Branch와 warp efficiency는100%로 그대로이고, achieved_occupancy가 약간 향상된 것을 확인할 수 있다. 또한 global load/store throughput이 향상된 것을 확인할 수 있다.

```
==24415== Profiling application: ./gpu_HW2 512 512 1 512 1024 512
==24415== Profiling result:
==24415== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                       achieved_occupancy                     Achieved Occupancy    0.873748    0.873748    0.873748
          1                        branch_efficiency                      Branch Efficiency      100.00%     100.00%     100.00%
          1                           gld_throughput                 Global Load Throughput  1.2732GB/s  1.2732GB/s  1.2732GB/s
          1                         gld_transactions               Global Load Transactions  2.1319e+11  2.1319e+11  2.1319e+11
          1                           gst_throughput                Global Store Throughput   8e+04GB/s   8e+04GB/s   8e+04GB/s
          1                         gst_transactions              Global Store Transactions  6270361604  6270361604  6270361604
```

### 3) With your new code, measure the GPU execution time and profile the metrics that you picked. And then compare the experimental results to those in HW1

이제, 새로운 코드를 토대로 GPU execution time을 구하면 다음과 같다.
CPU의 execution time은 12.5275ms였었다. GPU의 execution time과 metrics profile을 실행하면, 다음과 같다.

SM은 128개, SM당 최대 쓰레드의 개수는 2048이므로, block은 128 x 2048 / thread로 결정하였다.

1. Block: 8192 Thread : 32

```
Average ElapsedTime = 6.980558 ms
==24182== Profiling application: ./gpu_HW2 8192 32 1 512 1024 512
==24182== Profiling result:
==24182== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                       achieved_occupancy                     Achieved Occupancy    0.481063    0.481063    0.481063
          1                        branch_efficiency                      Branch Efficiency      100.00%     100.00%     100.00%
          1                           gld_throughput                 Global Load Throughput  430.70MB/s  430.70MB/s  430.70MB/s
          1                         gld_transactions               Global Load Transactions      393218      393218      393218
          1                           gst_throughput                Global Store Throughput  430.70MB/s  430.70MB/s  430.70MB/s
          1                         gst_transactions              Global Store Transactions       98304       98304       98304
```

2. Block : 4096 Thread : 64

```
Average ElapsedTime = 4.922554 ms
==24191== Profiling application: ./gpu_HW2 4096 64 1 512 1024 512
==24191== Profiling result:
==24191== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                       achieved_occupancy                     Achieved Occupancy    0.943866    0.943866    0.943866
          1                        branch_efficiency                      Branch Efficiency      100.00%     100.00%     100.00%
          1                           gld_throughput                 Global Load Throughput  441.28MB/s  441.28MB/s  441.28MB/s
          1                         gld_transactions               Global Load Transactions      393218      393218      393218
          1                           gst_throughput                Global Store Throughput  441.28MB/s  441.28MB/s  441.28MB/s
          1                         gst_transactions              Global Store Transactions       98304       98304       98304
```

3. Block : 2048 Thread : 128

```
Average ElapsedTime = 6.827844 ms
```

```
==24203== Profiling application: ./gpu_HW2 2048 128 1 512 1024 512
==24203== Profiling result:
==24203== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                       achieved_occupancy                    Achieved Occupancy     0.931071    0.931071    0.931071
          1                        branch_efficiency                     Branch Efficiency      100.00%     100.00%     100.00%
          1                           gld_throughput                Global Load Throughput   441.10MB/s  441.10MB/s  441.10MB/s
          1                         gld_transactions              Global Load Transactions       393218      393218      393218
          1                           gst_throughput               Global Store Throughput   441.10MB/s  441.10MB/s  441.10MB/s
          1                         gst_transactions              Global Store Transactions       98304       98304       98304
```

4. Block : 1024 Thread : 256

```
Average ElapsedTime = 4.969820 ms
```

```
==24223== Profiling application: ./gpu_HW2 1024 256 1 512 1024 512
==24223== Profiling result:
==24223== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                       achieved_occupancy                    Achieved Occupancy     0.887953    0.887953    0.887953
          1                        branch_efficiency                     Branch Efficiency      100.00%     100.00%     100.00%
          1                           gld_throughput                Global Load Throughput   876.64MB/s  876.64MB/s  876.64MB/s
          1                         gld_transactions              Global Load Transactions  2.1319e+11  2.1319e+11  2.1319e+11
          1                           gst_throughput               Global Store Throughput     5e+04GB/s   5e+04GB/s   5e+04GB/s
          1                         gst_transactions              Global Store Transactions  6270361604  6270361604  6270361604
```

5. Block : 512 Thread : 512

```
Average ElapsedTime = 3.446301 ms
```

```
==24415== Profiling application: ./gpu_HW2 512 512 1 512 1024 512
==24415== Profiling result:
==24415== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                       achieved_occupancy                    Achieved Occupancy     0.873748    0.873748    0.873748
          1                        branch_efficiency                     Branch Efficiency      100.00%     100.00%     100.00%
          1                           gld_throughput                Global Load Throughput  1.2732GB/s  1.2732GB/s  1.2732GB/s
          1                         gld_transactions              Global Load Transactions  2.1319e+11  2.1319e+11  2.1319e+11
          1                           gst_throughput               Global Store Throughput     8e+04GB/s   8e+04GB/s   8e+04GB/s
          1                         gst_transactions              Global Store Transactions  6270361604  6270361604  6270361604
```

6. Block : 256 Thread : 1024

```
Average ElapsedTime = 5.747744 ms
```

```
==24238== Profiling application: ./gpu_HW2 256 1024 1 512 1024 512
==24238== Profiling result:
==24238== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: device_matrix_concat(int*, int*, int*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      100.00%     100.00%     100.00%
          1                       achieved_occupancy                    Achieved Occupancy     0.836453    0.836453    0.836453
          1                        branch_efficiency                     Branch Efficiency      100.00%     100.00%     100.00%
          1                           gld_throughput                Global Load Throughput   434.85MB/s  434.85MB/s  434.85MB/s
          1                         gld_transactions              Global Load Transactions  2.1319e+11  2.1319e+11  2.1319e+11
          1                           gst_throughput               Global Store Throughput     3e+04GB/s   3e+04GB/s   3e+04GB/s
          1                         gst_transactions              Global Store Transactions  6270361604  6270361604  6270361604
```

표로 정리하면 다음과 같다.

| block | thread | GPU_EXECUTION | SPEEDUP |
|---|---|---|---|
| 8192 | 32 | 6.980558 | 1.794627 |
| 4096 | 64 | 4.922554 | 2.544919 |
| 2048 | 128 | 6.827844 | 1.834767 |
| 1024 | 256 | 4.96982 | 2.520715 |
| 512 | 512 | 3.446301 | 3.635057 |
| 256 | 1024 | 5.747744 | 2.179551 |

표에서 확인할 수 있듯이, 512 512일 때 speed up이 3.6으로 가장 빠르게 나왔다. HW1에서 가장 빠른 speed up의 수치가 2.59였다는 점을 생각하면, 좋은 향상을 이루었음을 확인할 수 있다.

# 2. Finding a Maximum Value in a Large Vector

## 1) Implement a single-threaded program for CPU.

다음 사진은 CPU만을 사용하여 문제를 해결하는 code이다. Max를 찾기 위해, index 0번부터 차례로 무엇이 큰지 차례로 len만큼 비교한 후, 가장 큰 값을 구조체 포인터로 받은 구조체의 멤버에 전달한다. 그 값은 가장 큰 요소의 값(max_value)과 그 위치인 index이다.

```
void find_max(struct Element_info * max_element, const float* vector_arr, int len)
{
    float max_value = vector_arr[0];
    int max_index = 0;


    for (int i = 0; i < len; i++)
    {
        if (max_value < vector_arr[i])
        {
            max_value = vector_arr[i];
            max_index = i;
        }
    }

    max_element->value = max_value;
    max_element->index = max_index;

}
```

그 결과는 다음과 같다.

```
user14@lab14:~/cuda/Lim_Yong/HW2/HW2_2/CPU_HW2$ ./cpu_HW2 10
--------------------------------------------------------------------------------
CPU Execution Time Estimatation START
Input  vector lenghth : 100000000
Number of Test : 10
 Test : 1 , Elapsed Time :  586.315000 ms
 Test : 2 , Elapsed Time :  591.352000 ms
 Test : 3 , Elapsed Time :  594.218000 ms
 Test : 4 , Elapsed Time :  593.035000 ms
 Test : 5 , Elapsed Time :  593.087000 ms
 Test : 6 , Elapsed Time :  592.991000 ms
 Test : 7 , Elapsed Time :  594.208000 ms
 Test : 8 , Elapsed Time :  593.174000 ms
 Test : 9 , Elapsed Time :  586.291000 ms
 Test : 10 , Elapsed Time :  586.347000 ms
--------------------------------------------------------------------------------

 Average Elapsed Time : 591.101800 ms
--------------------------------------------------------------------------------
The Largest elements in a vector : 100.000000  and index : 43254
--------------------------------------------------------------------------------
```

Test를 몇 번 할지는 argv[]를 이용하여 받았다. 여기서 100,000,000개의 floating point-type element는 index=43254를 제외하고 모두 10으로 초기화하였고, index=43254일때는 100으로 초기화하였다.

따라서 위의 사진에서 볼 수 있듯이, 벡터안에 가장 큰 요소의 값은 100이고, 그 index는 43254임을 확인할 수 있다. 즉, 잘 동작함을 알 수 있다. 평균적으로 측정된 시간을 확인해보면, 591.1018ms 가 걸렸음을 확인할 수 있다.

**Average CPU Elapsed Time = 591.101800ms**

## 2) Implement a CUDA program, aiming at minimizing branch divergence and optimizing memory accesses. Include analysis of your own regarding

Branch divergence를 최소화하고, 메모리 접근을 최적화하기위해(공유 메모리 사용, 상수메모리 사용, memory coalescing 극대화)다음과 같이 코드를 작성하였다.

### A. Are there any possibilities to have branch divergence? How do you resolve it?

Branch divergence의 가능성을 줄이기 위해서, 가능한한 %연산을 사용하지 않았고, 각 warp가 최대한 하나의 명령어를 실행하게 설계하였다.(warp의 thread가 다른 control flow를 타지 않게 하였다)

### B. Are there any optimizations for better memory accesses? How do you apply them?

메모리 접근을 최적화 하기 위해서, 사용하는 vector 을 재사용하여 다시 쓴다는 점을 주목하였다. 그래서, 블록 내에서 공유메모리를 사용하여, 각 블록 내에서 최대의 값을 찾게 병렬적으로 설계하였고(공유메모리의 값을 로드하고, 최대값을 찾고, 다시 공유메모리값에 store하고를 반복하여 블록내에서 최대값을 찾는다), 그렇게 도출된 블록 내의 최대 값을 글로벌 메모리에 있는 vector에 차례로 순서대로 store하였다. (이는 memory coalescing을 극대화 하기 위해서, 인접한 쓰레드는 이웃 메모리 주소에 접근해야하기 위해서이다.) 하지만, block을 전부 synchronize하는 기능이 kernel함수내에 없으므로, 커널함수를 여러 번 실행하면서 block을 synchronize하여, bank 충돌을 해결하였다. 코드는 다음과 같다.

```
119         while(numElements != 1)
120         {
121             numOps = (numElements % (block * thread) == 0)? numElements / (block * thread) : numElements / (block * thread) + 1 ;
122             //Find MAX Element
123             find_max_element<<<block,thread, sizeof(float) * thread>>>(dvector_arr, numOps);
124 
125             numElements = (numElements % thread == 0)? numElements / thread : numElements / thread + 1;
126         }
```

```
23  __global__
24  void find_max_element(float * A, int numOps)
25  {
26      extern __shared__ float sMem[];
27      //load shared MEM
28      int tid = threadIdx.x;
29      int bid = blockIdx.x;
30      int index = blockIdx.x * blockDim.x + threadIdx.x; // index :  0 ~ 1억
31      int offset = gridDim.x * blockDim.x;               // block x thread
32 
33      for( int i = 0 ; i < numOps; i ++)
34      {
35          if(index + (offset) * i < LENGTH)
36          {
37              sMem[tid] = A[index + (offset) * i];
38              __syncthreads();
39              for(int s=blockDim.x/2; s>0; s/=2)
40              {
41                  if (tid < s)
42                  {
43                      if(sMem[tid] < sMem[tid + s])
44                      {
45                          sMem[tid] = sMem[tid + s];
46                      }
47                  }
48                  __syncthreads();
49              }
50              if(tid == 0)
51              {
52                  A[bid + gridDim.x * i] = sMem[tid];
53              }
54              __syncthreads();
55          }
56      }
57  }
```

디바이스쿼리에 따르면, Jetson nano는 128개의 cuda core를 갖고있고, 멀티프로세서당 최대 쓰레드의 개수는 2048개이다. 공유메모리는 블록 내에서만 서로 store load할 수 있고, 하나의 SM에는 49152 bytes가 최대 공유메모리이므로( 블록당 최대 공유메모리는 49152 byte) 다음과 같이 코딩하였다. 공유메모리의 크기는 동적으로 할당하여 sizeof(float)*thread만큼 공간을 설정하였다. 이제 이를 확인하기 위해 nvprof를 사용하여 branch divergence를 최소화하고, 메모리 접근의 효율을 띄는지 확인한다.

우선 nvprof로 block = 32 thread = 512를 했을 때의 결과이다. 그것은 다음과 같다.

```
==18795== Profiling application: ./gpu_HW2 1 32 512
==18795== Profiling result:
==18795== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA Tegra X1 (0)"
    Kernel: find_max_element(float*, int)
          3                  warp_execution_efficiency              Warp Execution Efficiency      98.74%      98.87%      98.79%
          3                        branch_efficiency                   Branch Efficiency     100.00%     100.00%     100.00%
          3                       achieved_occupancy                  Achieved Occupancy    0.939093    0.977785    0.956160
          3                           gld_throughput                Global Load Throughput  1.3527GB/s  1.6059GB/s  1.6058GB/s
          3                           gst_throughput               Global Store Throughput  21.643MB/s  25.695MB/s  25.693MB/s
```

여기서 볼 수 있듯이, branch_efficiency는 100%로 branch divergence를 줄였음은 분명하다. Shared memory를 사용하였으므로, 메모리 접근을 최적화 한것도 맞다. Througput을 보면 load는 1.6GB/s 이지만 store은 25.693MB/s임을 볼 수 있는데, 이는 코드에서 store을 할 때, tid ==0 일때에 대해서만 한 것을 보면 이해가 되는 부분이다. Achieved_occupancy = 0.956으로 잘 나왔음을 확인할 수 있고, warp_execution_efficiency는 vector length가 2의 거듭제곱꼴이 아닌 100,000,000개이므로 저정도는 어쩔 수 없는 것이다.(나누어떨어지지않으므로)

### 3) Measure the execution time with the CPU code and the GPU code. Find the speedup of your CUDA code over the code for CPU.

이제 execution time 을 비교하면 다음과 같다. 우선, CPU 의 execution time 은 591.1018ms 이다. 이제 gpu 의 execution time 을 구하고 speed up 을 구하면 다음과 같다.

1. Block : 512 Thread : 32

```
Average ElapsedTime = 399.943481 ms
```

2. Block : 256 Thread : 64

```
Average ElapsedTime = 466.217804 ms
```

3. Block : 128 Thread : 128

```
Average ElapsedTime = 334.428864 ms
```

4. Block : 64 Thread : 256

```
Average ElapsedTime = 344.118744 ms
```

5. Block : 32 Thread : 512

```
Average ElapsedTime = 366.122925 ms
```

6. Block : 16 Thread : 1024

```
Average ElapsedTime = 382.935791 ms
```

표로 정리하면 다음과 같다.

| block | thread | gpu_execution time | speedup |
|---|---|---|---|
| 512 | 32 | 399.943481 | 1.477963 |
| 256 | 64 | 466.217804 | 1.267866 |
| 128 | 128 | 334.428864 | 1.767496 |
| 64 | 256 | 344.118744 | 1.717726 |
| 32 | 512 | 366.122925 | 1.61449 |
| 16 | 1024 | 382.935791 | 1.543606 |

위에 6 개의 예시에서 block = 128 thread = 128 일 때 가장 동작이 빨랐다.

하지만 이론적으로 thread 를 1024 개로 하였을 때가 병렬처리가 가장 효율적이므로 thread 가 1024 일 때 가장 speedup 이 빠를것으로 예상되지만 jetson nano 는 1 개의 SM 과 최대 2048 개의 쓰레드를 하나의 멀티프로세서에서 처리하므로 그렇지만은 않다는 것을 알 수 있었다.

부가적으로 thread 를 고정시키고 블록 수를 다르게 하였을 때, 비교해보면 다음과 같다.

| block | thread | gpu_execution time | speed up |
|---|---|---|---|
| 64 | 32 | 365.184998 | 1.618637 |
| 32 | 64 | 342.46051 | 1.726044 |
| 16 | 128 | 320.65979 | 1.843392 |
| 8 | 256 | 467.23337 | 1.26511 |
| 4 | 512 | 352.570587 | 1.676549 |
| 2 | 1024 | 443.5647 | 1.332617 |

Block = 16 이고 thread 가 128 일 때 가장 speed up 이 높음을 확인할 수 있다