CUDA Lab 3. CUDA memory management

1. Understand how to allocate shared memory
2. Understand how to allocate managed memory
3. Understand where shared memory can be used to enhance the performance of a parallel program
4. Learn how to find the subtotal of the set of numbers processed in a thread block for a 1D array of numbers using shared memory

Exercise 1. Vector dot-product

For *n*-dimensional vectors **a**=($a_0, a_1, a_2, \ldots, a_{n-1}$) and **b**=($b_0, b_1, b_2, \ldots, b_{n-1}$), their dot-product is defined as

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_0 b_0 + a_1 b_1 + a_2 b_2 + \cdots\cdots + a_{n-1} b_{n-1}$$

(1). **CPU only solution**.

Write a C++ program to calculate the dot-product of two vectors used in the template CUDA program created in VS2019.

(2). **CPU + GPU solution**.

The dot-product of vectors **a**=($a_0, a_1, a_2, \ldots, a_{n-1}$) and **b**=($b_0, b_1, b_2, \ldots, b_{n-1}$), can be found in two steps:

Step 1. Per-element multiplication: In this step, we calculate a vector **c**:

$$\boldsymbol{c} = (a_0 b_0, \quad a_1 b_1, \quad a_2 b_2, \quad \cdots, \quad a_{n-1} b_{n-1})$$

This task can be done in parallel on GPU.

Step 2. Calculate on CPU the sum of elements of vector **c** found in step 1.
Write a CUDA program to accelerate the calculation of the dot-product by doing the per-element multiplication on the GPU.

Exercise 2. Vector dot-product using unified memory

Unified Memory is a kind of managed memory space in which all processors see a single coherent memory image with a common address space. Data access to unified memory is entirely managed by the underlying system, without any need for explicit memory copy calls. Unified Memory simplifies the GPU programming and enables the writing of simpler and more maintainable code.

Exercise 2.1 Vector dot-product using managed memory

1) Create a CUDA program using the template CUDA program created in VS2019.
2) Write a new kernel function

```
__global__ void PerElement_AtimesB(int *c, int *a, int  *b) {

   c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

}
```

3) In the main( ) function, allocate managed memories using cudaMallocManaged( ) for vectors a, b, c and initialize their values.

```
int main() {

    int *a, *b, *c;

    int arraySize = 5;

    cudaMallocManaged(&a, arraySize * sizeof(int));

    cudaMallocManaged(&b, arraySize * sizeof(int));

    cudaMallocManaged(&c, arraySize * sizeof(int));

     // initialize arrays a, b, c:

     ... ...

    //calculate per-element vector multiplication on GPU:

    PerElement_AtimesB <<< 1, arraySize >>>(c, a, b);

    cudaDeviceSynchronize();

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",

            c[0], c[1], c[2], c[3], c[4]);

    cudaFree(a);

    cudaFree(b);

    cudaFree(c);

    return 0;

}
```

Notice that the use of the cudaMallocManaged() routine, which returns a pointer valid from both host and device code. So, there is no need to copy data between host and device, greatly simplifying and reducing the size of the program.

4) Extend the sample code shown above to calculate the dot-product of the two vectors.

Exercise 2.3 Vector dot-product using GPU-declared __managed__ memory

The above program can be further simplified by using GPU-declared __managed__ memory

1) Starting from the default CUDA program created in VS2019.
2) Allocate managed memory for vector a[ ], b[ ] and c[ ] using device-declared __managed__ memory. For example,

```
__device__ __managed__ int a[5], b[5], c[5];
```

3) Same as above, write a kernel function to calculate element-wise vector multiplication:

```
__global__ void PerElement_AtimesB(int *c, int *a, int  *b) {

   c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

}
```

4) In the main( ) function, initialize a[ ] and b[ ] and calculate c[ ]=a[ ]+ b[ ] on GPU in parallel.

```
            for (int i=0; i<5; i++) {

                        a[i]=i;

                        b[i]=i*10;

            }

            PerElement_AtimesB <<< 1, 5 >>>(c, a, b);

            cudaDeviceSynchronize();

            printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",

            c[0], c[1], c[2], c[3], c[4]);

            ... ...

            ... ...
```

Note that in this example implementation of the element-wise vector multiplication, cudaMallocManaged( ) is not used, which further simplifies the code.

5) Extend the sample code to calculate the dot-product of the two vectors.

Exercise 3. Vector dot-product using shared Memory

You have attempted the problem of computing vector dot-product by performing per-element multiplication in parallel in GPU and then copy back the result to CPU and let CPU to find the sum of the array of numbers produced by the parallel process. There are various ways to improve the efficiency of calculating the dot-product by asking GPU to do some more work. Let’s consider two vectors **a** and **b** of 8 elements:

a=	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
b=	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7

c=	$a_0 \times b_0$	$a_1 \times b_1$	$a_2 \times b_2$	$a_3 \times b_3$	$a_4 \times b_4$	$a_5 \times b_5$	$a_6 \times b_6$	$a_7 \times b_7$

Instead of directly copying the result vector **c** back to CPU, we can let GPU to do some more work to improve the efficiency of computing the dot-product of the two vectors. For instance, if you execute the kernel with thread configuration <<<2, 4>>>, using TWO thread blocks with FOUR threads in each block, we can divide the data **c** into two blocks of data and let GPU to find a subtotal for each data block and then only ask CPU to calculate the sum of the two subtotals. That is, we only copy back the two subtotals to CPU to find the dot-product of the two vectors. One way of achieving this is with the use of shared memory. For example, we can rewrite our code in the following way:

- We divide the data into two sub-datasets, with one being copied to the shared memory for the first thread block and the other copied to the shared memory for the second thread block. In this example, we have 8 data elements and two thread blocks, we can create an array in shared memory to store just four elements. (This can be declared inside or outside of the kernel):

  ```
  __shared__ int dataPerBlock[4];
  ```

- To distribute different sub-datasets of data to different thread blocks, we need to identify each data element of **c**[ ] globally. We can get this by looking at the global ID for each thread, which is:

  ```
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  ```

- Then we copy the part of data c[ ] to be processed by a block to dataPerBlock[ ]. In this example, as the size of shared data is the same as the size of block, this can be easily done as:

  ```
  dataPerBlock[threadIdx.x] = c[i];
  ```

- Find the sum of all the elements in `dataPerBlock`[ ] and store the subtotal calculated in each thread block in array c with index blockIdx.x:

  ```
  float subtotal = 0;

  for (int k = 0; k < blockDim.x; k++)
         subtotal += dataPerBlock[k];

  c[blockIdx.x] = subtotal;
  ```

- Copy c[0], c[1] back to host to let host calculate c[0]+c[1] to get the dot-product of the two vectors.

Shared memory can be used to optimize CUDA code in many ways, as the access to shared memory is much faster than the access to the global memory because it is located on chip.

Task 1. Threads synchronization.

Analyse the above process and identify areas where thread execution needs to be synchronized by calling CUDA function: __syncthreads();

Task 2. Consider different thread configurations, for example, <<<1, 8>>>, <<<2, 4>>>, <<<4, 2>>> and observe if the above program can calculate the vector dot-product correctly. If not, analyse the issues and consider how to fix them.