600086 Lab Book

Week 3 – Lab C

Date: 22nd Feb 2022

E1. Vector dot-product

Question:

> (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.

Solution:

```
int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c = 0;

    printf("array A = {1,2,3,4,5}\n");
    printf("array B = {10,20,30,40,50}\n");

    for (int i = 0; i < arraySize; i++)
    {
        c += a[i] * b[i];
    }

    printf("the dot product of arrays A and B is: %d", c);
```

Test data:

A	1	2	3	4	5
B	10	20	30	40	50

Sample output:

```
array A = {1,2,3,4,5}
array B = {10,20,30,40,50}
the dot product of arrays A and B is: 550
```

Question:

(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.

Solution:

```
__global__ void addKernel(int *c, int *a, int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] * b[i];
}
```

```
const int size = 12;
int a[size] = { 0 };
int b[size] = { 0 };
int c[size] = { 0 };
int d = 0;

for (int i = 0; i < size; i++)
{
    a[i] = i + 1;
    b[i] = (i + 1) * 10;
}
```

```
int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;
    addKernel << <1, size >> > (dev_c, dev_a, dev_b);

    cudaDeviceSynchronize();
    cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    printf("Array A = {%d",a[0]);
    for (int i = 1; i < size; i++){
        printf(",%d", a[i]);
    }
    printf("}\n");
    printf("Array B = {%d",b[0]);
    for (int i = 1; i < size; i++){
        printf(",%d", b[i]);
    }
    printf("}\n");
    printf("Array C = {%d", c[0]);
    for (int i = 1; i < size; i++) {
        printf(",%d", c[i]);
    }
    printf("}\n");
    for (int i = 0; i < size; i++){
        d += c[i];
    }
    printf("the dot product of A & B is: %d",d);
```

Test data:

A	1	2	3	4	5	6	7	8	9	10	11	12
B	10	20	30	40	50	60	70	80	90	100	110	120

Sample output:

```
Array A = {1,2,3,4,5,6,7,8,9,10,11,12}
Array B = {10,20,30,40,50,60,70,80,90,100,110,120}
Array C = {10,40,90,160,250,360,490,640,810,1000,1210,1440}
the dot product of A & B is: 6500
```

Reflection:

This is a very quick technique for calculating the dot product when increasing the lengths of the arrays used in the C++ method the completion time increases greatly whereas with the GPU solution the lengths of the test data array can be increased with almost no difference to the execution times.

Metadata:

“Dot-Product”,”thread”,”c++”,”GPU”

Further information:

E2. Vector dot-product using unified memory

Question:

Exercise 2.1 Vector dot-product using managed memory

Solution:

```
__global__ void PerElement_AtimesB(int *c, int *a, int *b)
{
    c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
}
```

```
int main()
{
    cudaError_t cudaStatus;
    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    const int size = 5;
    int* c, * a, * b;
    cudaMallocManaged(&a, size * sizeof(int));
    cudaMallocManaged(&b, size * sizeof(int));
    cudaMallocManaged(&c, size * sizeof(int));
    for (int i = 0; i < size; i++)
    {
        a[i] = i + 1;
        b[i] = (i + 1) * 10;
        c[i] = 0;
    }
    PerElement_AtimesB << <1, size >> > (c, a, b);
    cudaDeviceSynchronize();
    printf("A is :{ %d",a[0]);
    for(int i = 1;i<size;i++){
        printf(",%d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < size; i++){
        printf(",%d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < size; i++){
        printf(",%d", c[i]);
    }
    printf(" }\n");
    int d = c[0] + c[1] + c[2] + c[3] + c[4];
    printf("%d",d);
    cudaFree(c);
    cudaFree(a);
    cudaFree(b);
    cudaDeviceReset();
    return 0;
}
```

Test data:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
```

Sample output:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
C is :{ 10,40,90,160,250 }
550
```

Question:

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

Solution:

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

__global__ void PerElement_AtimesB(int* c, int* a, int* b)
{
    c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
}
int main()
{
    cudaSetDevice(0);

    for (int i = 0; i < 5; i++)
    {
        a[i] = i+1;
        b[i] = (i+1) * 10;
    }

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

    cudaDeviceSynchronize();

    printf("A is :{ %d", a[0]);
    for (int i = 1; i < 5; i++) {
        printf(",%d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < 5; i++) {
        printf(",%d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < 5; i++) {
        printf(",%d", c[i]);
    }
    printf(" }\n");
    int d = c[0] + c[1] + c[2] + c[3] + c[4];
    printf("dot product of A & B is: %d", d);

    cudaDeviceReset();
    return 0;
}
```

Test data:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
```

Sample output:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
C is :{ 10,40,90,160,250 }
dot product of A & B is: 550
```

Reflection:

Declaring a shared variable makes the code much neater and easier to read I struggled getting cudaMallocManaaged() to work as I had expected it however the declared shared memory worked immediately and felt much more intuitive. Added the for loops on the print statements sin each section as I had built them with extension in mind so the size characteristic can be adjusted to suit larger vectors and the code will run without fault limited purely by the thread limit.

Metadata:

“Dot-Product”,”thread”,”c++”,”GPU”,”declared”,”managed”

Further information:

E3. Vector dot-product using shared Memory

Question:

Task 1. Threads synchronization.
Analyse the above process and identify areas where thread execution needs to be synchronized by calling CUDA function: __syncthreads();

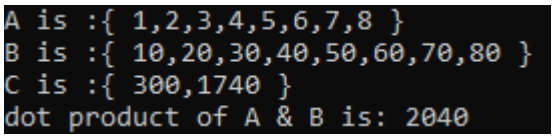
Solution:

```
__device__ __managed__ int a[8], b[8], c[8];

__shared__ int dataPerBlock[4];

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

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

    c[i] = a[i] * b[i];
    __syncthreads();
    dataPerBlock[threadIdx.x] = c[i];

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

    c[blockIdx.x] = subtotal;
}
int main()
{
    cudaSetDevice(0);

    for (int i = 0; i < 8; i++)
    {
        a[i] = i + 1;
        b[i] = (i + 1) * 10;
    }

    PerElement_AtimesB << <2, 4 >> > (c, a, b);

    cudaDeviceSynchronize();

    printf("A is :{ %d", a[0]);
    for (int i = 1; i < 8; i++) {
        printf(",%d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < 8; i++) {
        printf(",%d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < 2; i++) {
        printf(",%d", c[i]);
    }
    printf(" }\n");
    int d = c[0] + c[1];
    printf("dot product of A & B is: %d", d);

    cudaDeviceReset();
    return 0;
}
```

Test data:

A	1	2	3	4	5	6	7	8
B	10	20	30	40	50	60	70	80

Sample output:



Reflection:

The __sync thread function has been added into the kernel prior to the subtotal calculation this is to ensure all active threads reach this point before collapsing the values this is because a thread may still require access to c[0] and c[1] which are overwritten as a result of this subtotal calculation.

Question:

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.

Solution:

```
const int ArrayLength = 8;
const int threadsPerBlock = 2;
const int blocks = 4;

__device__ __managed__ int a[ArrayLength], b[ArrayLength], c[ArrayLength];
__shared__ int dataPerBlock[threadsPerBlock];

__global__ void PerElement_AtimesB(int* c, int* a, int* b)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] * b[i];
    __syncthreads();
    dataPerBlock[threadIdx.x] = c[i];
    int subtotal = 0;
    for (int k = 0; k < blockDim.x; k++)
    {
        subtotal += dataPerBlock[k];
    }
    printf("subtotal is: %d", subtotal);
    c[blockIdx.x] = subtotal;
}
int main()
{
    cudaSetDevice(0);

    for (int i = 0; i < ArrayLength; i++)
    {
        a[i] = i + 1;
        b[i] = (i + 1) * 10;
    }

    PerElement_AtimesB << <blocks, threadsPerBlock >> > (c, a, b);

    cudaDeviceSynchronize();

    printf("A is :{ %d", a[0]);
    for (int i = 1; i < ArrayLength; i++) {
        printf(", %d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < ArrayLength; i++) {
        printf(", %d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < blocks; i++) {
        printf(", %d", c[i]);
    }
    printf(" }\n");
    int d = 0;
    for (int i = 0;i<blocks;i++){
        d += c[i];
    }
    printf("dot product of A & B is: %d", d);

    cudaDeviceReset();
    return 0;
}
```

Test data:

Sample output:

Reflection:

In its initial state the program cannot compile as it only handles up to 2 return blocks to change the code so that it can handle any combination of threads and blocks then adjustments need to be made to the shared variable this is done by adding a set of variables to define these at the head of the solution adjusting these allows for any size of array the only restriction is that the blocks*threadsPerBlock must be greater than the ArrayLength variable

Metadata:

“Dot-Product”,”thread”,”c++”,”GPU”,”declared”,”managed”

Further information:

N/A