600086 Lab Book

Week 2 – Lab B

Date: 17th Feb 2022

Callum Gray

Q1. Understand the block and thread indices

Question:

List the values for the built-in variables **threadIdx.x** and **blockIdx.x** corresponding to the following thread configurations used for executing the kernel *addKernel*( ) function on GPU:

1) addKernel << <1, 5 >> > (dev_c, dev_a, dev_b);
2) addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);
3) addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);
4) addKernel << <6, 1 >> > (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : %d Block: ID: %d", threadIdx.x, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out .

```
addKernel << <1, 5 >> > (dev_c, dev_a, dev_b);
addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);
addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);
addKernel << <6, 1 >> > (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("{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]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output

`addKernel <<<1, 5 >>> (dev_c, dev_a, dev_b);`	`Thread ID : 0 Block: ID: 0`
	`Thread ID : 1 Block: ID: 0`
	`Thread ID : 2 Block: ID: 0`
	`Thread ID : 3 Block: ID: 0`
	`Thread ID : 4 Block: ID: 0`

`addKernel <<<2, 3 >>> (dev_c, dev_a, dev_b);`	`Thread ID : 0 Block: ID: 0`
	`Thread ID : 1 Block: ID: 0`
	`Thread ID : 2 Block: ID: 0`
	`Thread ID : 0 Block: ID: 1`
	`Thread ID : 1 Block: ID: 1`
	`Thread ID : 2 Block: ID: 1`

`addKernel <<<3, 2 >>> (dev_c, dev_a, dev_b);`	`Thread ID : 0 Block: ID: 0`
	`Thread ID : 1 Block: ID: 0`
	`Thread ID : 0 Block: ID: 2`
	`Thread ID : 1 Block: ID: 2`
	`Thread ID : 0 Block: ID: 1`
	`Thread ID : 1 Block: ID: 1`

`addKernel <<<6, 1 >>> (dev_c, dev_a, dev_b);`	`Thread ID : 0 Block: ID: 5`
	`Thread ID : 0 Block: ID: 0`
	`Thread ID : 0 Block: ID: 3`
	`Thread ID : 0 Block: ID: 2`
	`Thread ID : 0 Block: ID: 4`
	`Thread ID : 0 Block: ID: 1`

Reflection:

The block number indicates which thread block the thread is located in and the thread number indicates which thread of the block is currently executing. This could be useful to identify threads when in a large-scale process where the parallel processing becomes more complex.

Metadata:

Threadidx.x, blockidx.x

Further information:

N/A

Q2. Exercise 2. Find vector addition using multiple 1D thread blocks

Question:

For the vector addition problem considered in the CUDA template, find the solution based on the following thread configurations by modifying the following line of CUDA code:

```
int i = threadIdx.x;
```

1) addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);
2) addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);
3) addKernel << <6, 1 >> > (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : %d Block: ID: %d", threadIdx.x, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out .

```
addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);
addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);
addKernel << <6, 1 >> > (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("{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]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
`addKernel <<<2, 3 >>> (dev_c, dev_a, dev_b);`	`{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,0,0}`
`addKernel <<<3, 2 >>> (dev_c, dev_a, dev_b);`	`{1,2,3,4,5} + {10,20,30,40,50} = {11,22,0,0,0}`
`addKernel <<<6, 1 >>> (dev_c, dev_a, dev_b);`	`{1,2,3,4,5} + {10,20,30,40,50} = {11,0,0,0,0}`

Reflection:

The output is limited by the number of threads generated in in each block as the calculation uses the thread ID to index each array. SO if more than 5 arrays are used then an error will be raised.

Metadata:

Multithread

Further information:

N/A

Q3. Exercise 3. Understand the thread indices for 2D blocks

Question:

List the values for the built-in variables **threadIdx.x** and **threadIdx.y** corresponding to the following thread configurations used for executing the kernel *addKernel( )* function on GPU:

1) addKernel << <1, dim3(2, 3) >> > (dev_c, dev_a, dev_b);
2) addKernel << <1, dim3(3, 3) >> > (dev_c, dev_a, dev_b);
3) addKernel << <1, dim3(5, 1) >> > (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : (X = %d,Y = %d) Block: ID: %d\n", threadIdx.x, threadIdx.y, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out .

```
addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(3,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(5,1) >> > (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("{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]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
`addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);`	Thread ID : (X = 0,Y = 0) Block: ID: 0 Thread ID : (X = 1,Y = 0) Block: ID: 0 Thread ID : (X = 0,Y = 1) Block: ID: 0 Thread ID : (X = 1,Y = 1) Block: ID: 0 Thread ID : (X = 0,Y = 2) Block: ID: 0 Thread ID : (X = 1,Y = 2) Block: ID: 0
`addKernel << <1, dim3(3,3) >> > (dev_c, dev_a, dev_b);`	Thread ID : (X = 0,Y = 0) Block: ID: 0 Thread ID : (X = 1,Y = 0) Block: ID: 0 Thread ID : (X = 2,Y = 0) Block: ID: 0 Thread ID : (X = 0,Y = 1) Block: ID: 0 Thread ID : (X = 1,Y = 1) Block: ID: 0 Thread ID : (X = 2,Y = 1) Block: ID: 0 Thread ID : (X = 0,Y = 2) Block: ID: 0 Thread ID : (X = 1,Y = 2) Block: ID: 0 Thread ID : (X = 2,Y = 2) Block: ID: 0
`addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);`	Thread ID : (X = 0,Y = 0) Block: ID: 0 Thread ID : (X = 1,Y = 0) Block: ID: 0 Thread ID : (X = 2,Y = 0) Block: ID: 0 Thread ID : (X = 3,Y = 0) Block: ID: 0 Thread ID : (X = 4,Y = 0) Block: ID: 0

Reflection:

The aim of the above task is to display how to Identify individual threads within a multi-dimensional block. Each thread has a unique Id based on its location in the thread which can be thought of as a coordinate to locate it e.g 2d array of threads has threadIdx.x and threadIDx.y and a 3D array of threads has threadIDx.x ,threadIdx.y and threadIdx.z to locate it.

Metadata:

Multithread threadId.x threadId.y threadId.z

Further information:

N/A

Q4. Exercise 4. Find vector addition using one 2D thread block

Question:

For the vector addition problem considered in the CUDA template, find the solution based on the following thread configurations by modifying the following line of CUDA code:

```
int i = threadIdx.x;
```

1) addKernel << <1, dim3(2, 3) >> > (dev_c, dev_a, dev_b);
2) addKernel << <1, dim3(3, 2) >> > (dev_c, dev_a, dev_b);
3) addKernel << <1, dim3(5, 1) >> > (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x + threadIdx.y + (threadIdx.x * (blockDim.y-1));
    c[i] = a[i] + b[i];
    printf("Thread ID : (%d,%d) Block: ID: (%d)\n", threadIdx.x, threadIdx.y, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out.

```
addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(3,2) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(5,1) >> > (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("{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]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output

`addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);`	`Thread ID : (0,0) Block: ID: (0)`
	`Thread ID : (1,0) Block: ID: (0)`
	`Thread ID : (0,1) Block: ID: (0)`
	`Thread ID : (1,1) Block: ID: (0)`
	`Thread ID : (0,2) Block: ID: (0)`
	`Thread ID : (1,2) Block: ID: (0)`
	`{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}`

`addKernel << <1, dim3(3,2) >> > (dev_c, dev_a, dev_b);`	`Thread ID : (0,0) Block: ID: (0)`
	`Thread ID : (1,0) Block: ID: (0)`
	`Thread ID : (2,0) Block: ID: (0)`
	`Thread ID : (0,1) Block: ID: (0)`
	`Thread ID : (1,1) Block: ID: (0)`
	`Thread ID : (2,1) Block: ID: (0)`
	`{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}`

`addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);`	`Thread ID : (0,0) Block: ID: (0)`
	`Thread ID : (1,0) Block: ID: (0)`
	`Thread ID : (2,0) Block: ID: (0)`
	`Thread ID : (3,0) Block: ID: (0)`
	`Thread ID : (4,0) Block: ID: (0)`
	`{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}`

Reflection:

The tasks' goal was to perform vector addition using a multi-dimensional array of threads however the vector addition only occure3d on the column number for the thread blocks.

Metadata:

Multithread threadId.x threadId.y threadId.z

Further information:

Unsure how to resolve issue of only adding columns do we need to add an id checking method that will adapt to include rows as an overflow when there aren't enough columns?

Q5. Exercise 5. Find vector addition using multiple 2D thread blocks

Question:

For the vector addition problem considered in the CUDA template, find the solution based on the following thread configurations by modifying the following line of CUDA code:

```
int i = threadIdx.x;
```

1) addKernel << < dim3(1, 3), dim3(3, 1) >> > (dev_c, dev_a, dev_b);
2) addKernel << < dim3(2, 3), dim3(2, 2) >> > (dev_c, dev_a, dev_b);
3) addKernel << < dim3(2, 2), dim3(2, 3) >> > (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : (%d,%d) Block: ID: (%d,%d)\n", threadIdx.x, threadIdx.y, blockIdx.x,blockIdx.y);
}
```

In the below image each add kernel function is run separately and the others commented out.

```
addKernel << <dim3(1,3), dim3(2,3) >> > (dev_c, dev_a, dev_b);
addKernel << <dim3(2,3), dim3(3,2) >> > (dev_c, dev_a, dev_b);
addKernel << <dim3(2,2), dim3(5,1) >> > (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("{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]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
addKernel << < dim3(1, 3), dim3(3, 1) >> > (dev_c, dev_a, dev_b);	
addKernel << < dim3(2, 3), dim3(2, 2) >> > (dev_c, dev_a, dev_b);	
addKernel << < dim3(2, 2), dim3(2, 3) >> > (dev_c, dev_a, dev_b);	

Reflection:

This is threading 101

Metadata:

Threads

Further information:

Unsure of the use of mut?