

Introduction

In the following exercises, we will start using CUDA threads for something more useful and interesting than printing the HelloWorld messages or thread indices.

We will start by writing an application that will perform a simple vector addition on GPU. In addition, this example demonstrates memory allocation, movement and also the measurement of execution time on GPU.

The second exercise will be slightly more complex and will involve matrix-matrix addition and if you get some extra time then you are encouraged to extend the code to implement matrix-matrix multiplication. This example is very interesting not only because matrix-matrix operations are very common but also because we will be able to see the CPU-GPU performance differences.

Exercise 1 – Vector Addition

Add includes and define N, which is the size of our array.

```
#include <stdio.h>
#include <cuda.h>
#define N 4096
```

Initialise grid and block variables. Only the first number, the x dimension, will be set

```
dim3 grid(1, 1, 1);    //e.g. dim3 grid(4,1,1)
dim3 block(1, 1, 1);   //e.g. dim3 bock(128,1,1)
```

Initialise host arrays and device memory pointers. Notice that we use the *_h* suffix for host and *_d* for device.

```
int b_h[N];
int c_h[N];
int *a_d;
int *b_d;
int *c_d;
```

Load the host arrays with some data. Alternatively, you can try to use the *rand()* function.

```
for(int i=0; i<N; i++)
{
    a_h[i] = i;
    b_h[i] = i*1;
}
```

Now we need to allocate device memory using *cudaMalloc* function.

```
cudaMalloc((void**)&a_d, N*sizeof(int));
cudaMalloc((void**)&b_d, N*sizeof(int));
cudaMalloc((void**)&c_d, N*sizeof(int));
```

Let's copy the host memory to device memory so we can do the vector addition. Notice the use of the last parameter.

```

cudaMemcpy(a_d, a_h, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b_h, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(c_d, c_h, N*sizeof(int), cudaMemcpyHostToDevice);

```

We can precisely measure the time it takes to run the vector addition kernel that you are about to write. This can be done using CUDA events. Declare the following variables:

```

cudaEvent_t start;
cudaEvent_t stop;
float elapsedTime;

```

Start the timer just before the kernel launch.

```

cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

```

Launch a kernel to do the vector addition. Try to write your own kernel that will add a_d and b_d arrays and store the results in the c_d array.

```

vectorAddKernel<<< >>>();

```

The kernel is now finished and you can measure the time it took, which will be stored in the `elapsedTime` variable.

```

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

```

Copy the results from the device and print them out. Again notice the use of the last parameter.

```

cudaMemcpy(c_h, c_d, N*sizeof(int), cudaMemcpyDeviceToHost);

for(int i=0; i<N; i++)
{
    printf("%i+%i = %i\n", a_h[i], b_h[i], c_h[i]);
}

```

Print the execution time and free the memory.

```

printf("Time to calculate results: %f ms.\n", elapsedTime);

cudaFree(a_h);
cudaFree(b_h);
cudaFree(c_h);

cudaEventDestroy(start);
cudaEventDestroy(stop);

```

Now you should have got all working and you can see how a lot of parallel threads can solve can work on different parts of data. Nvidia calls this *Single*

Instruction, Multiple Threads (SIMT), which is related to *Single Instruction, Multiple Data (SIMD)* ([see here](#))

Exercise 2 – Matrix Addition

Add the standard includes

```
#include <stdio.h>
#include <cuda.h>
```

Variable that holds matrix size in each dimension. For example, if N=10 then the matrix will have 10x10 elements.

```
int N = 10;
```

Initialise grid and block sizes just like in the previous exercise.

```
dim3 grid(1, 1, 1);
dim3 block(1, 1, 1);
```

Pointers to host and device memory. In this exercise, we will use dynamically allocated memory as opposed to a static memory, which is allocated at a compile time. This will allow us to modify matrix size while the program is running.

```
int *a_h;
int *b_h;
int *c_h;
int *d_h;
int *a_d;
int *b_d;
int *c_d;
```

Additional variables, *size* holds the number of bytes required by arrays.

```
int size;
```

We are going to use the CUDA events again to measure the time.

```
cudaEvent_t start;
cudaEvent_t stop;
float elapsedTime;
```

Print out the information about number of blocks and threads.

```
printf("Number of threads: %i (%ix%i)\n", block.x*block.y, block.x, block.y);
printf("Number of blocks: %i (%ix%i)\n", grid.x*grid.y, grid.x, grid.y);
```

Dynamically allocate host memory and load the arrays with some data.

```
size = N * N * sizeof(int);

a_h = (int*) malloc(size);
```

```

b_h = (int*) malloc(size);
c_h = (int*) malloc(size);
d_h = (int*) malloc(size);

for(int i=0; i<N; i++)
{
    for(int j=0; j<N; j++)
    {
        a_h[i * N + j] = i;
        b_h[i * N + j] = i;
    }
}

```

Now we can allocate memory on device.

```

cudaMalloc((void**)&a_d, size);
cudaMalloc((void**)&b_d, size);
cudaMalloc((void**)&c_d, size);

```

Copy the host memory to device.

```

cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(c_d, c_h, size, cudaMemcpyHostToDevice);

```

Start the timer.

```

cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

```

Launch the kernel

```

matrixAddKernel<<<grid, block>>>(a_d, b_d, c_d, N);

```

Stop the timer and print out the execution time

```

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

printf("Time to calculate results on GPU: %f ms.\n",
elapsedTime);

```

Copy the results to host

```

cudaMemcpy(c_h, c_d, size ,cudaMemcpyDeviceToHost);

```

Until now, we were computing on the GPU side. Now let's run the same code on CPU and let's compare the execution times. You are encouraged to experiment with different matrix size and different number of threads/blocks.

Start the timer again, this time to measure the CPU side performance.

```

cudaEventRecord(start, 0);

```

Launch the matrixAdd function that executes on CPU.

```
matrixAdd(a_h, b_h, d_h, N);
```

Stop the timer and print out the results.

```
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop );
printf("Time to calculate results on CPU: %f ms.\n",
elapsedTime);
```

At this point we should have the GPU results stored in *c_h* and CPU results stored in *d_h*. Let's check if the results very correctly calculated and match.

```
for(int i=0; i<N*N; i++)
{
    if (c_h[i] != d_h[i]) printf("Error: CPU and GPU results
do not match\n");
    break;
}
```

Probably you realised that you will need the *matrixAdd* and *matrixAddKernel* functions. Here is the CPU code and try to write a GPU kernel based on this code. If you struggle, just ask me or check the source code at the end of this file.

```
void matrixAdd(int *a, int *b, int *c, int N)
{
    int index;

    for(int col=0; col<N; col++)
    {
        for(int row=0; row<N; row++)
        {
            index = row * N + col;
            c[index] = a[index] + b[index];
        }
    }
}
```

If you get some extra time, try to implement GPU kernel for matrix-matrix multiplication based on this CPU version code.

```
void matrixMult(int *a, int *b, int *c, int N)
{
    int row, col, k, sum;

    for(row=0; row<N; row++) // row of a
    {
        for(col=0; col<N; col++) // column of b
        {
            sum = 0;
            for(k = 0; k < N; k++) sum += a[row * N + k] * b[k * N + col];
            c[row * N + col] = sum;
        }
    }
}
```

Source code – vector addition

```
#include <stdio.h>
#include <cuda.h>

//size of array
#define N 4096

//vector addition kernel
__global__ void vectorAddKernel(int *a, int *b, int *c)
{
    int tdx = blockIdx.x * blockDim.x + threadIdx.x;
    if(tdx < N)
    {
        c[tdx] = a[tdx]+b[tdx];
    }
}

int main()
{
    //grid and block sizes
    dim3 grid(1, 1, 1);
    dim3 block(1, 1, 1);

    //host arrays
    int a_h[N];
    int b_h[N];
    int c_h[N];

    //device memory pointers
    int *a_d;
    int *b_d;
    int *c_d;

    //load arrays with some numbers
    for(int i=0; i<N; i++)
    {
        a_h[i] = i;
        b_h[i] = i*1;
    }

    //allocate device memory
    cudaMalloc((void**)&a_d, N*sizeof(int));
    cudaMalloc((void**)&b_d, N*sizeof(int));
    cudaMalloc((void**)&c_d, N*sizeof(int));

    //copy the host arrays to device
    cudaMemcpy(a_d, a_h, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b_h, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(c_d, c_h, N*sizeof(int), cudaMemcpyHostToDevice);

    //CUDA events to measure time
    cudaEvent_t start;
    cudaEvent_t stop;
    float elapsedTime;

    //start timer
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    //launch kernel
```

```

vectorAddKernel<<<grid, block>>>(a_d, b_d, c_d);

//stop timer
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

//copy the results to host
cudaMemcpy(c_h, c_d, N*sizeof(int), cudaMemcpyDeviceToHost);

//print the results
for(int i=0; i<N; i++)
{
    printf("%i+%i = %i\n",a_h[i], b_h[i], c_h[i]);
}

//print out execution time
printf("Time to calculate results: %f ms.\n", elapsedTime);

//clean up
cudaFree(a_h);
cudaFree(b_h);
cudaFree(c_h);

cudaEventDestroy(start);
cudaEventDestroy(stop);

return 0;
}

```

Source code – matrix-matrix addition

```

#include <stdio.h>
#include <cuda.h>

__global__ void matrixAddKernel(int *a,int *b, int *c, int N)
{
    int col = threadIdx.x + blockDim.x * blockIdx.x;

```

```

        int row = threadIdx.y + blockDim.y * blockIdx.y;

        int index = row * N + col;

        if(col < N && row < N)
        {
            c[index] = a[index]+b[index];
        }
    }
}

void matrixAdd(int *a, int *b, int *c, int N)
{
    int index;

    for(int col=0; col<N; col++)
    {
        for(int row=0; row<N; row++)
        {
            index = row * N + col;
            c[index] = a[index] + b[index];
        }
    }
}

int main(int argc, char *argv[])
{
    //matrix size in each dimension
    int N = 10;

    //grid and block sizes
    dim3 grid(1, 1, 1);
    dim3 block(1, 1, 1);

    //host memory pointers
    int *a_h;
    int *b_h;
    int *c_h;
    int *d_h;

    //device memory pointers
    int *a_d;
    int *b_d;
    int *c_d;

    //number of bytes in arrays
    int size;

    //variable used for storing keyboard input
    char key;

    //CUDA events to measure time
    cudaEvent_t start;
    cudaEvent_t stop;
    float elapsedTime;

    //print out summary
    printf("Number of threads: %i (%ix%i)\n", block.x*block.y,
        block.x, block.y);

    printf("Number of blocks: %i (%ix%i)\n", grid.x*grid.y, grid.x,
        grid.y);
}

```

```

//number of bytes in each array
size = N * N * sizeof(int);

//allocate memory on host, this time we are using dynamic
//allocation
a_h = (int*) malloc(size);
b_h = (int*) malloc(size);
c_h = (int*) malloc(size);
d_h = (int*) malloc(size);

//load arrays with some numbers
for(int i=0; i<N; i++)
{
    for(int j=0; j<N; j++)
    {
        a_h[i * N + j] = i;
        b_h[i * N + j] = i;
    }
}

//GPU computation////////////////////////////////////

//allocate device memory
cudaMalloc((void**)&a_d, size);
cudaMalloc((void**)&b_d, size);
cudaMalloc((void**)&c_d, size);

//copy the host arrays to device
cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(c_d, c_h, size, cudaMemcpyHostToDevice);

//start timer
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

//launch kernel
matrixAddKernel<<<grid, block>>>(a_d, b_d, c_d, N);

//stop timer
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

//print out execution time
printf("Time to calculate results on GPU: %f ms.\n",
elapsedTime);

//copy the results to host
cudaMemcpy(c_h, c_d, size ,cudaMemcpyDeviceToHost);
//grid and block sizes

//CPU computation////////////////////////////////////

//start timer
cudaEventRecord(start, 0);

//do the calculation on host
matrixAdd(a_h, b_h, d_h, N);

```

```

        //stop timer
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&elapsedTime, start, stop );

        //print out execution time
        printf("Time to calculate results on CPU: %f ms.\n",
        elapsedTime);

        //check if the CPU and GPU results match
        for(int i=0; i<N*N; i++)
        {
            if (c_h[i] != d_h[i]) printf("Error: CPU and GPU results
            do not match\n");
            break;
        }

        //clean up
        free(a_h);
        free(b_h);
        free(c_h);
        free(d_h);

        cudaFree(a_d);
        cudaFree(b_d);
        cudaFree(c_d);

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        return 0;
}

```