

INTRODUCTION TO PROGRAMMING IN CUDA C

Will Landau, Matt Simpson, Prof. Jarad Niemi

BASIC C PROGRAM

```
#include <iostream>

int main ( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

BASIC CUDA C PROGRAM

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

```
#include <iostream>

__global__ void kernel ( void ) {
}
```

Prefix that says: "only run this function on the GPU"

```
int main ( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) { ← Started on CPU
    kernel<<<1,1>>>(); ← Sent to GPU
    printf( "Hello, World!\n" ); ← Done
    return 0; ← Done on CPU
}
```

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) {
    kernel<<<1,1>>>(); ?
    printf( "Hello, World!\n" );
    return 0;
}
```

REVIEW OF PARALLELIZATION

Parallelization: Running different calculations simultaneously.

Kernel: An instruction set executed on the GPU. (All others are executed on the CPU.)

In CUDA C, a kernel is any function prefixed with the keyword, `--global__`. For example:

```
--global__ void kernel ( void ) {  
}
```

PARALLELIZATION AND THE SIMD PARADIGM

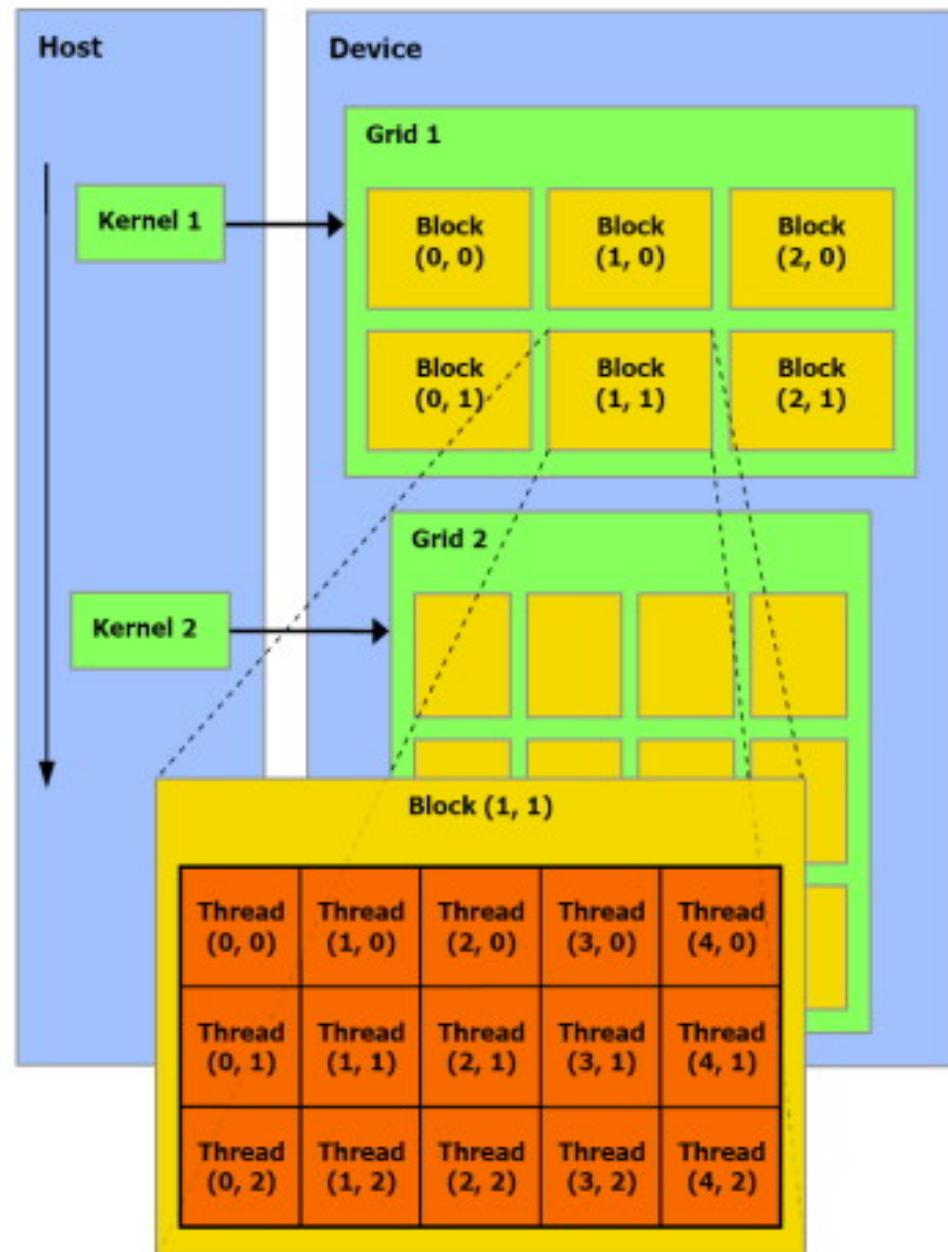
SIMD: Single Instruction Multiple Data. The paradigm works like this:

1. The CPU sends a kernel (a SINGLE INSTRUCTION set) to the GPU.
2. The GPU executes the kernel MULTIPLE times simultaneously in **PARALLEL**. Each such execution of the kernel is called a **thread**, and each thread is executed on different parts of a giant data set.

ORGANIZATION OF THREADS

Grid; The collection of all the threads that are spawned when the CPU sends a kernel to the GPU.

Block: A collection of threads within a grid that share memory quickly and easily.



NOW BACK TO THE STUFF IN ANGLE BRACKTES...

```
#include <iostream>

__global__ void kernel ( void ) {
} Uses one grid.      Number of blocks
int main ( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Number of threads per block

The code demonstrates a simple CUDA kernel named 'kernel' that prints 'Hello, World!' to the console. It uses a single grid and one block. The kernel is launched with 1 thread per block. The code is annotated with red text and arrows to explain these parameters.

Here, the GPU runs kernel() one time:

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!" \n" )
    return 0;
}
```

Here, the GPU runs `kernel()` 5 times:

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) {
    kernel<<<5,1>>>();
    printf( "Hello, World!" \n" )
    return 0;
}
```

Here, the GPU runs `kernel()` 5 times:

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) {
    kernel<<<1,5>>>();
    printf( "Hello, World!" \n" )
    return 0;
}
```

Here, the GPU runs kernel() 20 times:

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) {
    kernel<<<4,5>>>();
    printf( "Hello, World!" \n" )
    return 0;
}
```

PASSING DATA TO AND FROM THE GPU: simple1.cu

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

__global__ void colonel(int *dev_a){
    *dev_a = 1;
}

int main(){

    // Declare variables and allocate memory on the GPU.
    int a[1], *dev_a;
    cudaMalloc((void**) &dev_a, sizeof(int));

    // Execute kernel and copy the result to CPU memory.
    colonel<<<1,1>>>(dev_a);
    cudaMemcpy(a, dev_a, sizeof(int), cudaMemcpyDeviceToHost);

    // Print result and free dynamically allocated memory.
    printf("a[0] = %d\n", a[0]); // REMEMBER: INDEXING IN C STARTS FROM 0.
    cudaFree(dev_a);

}
```

```
[landau@impact1 simple]$ nvcc simple1.cu -o simple1.out
[landau@impact1 simple]$ ./simple1.out
a[0] = 1
[landau@impact1 simple]$ |
```

PASSING DATA TO AND FROM THE GPU: simple2.cu

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

__global__ void colonel(int *dev_a){
    *dev_a = *dev_a + 1;
}

int main(){
    // Declare variables and allocate memory on the GPU.
    int a[1], *dev_a;
    cudaMalloc((void**) &dev_a, sizeof(int));

    // Initialize argument a, executed kernel, and store result back in a.
    a[0] = 1; // REMEMBER: INDEXING IN C STARTS FROM 0.
    cudaMemcpy(dev_a, a, sizeof(int), cudaMemcpyHostToDevice);
    colonel<<<1,1>>>(dev_a);
    cudaMemcpy(a, dev_a, sizeof(int), cudaMemcpyDeviceToHost);

    // Print result and free dynamically allocated memory.
    printf("a[0] = %d\n", a[0]); // REMEMBER: INDEXING IN C STARTS FROM 0.
    cudaFree(dev_a);
}
```

```
[landau@impact1 simple]$ nvcc simple2.cu -o simple2.out
[landau@impact1 simple]$ ./simple2.out
b[0] = 2
[landau@impact1 simple]$
```

threadIdx.x AND blockIdx.x

Say the CPU sends a kernel to the GPU with B blocks and T threads per block.

The following CUDA C variables are available for use in the kernel:

blockIdx.x: the block ID corresponding to the current thread, an integer from 0 to $B - 1$ inclusive.

threadIdx.x: the thread ID of the current thread within its block, an integer from 0 to $T - 1$ inclusive.

NOTE: there are also variables like `blockIdx.y` and `threadIdx.y`, but those are only useful if you deliberately organize your blocks and threads in two-dimensional arrays (which can be done).

USING blockIdx.x AND threadIdx.x: identify_naive1.cu

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

__global__ void isExecuted(int *dev_a, int blockid, int threadid){

    if(blockIdx.x == blockid && threadIdx.x == threadid)
        *dev_a = 1;
    else
        *dev_a = 0;

}

int main(){
```

```
// Declare variables and allocate memory on the GPU.  
int a[1], *dev_a;  
cudaMalloc((void**) &dev_a, sizeof(int));  
  
// Execute kernel and copy the result to CPU memory.  
isExecuted<<<1,1>>>(dev_a, 0, 0);  
    // NOTE: INDEXING OF THREADS AND BLOCKS STARTS FROM 0.  
cudaMemcpy(a, dev_a, sizeof(int), cudaMemcpyDeviceToHost);  
  
// Print result and free dynamically allocated memory.  
printf("a[0] = %d\n", a[0]); // REMEMBER: INDEXING IN C STARTS FROM 0.  
cudaFree(dev_a);  
}
```

```
[landau@impact1 simple]$ nvcc identify_naive1.cu -o identify_naive1.out
[landau@impact1 simple]$ ./identify_naive1.out
a[0] = 1
[landau@impact1 simple]$
```

You can verify that when the line,

```
isExecuted<<<1,1>>>(dev_a, 0, 0);
```

is changed to:

```
isExecuted<<<1,1>>>(dev_a, 2, 4);
```

the output of the program:

```
a[0] = 0
```

which makes sense because there was no block number 2 or thread number 4.

However, if I change that same line to:

```
isExecuted<<<100,100>>>(dev_a, 2, 4);
```

the output is still:

```
a[0] = 0
```

Even though thread number 4 in block number 2 executed.

Why? Take a look at identify_naive2.cu on the following page:

```

#include <stdio.h>
#include <stdlib.h>

__global__ void isExecuted(int *dev_a, int blockid, int threadid){

    if(blockIdx.x == blockid && threadIdx.x == threadid)
        *dev_a = 1;
    else
        *dev_a = 0;

}

int main(){

    // Declare variables and allocate memory on the GPU.
    int a[1], *dev_a;
    cudaMalloc((void**) &dev_a, sizeof(int));

    // Execute kernel and copy the result to CPU memory.
    isExecuted<<<100,100>>>(dev_a, 2, 4);
    // NOTE: INDEXING OF THREADS AND BLOCKS STARTS FROM 0.
    cudaMemcpy(a, dev_a, sizeof(int), cudaMemcpyDeviceToHost);

    // Print result and free dynamically allocated memory.
    printf("a[0] = %d\n", a[0]); // REMEMBER: INDEXING IN C STARTS FROM 0.
    cudaFree(dev_a);

}

```

```
[lalandau@impact1 simple]$ nvcc identify_naive2.cu -o identify_naive2.out  
[lalandau@impact1 simple]$ ./identify_naive2.out  
a[0] = 0
```

The problem is in:

```
__global__ void isExecuted(int *dev_a, int blockid,
                           int threadid){

    if(blockIdx.x == blockid && threadIdx.x == threadid)
        *dev_a = 1;
    else
        *dev_a = 0;

}
```

The problem is in:

```
--global__ void isExecuted(int *dev_a, int blockid,
                           int threadid){

    if(blockIdx.x == blockid && threadIdx.x == threadid)
        *dev_a = 1;
    else
        *dev_a = 0;

}
```

The thread with blockIdx.x = blockid and threadIdx.x = threaded will set *dev_a to 1.

However, all the other threads will set *dev_a to 0.

IMPROVED CODE: identify.cu

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

__global__ void isExecuted(int *dev_a, int blockid, int threadid){

    if(blockIdx.x == blockid && threadIdx.x == threadid)
        *dev_a = 1;

}

int main(){

    // Declare variables and allocate memory on the GPU.
    int init[1], a[1], *dev_a;
    cudaMalloc((void**) &dev_a, sizeof(int));
```

```
// Initialize dev_a, execute kernel, and copy the result to CPU memory.  
init[1] = 0;  
cudaMemcpy(dev_a, init, sizeof(int), cudaMemcpyHostToDevice);  
isExecuted<<<100,100>>>(dev_a, 2, 4);  
    // NOTE: INDEXING OF THREADS AND BLOCKS STARTS FROM 0.  
cudaMemcpy(a, dev_a, sizeof(int), cudaMemcpyDeviceToHost);  
  
// Print result and free dynamically allocated memory.  
printf("a[0] = %d\n", a[0]); // REMEMBER: INDEXING IN C STARTS FROM 0.  
cudaFree(dev_a);  
}
```

```
[landau@impact1 simple]$ nvcc identify.cu -o identify.out  
[landau@impact1 simple]$ ./identify.out  
a[0] = 1  
[landau@impact1 simple]$ |
```

EXAMPLE: VECTOR SUMMATION (Sanders, et. al.)

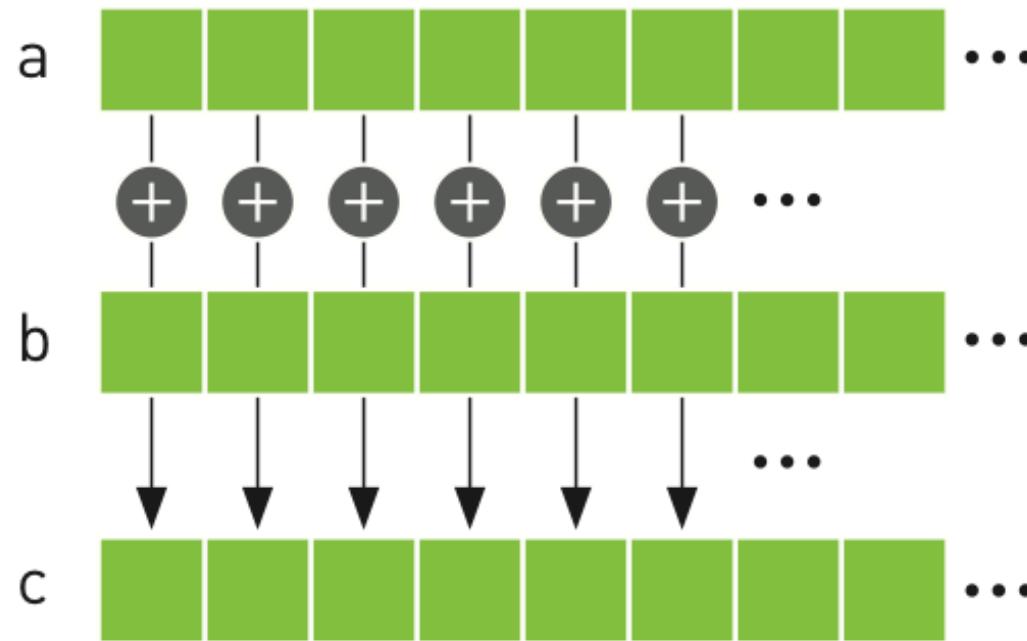


Figure 4.1 Summing two vectors

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;      // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

```
#define N    10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
```

```
// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                         cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                         cudaMemcpyHostToDevice ) );

add<<<N,1>>>( dev_a, dev_b, dev_c );

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                         cudaMemcpyDeviceToHost ) );

// display the results
for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
```

```
// free the memory allocated on the GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}
```

REFERENCES

J. Sanders and E. Kandrot. *CUDA by Example*. Addison-Wesley, 2010.