INTRODUCTION TO PROGRAMMING IN CUDA C

Will Landau, 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;
}
```

__global__: Call from CPU and run only on GPU.

__device__: Call from GPU and run only on GPU.
                           (More specifically, call only from within
                           a __global__ or another __device__ function.)

__host__: Call from CPU and run only on CPU.
                           (i.e., a traditional C function.)

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

```
__device__ int dev1( void ){
}


__device__ int dev2( void ){
}


__global__ void kernel ( void ) {
  dev1();
  dev2();
}


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 on CPU
  return 0;                  ← Done on CPU
}
```

```cpp
#include <iostream>

__global__ void kernel ( void ) {
}

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

REVIEW OF PARALLELISM

Parallelism: 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 ) {
}
```
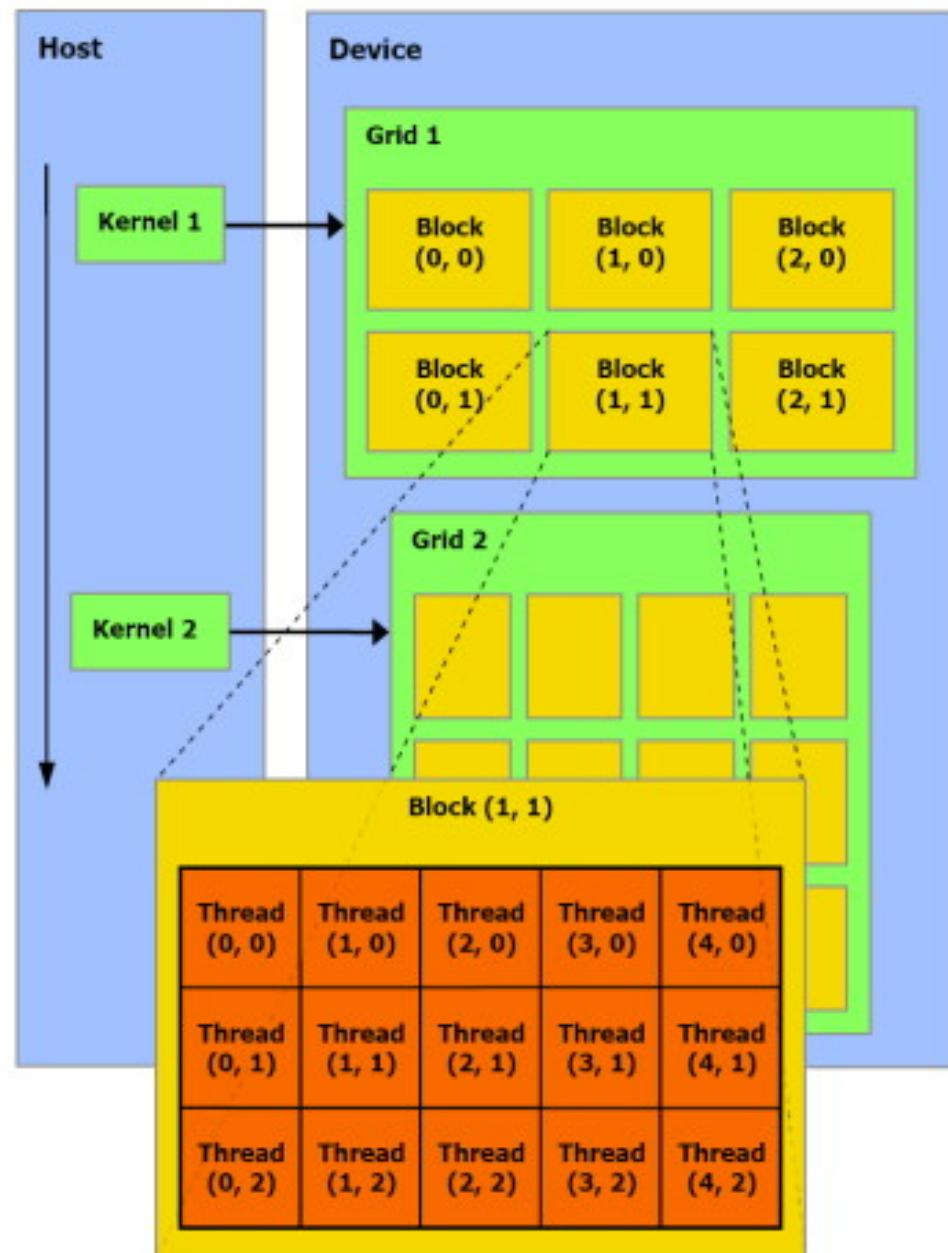
PARALLELISM AND THE SIMD PARADIGM

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.

PARALLELIZING A WORKLOAD ON A GPU
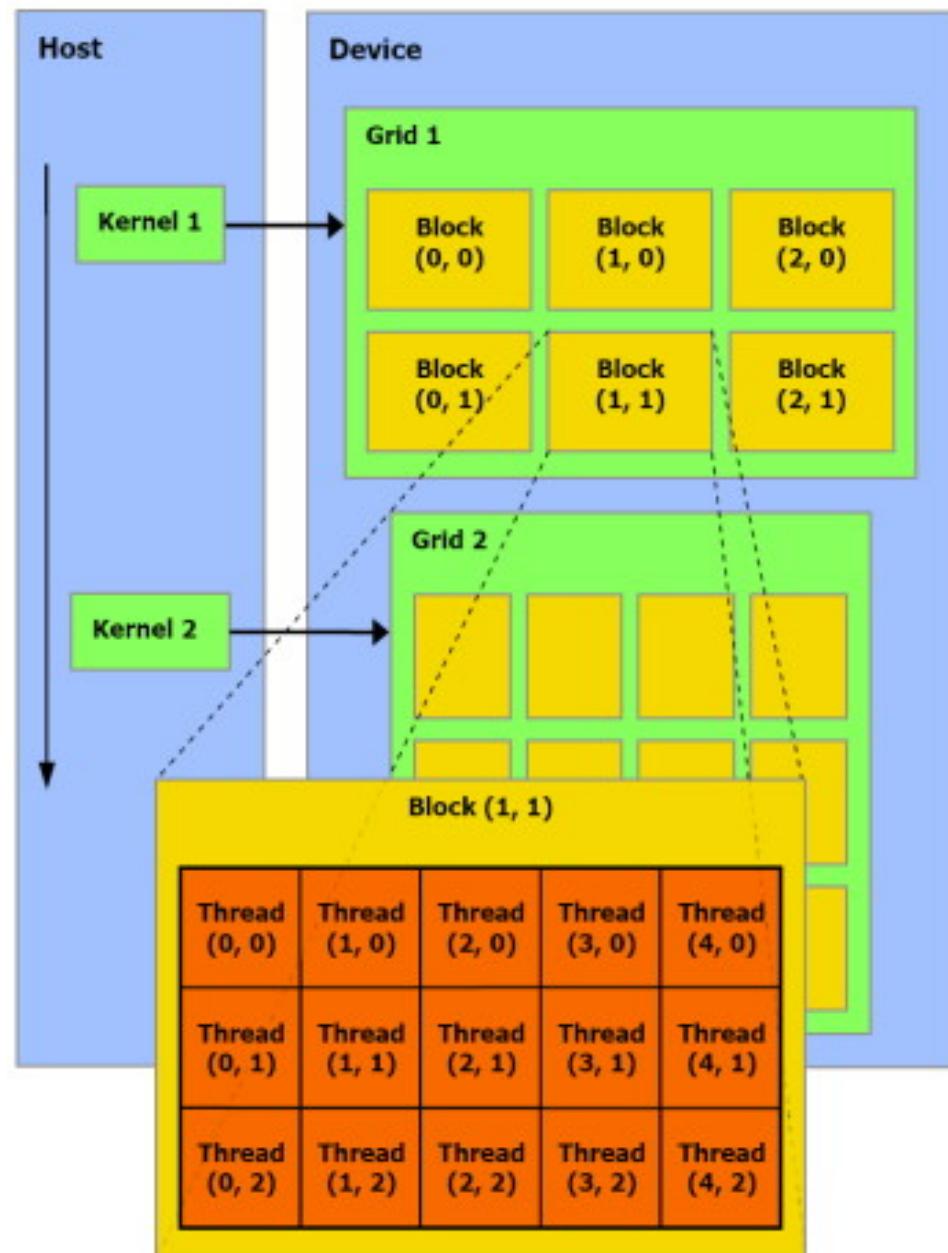
1. The CPU sends a CPU-to-GPU command called a **kernel** to a single GPU core.

2. The GPU core multitasks to execute the command:

    a. The GPU makes $N \cdot M$ copies of the kernel's code, and then runs all those copies simultaneously. Those parallel copies are called **threads**.

    b. The $N \cdot M$ threads are partitioned into N groups, called **blocks**, of M threads each.

    c. The sum total of all the threads from a kernel call is a **grid**.

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

Number of threads
per block

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

Here, the GPU runs kernel() one time:

```cpp
#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:

```cpp
#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:

```cpp
#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

```c
#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

```c
#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));

  // Intitialize 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.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]$
```

isExecuted<<<2,3>>>(dev_a, blockid = 1, threadid = 1);

isExecuted(dev_a, blockid = 1, threadid = 1, blockIdx.x = 0, threadIdx.x = 0, ...); ⟶ *dev_a is unchanged.

isExecuted(dev_a, blockid = 1, threadid = 1, blockIdx.x = 0, threadIdx.x = 1, ...); ⟶ *dev_a is unchanged.

isExecuted(dev_a, blockid = 1, threadid = 1, blockIdx.x = 0, threadIdx.x = 2, ...); ⟶ *dev_a is unchanged.

isExecuted(dev_a, blockid = 1, threadid = 1, blockIdx.x = 1, threadIdx.x = 0, ...); ⟶ *dev_a is unchanged.

isExecuted(dev_a, blockid = 1, threadid = 1, blockIdx.x = 1, threadIdx.x = 1, ...); ⟶ *dev_a is set to 1.

isExecuted(dev_a, blockid = 1, threadid = 1, blockIdx.x = 1, threadIdx.x = 2, ...); ⟶ *dev_a is unchanged.

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.

ASIDE: AVOIDING PITFALLS IN PARALLELISM

What problem would arise if I used the function, isExectued_goofup(), instead
of Executed():

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

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

}
```

isExecuted_goofup<<<2,3>>>(dev_a, blockid = 1, threadid = 1);

isExecuted_goofup(dev_a, blockid = 1, threadid = 1, blockIdx.x = 0, threadIdx.x = 0, ...); ⟶ *dev_a is set to 0.

isExecuted_goofup(dev_a, blockid = 1, threadid = 1, blockIdx.x = 0, threadIdx.x = 1, ...); ⟶ *dev_a is set to 0.

isExecuted_goofup(dev_a, blockid = 1, threadid = 1, blockIdx.x = 0, threadIdx.x = 2, ...); ⟶ *dev_a is set to 0.

isExecuted_goofup(dev_a, blockid = 1, threadid = 1, blockIdx.x = 1, threadIdx.x = 0, ...); ⟶ *dev_a is set to 0.

isExecuted_goofup(dev_a, blockid = 1, threadid = 1, blockIdx.x = 1, threadIdx.x = 1, ...); ⟶ *dev_a is set to 1.

isExecuted_goofup(dev_a, blockid = 1, threadid = 1, blockIdx.x = 1, threadIdx.x = 2, ...); ⟶ *dev_a is set to 0.

If block 1 thread 1 finishes last, then *dev_a = 1.
If block 1 thread 1 does not finish last, *dev_a = 0.

We can't know or control which thread finishes last!

LECTURE SERIES MATERIALS

These lecture slides, a tentative syllabus for the whole lecture series, and code are available at:

https://github.com/wlandau/gpu.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the course materials.

REFERENCES

David B. Kirk and Wen-mei W. Hwu. "Programming Massively Parallel Processors: a Hands-on Approach." Morgan Kaufman, 2010.

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