Using Blocks and Threads in CUDA

slides taken from:

(C-) CUDA programming for (multi) GPU

Massimo Bernaschi
http://www.iac.cnr.it/~massimo

Parallel Programming in CUDA C

- GPU computing is about **massive** parallelism

- So how do we run code ***in parallel*** on the device?

- Solution lies in the parameters between the triple angle brackets:

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

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

- Instead of executing `add()` once, `add()` executed **N** times in parallel

Parallel Programming in CUDA C

- With `add()` running in parallel…let's do *vector* addition
- Terminology: Each parallel invocation of `add()` referred to as a ***block***
- Kernel can refer to its block's index with the variable **`blockIdx.x`**
- Each block adds a value from **`a[]`** and **`b[]`**, storing the result in **`c[]`**:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x]+b[blockIdx.x];
}
```

- By using **`blockIdx.x`** to index arrays, each block handles a different index
- **`blockIdx.x`** is the first example of a CUDA predefined variable.

Parallel Programming in CUDA C

- We write this code:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x]+b[blockIdx.x];
}
```

- This is what runs in parallel on the device:

Block 0	Block 1
`c[0]=a[0]+b[0];`	`c[1]=a[1]+b[1];`

Block 2	Block 3
`c[2]=a[2]+b[2];`	`c[3]=a[3]+b[3];`

Parallel Addition: main()

```
#define N  512
int main( void ) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = N * sizeof( int );   // we need space for 512
                                   // integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition: main() (cont)

```
    // copy inputs to device
   cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
   cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N parallel blocks
    add<<< N, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
   cudaMemcpy( c, dev_c, size,   cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Review

- Difference between “host” and “device”
  - Host = CPU
  - Device = GPU

- Using `__global__` to declare a function as device code
  - Runs on device
  - Called from host

- Passing parameters from host code to a device function

Review (cont)

- Basic device memory management
  - **cudaMalloc**()
  - **cudaMemcpy**()
  - **cudaFree**()

- Launching parallel kernels
  - Launch **N** copies of **add()** with: **add<<< N, 1 >>>();**
  - **blockIdx.x** allows to access block's index

- Exercise: look at, compile and run the *add_simple_blocks.cu* code

Threads

- Terminology: A block can be split into parallel *threads*

- Let's change vector addition to use parallel threads instead of parallel blocks:

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

- We use **threadIdx.x** instead of **blockIdx.x** in **add**()

- **main**() will require one change as well…

Parallel Addition (Threads): main()

```
#define N  512
int main( void ) {
    int *a, *b, *c;             //host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;//device copies of a, b, c
    int size = N * sizeof( int )//we need space for 512  integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Threads): `main()`

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N threads
    add<<< 1, N >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b );  free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Exercise: compile and run the *add_simple_threads.cu* code

Using Threads *And* Blocks

- We've seen parallel vector addition using
  - Many blocks with 1 thread apiece
  - 1 block with many threads

- Let's adapt vector addition to use lots of ***both*** blocks and threads

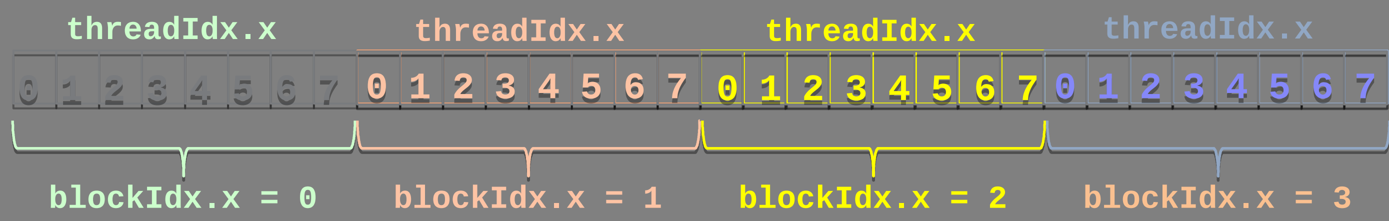
- After using threads and blocks together, we'll talk about ***why*** threads

- First let's discuss data indexing…

Indexing Arrays With Threads & Blocks

- No longer as simple as just using **threadIdx.x** or **blockIdx.x** as indices

- To index array with 1 thread per entry (using 8 threads/block)

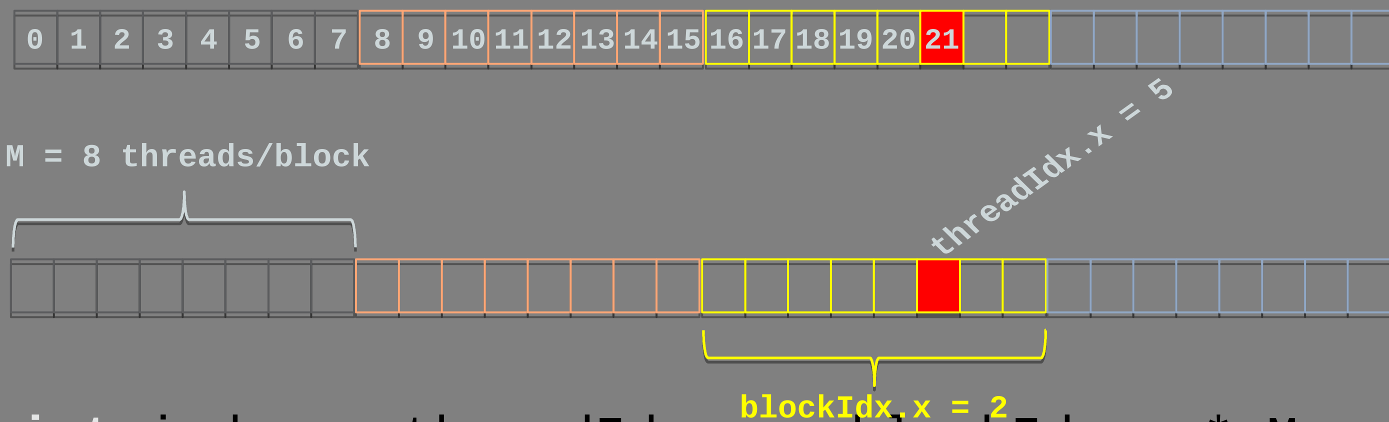


- If we have M threads/block, a unique array index for each entry is given by

```
int index = threadIdx.x + blockIdx.x * M;

int index =          x  +          y  * width;
```

Indexing Arrays: Example

- In this example, the **red** entry would have an index of 21:




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +      2      * 8;
          = 21;
```

Indexing Arrays: other examples
(4 blocks with 4 threads *per* block)

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}                                Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
                                 Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
                                 Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
}
```

Addition with Threads and Blocks

- **`blockDim.x`** is a built-in variable for threads per block:

```
int index= threadIdx.x + blockIdx.x * blockDim.x;
```

- **`gridDim.x`** is a built-in variable for blocks in a grid;

- A combined version of our vector addition kernel to use blocks *and* threads:

```
__global__ void add( int *a, int *b, int *c ) {
int index = threadIdx.x + blockIdx.x * blockDim.x;
     c[index] = a[index] + b[index];
}
```

- So what changes in `main()` when we use both blocks and threads?

Parallel Addition (Blocks/Threads)

```
#define N  (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;               // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;//device copies of a, b, c
    int size = N * sizeof( int ); // we need space for N
                                 // integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Blocks/Threads)

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with blocks and threads
 add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c);

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

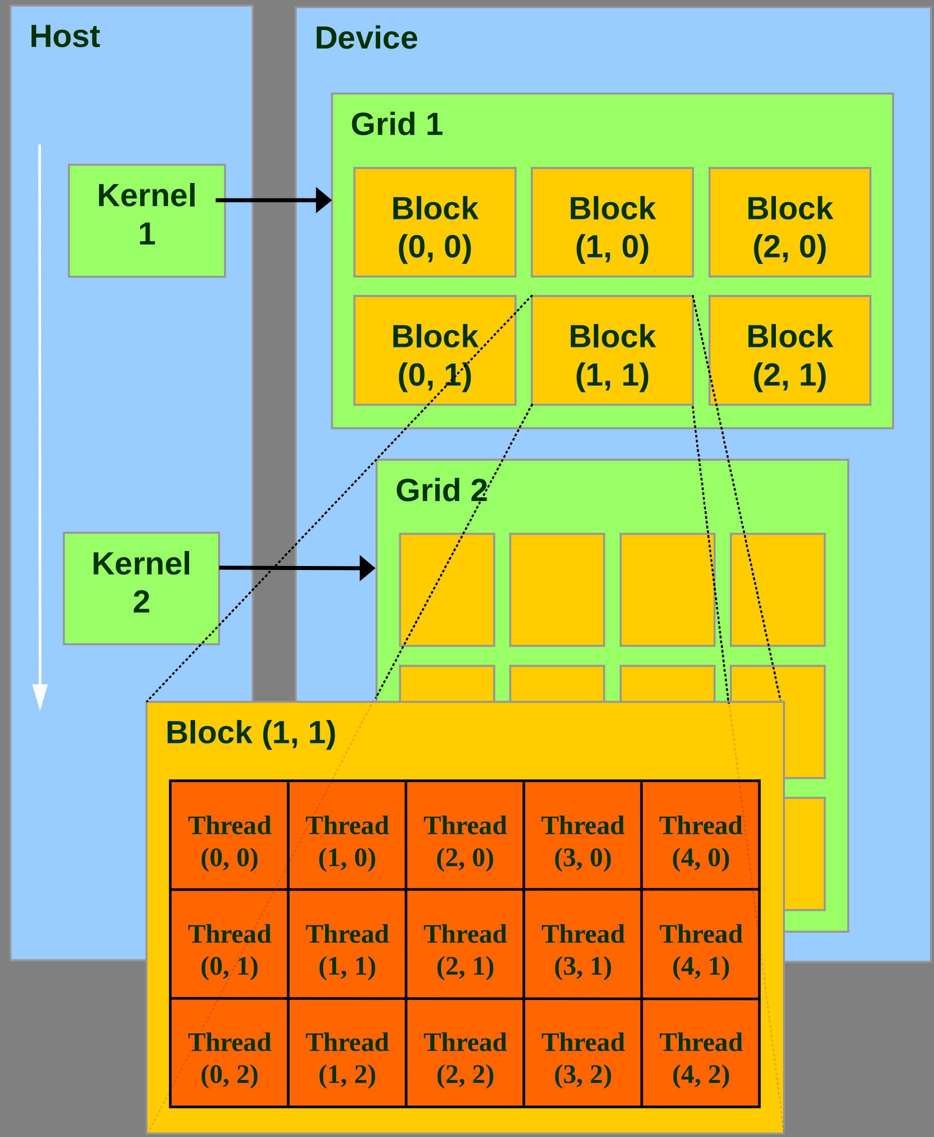
Exercise: compile and run the *add_simple.cu* code

Exercise: dynamic size floating point vector add

- Start from the ***vector_add.cu*** code which has an implementation for the vector addition on the CPU. The code must do an addition of two vectors and produce the same result as on the CPU.
- The comments containing XXX indicate what the CUDA code should do, that you will have to write.
- Remember that:
  - **`blockDim.x`** is the number of threads per block;
  - **`threadIdx.x`** is the index of the current thread in the block;
  - **`gridDim.x`** is the number of blocks in a grid;
  - **`blockIdx.x`** is the index of the current block in the grid;

CUDA Thread organization: Grids and Blocks

- A kernel is executed as a 1D, 2D or 3D grid of thread blocks.
  - All threads share the *global* memory
- A thread block is a 1D, 2D or 3D batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Threads blocks are independent of each other and can be executed **in any order!**



Courtesy: NVIDIA

Built-in Variables to manage grids and blocks

`dim3`: a new datatype defined by CUDA: **struct dim3 { unsigned int x, y, z };** three unsigned ints where any unspecified component defaults to 1.

- **`dim3 gridDim;`**
  - Dimensions of the grid in blocks
- **`dim3 blockDim;`**
  - Dimensions of the block in threads
- **`dim3 blockIdx;`**
  - Block index within the grid
- **`dim3 threadIdx;`**
  - Thread index within the block

Bi-dimensional threads configuration by example: set the elements of a square matrix (assume the matrix is a single block of memory!)

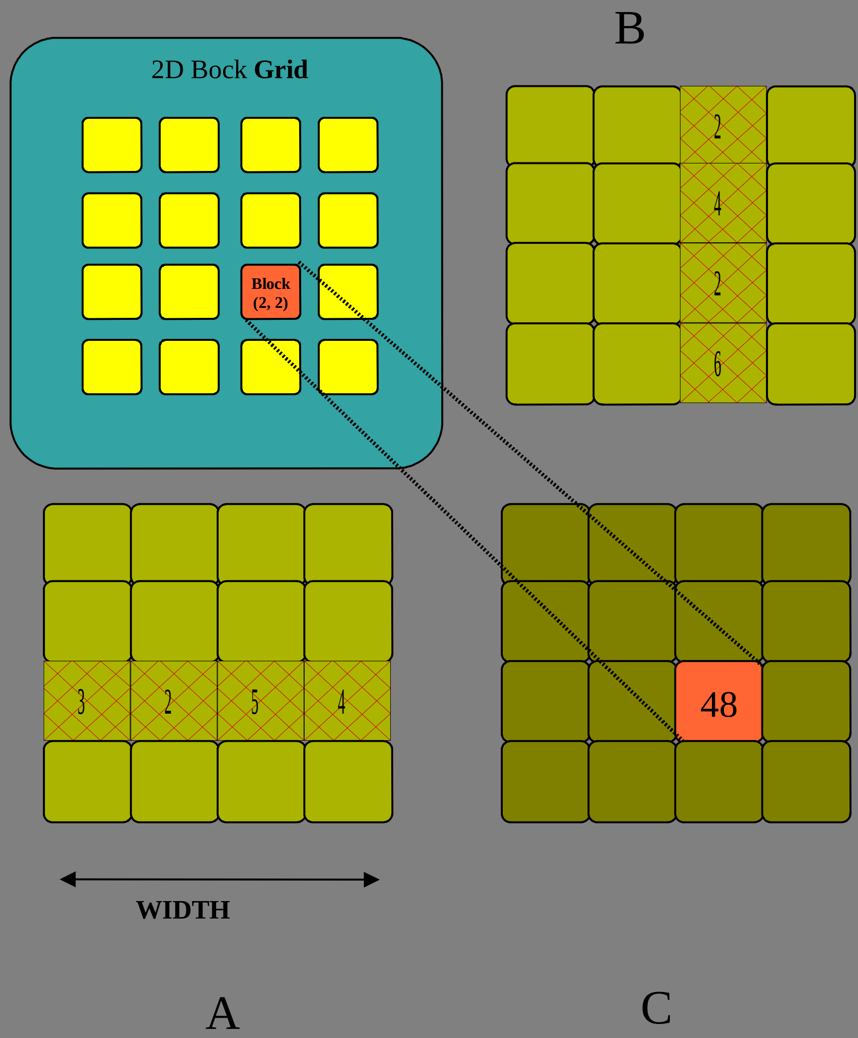
```
__global__ void kernel( int *a, int dimx, int dimy ) {
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = idx+1;
}
```

```
int main() {
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy(h_a,d_a,num_bytes,
                        cudaMemcpyDeviceToHost);

    for(int row=0; row<dimy; row++) {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );
    return 0;
}
```

Exercise: compile and run: **setmatrix.cu**

Matrix multiply with thread blocks

- One block of threads computes one matrix element of matrix C
- Each block loops over
  - a row of matrix A
  - a column of matrix B
  - Perform one multiply and sum the result into a temporary variable
- Store the result into the proper element of matrix C
- Size of matrix limited by the number of blocks per dimension!



Exercise: dynamic size matrix multiply

- Start from the ***matrix_multiply.cu*** code which has an implementation for a unoptimized matrix multiplication on the CPU
- The code must do the multiplication of the matrices and produce the same result as on the CPU.
- The comments containing XXX indicate what the CUDA code should do, that you will have to write.
- Use a 2D-grid of thread blocks where each block computes one matrix element of the result matrix.
- For tomorrow: parallelize the dot product over threads