GPGPU Programming Fundamentals — 1

Salvatore Filippone, PhD

DICII
salvatore.filippone@uniroma2.it

Nomenclature

Host Environment hosting and controlling the graphics card (CPU and main memory)

Device The graphics processor (GPU cores and memory)

Streaming Multi-processor (SM) A group of GPU cores designed to work together

Kernel Code that runs on the graphics device

(GPU) thread Lightweight process that executes on a GPU

Block (of threads) A group of threads executed together on the same SM

Consider the problem of adding two vectors:

$$w \leftarrow u + v$$

- Trivially data parallel (vector entries are independent of each other);
- Available parallelism depends on size (cannot use more processors than vector entries);
- Low arithmetic intensity (how many data accesses per floating point operation?);

Sample serial code:

Hello World!

Consider the problem of adding two vectors:

$$w \leftarrow u + v$$

- Trivially data parallel (vector entries are independent of each other);
- Available parallelism depends on size (cannot use more processors than vector entries);
- Low arithmetic intensity (how many data accesses per floating point operation?);

Sample serial code:

```
void VectorAdd(int n, const float* u,
               const float* v, float* w) {
    int i;
    for (i=0; i<n; i++)
        w[i] = u[i] + v[i];
}
```

Hello World!

What do you expect the CUDA code to look like?

Hello World!

What do you expect the CUDA code to look like?

```
// Computes w = u + v
__global__ void VectorAdd(const float* u,
                          const float* v, float* w) {
    int idx = threadIdx.x;
    w[idx] = u[idx] + v[idx];
}
```

Adding two vectors

- Trivial to parallelise — one thread per vector element
- `threadIdx` — structure holding thread co-ordinates
- Pointers u, v and w should point to device (GPU) memory

```
// Computes w = u + v
__global__ void VectorAdd(const float* u,
                          const float* v, float* w) {
    int idx = threadIdx.x;
    w[idx] = u[idx] + v[idx];
}
```

`__global__` function

- Can only be called from the host
- Runs on the device
- Cannot return anything (must be a void function)
- Cannot be called recursively
- Cannot contain I/O statements;

Other Function Types

__device__ function

- Is executed on the device;
- Can only be called from device or global function;

__host__ function

- The same as a standard C global function;
- Useful to generate CPU and GPU function at the same time

```
__host__ __device__ float DegToRad(float deg) {
  return deg * M_PI / 180.0f;
}
```

Kernel Invocation

Or, how to get the GPU device running:

```
int main(int argc, char** argv) {
    // Allocate and initialise x, y and z vectors
    ...
    // Run VectorAdd kernel with N threads in one block
    VectorAdd<<<1, N>>>(x, y, z);
}
```

Kernel Invocation

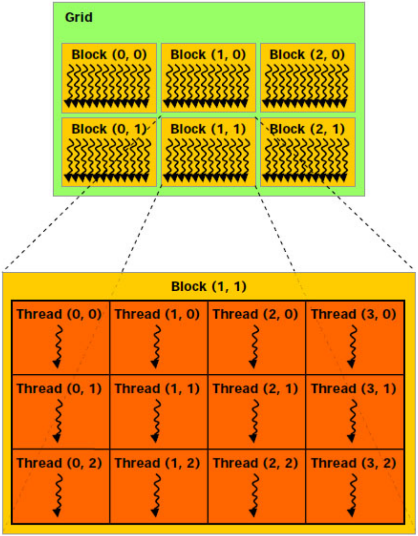
Or, how to get the GPU device running:

```
int main(int argc, char** argv) {
    // Allocate and initialise x, y and z vectors
    ...
    // Run VectorAdd kernel with N threads in one block
    VectorAdd<<<1, N>>>(x, y, z);
}
```

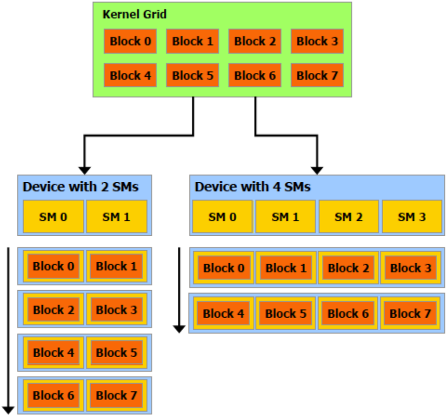
- `<<<...>>>` syntax is used to define the number of blocks and the size of each block of threads
- The number of blocks is (almost!) unlimited
- The maximum number of threads per block is relatively low — typically 1024
- Efficient communication between threads is only possible within a block

In This Section You Will Hear About:

- Thread hierarchy (threads, blocks, grid)
- Thread indexing
- Warps
- Choosing the block size



- Threads are grouped in blocks (of threads)
- Blocks form a grid
- Threads and blocks can be indexed in up to 3 dimensions

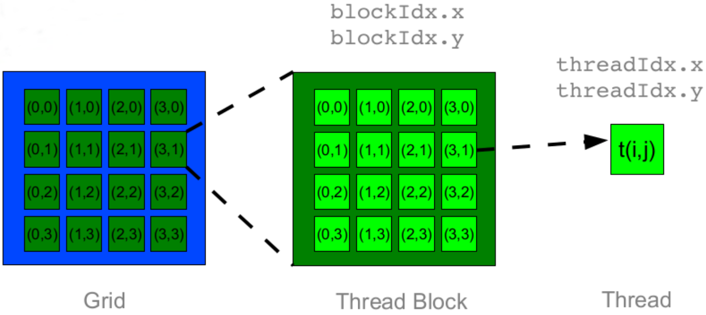


- Blocks are assigned to SMs and executed independently
- Multiple blocks can be executed on an *SM* concurrently
- Blocks are never preempted from SMs

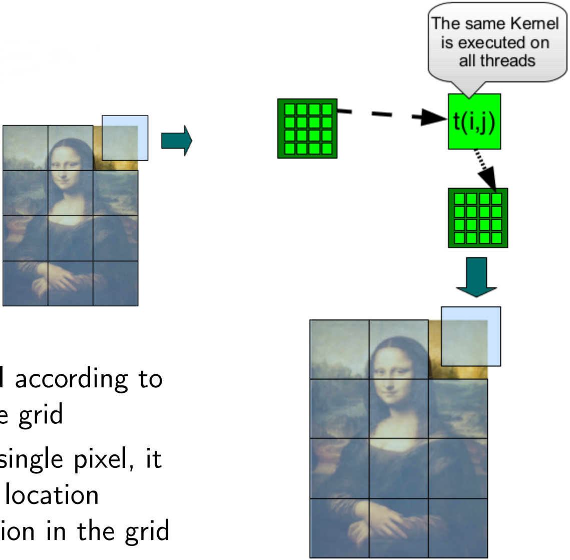
Built-in Variables

- `threadIdx` — thread position within a thread block
- `blockDim` — dimensions of the thread block
- `blockIdx` — block position in the grid
- `gridDim` — dimensions of the grid of thread blocks

All variables are of type `dim3`, a structure containing three `int` members: `x`, `y`, `z`



Actual index of thread t(i,j)
i: `blockIdx.x * blockDim.x + threadIdx.x`
j: `blockIdx.y * blockDim.y + threadIdx.y`



- Thread retrieves pixel according to its position within the grid
- After processing the single pixel, it stores the result at a location according to its position in the grid

Grids of thread blocks

- Size is related to the amount of data to be processed
  - Not related to number of processing cores
  - Significant difference to CPU threads
- Prerequisites for successful thread execution
  - Threads are independent of one another
  - They can be executed in any order

Improved Hello World (1/4)

Multiple blocks

```
__global__ void VectorAdd(const float* u,
                          const float* v, float* w) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   w[idx] = u[idx] + v[idx];
}
```

How many threads can we execute now?

- Number of blocks can be very large, but each grid dimension is limited to 65, 536
- Maximum number of threads in this case is 65, 536 * `blockDim.x`
- Can we do better?

Improved Hello World (2/4)

Two-dimensional grid of thread blocks

```
__global__ void VectorAdd(const float* u,
                          const float* v, float* w) {
    int idx = threadIdx.x + blockDim.x *
           (blockIdx.x + blockIdx.y * gridDim.x);
    w[idx] = u[idx] + v[idx];
}
```

What if...

...our vectors are of length 7, and we start 2×2 blocks of size 2?

Range checking required!

Improved Hello World (3/4)

Two-dimensional grid of thread blocks with range checking

```
__global__ void VectorAdd(int n, const float* u,
                                   const float* v, float* w) {
    int idx = threadIdx.x + blockDim.x *
          (blockIdx.x + blockIdx.y * gridDim.x);
    if (idx < n) {
       w[idx] = u[idx] + v[idx];
    }
}
```

- The kernel requires information about the vector size – `n`.
- Surplus threads will calculate their index and exit immediately

Improved Hello World (4/4)

Alternative implementation

```
__global__ void VectorAdd(int n, const float* u,
                                 const float* v, float* w) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   int thread_count = gridDim.x * blockDim.x;
   for (; idx < n; idx += thread_count) {
      w[idx] = u[idx] + v[idx];
   }
}
```

- Each thread can process more than one vector element
- Several thousand of threads are required to keep the GPU busy
- More threads impose additional scheduling overhead, thus this approach might actually be faster

Warps

CUDA threads are executed in groups called warps

- Threads in a block compose consecutive warps
- SM scheduler decides which warp to execute next
- Threads in a warp are executed in a lock-step, i.e. all execute the same operation concurrently
- Warps consist of 32 threads; this *might* change in future hardware (but don't hold your breath)
- A built-in variable warpSize is available to the kernels
- *Warp-awareness is essential to write highly-optimised code*

To be precise, warps are made up by threads with consecutive global indices (see below).

Kernel launch full syntax

```
dim3 BLOCK_DIM(XBDIM,YBDIM,ZBDIM);
dim3 GRID_DIM(XGDIM,YGDIM,ZGDIM);
int shared_memory_size = ....;
....
CudaKernelName<<<GRID_DIM,BLOCK_DIM,shared_memory_size>>>(...);
```

- Both grids and blocks can be 1D, 2D or 3D; different constructors for the `dim3` object will be invoked according to the declaration in the source code;
- The shared memory size is optional; it can be left out when the kernel does not use explicitly the shared memory, or when the shared declaration in the kernel is static;
- The indexing within the `dim3` structure has the `x` dimension changing more rapidly, then the `y`, then the `z`.

- Each thread has a unique index along each dimension in its block; each block has a size given by the product of its three dimensions (and limited to a relatively small number like 1024, depending on the device model):

```
int tx= threadIdx.x; int ty= threadIdx.y;
int tz= threadIdx.z;
int blockSize = blockDim.x*blockDim.y*blockDim.y;
```

- Each thread also has a unique combined index within its block;

```
int tid_in_block = threadIdx.x + threadIdx.y*blockDim.x +
          threadIdx.z*blockDim.x*blockDim.y;
```

- For a 2D block, we have `blockDim.z==1`, hence `threadIdx.z==0` and the previous expression reduces to

```
int tid_in_block = threadIdx.x + threadIdx.y*blockDim.x;
```

- For a 1D block, we also have `threadIdx.y==0` and the previous expression reduces to

```
int tid_in_block = threadIdx.x;
```

Indexing of a thread

- Each block has a unique index along each dimension in its grid, as well as a unique global index; the maximum size and number of blocks in a grid is much larger than the number of threads in a block:

```
long long int bx= blockIdx.x, by= blockIdx.y,
              bz= blockIdx.z;
long long int blockId = blockIdx.x //1D
        + blockIdx.y * gridDim.x //2D
        + gridDim.x * gridDim.y * blockIdx.z; //3D
```

- Each thread also has a unique global index obtained from the global block index plus the thread index within the block;

```
long long int global_tid = tid_in_block
                         + blockId*blockSize;
```

- For example, with a 1D grid of 2D blocks, we have

```
long long int global_tid = threadIdx.x
       + threadIdx.y*blockDim.x + blockIdx.x*gridDim.x;
```

Choosing the Optimal Block Size

Performance may depend highly on the block size:

- It is not easy to determine and highly problem-dependent
- Too small blocks limit the device utilisation
- Too large blocks limit the number of blocks concurrently executed on SM, which may also lead to reduced performance

So how many threads should I use per block?

- Sometimes it is enforced by an algorithm
- Shoul be a multiple of a warp size (32)
- Good initial guesses for 1D blocks: 192, 256, 384
- Good initial guesses for 2D blocks: 16×16, 32×8
- Just try it out and see if you can make it go faster!

Summary

- GPU kernels are defined as C global functions prefixed with `__global__` keyword
- GPU kernels are called using the `<<<...>>>` syntax
- GPU threads are grouped into blocks, each block is assigned to an SM and executed independently
- The maximum thread block size is relatively small (1,024)
- The number of blocks can be very large, although may require more complex indexing
- GPU threads are executed in warps of typically 32 threads
- The block size may have a significant impact on the kernel performance
- The optimal block size is difficult to determine, although some good initial guesses are known