GPGPU Programming

Donato D'Ambrosio

Department of Mathematics and Computer Science
Cubo 30B, University of Calabria, Rende 87036, Italy
mailto: donato.dambrosio@unical.it
homepage: http://www.mat.unical.it/~donato

Academic Year 2020/21

Table of contents

Scalable Parallel Execution

Scalable Parallel Execution

Cuda Thread Organization

- CUDA threads:
  - Execute different instances of the same kernel function;
  - Use the thread index to access data element to process.
- In general, a grid is a three-dimensional array of blocks, and each block is a three-dimensional array of threads.
- The grid is defined when the kernel is launched by appending `<<<number_of_blocks, block_size>>>` to the kernel name (before the kernel parameter list):

```
dim3 block_size(256, 1, 1);
dim3 number_of_blocks(ceil(n/(float)block_size.x), 1, 1);
vecAddKernel<<<number_of_blocks, block_size>>>(...);
```

- Both `number_of_blocks` and `block_size` parameters are of type `dim3`, which is a C struct with three unsigned integer fields: x, y, and z.
- The programmer can use fewer than three dimensions by setting the size of the unused dimensions to 1.

Cuda Thread Organization

- For convenience, CUDA C allows to use an integer expression instead of `dim3` for 1D grids and blocks.
- In the example

```
dim3 block_size(256, 1, 1);
dim3 number_of_blocks(ceil(n/(float)block_size.x), 1, 1);
vecAddKernel<<<number_of_blocks, block_size>>>(...);
```

  the number of threads per block is fixed to 256, while the number of blocks is a function of both the data size (`n`) and the block size (`block_size`) in order to have as many threads as the data elements to be processed.
  - If `n` is 1000, the grid will consist of 4 blocks, each of 256 threads, for a total of 1024 threads;
  - If `n` is 4000, the grid will have 16 blocks, and so on.
- Once `vecAddKernel` is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.

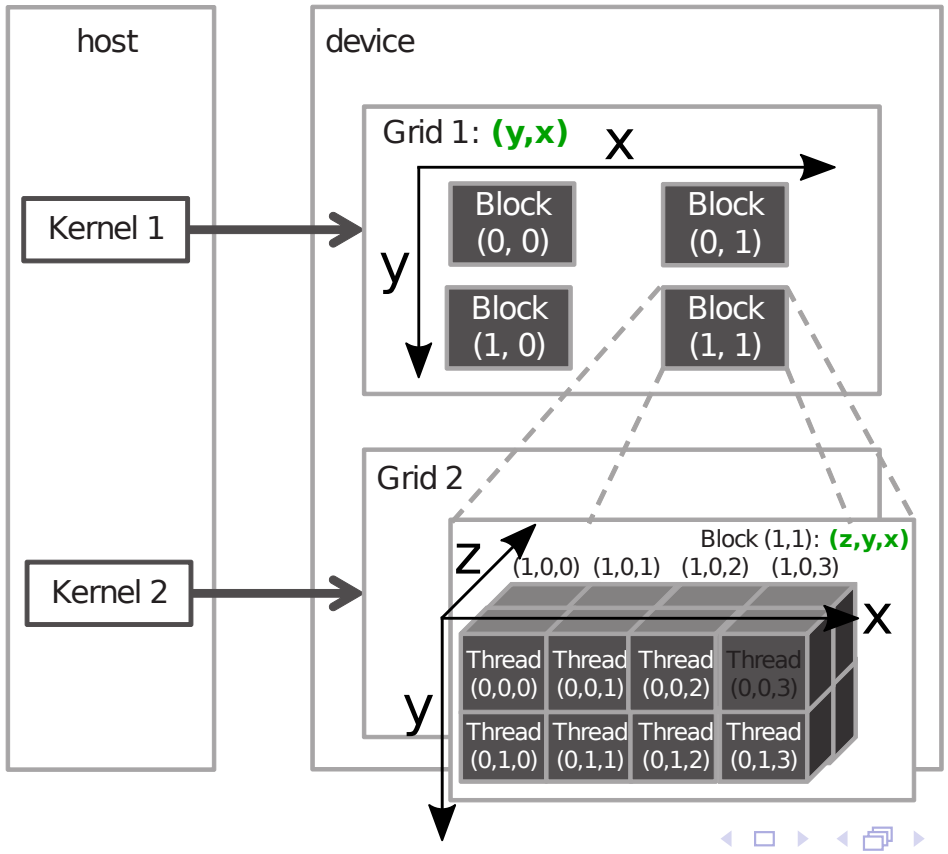
Cuda Thread Organization

- The grid size is limited to 65,536 blocks in each dimension.
- The block size is limited to a total of 1024 threads, with flexibility in distributing these elements into the three dimensions. For instance:
  - `block_size(512, 1, 1)`, `block_size(8, 16, 4)`, and `block_size(32, 16, 2)` are allowed values,
  - `block_size(32, 32, 2)` is not allowed because the total number of threads (i.e., 2048) would exceed 1024.
- All threads in a block share the same `blockIdx.x`, `blockIdx.y`, and `blockIdx.z` values.
- Note that, in the grid of blocks,
  `blockIdx.x` ranges in `[0, gridDim.x-1]`
  `blockIdx.y` ranges in `[0, gridDim.y-1]`
  `blockIdx.z` ranges in `[0, gridDim.z-1]`.

Cuda Thread Organization

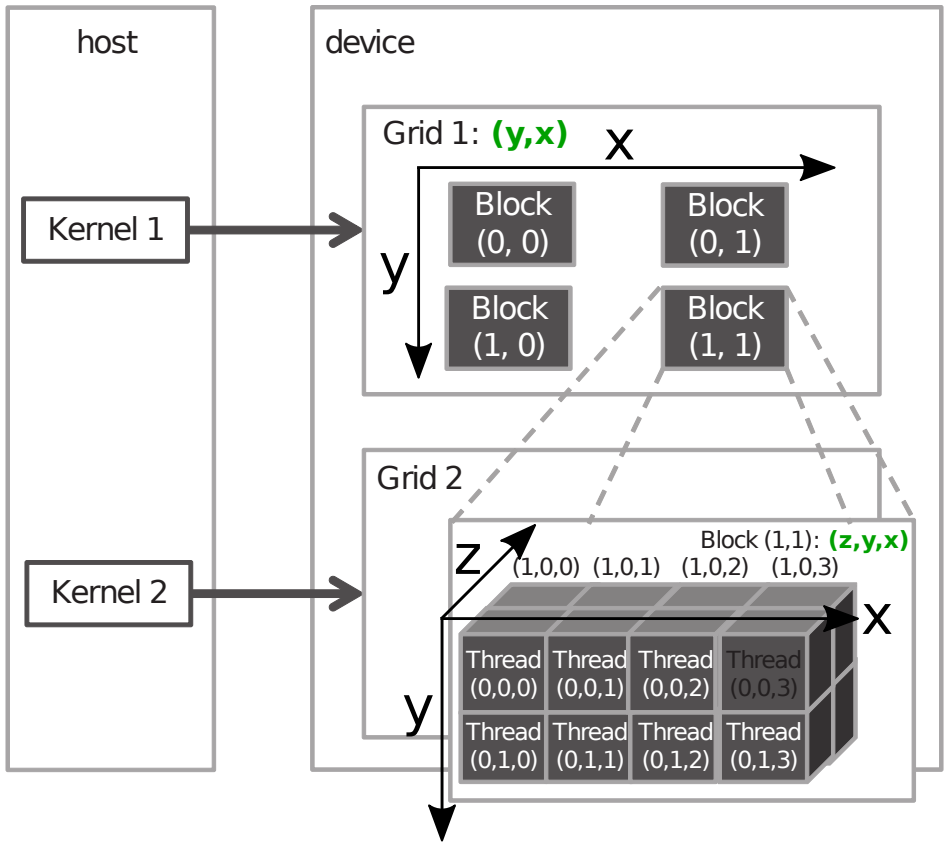
- An example grid is the following

```
dim3 block_size(4, 2, 2); //(x, y, z)
dim3 number_of_blocks(2, 2, 1); //(x, y, z)
cudaKernel<<<number_of_blocks, block_size>>>(...);
```



Cuda Thread Organization

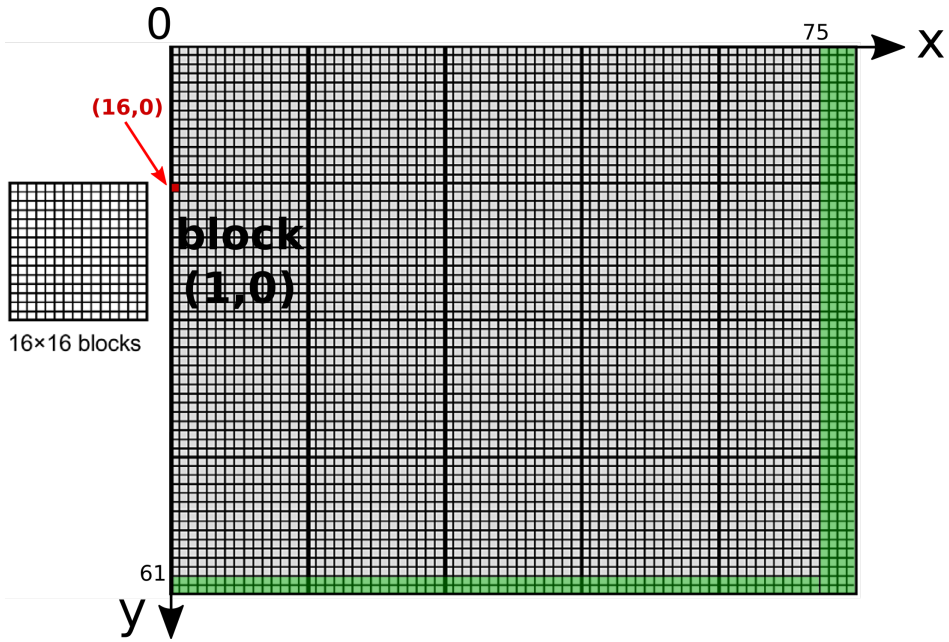
Note that blocks were labeled as `(blockIdx.y, blockIdx.x)`, e.g., Block(1,0) has `blockIdx.y=1` and `blockIdx.x=0`[1].



[1]The authors think this is better to illustrate the mapping of thread coordinates into data indexes in accessing multidimensional data!

Mapping Threads to Multidimensional Data

- 2D grids of 2D blocks are often convenient for 2D raster data.
  - Let consider a 76×62 picture (76 pixels along x, 62 pixels along y).
  - If we use a 16×16 block we need 5 blocks in the x direction and 4 blocks in the y direction, resulting in 5×4=20 blocks.



  - The element processed by thread(0,0) of block(1,0) is given by:

```
(blockIdx.y*blockDim.y+threadIdx.y, blockIdx.x*blockDim.x+threadIdx.x)=(16,0)
```

Mapping Threads to Multidimensional Data

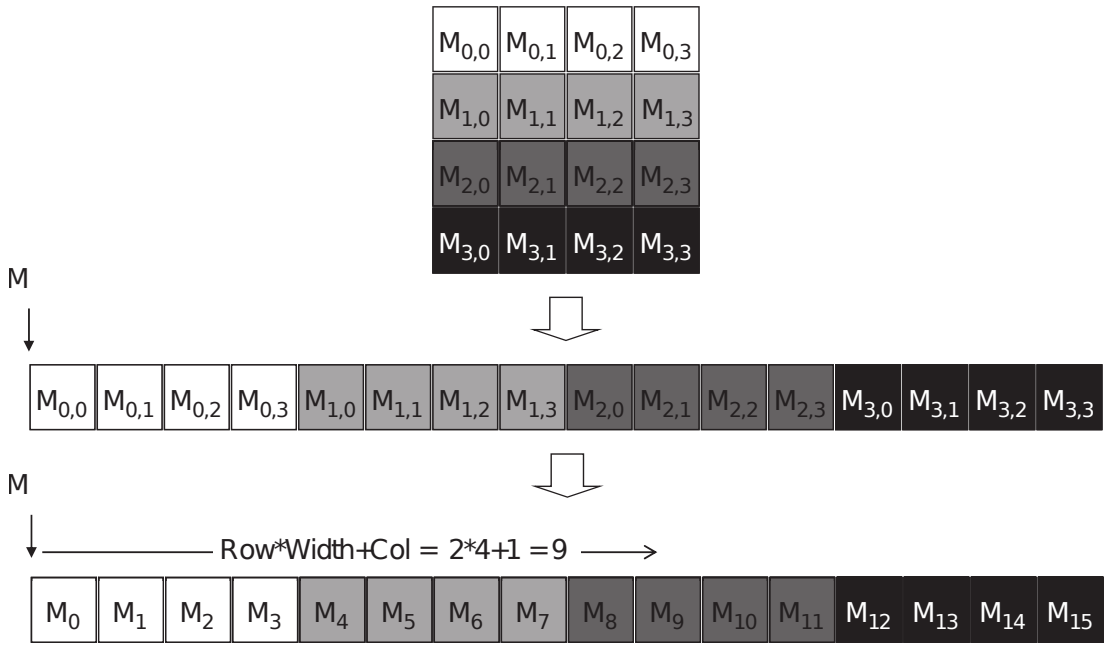
- If `m` and `n` are the number of pixels in the `x` and `y` directions, the following code can be used to launch a 2D kernel to process the image

```
dim3 number_of_blocks(ceil(m/16.0), ceil(n/16.0), 1);
dim3 block_size(16, 16, 1);
colorToGrey<<<number_of_blocks,block_size>>>(d_Pin,d_Pout,m,n);
```

- Before commenting the kernel code, we need to understand how to access data in global memory, since data must be stored as linear buffer, so that a single index must be used (this is due to the fact that CUDA C is compliant with a C standard prior to the C99).
- A two-dimensional array can be linearized in *row-major* order, by placing the rows one after another into the memory space.

Mapping Threads to Multidimensional Data

- CUDA C represents two-dimensional arrays in *row-major* order, by placing the rows one after another into the linear memory space.



- The linearized index for M in row j and column i is

$$j * 4 + i$$

The $j * 4$ term skips all elements of the rows before row j. The i term represents an offset on the row to the right element.

Mapping Threads to Multidimensional Data

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGrey(unsigned char* Pout, unsigned char* Pin, int width,
    int height) {
int Col = threadIdx.x + blockIdx.x * blockDim.x;
int Row = threadIdx.y + blockIdx.y * blockDim.y;
if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

Image Blur: A More Complex Kernel

- Blurring smooths out the variation of pixel values by updating the pixel values with a weighted sum of the surrounding pixel values (convolution).
- We consider the average value of the N×N patch of pixels surrounding, and including, our target pixel. To keep the algorithm simple, we will not consider the distance from the target pixel, which is common in a convolution blurring such as Gaussian blur.

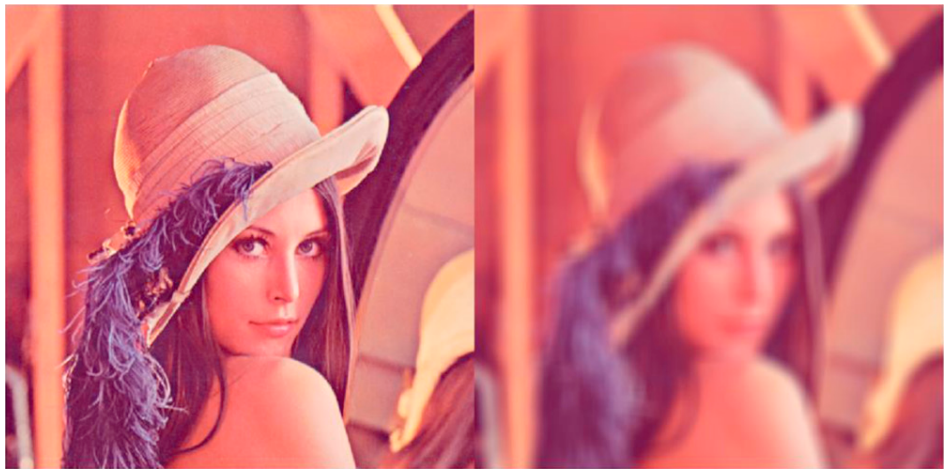


Image Blur: A More Complex Kernel

- Here an example of patch

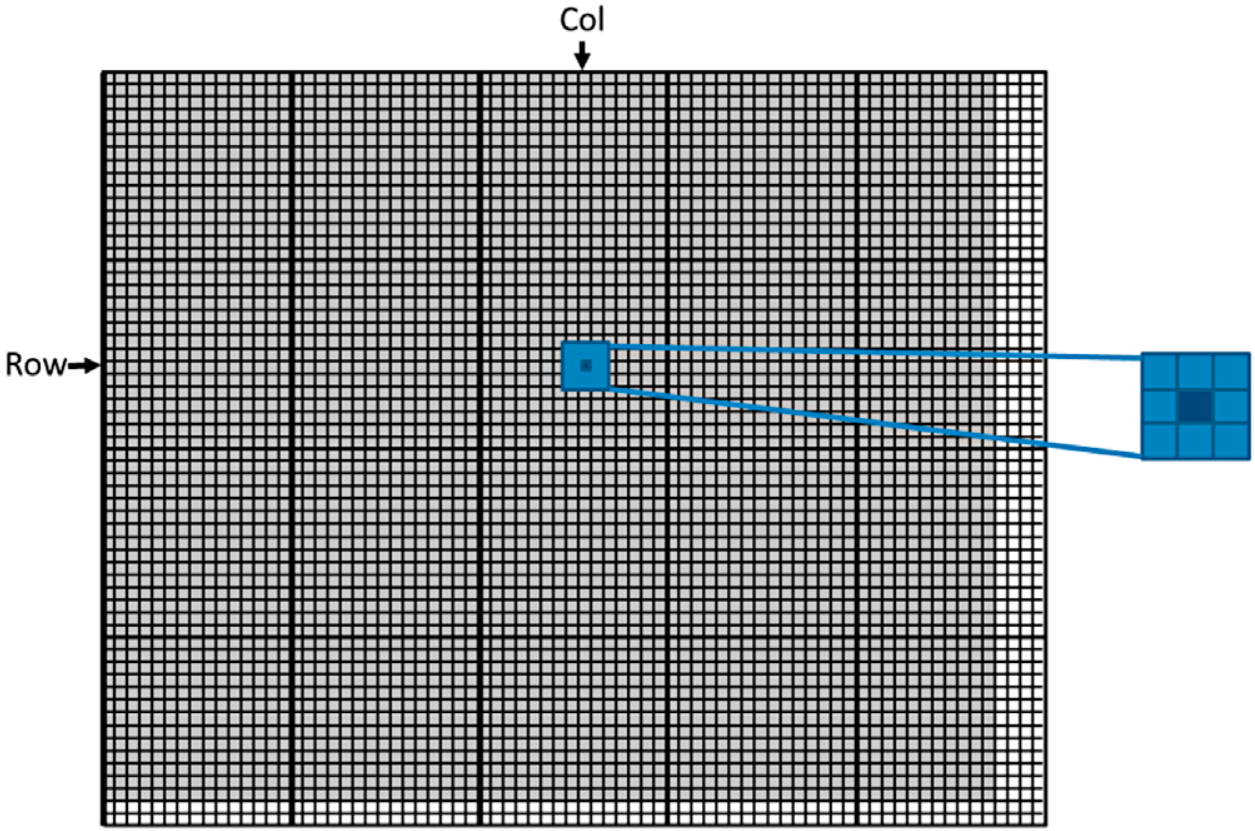
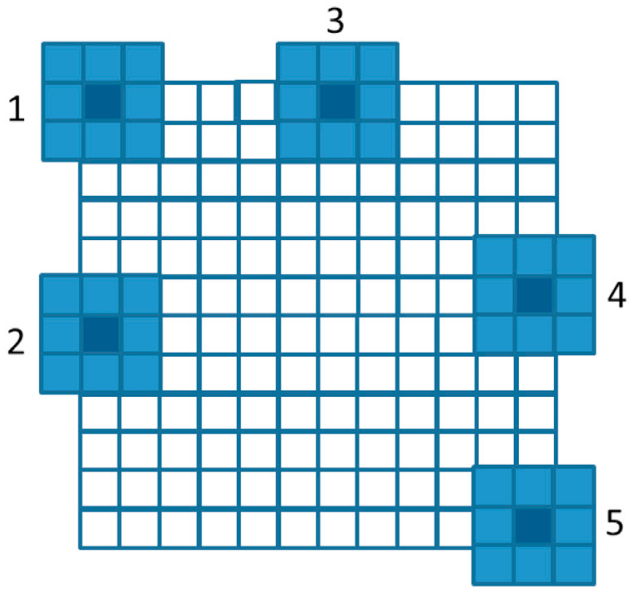


Image Blur: A More Complex Kernel

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
    {
  int Col  = blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
1.    int pixVal = 0;
2.    int pixels = 0;

    // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.    for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
     {

5.        int curRow = Row + blurRow;
6.        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
7.        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {

8.          pixVal += in[curRow * w + curCol];
9.          pixels++; // Keep track of number of pixels in the avg
          }
        }
      }

    // Write our new pixel value out
10.  out[Row * w + Col] = (unsigned char)(pixVal / pixels);
  }
 }
```

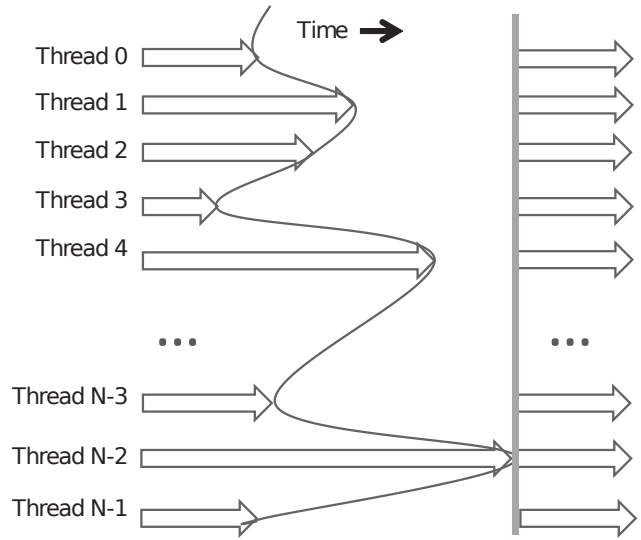
Image Blur: A More Complex Kernel

- How to mange boundary pixels?



Synchronization and Transparent Scalability

- CUDA allows threads in the same block to coordinate their activities by using a barrier synchronization function `__syncthreads()`

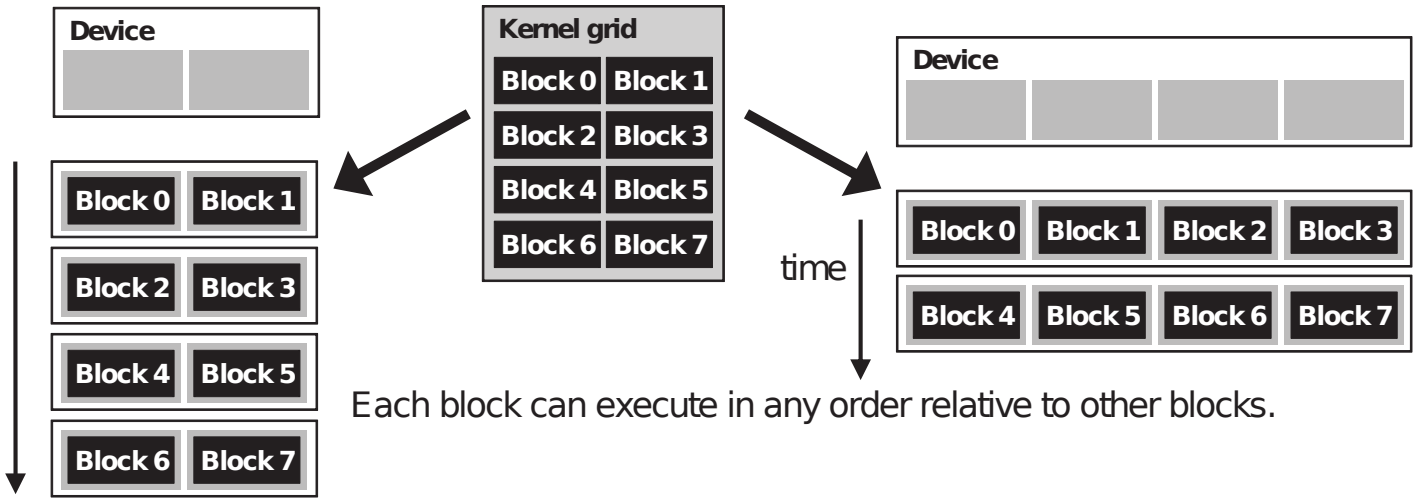


- Synchronization is not possible among threads of different blocks[2].

[2] Threads in a block can share the resources needed for the sync since they run on a specific SM (Streaming Multiprocessor), while threads of different blocks can not. Using the global memory to share resource for all the running threads could be inefficient and consume too much resources.

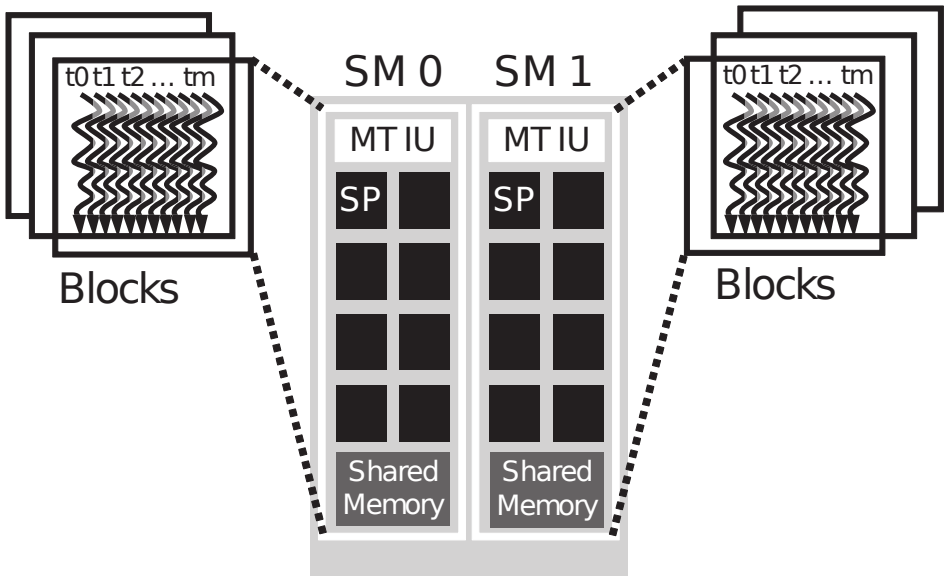
Synchronization and Transparent Scalability

- By not allowing threads in different blocks to perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other because none of them need to wait for each other. This flexibility enables scalable implementations (transparent scalability).
  - In a low-cost system with only a few execution resources, one can execute a small number of blocks simultaneously.
  - In a high-end implementation with more execution resources, one can execute a large number of blocks simultaneously.



Resource Assignment

- Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads. As discussed, these threads are assigned to execution resources on a block-by-block basis.
- The execution resources are organized into Streaming Multiprocessors (SMs). Multiple thread blocks can be assigned to each SM.



Resource Assignment

- Each device sets a limit on the number of blocks that can be simulatneously assigned to each SM.
  - For instance, let us consider a CUDA device that may allow up to 8 blocks to be assigned to each SM.
  - In case of shortage of resources needed for the simultaneous execution of 8 blocks, CUDA automatically reduces the number of blocks assigned to each SM until their combined resource usage falls below the limit.
- The number of blocks that can be actively executed in a CUDA device is therefore limited. Most grids contain many more blocks than those that coulb be sismulaneosly processed by the avaliable SMs. In that cases, the runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as previously assigned blocks complete execution.

Resource Assignment

- One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled.
- It takes hardware resources (built-in registers) for SMs to maintain the thread and block indexes and track their execution status. Therefore, each generation of hardware sets a limit on the number of blocks and number of threads that can be assigned to an SM.
  - For instance in the Fermi architecture, up to 8 blocks and 1536 threads can be assigned to each SM. This could be in the form of 6 blocks of 256 threads each, 3 blocks of 512 threads each, and so on.
  - If a CUDA device has 30 SMs, and each SM can accommodate up to 1536 threads, the device can have up to 46,080 threads simultaneously residing in the CUDA device for execution.

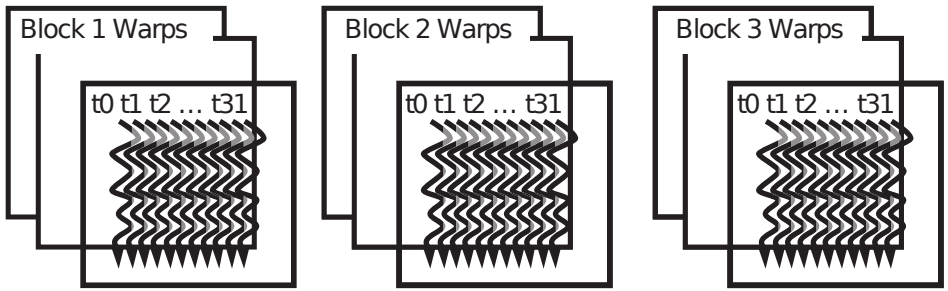
Querying Device Properties

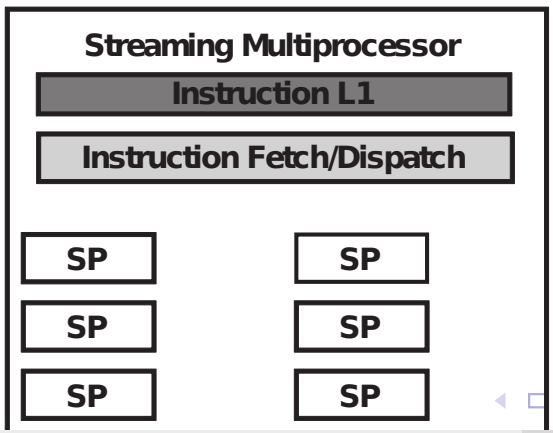
- In CUDA C, a built-in mechanism exists for a host code to query the properties of the devices available in the system.
- The CUDA runtime system (device driver) API function `cudaGetDeviceCount` gives the number of available CUDA devices in the system, while `cudaGetDeviceProperties` gets device info:

```
int dev_count; cudaDeviceProp dev_prop;
cudaGetDeviceCount(&dev_count);
for (int i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&dev_prop, i);
  printf("... %d ...\n", dev_prop.multiProcessorCount);
  printf("... %d ...\n", dev_prop.maxThreadsPerBlock);
  printf("... %d ...\n", dev_prop.maxThreadsDim[0];
  printf("... %d ...\n", dev_prop.maxThreadsDim[1];
  printf("... %d ...\n", dev_prop.maxThreadsDim[2];
  printf("... %d ...\n", dev_prop.maxGrisSize[0];
  printf("... %d ...\n", dev_prop.maxGrisSize[1];
  printf("... %d ...\n", dev_prop.maxGridSize[2];
  printf("... %d ...\n", dev_prop.warpSize); //See next slides
...
```

Thread Scheduling And Latency Tolerance

- In the majority of implementations, a block assigned to an SM is further divided into 32 thread units called warps.
- The size of warps is implementation-specific. The size of warps is a property of a CUDA device, which is in the warpSize field of the device query variable. The warp is the unit of thread scheduling in SMs.





Thread Scheduling And Latency Tolerance

- An SM is designed to execute all threads in a warp following the Single Instruction, Multiple Data (SIMD) model - i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp.
- These threads will apply the same instruction to different portions of the data. Consequently, all threads in a warp will always have the same execution timing.
- When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Instead, another resident warp that is no longer waiting for results will be selected for execution. This mechanism of filling the latency time of operations with work from other threads is often called latency tolerance or latency hiding.
- Warp scheduling is also used for tolerating other types of operation latencies, such as pipelined floating-point arithmetic and branch instructions.

Thread Scheduling And Latency Tolerance

- Assume that a CUDA device allows up to
  - 8 blocks;
  - 512 threads in each block;
  - 1024 threads per SM;
- What should we use 8×8, 16×16, or 32×32 thread blocks?

8×8 blocks - 64 threads per block. We will need 1024/64 = 12 blocks to fully occupy an SM. However, each SM can only allow up to 8 blocks; thus, we will end up with only 64 × 8 = 512 threads in each SM. The SM execution resources will likely be underutilized.

16×16 blocks - 256 threads per block. Each SM can take 1024/256 = 4 blocks. This number is within the 8-block limitation and is a good configuration as it will allow us a full thread capacity in each SM and a maximal number of warps for scheduling around the long-latency operations.

32×32 blocks - 1024 threads in each block. It exceeds the 512 threads per block limitation of this device.