CUDA Programming

Srini Prakash Maiya

October 21, 2024

Contents

Chapter 1

Data Parallelism

- The phenomenon in which the **computation work of different parts of a dataset can be performed independently** of each other and thus can be executed in **parallel**.
- A large problem can be decomposed into n - *smaller problems which can be executed independently.* This entails (re-)organizing the computation around the data such that the resulting computation can be executed in parallel to complete the overall job faster.
- Examples:
  - Conversion of image from RGB → Gray as visualized in 1.1. Here each $O[0], O[1], \ldots O[N-1]$ can be calculated independently.

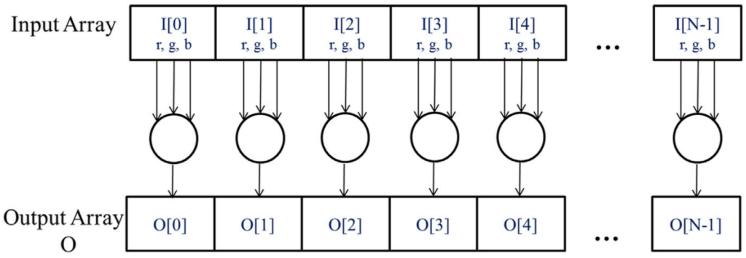
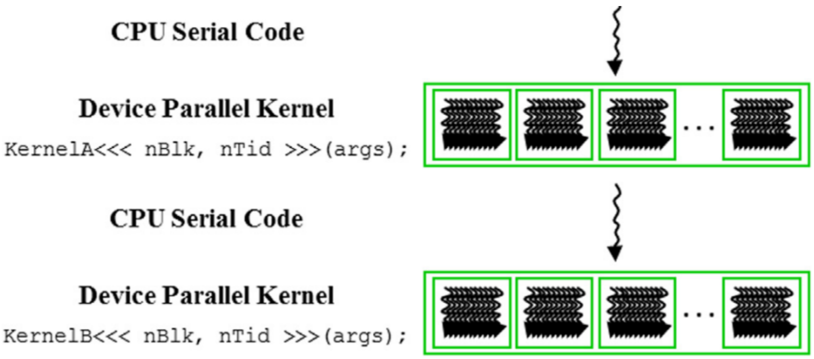


Figure 1.1: Data parallelism of RGB-to-grayscale. Each pixel can be independently converted to grayscale

1.1 CUDA program structure

- CUDA C → Extends ANSI C language with minimal new syntax and library functions.
- Enables programmers to target heterogeneous computing systems containing both CPU and GPUs.
- *host:* CPU , *device*: GPU
- Each CUDA C file can be a mixture of *host* code and *device* code.
- Simplified CUDA program execution:
  - Execution of host code (CPU serial code).
  - Call of Kernel function → A large number of threads are launched on *device* to execute kernel. (Collection of threads: *grid.*)



```
KernelA<<< nBlk, nTid >>>(args);
KernelB<<< nBlk, nTid >>>(args);
```

Figure 1.2: Simplified execution of a CUDA program with no CPU and GPU overlap

  - When all the threads of the grid finish execution, grid terminates → Execution on host continues until the next kernel is called.

- In the case of RGB → Gray conversion, each thread can be used to convert one pixel of the image to grayscale. In such case, the number of threads launched ≡ number of pixels in the image.

- The threads in GPU take very few clock cycles to generate and schedule, as compared to traditional CPU threads, which typically take thousands of clock cycles to generate and schedule.

- The suffix "`_h`" indicates the variables on host(CPU) and "`_d`" to indicate the variable resides on device(GPU).

- The error-prone regions should be surrounded with the code that catches the error condition to print it out. For example, to catch the errors occuring during the memory allocation on device using `cudaMalloc()` function:

Code 1.1: Error catching for Malloc

```c
        // Error handling while allocating 'size' bytes of memory for pointer 'A_d'. Catch the return value of cudaMalloc() in err.
        cudaError_t err = cudaMalloc((void **)&A_d, size);
        // If enum 'err' is not equal to cudaSuccess, then print the error type, error name, in which file and line number.
        if (err != cudaSuccess){
            printf("%s: %s in %s at line %d\n",cudaGetErrorName(err), cudaGetErrorString(err), __FILE__, __LINE__);
            exit(EXIT_FAILURE);
        }
```

1.2 Kernel functions and threading

- Kernel function specifies the code to be executed by all threads during the parallel phase. → Since all the threads execute the same code, CUDA C programming is an instance of *single-program multiple-data* **SPMD** parallel-programming style.

- Kernel call → launch of grid of threads. The threads are organized into two-level hierarchy.

  - Grid: Organized into array of *thread blocks* or **Blocks**.
  - Block: All blocks of the grid are of same size; each block **can contain *up to* 1024 threads**.
  - Total number of threads in each block is specified in the host code.

- The built-in variables that enable thread to distinguish itself from other are:

  - `blockDim`: *Build-in variable* specifying the number of threads in a block. Struct with 3 unsigned integer fields (x, y and z), enabling one to organize threads into one-(x), two-(x, y) or three-dimensional(x, y, z) array.
  - `blockIdx`: Struct with 3 unsigned integer fields (x, y and z). Gives all threads in a block a common block coordinate.
  - `threadIdx`: Struct with 3 unsigned integer fields (x, y and z). Gives each thread a unique coordinate within the block.

- **Unique identifier for a thread** is calculated as:
  `data_index=blockIdx.x * blockDim.x + threadIdx.x`

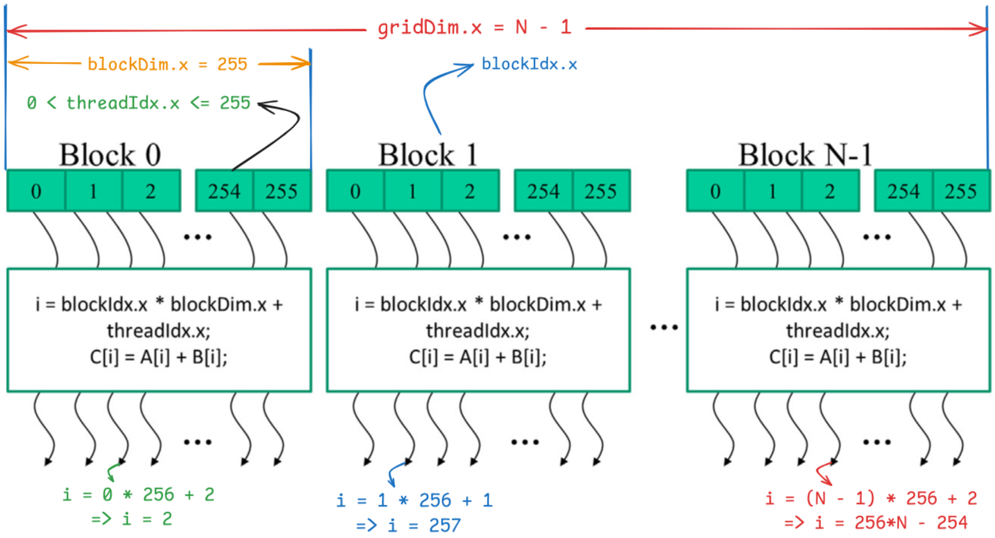


Figure 1.3: Hierarchy and organization of threads in a Grid

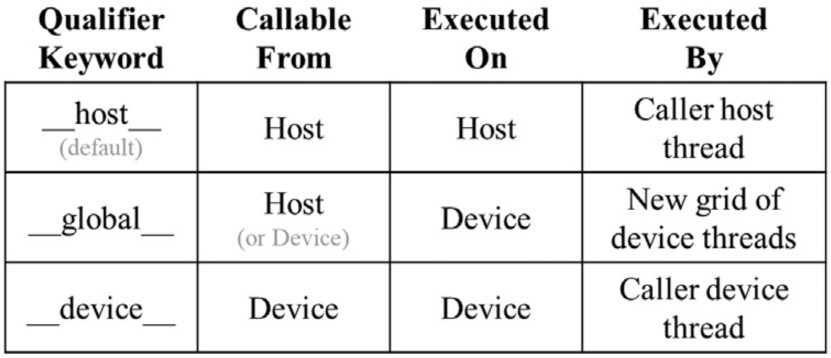
1.3 `__host__`,`__global__` and `__device__` keywords

- `__host__`: Indicates that the function being declared is a CUDA host function. Is a function that is **executed on the CPU**. By default, all functions in CUDA are *host* functions, if not specified otherwise.

- `__global__`: Indicates the function being declared is a **CUDA C function**. Such a kernel function is **executed on the device and can be called from the host**. This keyword indicates that the function is a kernel and that it **can be called to generate a grid of threads on a device**.

- `__device__`: Indicates that the function being declared is a *CUDA device* function. This function **executes only on device, can be called only from a kernel function or another device function**. The device function is executed by the device thread that calls it and does not result in any new device threads being launched.

- ***NOTE***: One can use both `__host__` and `__device__` keywords in a function declaration. $\Longrightarrow$ Two versions of the object code is compiled for same function. → One is a pure host function (call, execution) and the other is pure device function. Supports the common use case when the same function source code can be recompiled to generate device version. Ex. user-library functions.



Qualifier Keyword	Callable From	Executed On	Executed By
__host__ (default)	Host	Host	Caller host thread
__global__	Host (or Device)	Device	New grid of device threads
__device__	Device	Device	Caller device thread

Code 1.2: Declaration of a simple kernel.

```
// This kernel runs for each thread. Each thread computes the sum of A and B at specified index and saves it to C.
__global__
void vecAddKernel(float* A, float* B, float* C, int n){
    int global_threadID = blockIdx.x * blockDim.x + threadIdx.x;
    if (global_threadID < n){
        C[global_threadID] = A[global_threadID] + B[global_threadID];
    }
}
```

- The **automatic (local) variable** `data_index` in Code 1.2 is **private for each thread**. That is, if a grid launches 10,000 threads, there will be 10,000 unique versions of `data_index`, one for each thread. The **value assigned in one thread is not visible to other threads**.

- The CUDA kernel in Code 1.2 does not have an outer loop iterating over all elements sequentially, as the individual threads execute the same task for each index in parallel.

- The `if (data_index < n)` condition **cuts off the calculation when the** $number_{threads} > number_{elements}$ **in the array**. The minimum efficient thread block dimension is 32 (block size). As all vector length can not be expressed in multiples of 32, this allows the kernel to process vectors of arbitrary lengths.

1.4 Calling Kernel functions

- When the host code calls the kernel, it sets the grid and thread block dimensions via ***execution configuration parameters***. The configuration parameters are given between "<<<" and ">>>" before the traditional C argument functions.

- In Code 1.3, the thread block size (number of threads per block) is set to 256 **(line 24)**. Considering `n = 1000` elements, the grid size is ceil of (1000 / 256), which is 4 **(line 25)**.

- The *execution configuration parameters* take two arguments. Grid size → number of blocks per grid, and block size → number of threads in a block in that order **(line 28)**.

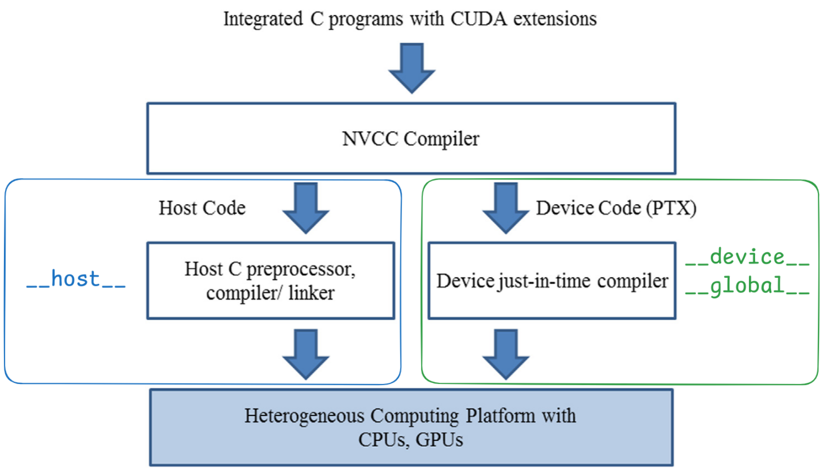
- By checking for `data_index < n`, the first 1000 threads perform the addition operation among the created 1024 threads (4 blocks * 256 threads).

- The thread blocks operate on different parts of the vector. They can be executed in arbitrary order. A small GPU with a small amount of execution resources may execute only one or two of these thread blocks in parallel. A larger GPU may execute 64 or 128 blocks in parallel. This gives CUDA kernels scalability in execution speed with hardware.

Code 1.3: Calling the kernel

```
#include <math.h>

__global__ void vecAddKernel(float *A, float *B, float *C, int n)
{
    int global_threadID = blockIdx.x * blockDim.x + threadIdx.x;
    if (global_threadID < n)
    {
        C[global_threadID] = A[global_threadID] + B[global_threadID];
    }
}

void vecAdd(float *A, float *B, float *C, int n)
{
    float *A_d, *B_d, *C_d;
    int bytes = n * sizeof(float);

    cudaMalloc(&A_d, bytes);
    cudaMalloc(&B_d, bytes);
    cudaMalloc(&C_d, bytes);

    cudaMemcpy(A_d, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, bytes, cudaMemcpyHostToDevice);

    int THREADBLOCK_SIZE = 256;
    int GRID_SIZE = (n + THREADBLOCK_SIZE - 1) / THREADBLOCK_SIZE;
    // int GRID_SIZE = (int)ceil((float)n / THREADBLOCK_SIZE);

    vecAddKernel<<<GRID_SIZE, THREADBLOCK_SIZE>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, bytes, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

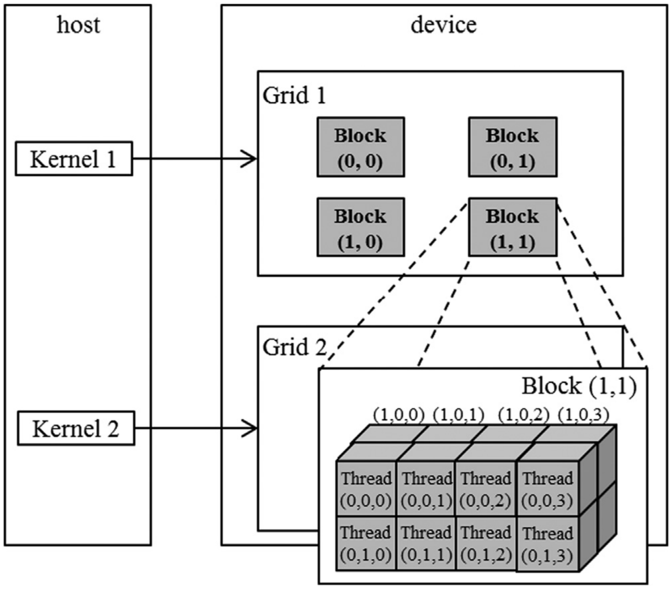
1.5 Compilation overview


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
__host__
Host C preprocessor, compiler/ linker
Device just-in-time compiler
__device__
__global__
Heterogeneous Computing Platform with CPUs, GPUs

Chapter 2

Multidimensional grids and data

The dimensions of the grid(containing blocks) and blocks(containing threads) is given by the built-in variables, `gridDim` and `blockDim` variables. These parameters are specified within the *execution configuration parameters,* <<< $\cdots$ >>> of the kernel call statement.
`<<< dimGrid, dimBlock >>>`

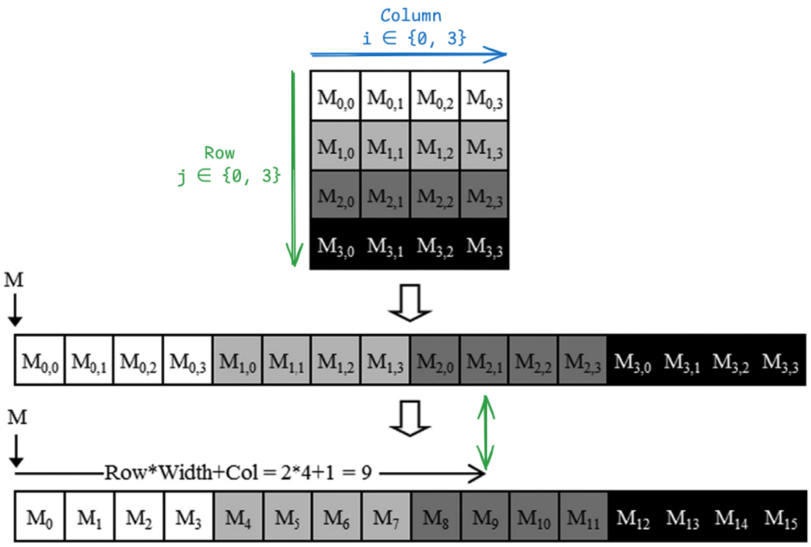
- 

- **Grid** → 3D array of blocks.
  - All blocks share the same `gridDim.x, gridDim.y, gridDim.z` values.
- **Block** → 3D array of threads.
  - All threads in a block share the same `blockIdx.x, blockIdx.y, blockIdx.z` values.
  - Total number of threads in a block is constrained to 1024 and can be distributed flexibly in the 3 dimensions.
  - `blockIdx.x` $\in \{0 \ldots$ `gridDim.x - 1`$\}$
    `blockIdx.y` $\in \{0 \ldots$ `gridDim.y - 1`$\}$
    `blockIdx.z` $\in \{0 \ldots$ `gridDim.z - 1`$\}$
- **Thread** → Has **unique identifier** in a block.
  - `threadIdx.x` $\in \{0 \ldots$ `blockDim.x - 1`$\}$
    `threadIdx.y` $\in \{0 \ldots$ `blockDim.y - 1`$\}$
    `threadIdx.z` $\in \{0 \ldots$ `blockDim.z - 1`$\}$

```
// 1D grid with 32 blocks in x direction.
dim3 dimGrid(32, 1, 1);
// 1D block with 128 threads in x direction.
dim3 dimBlock(128, 1, 1);
// Kernel call
vecAddKernel<<<dimGrid, dimBlock>>>(...);
// Shorthand convention
vecSubKernel<<<32, 128>>>(...);
```

2.1 Linearization of 2D array

The main two ways in which a 2D array can be linearized, *row-major layout* and *column-major layout*. In **row-major layout**, the same elements of the same rows are placed into consecutive locations. The rows are placed one after other consecutively in memory space. An element at jth row and ith column is indexed as $M_{j,i}$. *CUDA C uses the row-major layout.*


Column
i ∈ {0, 3}
Row
j ∈ {0, 3}
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
Row*Width+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

The formula to convert an image from color to grayscale is described as:

$$L = 0.21 * r + 0.72 * g + 0.07 * b$$

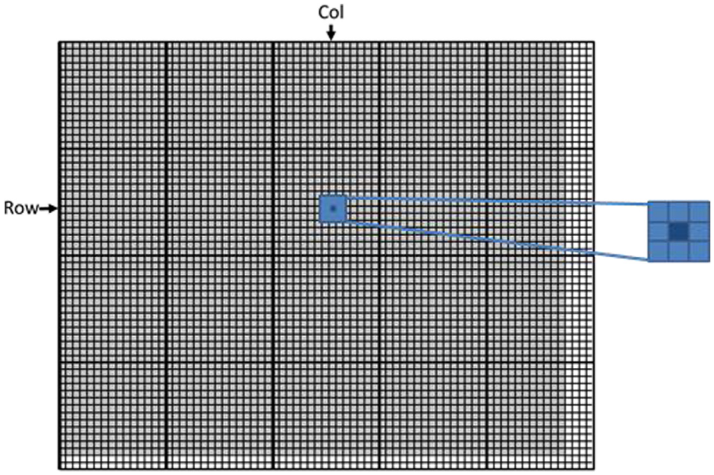
```
// The input image is encoded as unsigned chars [0, 255]
// Each pixel is 3 consecutive chars for 3 channels (RGB)
__global__
void colortoGrayscaleKernel(unsigned char* Pout, unsigned char* Pin,
                                        int width, int height) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col < width && row < height){
        // Get index of the current element of output
        int grayOffset = row * width + col;
        // RGB array has CHANNELS * elements of grayscale image.
        int rgbOffset = grayOffset * CHANNELS;
        // Red, Green and Blue values
        unsigned char r = Pin[rgbOffset];
        unsigned char g = Pin[rgbOffset + 1];
        unsigned char b = Pin[rgbOffset + 2];
        // Perform rescaling and store in the out array (Grayscale img)
```

```
            Pout[grayOffset] = 0.21f * r + 0.72f * g + 0.07f * b;
        }
    }
```

For an image of size 62 × 76, and block size in *x, y* of (16, 16), the linearized 1D index of the `Pout` pixel at thread (0, 0) and block (1, 0) is calculated as:

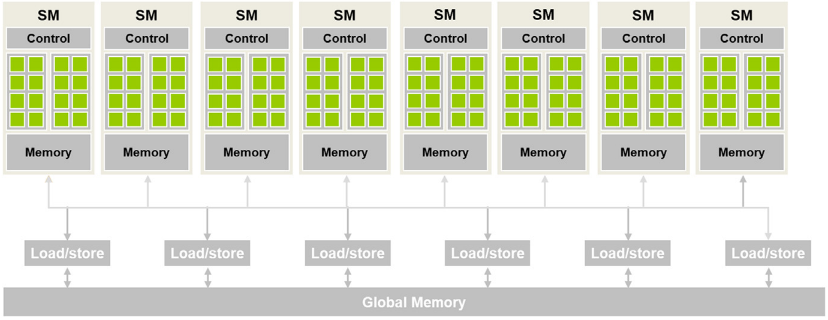
$$\begin{aligned}&\text{Pout}_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}\\ &=\text{Pout}_{1*16+0,0*16+0}=\text{Pout}_{16,0}=\text{Pout}[16*76+0]=\text{Pout}[1216]\end{aligned}$$

2.2 Image blur Kernel



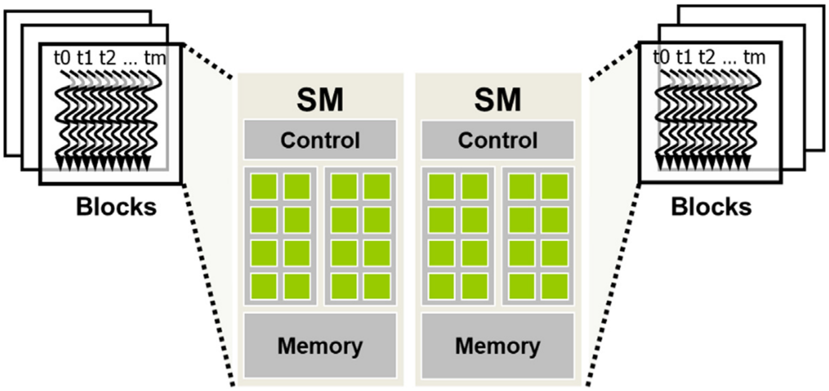
Chapter 3

Compute architecture and scheduling



- A CUDA capable GPU is organized into an array of highly threaded streaming multiprocessors(SMs).
  - Each SM → has several processing units called CUDA Cores (cores).
    - Each Core → shares control logic and memory resources.

3.1 Block scheduling



When the kernel is called, the CUDA runtime system launches a grid of threads that execute kernel code. The threads are attached to Streaming Multiprocessors (SMs) on a block-by-block basis.

- i.e, **all the threads in a block are assigned to the same SM**.
- Multiple blocks are assigned to the same SM.

- **Blocks need to reserve hardware resources** to execute. $\implies$ A **limited number of blocks can be assigned to an SM.**
  - As the **number of SMs** in a GPU is **limited**, the **total number of blocks that can be simultaneously executing on a CUDA device is limited**.
  - Most grids contain many more blocks than this number. To ensure all the blocks in a grid get executed, the runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs when previously assigned blocks complete execution.
- This fashion of assignment of threads to SMs **guarantees that the threads in the same block are scheduled simultaneously on the same SM**. This **guarantee enables the threads in the same block to interact with each other** in ways that the threads across the blocks cannot.

3.2 Synchronization and transparent scalability