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 $\rightarrow$ Gray as visualized in 1.1. Here each $O[0], O[1], \ldots O[N-1]$ can be calculated independently.

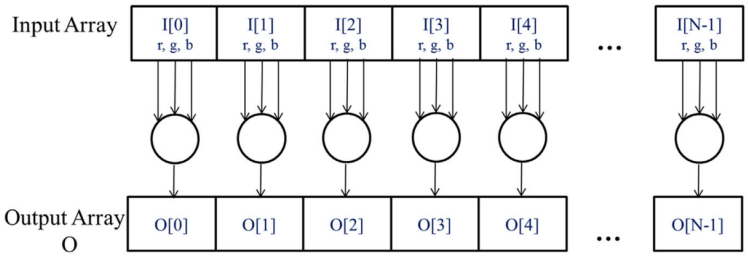


Figure 1.1: Data parallelism of RGB-to-grayscale. Each pixel can be independently converted to grayscale

1.1 CUDA program structure

- CUDA C $\rightarrow$ 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 $\rightarrow$ A large number of threads are launched on *device* to execute kernel. (Collection of threads: *grid*.)

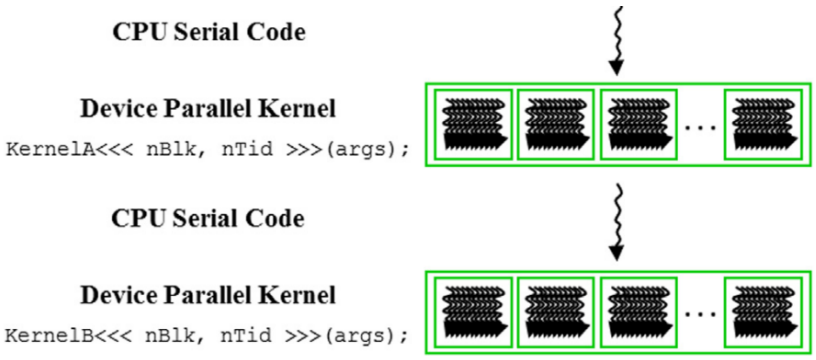


Figure 1.2: Simplified execution of a CUDA program with no CPU and GPU overlap

  - 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

```
1        // Error handling while allocating 'size' bytes of memory for pointer 'A_d'.
         ↪  Catch the return value of cudaMalloc() in err.
2        cudaError_t err = cudaMalloc((void **)&A_d, size);
3        // If enum 'err' is not equal to cudaSuccess, then print the error type,
         ↪  error name, in which file and line number.
4        if (err != cudaSuccess){
5            printf("%s: %s in %s at line %d\n",cudaGetErrorName(err),
             ↪  cudaGetErrorString(err), __FILE__, __LINE__);
6            exit(EXIT_FAILURE);
7        }
```

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`

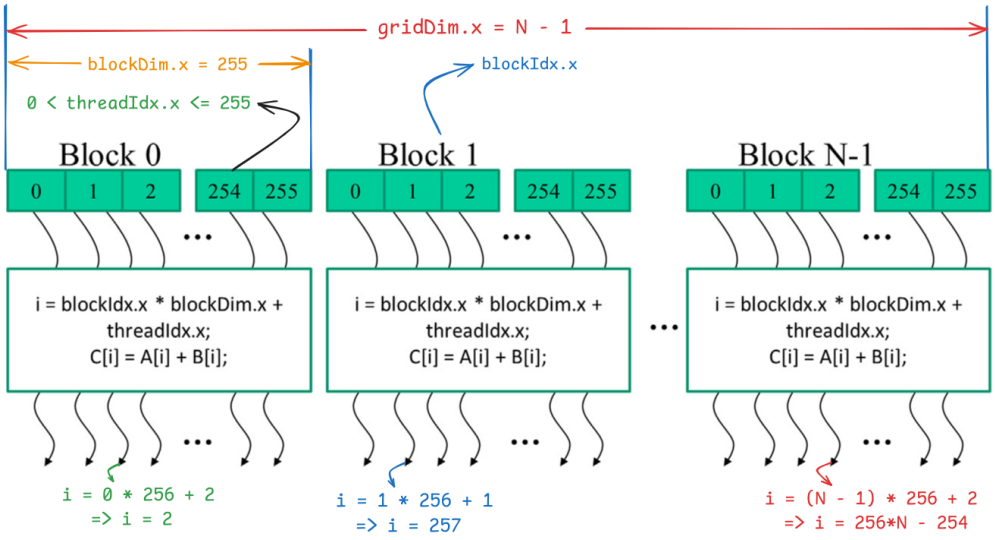


Figure 1.3: Hierarchy and organization of threads in a Grid

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. $\rightarrow$ 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.

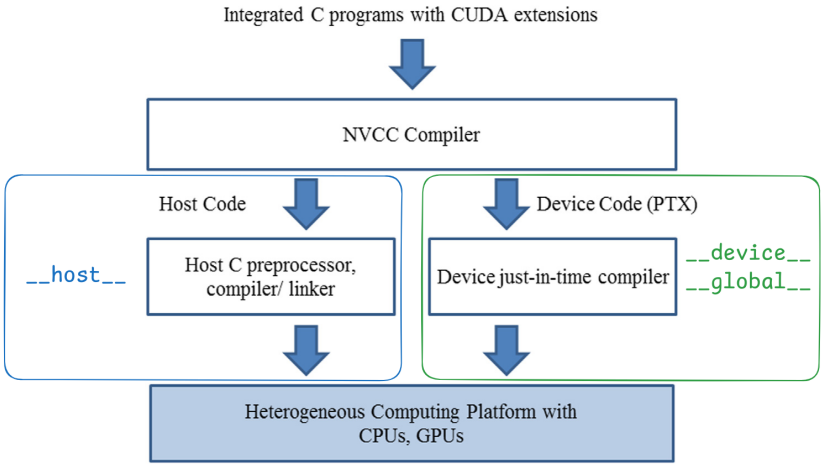
1.4 Calling Kernel functions

- 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);
}
```

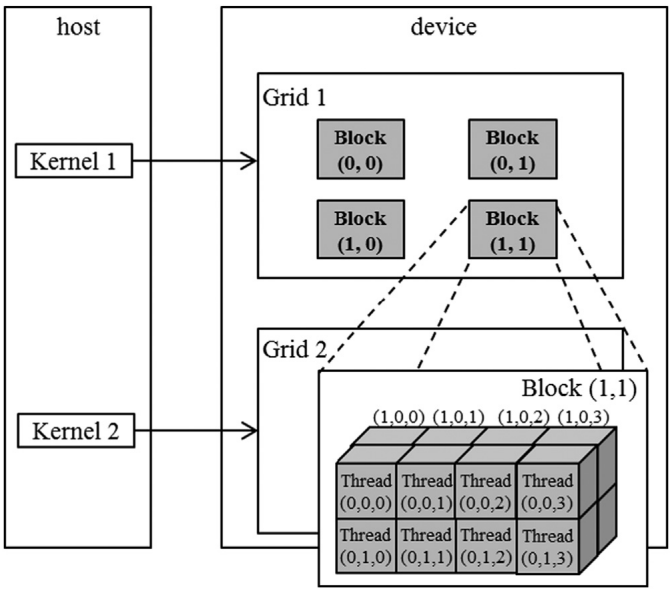
1.5 Compilation overview



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 >>>`

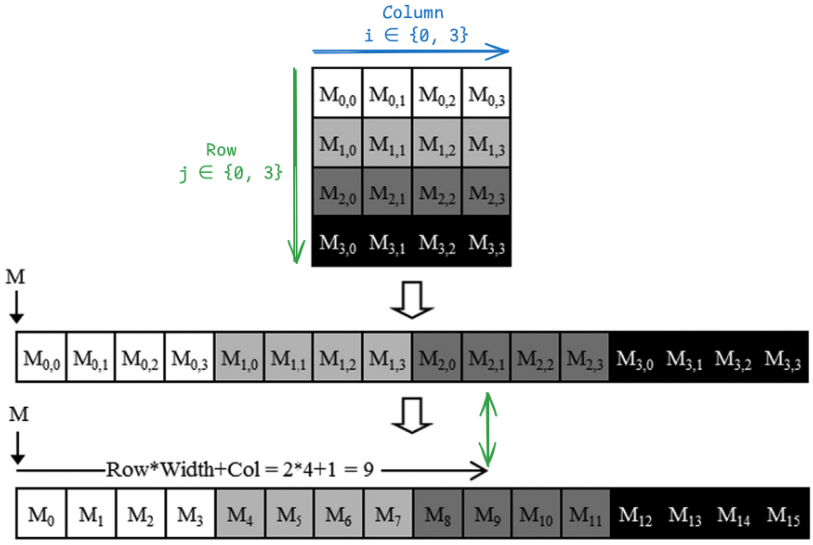


- **Grid** → 3D array of blocks.
  - All blocks share the same `gridDim.x, gridDim.y, gridDim.z` values.
- **Block** → 3D array of threads.
  - All threads in a block share the same `blockIdx.x, blockIdx.y, blockIdx.z` values.
  - Total number of threads in a block is constrained to 1024 and can be distributed 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.*



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

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

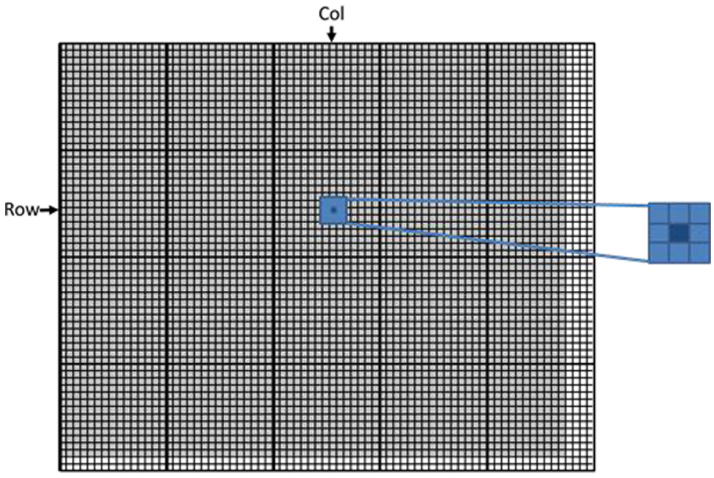
```
    // The input image is encoded as unsigned chars [0, 255]
    // Each pixel is 3 consecutive chars for 3 channels (RGB)
    __global__
    void colortoGrayscaleKernel(unsigned char* Pout, unsigned char* Pin,
                                        int width, int height) {
        int col = blockIdx.x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

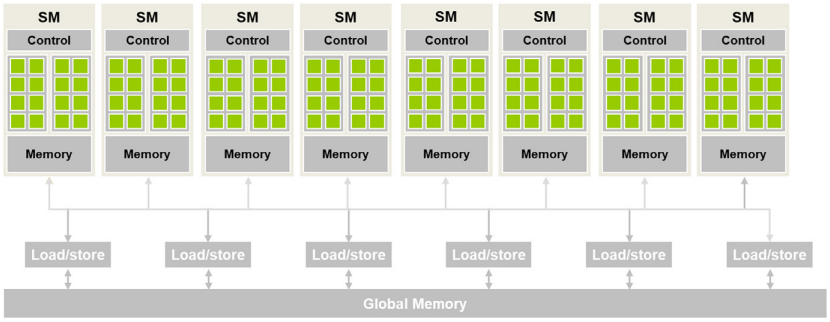
$$\begin{aligned} &\text{Pout}_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} \\ &= \text{Pout}_{1*16+0,0*16+0} = \text{Pout}_{16,0} = \text{Pout}[16 * 76 + 0] = \text{Pout}[1216] \end{aligned}$$

2.2 Image blur Kernel



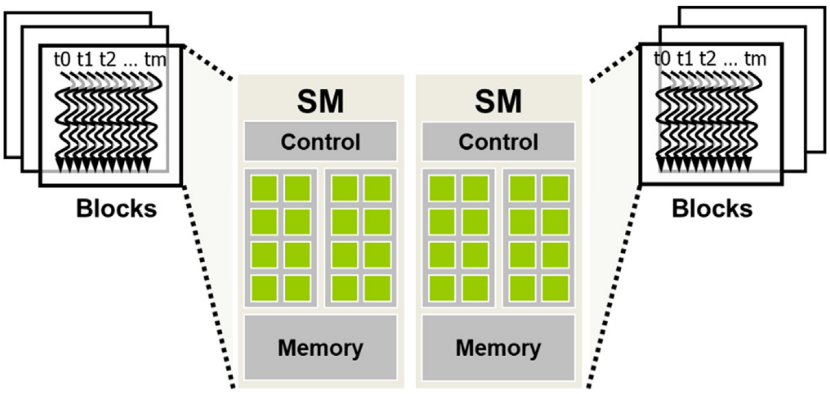
Chapter 3

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.

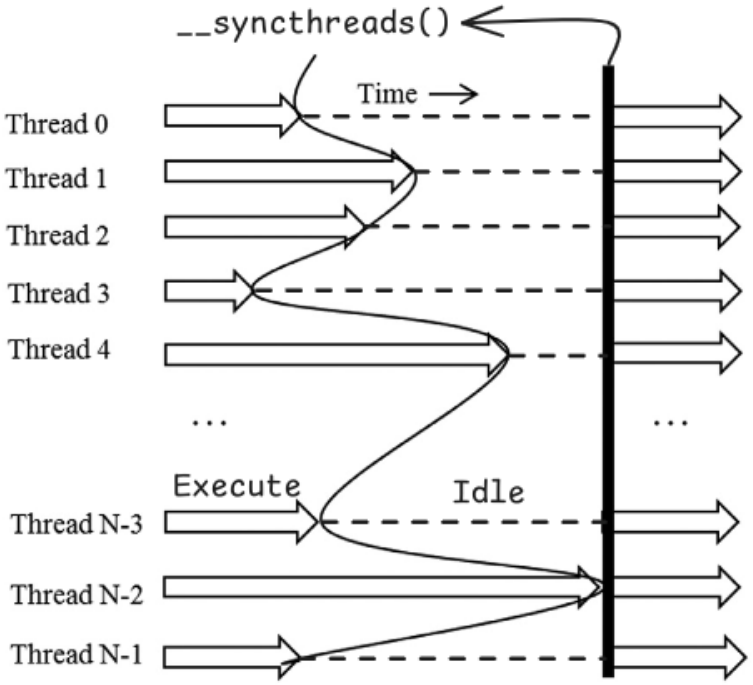
- i.e, **all the threads in a block are assigned to the same SM**.
- Multiple blocks are assigned to the same SM.

- **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

3.2.1 __syncthreads()

- The threads from the same block can coordinate their activities using the barrier synchronization function `__syncthreads()`.
- If a thread calls `__syncthreads()`, the thread will be held at the program location of the call, until all threads in the block reach that location. Ensures all the threads have completed a phase of their execution before moving to the next phase.



- Among the *N threads* in the image above, the *N-2* nd thread is the last to execute. But all the other threads wait for the *N-2* nd thread to finish the execution, thus ensuring the same program location / state among all the threads in the block.
- With barrier synchronization, "no one is left behind."
- In CUDA, if a `__syncthreads()` statement is present, it **must be executed by all threads in a block**.

- `__syncthreads()` in `if` block: The `__syncthreads()` is **either executed by all the threads or none of the threads** in the block.
- `__syncthreads()` in `if-else` block: if each path has `__syncthreads()` statement, then either all the threads execute the `if-then` path or `else` path. The two
- `__syncthreads()` in `if block` are different barrier synchronization points. Since not all threads in a block are guaranteed to execute the same barrier, the code violates the rules for using `__syncthreads()` and will result in undefined execution behavior.

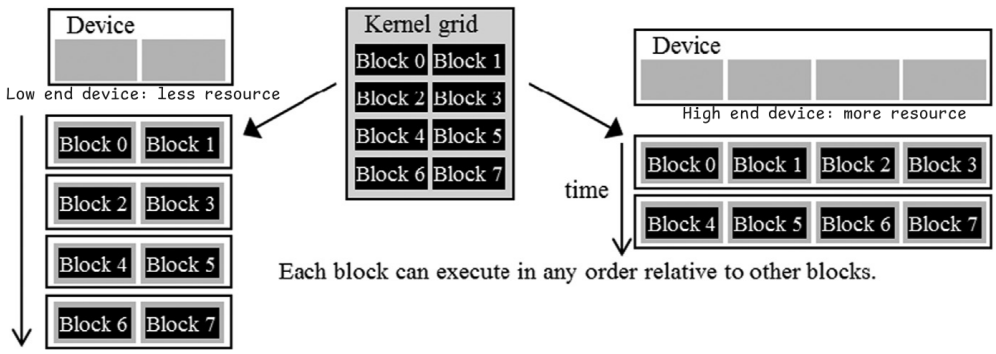
```
1   void incorrect_barrier_example(int n){
2       ...
3       if (threadIdx.x % 2 == 0) {
4           ...
5           // All the even threads wait here.
6           __syncthreads();
7       }
8       else {
9           ...
10          // All the odd threads wait here.
11          __syncthreads();
12      }
13  }
```

- Barrier synchronization **imposes execution constrains on the threads within a block. The threads should operate in close time proximity with each other to avoid excessively long waiting times.**
- The **system needs to make sure that all the threads involved in synchronization should have the access to necessary resources to arrive at the barrier**. Otherwise, a thread may never arrive at barrier synchronization point, which **can cause a deadlock**.

  $\Longrightarrow$ All threads of the block should be assigned to the same SM, but should be assigned simultaneously to the SM (whole block executes once). i.e, a block can begin execution only when the runtime system has secured all the resources needed by all threads in the block to complete execution. Ensuring the time proximity of all threads in a block preventing excessive or indefinite waiting time during barrier synchronization.

3.2.2 Transparent scalability

- As the threads from different blocks can't communicate with one another, CUDA runtime system can execute the blocks in any order relative to each other. This flexibility enables scalable implementations.



- The ability to execute the same application code with a wide range of speeds allows the production of a wide range of implementations according to the cost, power, and performance requirements of different market segments.

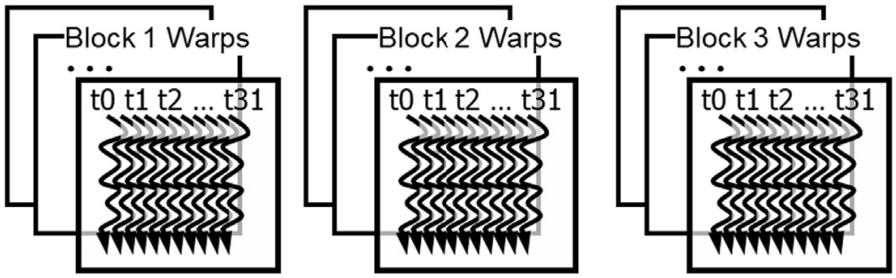
- The ability to execute the same application code on different hardware with different amounts of execution resources is referred to as ***transparent scalability***, which reduces the burden on application developers and improves the usability of applications.

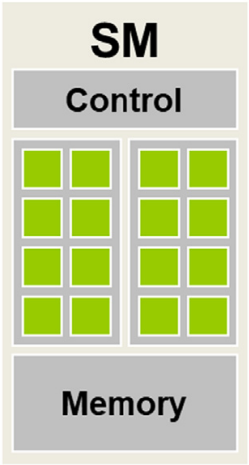
3.3 Warps and SIMD hardware

- One should assume that in a block, threads can execute in any order w.r.t to each other.
- In algorithms with phases, barrier synchronization should be used between all the threads in a block, whenever the threads must move from one phase to the other. The **correctness of executing a kernel *shouldn't* depend on any assumption that threads will synchronize without the use of barrier synchronization**.

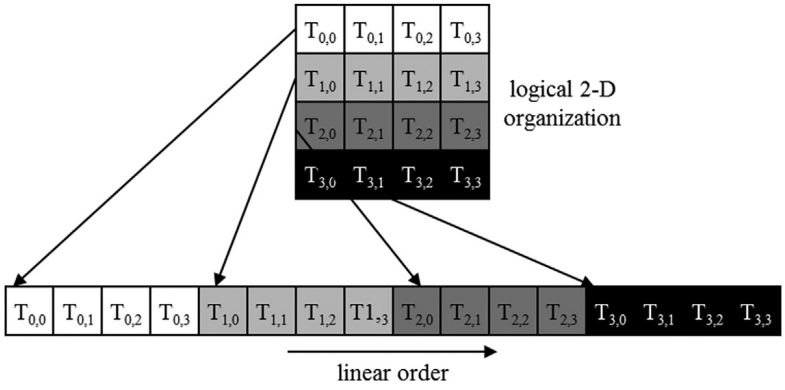
3.3.1 Warps

- A block is further divided into 32-thread units called warps. *The size of the warps can vary in the future.* A warp is the unit of thread scheduling in SMs.



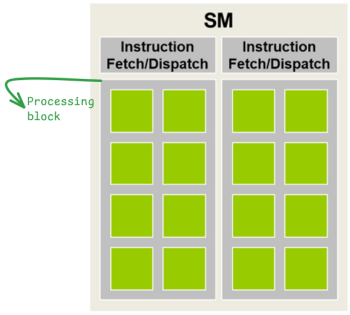


- Block 1, Block 2, Block 3 - all are assigned to the single SM. Each block is further divided into warps for scheduling purposes.
  - Each warp consists of 32 consecutive `threadIdx` values: threads 0 through 31 form the first warp, threads 32 through 64 the second warp etc.
  - If each block has 256 threads, a block has $256/32 = 8$ warps.
  - With 3 blocks in the SM, there are $8 \times 3 = 24$ warps in the SM.
- Block $\rightarrow$ Warp: Index z axis, then y then x.



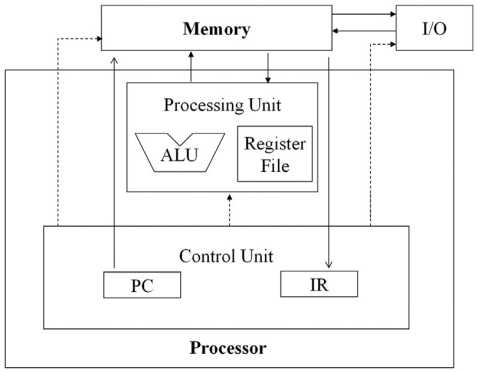
- For 2D grid: $T_{y,x}$, y being `threadIdx.y` and x being `threadIdx.x`, a 2D grid is linearized by placing all the threads whose `threadIdx.y` is 0 then `threadIdx.y` is 1 and so on. Threads with the same threadIdx.y value are placed in consecutive positions in increasing threadIdx.x order.
- In the above figure, the 16 threads form half a warp. The warp is padded with 16 other threads to form a 32-thread warp.
- 2D Block with 8×8 threads form 2 warps of 32 threads.
  Warp 1: $T_{0,0}$ to $T_{3,7}$
  warp 2: $T_{4,0}$ to $T_{7,7}$
- 3D Block with $2 \times 8 \times 4$ threads, form 2 warps of 32 threads.
  warp 1: $T_{0,0,0}$ to $T_{0,7,3}$
  warp 2: $T_{1,0,0}$ to $T_{1,7,3}$

3.3.2 SIMD



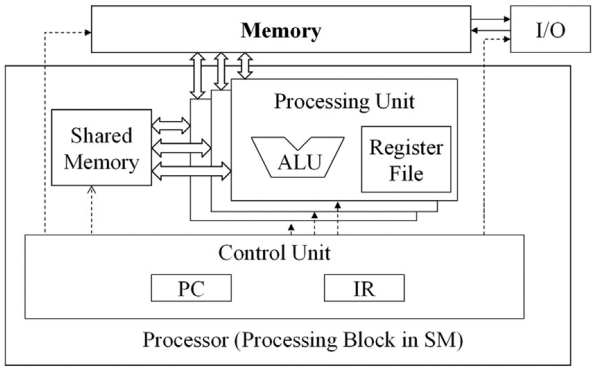
- An SM is designed to execute all threads in a warp following the single-instruction, multiple-data (SIMD) model. That is, at any instant in time, one instruction is fetched and executed for all threads in the warp.
- As an example, in a GPU of 16 cores/SM, all the cores in a processing block share an instruction fetch/dispatch unit. **Threads in the same warp are assigned to the same processing block, which fetches the instruction for the warp and executes it for all the threads in the warp at the same time. These threads apply the same instruction to different portions of the data. Because the SIMD hardware effectively restricts all threads in a warp to execute the same instruction at any point in time, the execution behavior of a warp is often referred to as single instruction, multiple-thread.**
- The **advantage of SIMD** is that the **cost of the control hardware**, such as the instruction fetch/dispatch unit, **is shared across many execution units**. This design choice **allows for a smaller percentage of the hardware to be dedicated to control and a larger percentage to be dedicated to increasing arithmetic throughput**.

- von Neuman Model



- The computer has an I/O (input/output) that allows both programs and data to be provided to and generated from the system.
- Computer loads the program and data into memory, before executing the program. Program is a set of instructions.
- The *Control Unit* maintains a *Program Counter*(PC), which contains the memory address of the next instruction that is to be executed.
- In each *"instruction cycle"*, the *Control Unit* uses the *PC* to fetch an instruction into the *Instruction Register*(IR).
- The instruction bits are then examined to determine the action to be taken by all the components of the computer.

- Modified von Neuman Model for SIMD



- Since all processing units are controlled by the same *Instruction Register* (IR) of the *Control Unit*, the only execution differences in these units are the input data (in register files) on which they operate on.
- This is *Single Instruction Multiple-Data* in processor design.
- Control units in modern processors are complex and sophisticated for fetching instructions and access instruction cache. Having multiple processing units share the same control unit can result in significant reduction in hardware manufacturing and power consumption.

3.4 Control Divergence

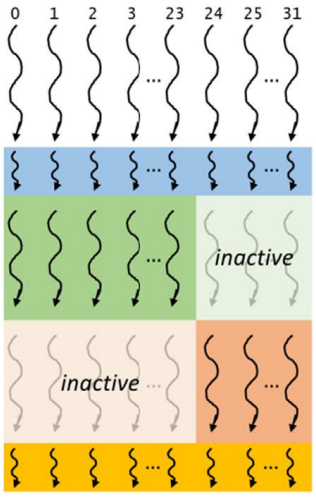
SIMD provides massive performance gains when all the threads within a warp follow the same execution path, or *control flow* when working on their data.
For example, in an `if-else` construct, the execution works well when either all the threads follow the

`if`-path or all execute the `else`-path.
However, when the threads in the warp take different control paths, the SIMD hardware has to take multiple passes through these paths, with one pass per each path.

- When threads in the same warp follow different execution paths, we say that these threads exhibit control divergence, that is, they diverge in their execution.
- While the **hardware** executes the same instruction for all threads in a warp, it **selectively lets these threads take effect in only the pass that corresponds to the path that they took**, allowing every thread to appear to take its own control flow path.
- This preserves the independence of threads while taking advantage of the reduced cost of SIMD hardware.
- The cost of divergence, however, is the extra passes the hardware needs to take as well as the execution resources the inactive threads consume in each pass.

- Example: `if-else-loop`

```
if(threadIdx.x < 24) {
    A
} else {
    B
}
C
```



- First Pass:

  threads with `threadIdx.x` $<$ 24 execute `A`.

  threads with 24 $<$ `threadIdx.x` $<=$ 32 are inactive.

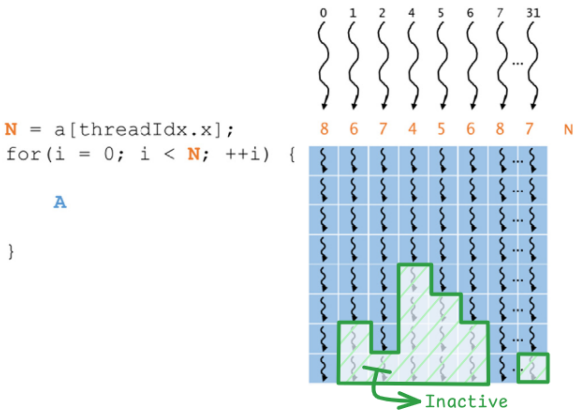
- Second Pass:

  threads with 24 $<$ `threadIdx.x` $<=$ 32 execute `B`.

  threads with `threadIdx.x` $<$ 24 are inactive.

- The threads in the warp reconverge and then execute `C`.
- From the Volta architecture onwards, the passes may be executed concurrently, meaning that the execution of one pass may be interleaved with the execution of another pass, known as *independent thread scheduling*.

- Example: `for-loop`



- The for loop varies between 4 and 8.
- For the first four iterations, all threads are active and execute A.
- For the remaining iterations, some threads execute A, while others are inactive because they have completed their iterations.

- To guess thread/control divergence

- If the decision condition is based on threadIdx values, the control statement can potentially cause thread divergence.
- For example, the statement `if(threadIdx.x > 2) {...}` causes the threads in the first warp of a block to follow two divergent control flow paths.

  Threads 0, 1, and 2 follow a different path than that of threads 3, 4, 5, and so on.

- Reasons to use control construct with thread control divergence

- To handle boundary conditions when mapping threads to data.
  - **Example**: Vector length: 1003, Block size: 64 $\implies$ (1003 + 64 - 1)/64, i.e, 16 thread blocks to process the 1003 elements.
  - 16 thread blocks $\rightarrow$ have 1024 total threads.
  - We need to disable the last 21 threads in thread block 15 from doing work that is not expected or not allowed by the original program.
  - The 16 blocks are partitioned into 32 warps. **Only the last warp** (i.e., the second warp in the last block) **will have control divergence.**
- The performance impact of control divergence decreases as the size of the vectors being processed increases.
  - For a vector length of 100, one of the four warps will have control divergence. (25%)
  - For a vector size of 1000, only one of the 32 warps will have control divergence. (4%)
  - For a vector of length 10,000, only one of the 313 warps will have control divergence. (0.3%)
- Due to the control divergence, one can not confidently assume that all the threads in the warp have the same execution timing. If all threads in a warp must complete one phase before moving to the next phase, barrier synchronization mechanism such as `__syncwarp()` is to be used.