CUDA Programming, Draft.

Srini Prakash Maiya

October 26, 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.

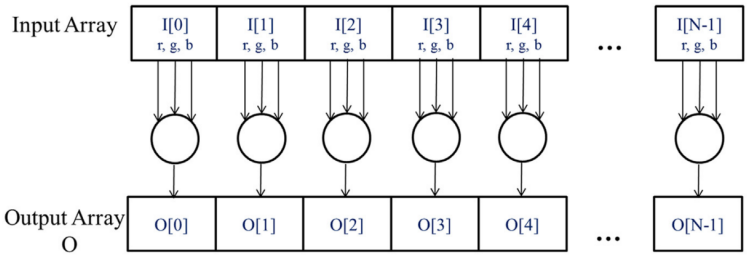


Figure 1.1: Data parallelism of RGB-to-grayscale. Each pixel can be independently converted to grayscale

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*.)

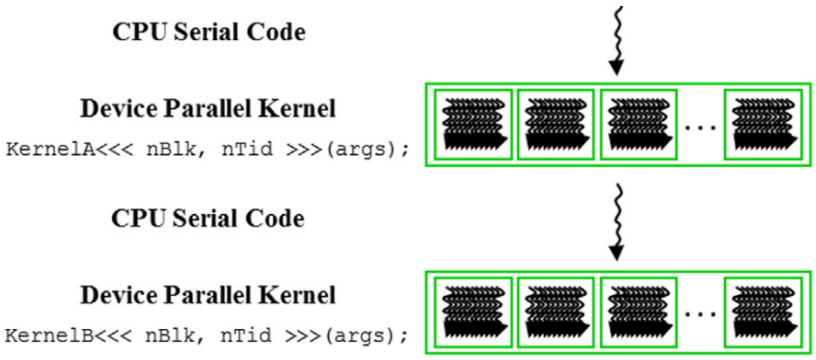


Figure 1.2: Simplified execution of a CUDA program with no CPU and GPU overlap

  - 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`

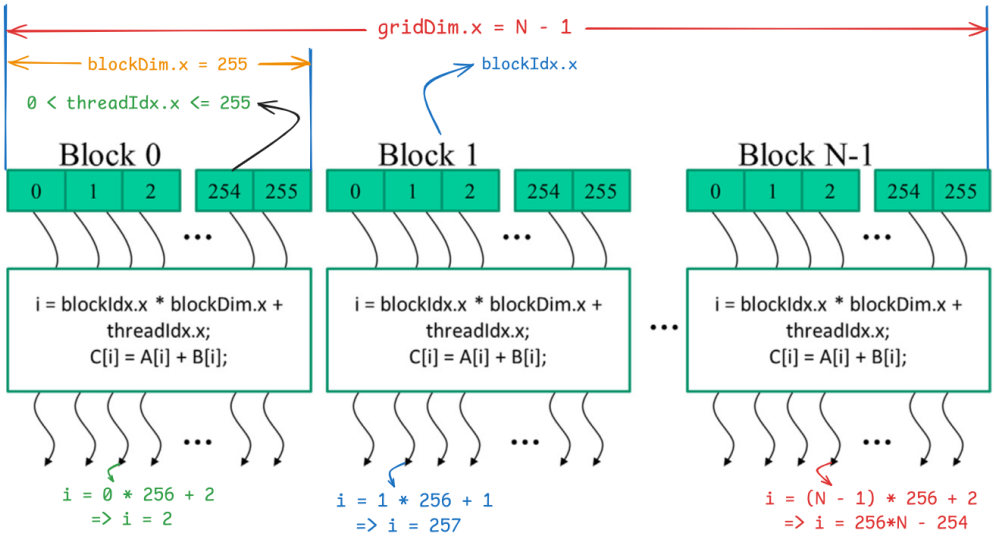


Figure 1.3: Hierarchy and organization of threads in a Grid

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.

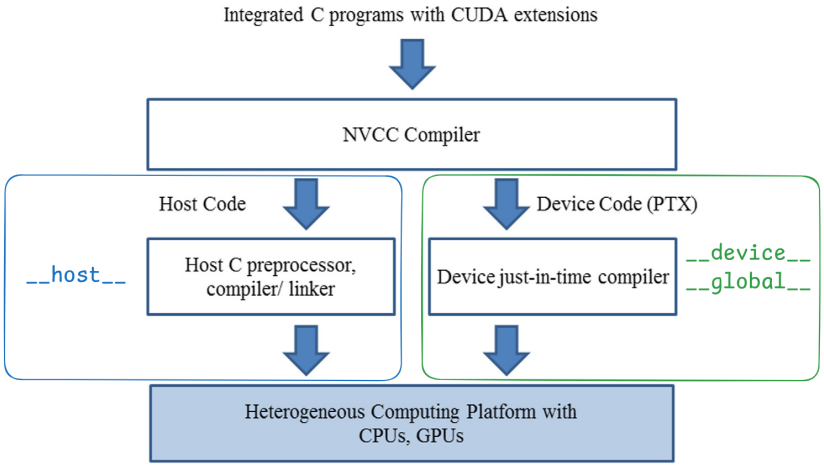
1.4 Calling Kernel functions

- 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);
}
```

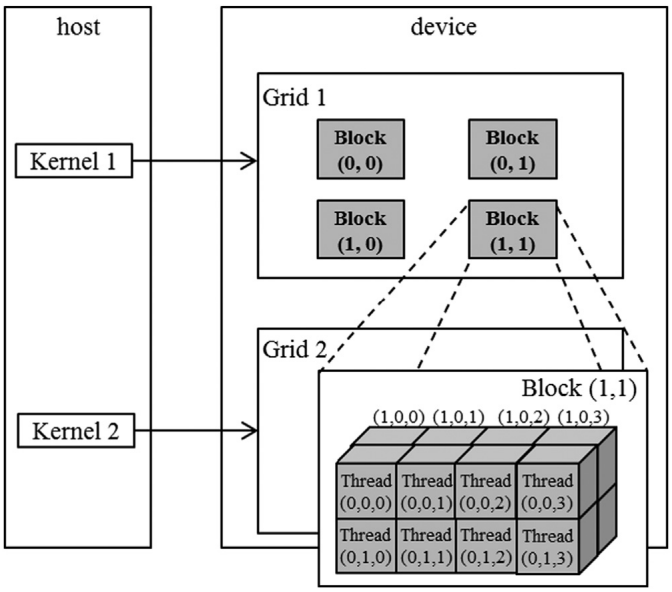
1.5 Compilation overview



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 >>>`

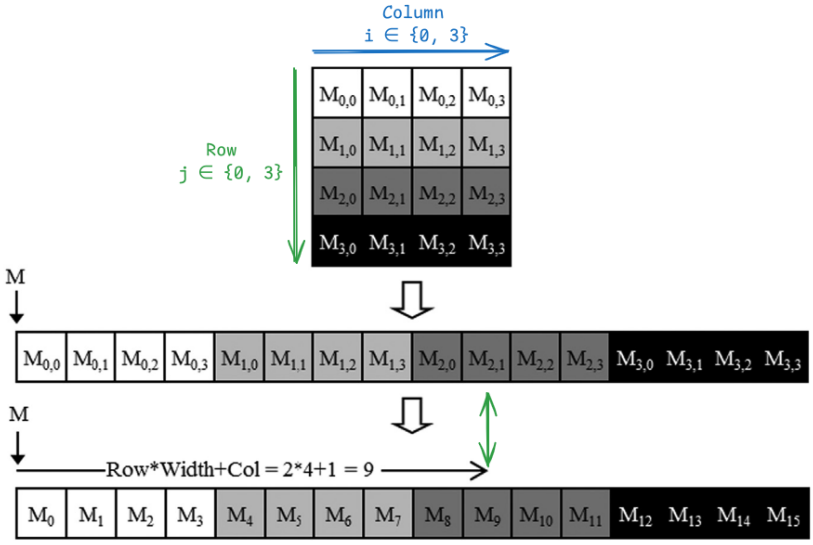


- **Grid** → 3D array of blocks.
  - All blocks share the same `gridDim.x, gridDim.y, gridDim.z` values.
- **Block** → 3D array of threads.
  - All threads in a block share the same `blockIdx.x, blockIdx.y, blockIdx.z` values.
  - Total number of threads in a block is constrained to 1024 and can be distributed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

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

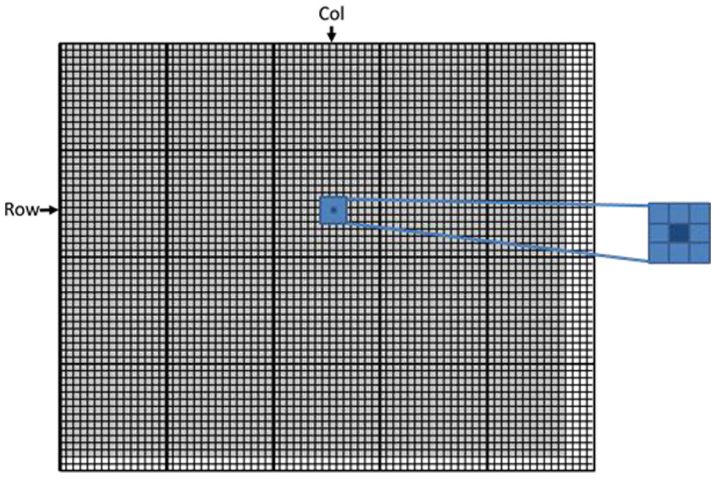
```
    // The input image is encoded as unsigned chars [0, 255]
    // Each pixel is 3 consecutive chars for 3 channels (RGB)
    __global__
    void colortoGrayscaleKernel(unsigned char* Pout, unsigned char* Pin,
                                        int width, int height) {
        int col = blockIdx.x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

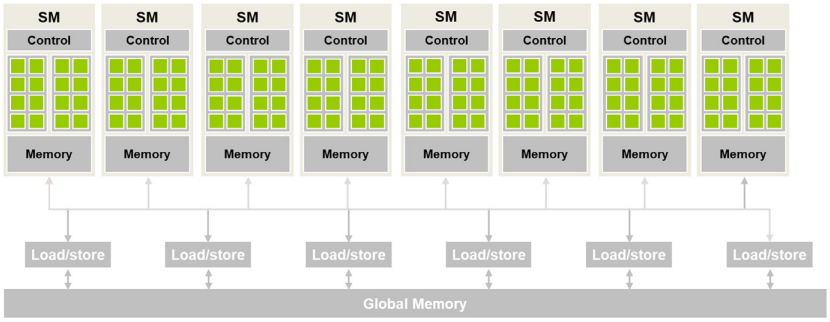
$$\begin{aligned} &\text{Pout}_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} \\ &= \text{Pout}_{1*16+0,0*16+0} = \text{Pout}_{16,0} = \text{Pout}[16 * 76 + 0] = \text{Pout}[1216] \end{aligned}$$

2.2 Image blur Kernel



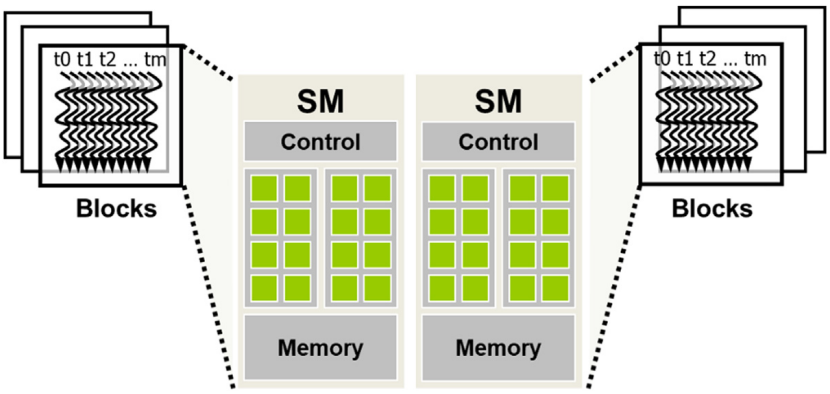
Chapter 3

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.

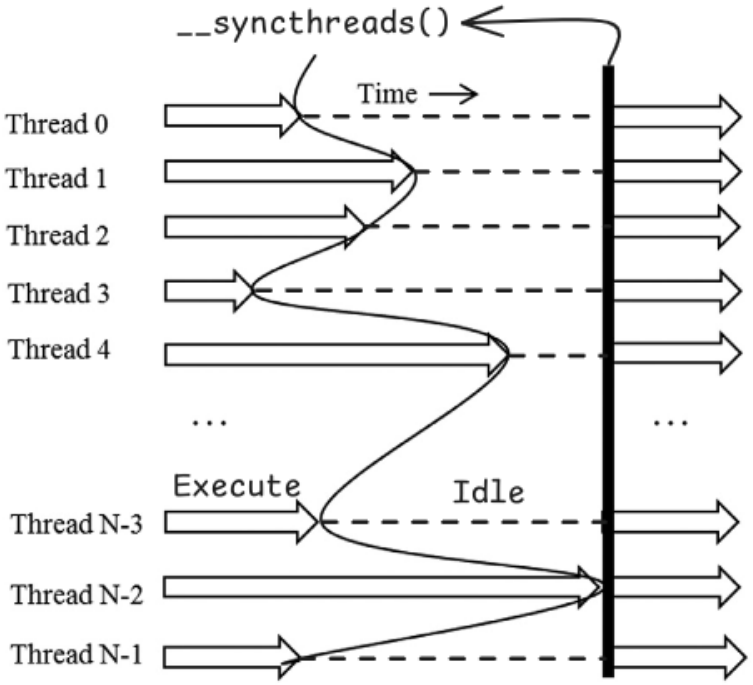
- i.e, **all the threads in a block are assigned to the same SM**.
- Multiple blocks are assigned to the same SM.

- **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.

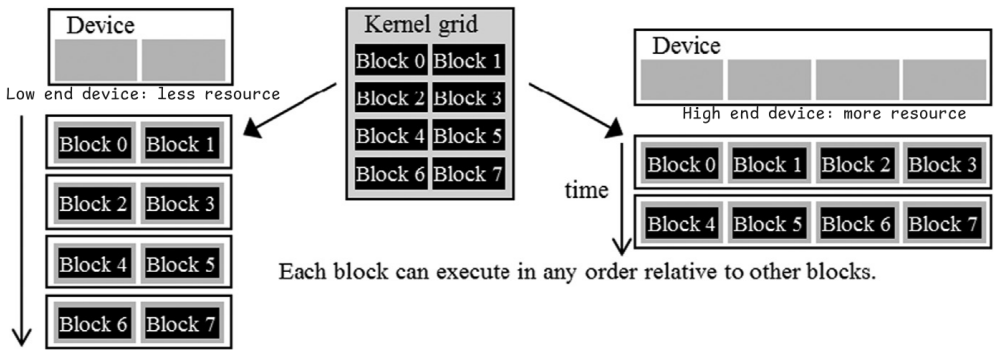
```
1   void incorrect_barrier_example(int n){
2       ...
3       if (threadIdx.x % 2 == 0) {
4           ...
5           // All the even threads wait here.
6           __syncthreads();
7       }
8       else {
9           ...
10          // All the odd threads wait here.
11          __syncthreads();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

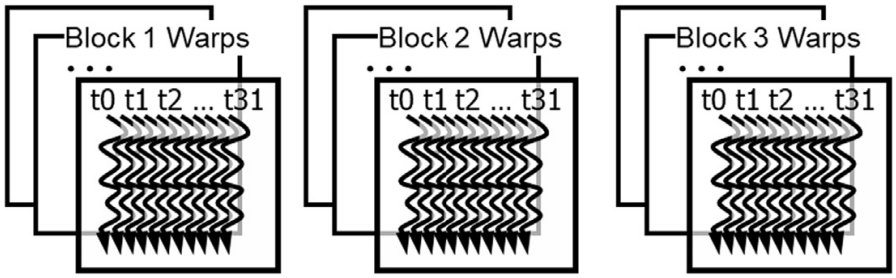
- The ability to execute the same application code on different hardware with different amounts of execution resources is referred to as ***transparent scalability***, which reduces the burden on application developers and improves the usability of applications.

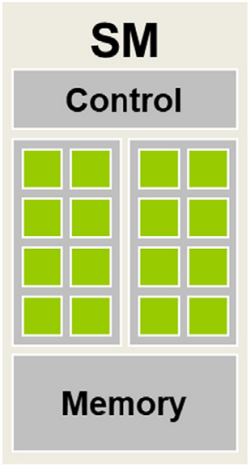
3.3 Warps and SIMD hardware

- 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**.

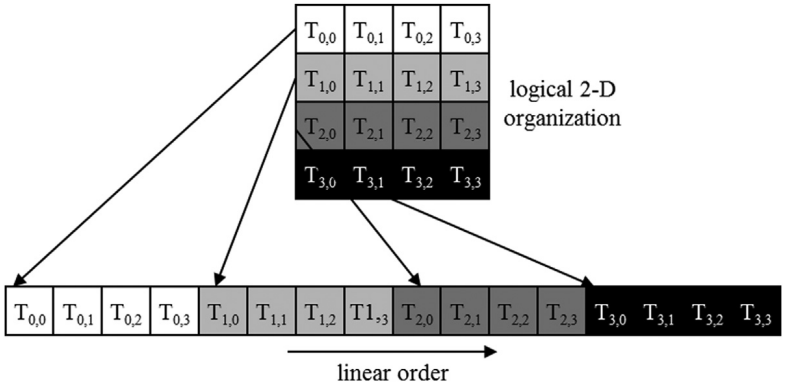
3.3.1 Warps

- 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.



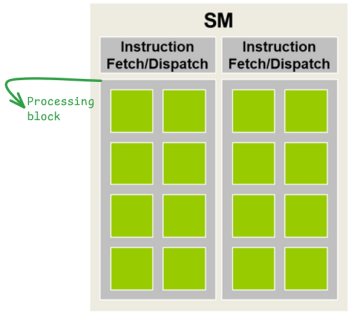


- Block 1, Block 2, Block 3 - all are assigned to the single SM. Each block is further divided into warps for scheduling purposes.
  - Each warp consists of 32 consecutive `threadIdx` values: threads 0 through 31 form the first warp, threads 32 through 64 the second warp etc.
  - If each block has 256 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.



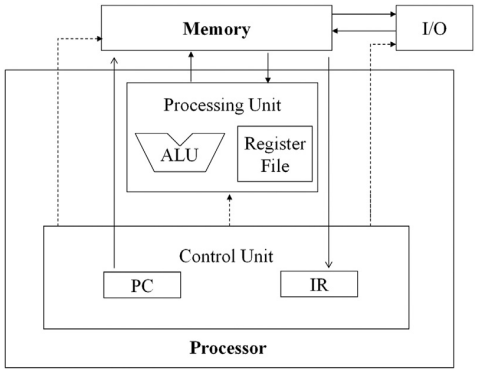
- For 2D grid: $T_{y,x}$, y being `threadIdx.y` and x being `threadIdx.x`, a 2D grid is linearized by placing all the threads whose `threadIdx.y` is 0 then `threadIdx.y` is 1 and so on. Threads with the same threadIdx.y value are placed in consecutive positions in increasing threadIdx.x order.
- In the 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



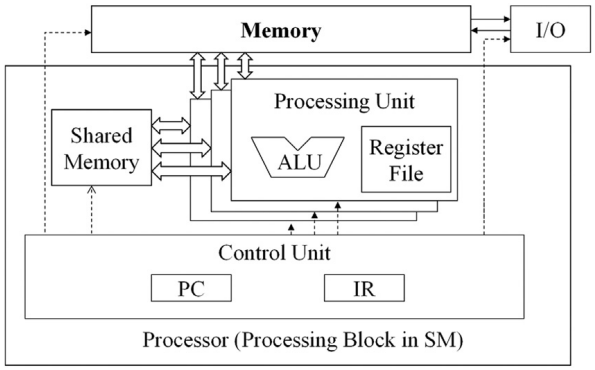
- An SM is designed to execute all threads in a warp following the single-instruction, multiple-data (SIMD) model. That is, at any instant in time, one instruction is fetched and executed for all threads in the warp.
- As an example, in a GPU of 16 cores/SM, all the cores in a processing block share 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

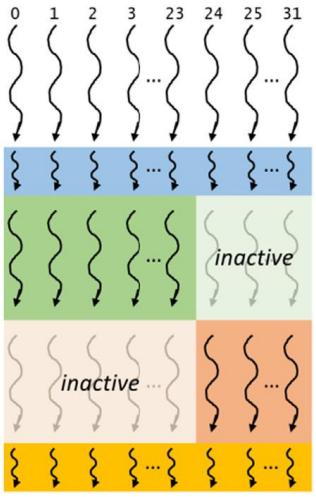
SIMD provides massive performance gains when all the threads within a warp follow the same execution path, or *control flow* when working on their data.
For example, in an `if-else` construct, the execution works well when either all the threads follow the

`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.

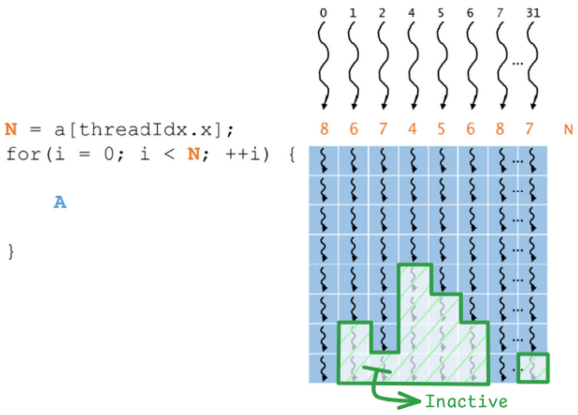
- Second Pass:

  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.

3.5 Warp scheduling and latency tolerance

When the threads are assigned to SMs, usually the number of threads exceeds the number of cores in the SM. This abundance of the warps/threads help the CUDA system to effectively schedule the warps.
In a warp, when the instruction that needs to be executed has to wait on the results of the previously initiated long-latency operation, the warp is not selected for execution. Instead, another warp resident in the SM is selected which is no longer waiting for the results of the previous instructions. If more than one warp is ready for execution, a priority mechanism is used to select one for execution.

This **mechanism of filling the latency time of operations from some threads 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. With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware while the instructions of some warps wait for the results of these long-latency operations.

3.6 Resource partitioning and occupancy

- **Occupancy**: The ratio of number of warps assigned to an SM to the maximum number of warps the SM supports.
- As the threads in an SM share the resources, registers and shared memory of that SM, a limited number of thread blocks can be assigned to an SM.
- *Example*: an Ampere A100 GPU can support maximum of:

  32 blocks / SM.

  64 warps / SM.

  2048 threads / SM.

  1024 threads / block.

  If a grid is launched with a block size of 1024 threads (maximum allowed), the 2048 thread slots in SM are partitioned into 2 blocks. $\implies$ Each SM can accommodate 2 blocks.

  Block size: 512 $\implies$ 4 blocks / SM

  Block size: 256 $\implies$ 8 blocks / SM

  Block size: 128 $\implies$ 16 blocks / SM

  Block size: 64 $\implies$ 32 blocks / SM

- The ability to dynamically partition the thread slots among blocks make SMs robust. They can either execute many blocks with few threads or few blocks with many threads.

  If fixed partition was the norm, then there would be either wasted thread slots when blocks require fewer threads than what the partition supports or fail in execution of the block when the requirement of threads is greater than supported.

- Underutilization with dynamic partitioning

1. With block size of 32 threads, the SM is partitioned into 2048 / 32 = 64 blocks. But an SM can support only 32 blocks at once. Means the supported 32 blocks end up using only 1024 thread slots (32 blocks, 32 threads each). The occupancy in this case is (1024 assigned threads)/(2048 maximum threads) = 50%. $\implies$ To utilize the thread slots in an SM fully, one needs at least 64 threads / block.
2. When the number of maximum threads is not divisible by the block size. If a block size of 700 is selected, 2048 // 700 = 2 blocks can be created in the SM. The resulting 2048 %700 = 648 threads are left unutilized. In this case, neither maximum thread slots nor maximum number of

blocks per SM are reached. The occupancy is (1400 assigned threads) / (2048 maximum threads) = 68.4%.

- Register resource limitation on occupancy

For example, the Ampere A100 GPU allows a maximum of 65,636 registers per SM. To run at full capacity, each of the maximum allowed 2048 threads of SM should have enough registers, which means that each thread should not use more than (64,536 registers) / (2048 threads) = 32 registers.

- For example, if a thread needs 64 registers to complete its execution successfully, the maximum number of threads that can be supported in an SM is (64,536 registers) / (64 registers / thread) = 1024 threads. In this case, the kernel can not run at full occupancy regardless of what block size is set. Instead, the occupancy will be at most 50%.
- Given a scenario of a kernel that uses 31 registers / thread, and has 512 threads per block. In this case, SM will have (2048 threads) / (512 threads / block) = 4 blocks running simultaneously. These threads will use a total of (2048 threads) × (31 registers/thread) = 63488 registers < register limit of 64,536.
  - If only the registers / thread is changed 33, the 2048 threads in the SM will need 67,584 threads > maximum allowed 64,536 registers. The CUDA runtime system might assign only 3 blocks with 512 threads/block in this case. Reducing the total required registers to 3 blocks × 512 threads/ block × 33 registers / thread = 50,688 registers.
    In this process, the threads / SM are reduced from 2048 to 1566 (3 × 512). That is, by using 2 automatic variables, the occupancy drops from 100% to 75%. This is sometimes referred as "***Performance cliff***", where a slight increase in resource usage (31 registers → 33 registers) can result in significant drop in parallelism performance achieved.

3.7 Device query

```
    // Get number of devices avilable.
    int devCount;
    cudaGetDeviceCount(&devCount);
    // Device property struct.
    cudaDeviceProp devProp;
    // Get properties and statisics of first GPU.
    cudaGetDeviceProperties(&devProp, 0);
    // View device statistics and properties from the devProp object.
    // Example:
    printf("Maximum threads per block  : %d", devProp.maxThreadsPerBlock);
    printf("Clockrate of the GPU       : %d GHz", devProp.clockRate / 1e6);
    printf("Man threads in x dim       : %d", devProp.maxThreadsDim[0]);
    printf("Man threads in y dim       : %d", devProp.maxThreadsDim[1]);
    printf("Man threads in z dim       : %d", devProp.maxThreadsDim[2]);
    // And many more.

```

3.8 Q & A

1. Consider the following CUDA kernel and the corresponding host function that calls it:

```
01    __global__ void foo_kernel(int* a, int* b) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        if(threadIdx.x < 40 || threadIdx.x >= 104) {
04            b[i] = a[i] + 1;
05        }
06        if(i%2 == 0) {
07            a[i] = b[i]*2;
08        }
09        for(unsigned int j = 0; j < 5 - (i%3); ++j) {
10            b[i] += j;
11        }
12    }
13    void foo(int* a_d, int* b_d) {
14        unsigned int N = 1024;
15        foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d);
16    }
```

   (a) What is the number of warps per block?
   A: Block size = 128 $\implies$ Number of warps per block = 128/32 = **4**.

   (b) What is the number of warps in the grid?
   A: Total blocks = (1024 + 128 - 1) / 128 = 8.
   Number of warps = 8 × 4 = **32**.

   (c) For the statement on line 04:

   - *How many warps in the grid are active?*
     A: **In a block**, the warps which contain the elements between `threadIdx.x` $\in$ 0,...,39 and 104,...,127 are active. *With warp size of 32, Warp 0 (0 to 31), Warp 1 (32 to 63), Warp 3 (96 to 128) are active (3 active warps).* $\implies$ 3 / 4 warps in a block are active. $\implies$ 3/4 × (32 warps in the grid) = **24 warps are active.**
   - *What is the SIMD efficiency (in %) of warp 0 of block 0?*
     A: In warp 0 all threads in index 0 ... 31 are executed. Thus, the efficiency of the warp is **100%**.
   - *What is the SIMD efficiency (in %) of warp 1 of block 0?*
     A: In warp 1 the threads with indices between 32 ... 39 are executed (8 threads). The efficiency of warp 1 is (8 threads execute) / (32 threads in the warp) = **25%**.
   - *What is the SIMD efficiency (in %) of warp 3 of block 0?*
     A: In warp 3 the threads with indices between 104 ... 127 are executed (23 threads). The efficiency of warp 3 is (23 threads execute) / (32 threads in the warp) = **75%**.

   (d) For the statement on line 07:

   - *How many warps in the grid are active?*
     A: All the 32 warps are active as threads with even indices are found in all the warps.
   - *How many warps in the grid are divergent?*
     A: All the 32 warps are divergent, as the even threads are distributed in all the warps. The threads with even index take the if-then path, while the threads with odd indices do not enter the if statement.
   - *What is the SIMD efficiency (in %) of warp 0 of block 0?*
     A: Among the 32 threads, only 16 of them are active in line 07. $\implies$ **50% SIMD efficiency.**

   (e) For the loop on line 09:

   - *How many iterations have no divergence?*
     A: $\min(\texttt{5 - i\%3}) \equiv 5 - \max(i\%3) \equiv \texttt{5 - 2} = 3$. The loop reduces to: `for (int j = 0; j < 3; +j)`. As all the blocks execute the for loop until j $\in$ 0, 1, 2, **3 iterations have no divergence**.
   - *How many iterations have divergence?*
     A: $\max(\texttt{5 - i\%3}) \equiv 5 - \min(i\%3) \equiv \texttt{5 - 0} = 5$. There are 5 iterations possible. Out of the 5, 3 are not divergent. $\implies$ **2 divergent iterations.**

(f) For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?
A: Block size : 512. To include all 2000 elements of the vector, there has to be (2000 + 512 - 1) / 512 = 4 blocks. $\implies$ 4 $\times$512 = **2048 threads.**

(g) For the previous question, how many warps do you expect to have divergence due to the boundary check on vector length?
A: Number of warps: (2048 threads) / (32 threads per warp) = 64 warps. 63 warps cover the entire vector of 2000 elements. (63 * 32 = 2016 threads.) $\implies$ As 62 * 32 = 1984 < 2000, the last (63^{rd}) warp will have divergence. **One warp with divergence.**

(h) Consider a hypothetical block with 8 threads executing a section of code before reaching a barrier. The threads require the following amount of time (in microseconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, and 2.9; they spend the rest of their time waiting for the barrier. What percentage of the threads' total execution time is spent waiting for the barrier?
A: The maximum time required is 3.0 μs.

$$Time_{idle} = \frac{1}{n}\sum_{n}^{8} max(T) - t$$
$$= 0.5125\mu s. \implies \frac{100 \times 0.5125}{3} = \textbf{17.08\% is spent waiting.}$$

(i) A CUDA programmer says that if they launch a kernel with only 32 threads in each block, they can leave out the `__syncthreads()` instruction wherever barrier synchronization is needed. Do you think this is a good idea? Explain.
A: Launching the kernel with 32 threads per block does not promise the timely execution of all the threads like the barrier synchronization does. Assuming and tweaking the block, warp size does not ensure the correctness of the execution when the execution happens in phases.

(j) If a CUDA device's SM can take up to 1536 threads and up to 4 thread blocks, which of the following block configurations would result in the most number of threads in the SM?

a. *128 threads per block*
b. *256 threads per block*
c. *512 threads per block*
d. *1024 threads per block*

A: 1536 % 3 = 0 && 1536 / 3 = 512 $\implies$ 3 blocks with 512 threads each yield in 100% occupancy.

(k) Assume a device that allows up to 64 blocks per SM and 2048 threads per SM. Indicate which of the following assignments per SM are possible. In the cases in which it is possible, indicate the occupancy level.

a. *8 blocks with 128 threads each*
A. 8 $\times$ 128 = 1024 $\implies$ Possible with Occupancy = 50%
b. *16 blocks with 64 threads each*
A. 16 $\times$ 64 = 1024 $\implies$ Possible with Occupancy = 50%
c. *32 blocks with 32 threads each*
A. 32 $\times$ 32 = 1024 $\implies$ Possible with Occupancy = 50%
d. *64 blocks with 32 threads each*
A. 64 $\times$ 32 = 2048 $\implies$ Possible with Occupancy = 100%
e. *32 blocks with 64 threads each*
A. 32 $\times$ 64 = 2048 $\implies$ Possible with Occupancy = 100%

(l) Consider a GPU with the following hardware limits: 2048 threads per SM, 32 blocks per SM, and 64K (65,536) registers per SM. For each of the following kernel characteristics, specify whether the kernel can achieve full occupancy. If not, specify the limiting factor.

a. *The kernel uses 128 threads per block and 30 registers per thread.*
A. Num. of blocks= (2048 threads) / (128 threads/block) = 16 blocks $<$ 32 blocks. OK.
• Total registers = total threads × registers per thread. = 16 × 128 × 30
= 61,440 $<$ 65,536 total registers OK. Total occupancy with threads can be reached.

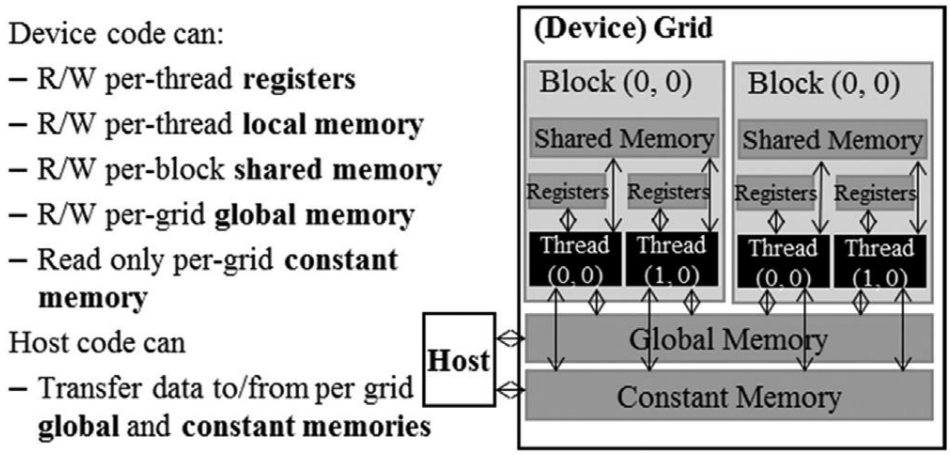
b. *The kernel uses 32 threads per block and 29 registers per thread.*
A. Num of blocks = 2048 / 32 = 64 blocks $>$ 32 blocks. NOK.

c. *The kernel uses 256 threads per block and 34 registers per thread.*
A. Num of blocks = 2048 / 256 = 8 blocks $<$ 32 blocks. OK.
• Total registers = 8 × 256 × 34 = 69632 $>$ 32 blocks. NOK.

(m) A student mentions that they were able to multiply two 1024 × 1024 matrices using a matrix multiplication kernel with 32 × 32 thread blocks. The student is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. The student further mentions that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?
A. Total threads in a block: 32 × 32 = 1024 $>$ allowed 512 threads / block. Not possible.

Chapter 4

Memory architecture and data locality



- **Global memory**:
  - Can be both read and written by *Device*.
  - Can be both read and written by *Host*.
- **Constant memory**:
  - Offers short-latency high-bandwidth ***read-only access*** to *Device*.
  - Can be both read and written by *Host*.
- **Local memory**:
  - Resides in the global memory, but is not shared across threads. Each thread has its own private local memory, where places the data that is private to the thread but cannot be allocated in the registers. This data includes statically allocated arrays, spilled registers, and other elements of the thread's call stack.
- **Registers and Shared memory**:
  - Are on-chip memories. Variables that reside in registers/shared memory can be accessed at very high speed in highly parallel manner.
  - Registers are allocated to individual threads; each thread can access its own set of registers.
  - *Kernel function typically uses registers* to store frequently accessed variables that are private to each thread.

- *Shared memory is allocated to thread blocks*; all threads in a block can access shared memory variables declared for the block.
  - Shared memory is an efficient means by which threads can cooperate by sharing their input data and intermediate results.
  - By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

CPU vs GPU Register architecture

- When CPUs context switch between different threads, they save the registers of the outgoing thread to memory and restore the registers of the incoming thread from memory.
- In contrast, GPUs achieve zero-overhead scheduling by keeping the registers of all the threads that are scheduled on the processing block in the processing block's register file.
- This way, switching between warps of threads is instantaneous because the registers of the incoming threads are already in the register file. Consequently, GPU register files need to be substantially larger than CPU register files

4.1 - Advantages of Registers over Global Memory

- Global memory is implemented with DRAM technology, implying long access latencies and relatively low access bandwidths.
- The registers on the other hand, are on the processor chip, which implies very short access latency and very large access bandwidth, compared to the global memory.
  - Floating-point addition with operands on global memory:

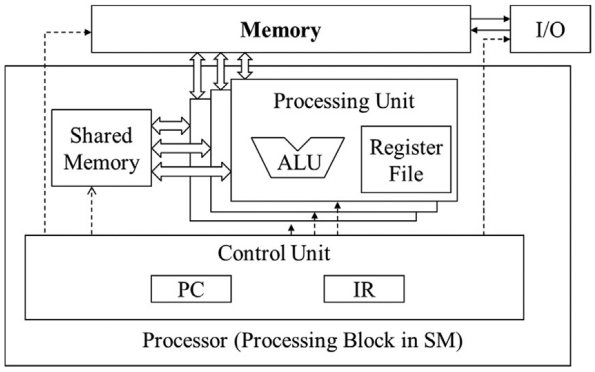
```
load r2, r4, offset
fadd r1, r2, r3
```

    `load` instruction adds an offset value to the contents of `r4` to form an address for the operand value. It then accesses the global memory and places the value into register `r2`. Once the operand value is in `r2`, the `fadd` instruction performs the floating-point addition using the values in `r2` and `r3` and places the result into `r1`.
    Since the processor can fetch and execute only a limited number of instructions per clock cycle, the version with an additional load will likely take more time to process than the one without.
- Whenever a variable is stored in a register, its accesses no longer consume off-chip global memory bandwidth. *A subtler point is that each access to registers involves fewer instructions than access to the global memory.*
  - Floating-point addition with operands on built-in registers:

```
fadd r1, r2, r3
```

    The register files `r2` and `r3` are the files where input values can be found. The addition of `r2` and `r3` are stored in `r1`.
- In modern computers the energy that is consumed for accessing a value from the register file is at least an order of magnitude lower than for accessing a valuefrom the global memory. Accessing a value from registers has a tremendous advantage in energy efficiency over accessing the value from the global memory.

4.2 Shared memory vs Registers



- Shared memory
  - Is designed as part of the memory space that resides on the processor chip.
  - When the processor needs to access the data, it still has to perform the `load` operation same as when accessing the data from the global memory.
  - However, as the shared memory resides on-chip, it can be accessed with much lower latency and much higher throughput than in global memory access.
  - Because of the need to perform a load operation, shared memory has longer latency and lower bandwidth than registers.
  - In computer architecture terminology the shared memory is a form of *scratchpad memory*.
- One important difference between *Shared Memory* and the *Registers* is that, *Shared Memory is accessible by all the threads in a block*. This is contrary to the *register data, which is private to a thread*.
- CUDA device SM typically employs multiple processing units to allow multiple threads to make simultaneous progress.
- Threads in a block can be spread across these processing units.
- Therefore, the hardware implementations of the shared memory in these CUDA devices are typically designed to allow multiple processing units to simultaneously access its contents to support efficient data sharing among threads in a block.

4.3 - Declaring program variables into the various memory types

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

1. If a variable's scope is a single thread, a private version of the variable will be created for every thread; each thread can access only its private version of the variable.
2. Lifetime tells the portion of the program's execution duration when the variable is available for use: either within a grid's execution or throughout the entire application.
3. If a **variable's lifetime is within a grid's execution, it must be declared within the kernel function body and will be available for use only by the kernel's code**. If the kernel is invoked several times, the value of the variable is not maintained across these invocations. Each invocation must initialize the variable in order to use it.

4. If a variable's lifetime is throughout the entire application, it must be declared outside of any function body. The contents of these variables are maintained throughout the execution of the application and available to all kernels.

5. Variables that are not arrays are *scalar* variables. **All automatic scalar variables that are declared in kernel and device functions are placed into registers.** The scopes of these automatic variables are within individual threads. When a **kernel function declares an automatic variable, a private copy of that variable is generated for every thread that executes the kernel function. When a thread terminates, all its automatic variables cease to exist.**

6. Automatic array variables are not stored in registers. They are stored in thread's local memory and may incur access delays. A private version of each automatic array is created for and used for every thread. Once a thread terminates its execution, the contents of its automatic array variables cease to exist.

7. If a variable declaration is preceded by `__shared__` keyword, it is declared as a shared variable in CUDA. Such a declaration is typically made within a kernel function or a device function.

    - **Shared variables reside in the shared memory. The scope of a shared variable is within a thread block**; that is, all threads in a block see the same version of a shared variable.
    - **A private version of the shared variable is created for and used by each block during kernel execution. The lifetime of a shared variable is within the duration of the kernel execution.** When a kernel terminates its grid's execution, the contents of its shared variables cease to exist.
    - Accessing shared variables from the shared memory is extremely fast and highly parallel. CUDA programmers often use shared variables to hold the portion of global memory data that is frequently used and reused in an execution phase of the kernel.

8. If a variable declaration is preceded by the `__constant__` keyword, then the variable is declared as a constant in CUDA. **Declaration of constant variables must be outside any function body**.

    - The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution.
    - Constant variables are **stored in the global memory but are cached for efficient access**. Currently, the total size of constant variables in an application is limited to 65,536 bytes.

9. If a variable declaration is preceded by the `__device__` keyword, **the global variable is placed in the global memory.**

    - Accesses to the global variable are slow.
    - One important advantage of global variables is that they are visible to all threads of all kernels. Their contents also persist through the entire execution.
    - Thus, global variables can be used as a means for threads to collaborate across blocks.
    - *There is currently no easy way to synchronize between threads from different thread blocks or to ensure data consistency across threads in accessing global memory other than using atomic operations or terminating the current kernel execution.*
    - **Therefore, global variables are often used to pass information from one kernel invocation to another kernel invocation.**

10. In CUDA, pointers can be used to point to data objects in the global memory.

    - If an object is allocated by a host function, the pointer to the object is initialized by memory allocation API functions such as `cudaMalloc` and `cudaMemCpy` functions and can be passed to the kernel function as a parameter.

- Assign the address of a variable is declared in the global memory to a pointer variable in a kernel function.
  `float* ptr = &GlobalVar;`, where `float *ptr` is an automatic pointer variable which points to the global variable.

4.4 Q & A

1. Consider matrix addition. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: Analyze the elements that are accessed by each thread and see whether there is any commonality between threads.
   A: No, the use of shared memory has no effect here. There is no commonality between the elements accessed by any two or more given threads.

2. What type of incorrect execution behavior can happen if one forgot to use one or both `__syncthreads()` in the kernel?

```
01    #define TILE_WIDTH 16
02    __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07        int bx = blockIdx.x;  int by = blockIdx.y;
08        int tx = threadIdx.x; int ty = threadIdx.y;
09
10        // Identify the row and column of the P element to work on
11        int Row = by * TILE_WIDTH + ty;
12        int Col = bx * TILE_WIDTH + tx;
13
14        // Loop over the M and N tiles required to compute P element
15        float Pvalue = 0;
16        for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18            // Collaborative loading of M and N tiles into shared memory
19            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21            __syncthreads();
22
23            for (int k = 0; k < TILE_WIDTH; ++k) {
24                Pvalue += Mds[ty][k] * Nds[k][tx];
25            }
26            __syncthreads();
27
28        }
29        P[Row*Width + Col] = Pvalue;
30
31    }
```

   A: If the `__syncthreads()` in `line 21` is not used, the loading of elements from global memory into the shared memory `Mds` and `Nds` is not synchronized/ensured for the entire block. This would in turn pass wrong and incomplete values of `Mds` and `Nds` during the calculation of `Pvalue` in `line 24`.
   If the `__syncthreads()` in `line 26` is not used, the threads that write might get ahead of the other thread and write false values into the output matrix `P`.

3. Assuming that capacity is not an issue for registers or shared memory, give one important reason why it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.
   A: The values stored in the shared memories can be accessed by all the threads in the block. By loading the values from the global memory into shared memory, the loaded data can be reused by multiple threads with single variable/memory fetch.
   If the data is directly loaded into registers directly, the data is restricted to a thread. If the same values are required by multiple threads, then they need to be loaded so many times, once per each thread.

4. For our tiled matrix-matrix multiplication kernel, if we use a 32 3 32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?
   A: 32× reduction.

5. Assume that a CUDA kernel is launched with 1000 thread blocks, each of which has 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?
A: 512,000 instances. 1 instance of the variable per thread.

6. In the previous question, if a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?
A: 1,000 instances. 1 per thread block.

7. Consider performing a matrix multiplication of two input matrices with dimensions N×N. How many times is each element in the input matrices requested from global memory when:

   - a. There is no tiling?
   A: N times.
   - b. Tiles of size T×T are used?
   A: N / T times.

8. A kernel performs 36 floating-point operations and seven 32-bit global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute-bound or memory-bound.

   a. Peak FLOPS=200 GFLOPS, peak memory bandwidth=100 GB/second
   A:

$$\begin{aligned}\text{computational intensity} &= \frac{\text{Floating point operations}}{\text{Num. Memory accesses} \times \text{size of access(bytes)}} \\ &= \frac{36}{7 \times 4} \\ &= 1.286 \text{ FLOP/Byte}\end{aligned}$$

$$\begin{aligned}\text{memory bandwidth limit} &= \text{peak memory bandwidth} \times \text{computational intensity} \\ &= 100 \times 1.286 = \textbf{128.6 GFLOPS.}\end{aligned}$$

$$128.6 \text{ (memory bandwidth limit)} > 100 \text{ (peak memory bandwidth)}$$
$$\implies \textbf{The kernel is memory bound.}$$

   b. Peak FLOPS=300 GFLOPS, peak memory bandwidth=250 GB/second
   A:

$$\begin{aligned}\text{computational intensity} &= 1.286 \text{ FLOP/Byte} \\ \text{memory bandwidth limit} &= 250 \times 1.286 = \textbf{321.5 GFLOPS}\end{aligned}$$

$$321.5 \text{(memory bandwidth limit)} > 300 \text{(Peak FLOPS)}$$
$$\implies \textbf{The kernel is compute bound.}$$

9. To manipulate tiles, a new CUDA programmer has written a device kernel that will transpose each tile in a matrix. The tiles are of size BLOCK_WIDTH by BLOCK_WIDTH, and each of the dimensions of matrix A is known to be a multiple of BLOCK_WIDTH. The kernel invocation and code are shown below. BLOCK_WIDTH is known at compile time and could be set anywhere from 1 to 20.

```
01  dim3 blockDim(BLOCK_WIDTH,BLOCK_WIDTH);
02  dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
03  BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

04  __global__ void
05  BlockTranspose(float* A_elements, int A_width, int A_height)
06  {
07      __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];

08      int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
09      baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A_width;

10      blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];

11      A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
12  }
```

   a. Out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will this kernel function execute correctly on the device?
   A:
   After loading the elements of `A_elements` into the shared memory `blockA`, there is no barrier function such as `__syncthreads()` to ensure the completion of write operations of all the threads into the shared memory.

   b. If the code does not execute correctly for all BLOCK_SIZE values, what is the root cause of this incorrect execution behavior? Suggest a fix to the code to make it work for all BLOCK_SIZE values.
   A: Placement of `__syncthreads()` directly after line 10.

10. Consider the following CUDA kernel and the corresponding host function that calls it:

```
01  __global__ void foo_kernel(float* a, float* b) {
02     unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03     float x[4];
04     __shared__ float y_s;
05     __shared__ float b_s[128];
06     for(unsigned int j = 0; j < 4; ++j) {
07        x[j] = a[j*blockDim.x*gridDim.x + i];
08     }
09     if(threadIdx.x == 0) {
10        y_s = 7.4f;
11     }
12     b_s[threadIdx.x] = b[i];
13     __syncthreads();
14     b[i] = 2.5f*x[0] + 3.7f*x[1] + 6.3f*x[2] + 8.5f*x[3]
15            + y_s*b_s[threadIdx.x] + b_s[(threadIdx.x + 3)%128];
16  }
17  void foo(int* a_d, int* b_d) {
18     unsigned int N = 1024;
19     foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d);
20  }
```

   a. How many versions of the variable `i` are there?
   A: **Automatic variable.** Each thread has a unique `i` $\implies$ **1024 versions.**

   b. How many versions of the array `x[]` are there?
   A: **Automatic variable.** Each thread has a unique `x[]` $\implies$ **1024 versions.**

c. How many versions of the variable `y_s` are there?
A: **Shared variable**. Each block has a unique `y_s` $\implies$ (1024 + 128 - 1) / 128 = **8 versions**

d. How many versions of the array `b_s[]` are there?
A: **Shared variable**. Each block has a unique `y_s` $\implies$ (1024 + 128 - 1) / 128 = **8 versions**

e. What is the amount of shared memory used per block (in bytes)?
A: `__shared__ float y_s` uses 4 bytes + `__shared__ float b_s[128]` uses 4 *times* 128 bytes of shared memory. (4: `sizeof(float)`) $\implies$ 516 bytes.

f. What is the floating-point to global memory access ratio of the kernel (in OP/B)?
A:

In line 14,

FLOPs per thread = 5 Multiplications + 1 Modulo operation + 5 Additions

Global Memory access count = 4 (line 06) + 1 (line 12) + 1 (line 14)

FLOP to GMEM ratio = 11/6 = **1.83**