CSE 531

Heterogeneous Computing and GPU Programming

Spring 2023

Mahmut Taylan Kandemir

ACK: G. Barlas, X. Tang

Heterogeneous computing

Traditionally, heterogeneous computing refers to a system that uses more than one *type* of computing cores, such as CPU, GPU, DPU, VPU, FPGA, or ASIC. By assigning different workloads to specialized processors suited for diverse purposes, performance and energy efficiency can be vastly improved.

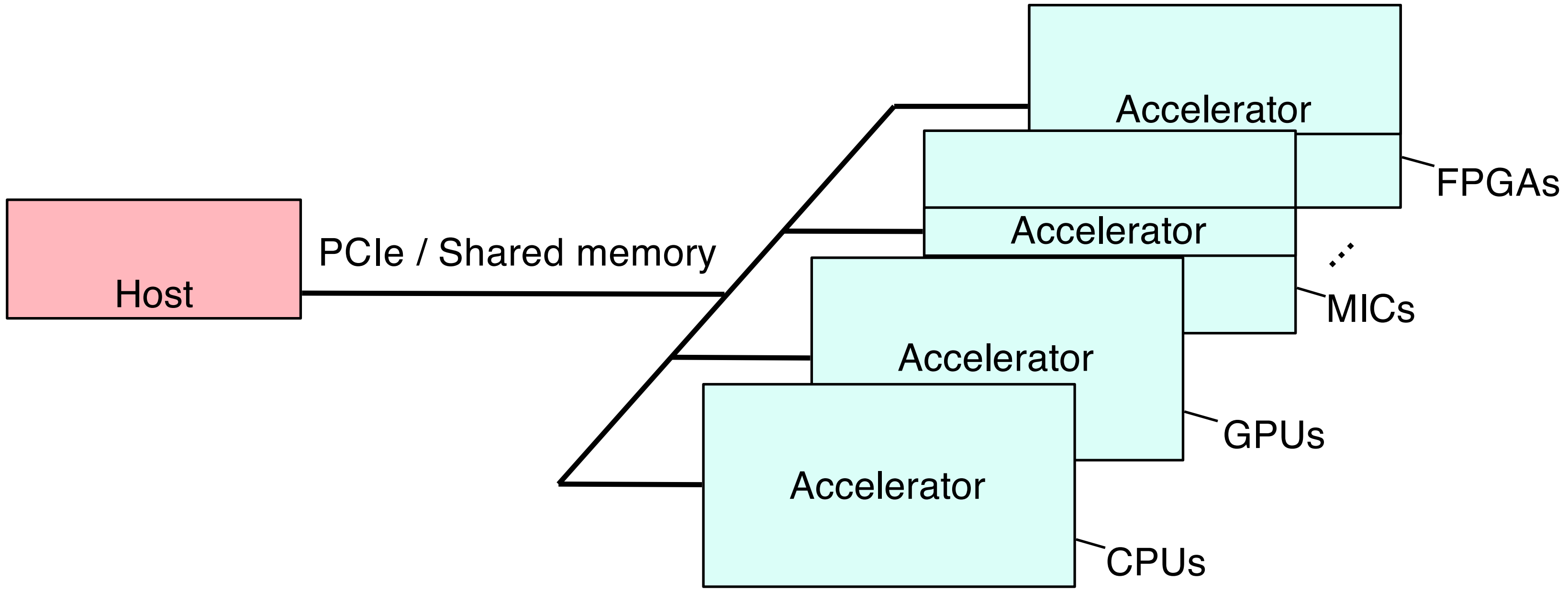
In recent years however, the definition of heterogeneous computing has expanded to encompass processors based on *different* computer architectures. For example, processors based on the ARM architecture may be a better choice for some tasks, due to their higher number of cores, better power-efficiency, and compatibility with ARM-based mobile devices.

Heterogeneous platforms

- Systems combining main processors and accelerators
  - e.g., CPU + GPU, CPU + Intel MIC, AMD APU, ARM SoC
  - Everywhere from supercomputers to mobile devices

Heterogeneous platforms

- Host-accelerator hardware model
- a.k.a. **Accelerated Computing**



Pros and cons of heterogeneous computing

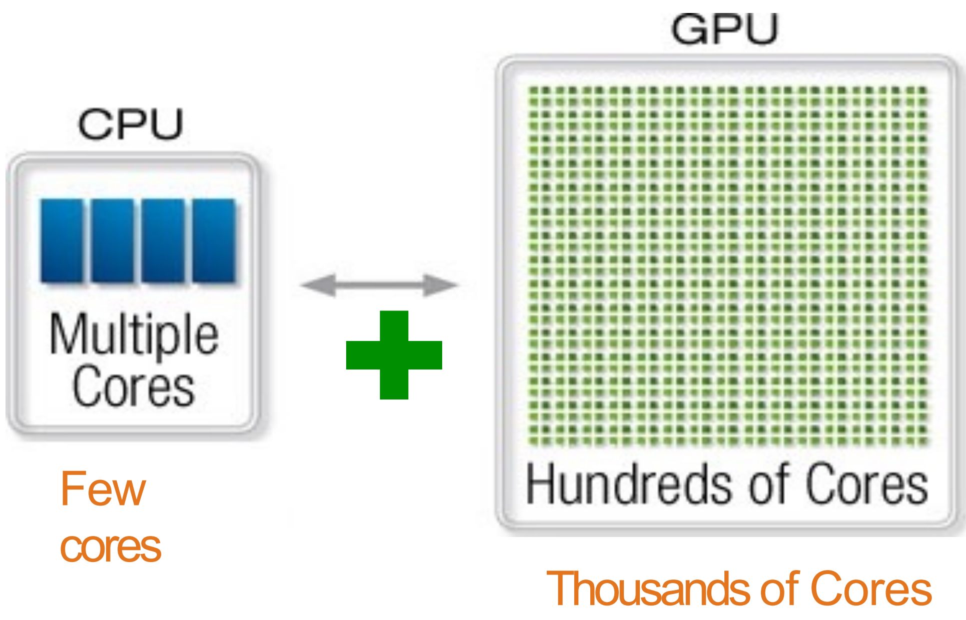
Heterogeneous systems offer the opportunity to significantly increase system performance and reduce system power consumption (increase power efficiency), enabling systems to continue to scale beyond the limitations imposed by ever shrinking process geometries.

Heterogeneous systems pose a unique challenge in computing as they contain distinct sets of compute units with their very own architecture/ISA. Since the ISA of/interface to each of the compute units is different, it becomes difficult to program and achieve load balancing between them.

Utilization is another big issue.

Our focus for now …

- A heterogeneous platform = **CPU** + **GPU**
  - Most solutions work for other/multiple accelerators as well
- An application workload = an application + its input dataset
- Workload partitioning = workload distribution among the processing units of a heterogeneous system

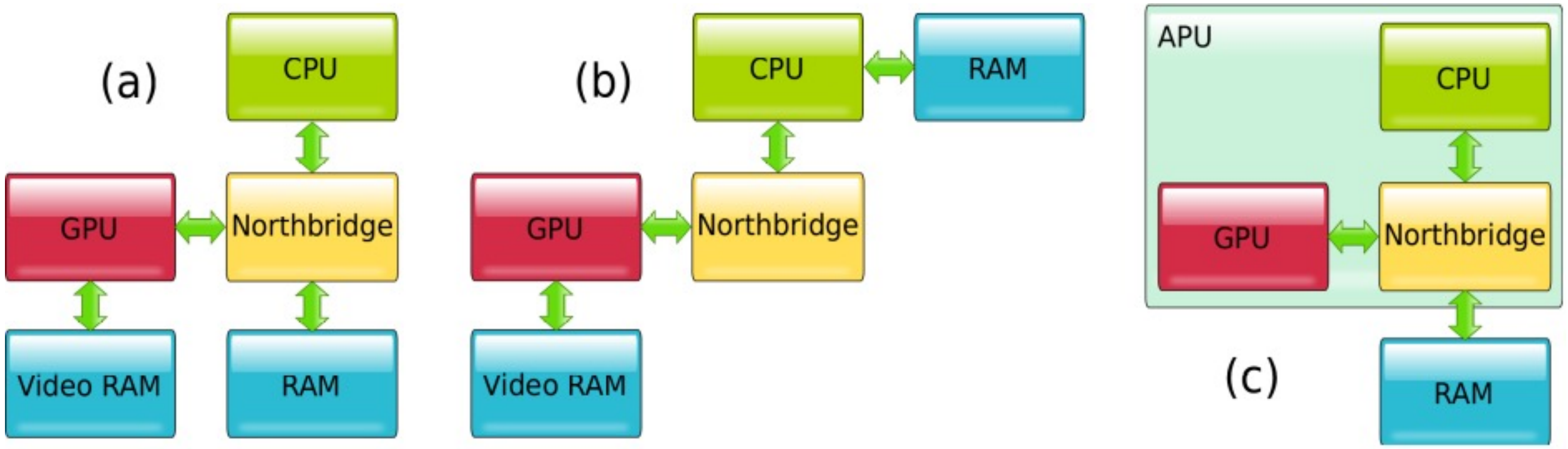


Graphics Processing Units (GPUs)

- Graphics Processing Units (GPUs) offer a *leap* in performance that makes tackling bigger and more difficult problems feasible.
- In addition, GPUs offer unparalleled “power efficiency” in terms of TFLOPS/Watt, a feature that is becoming all the more relevant in a world of dwindling natural resources.
- The only problem is that GPU architectures are a breed apart from traditional multicore CPUs. Thousands of cores, coupled with complex hierarchies of memory subsystems that are *not* transparent to the programmer, constitute a real challenge.

GPU : discrete or integrated?

- Integrated solutions are compromises. Trading speed for overall system simplicity and price.



- (a) and (b) represent **discrete GPU** solutions, with a CPU-integrated memory controller in (b). Diagram (c) corresponds to **integrated CPU-GPU** solutions, as the AMD's Accelerated Processing Unit (APU) chips.

As of 2019, Intel and AMD had both released chipsets in which all northbridge functions had been integrated into the CPU. Modern Intel Core processors have the northbridge integrated on the CPU die, where it is known as the uncore or system agent.

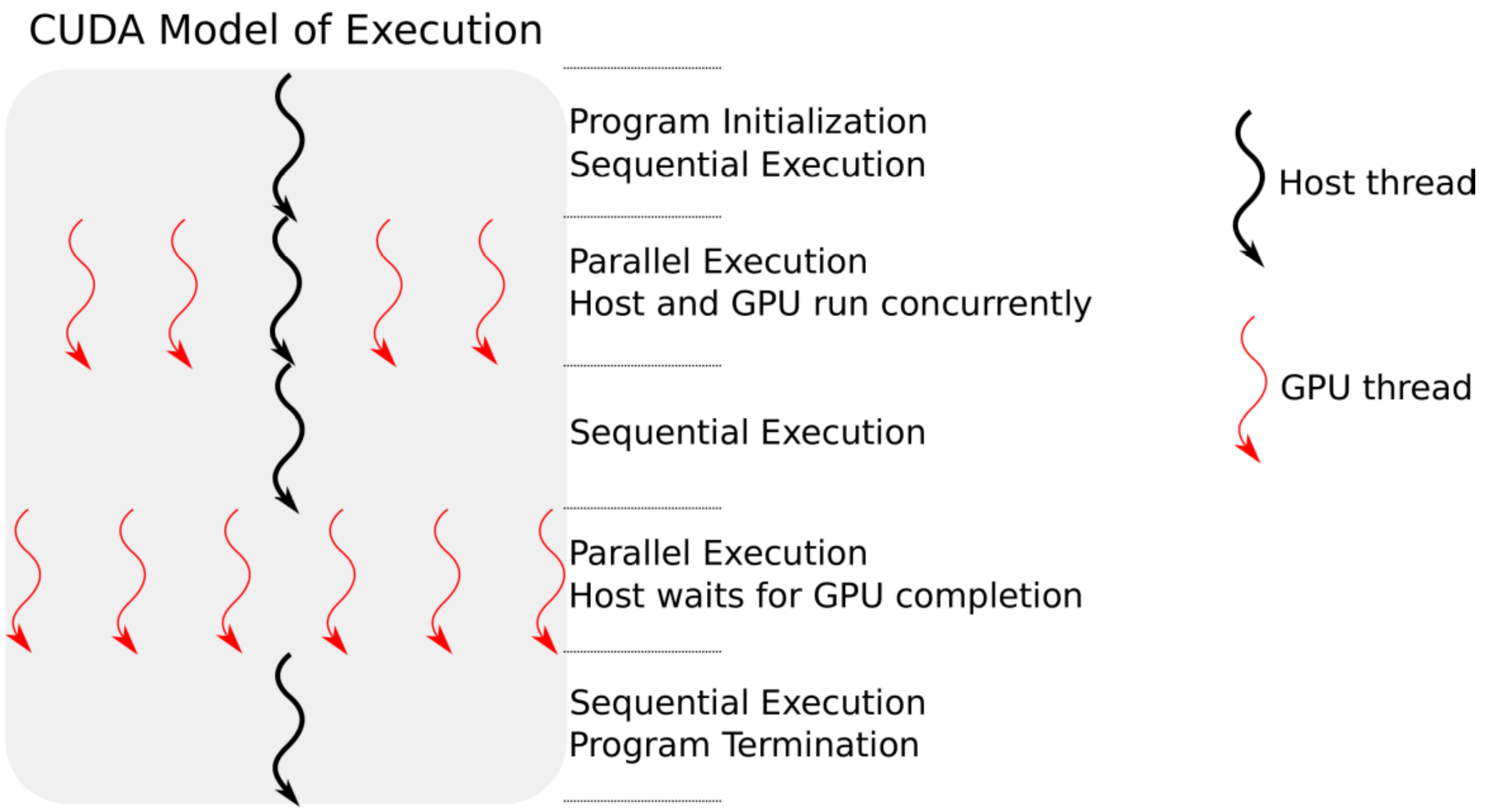
GPU software development platforms

- **CUDA** : Compute Unified Device Architecture. CUDA provides two sets of APIs (a low and a higher level one) and it is available freely for Windows, MacOS X and Linux operating systems. Major drawback : NVidia hardware only.
- **OpenCL** : Open Computing Language is an open standard for writing programs that can execute across a variety of heterogeneous platforms that include GPUs, CPU, DSPs or other processors. OpenCL is supported by both NVidia and AMD. It is the primary development platform for AMD and Intel GPUs. OpenCL's programming model matches closely the one offered by CUDA.
- **SYCL** : C++ abstraction of OpenCL. Intel's **oneAPI** is based on SYCL.
- **OpenMP** : the Open Multi-Processing standard was originally developed for shared-memory programming of multicore CPUs. Since the OpenMP 5.0 specification published in November 2018, it can also target GPUs.

GPU software development platforms (II)

- **OpenACC** : An open specification for an API that allows the use of compiler directives (e.g. #pragma acc, in a similar fashion to OpenMP) to automatically map computations to GPUs or multicore chips, according to a programmer's hints.
- **Thrust** : A C++ template library that accelerates GPU software development by utilizing a set of container classes and a set of algorithms to automatically map computations to GPU threads. Thrust used to rely solely on a CUDA back-end, but since version 1.6 it can target multiple device back-ends, including CPUs.
- **ArrayFire** : A comprehensive GPU function library that covers mathematics, signal and image processing, statistics, and other scientific domains. ArrayFire functions operate on arrays of data, in a similar fashion to Thrust. ArrayFire has been released under an Open Source Software license since 2014.

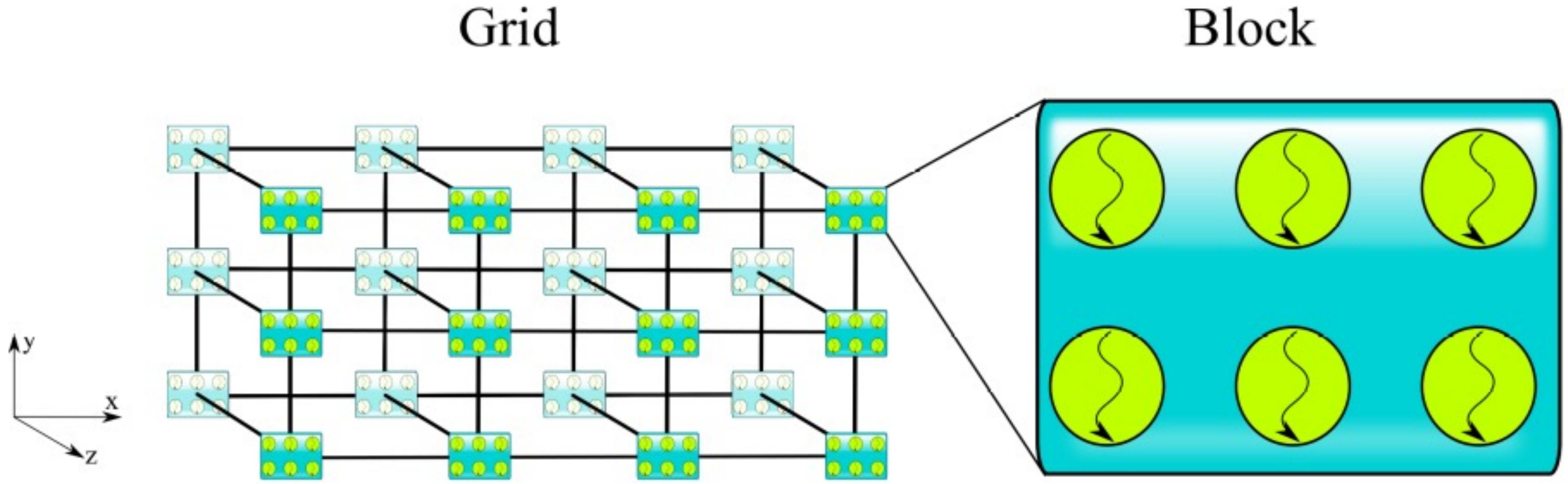
CUDA execution model



CUDA threads, blocks, and grids

- How do we spawn the hundreds or thousands of threads required?
  - CUDA's answer to this is to spawn all the threads as a group, running the “same function” with a set of parameters that apply to all.
  - The function is called a **kernel**.
- How do we initialize the individual threads so that each one does a different part of the work?
  - Explicit initialization with different parameters is impossible.
  - Instead, CUDA organizes the threads in a 6-D structure (lower dimensions are also possible).
  - Each thread is aware of its “position” in the overall structure, via a set of **intrinsic variables**/structures. With this information a thread can map its position to the subset of data that it is assigned to.

CUDA threads, blocks and grids (2)



- The above thread configuration is launched via:

```
dim3 block(3,2);
dim3 grid(4,3,2);
foo<<<grid, block>>>();
```

- The <<<>>> part of the launch statement, is called the **execution configuration**.

CUDA threads, blocks and grids (3)

- The dimension limits are determined by the **Compute Capability** of a device.
- The Compute Capability (CC) of a device references the underlying GPU **core*** architecture and associated hardware characteristics.

	Compute Capability			
Item	**5.x**	**6.x**	**7.x**	**8.0**
GPU family	Maxwell	Pascal	Turing	Ampere
Sample GPU	980	1080	2080	A100
Max. grid dimensions	3			
Grid maximum x-dimension	$2^{31}-1$			
Grid maximum y/z-dimension	$2^{16}-1$			
Max. block dimensions	3			
Block max. x/y-dimension	1024			
Block max. z-dimension	64			
Max. threads per block	1024			

(*) Core and not chip. GPU chips of the same architecture (and CC) can have different number of cores.

Exec. configuration examples

- Assuming we have

```
dim3 b(3,3,3);
dim3 g(20,100);
```

- `foo<<<g, b>>>();` // Run a 20x100 grid made of 3x3x3 blocks
- `foo<<<10, b>>>();` // Run a 10-block grid, each block made by 3x3x3 threads
- `foo<<<g, 256>>>();` // Run a 20x100 grid, made of 256 threads
- `foo<<<g, 2048>>>();` // An invalid example: maximum block size is 1024 threads for all compute capabilities
- `foo<<<5, g>>>();` // Another invalid example, that specifies a block size of 20x100=2000 threads
- `foo<<<10, 256>>>;` // simplified configuration for a 1D grid of 1D blocks

Hello World in CUDA

```
// File : hello.cu
#include <stdio.h>
#include <cuda.h>

__global__ void hello()
{
    printf("Hello world\n");
}

int main()
{
   hello<<<1,10>>>();
   cudaDeviceSynchronize();
   return 1;
}
```

Can be called from the host or the device

A kernel is always declared as `void`

Blocks until the CUDA kernel terminates

- To compile and run:

```
$ nvcc hello.cu -o hello
$ ./hello
```

Compiles for CC 3.0 by default

Function decorations

- **`__device__`** : A function that can only be called from within a kernel, i.e., not from the host.
- **`__host__`** : A function that can only run on the host.
- The `__host__` qualifier is typically omitted, unless used in combination with `__device__` to indicate that the function can run on both the host and the device.

`__global__` : can be called from the host and executed on the device. Device code can also call `__global__` functions, using a feature called dynamic parallelism.

Execution configuration related intrinsic variables

- `blockDim` : Contains the size of each block, e.g. (B_x, B_y, B_z)
- `gridDim` : Contains the size of the grid, in blocks, e.g. (G_x, G_y, G_z).
- `threadIdx` : The (x, y, z) position of the thread within a block,with $x \in [0, B_x - 1]$, $y \in [0, B_y - 1]$ and $z \in [0, B_z - 1]$
- `blockIdx` : The (b_x, b_y, b_z) position of a thread's block within the grid, with $b_x \in [0, G_x - 1]$, $b_y \in [0, G_y - 1]$ and $b_z \in [0, G_z - 1]$

The ID of a thread

- The only way a thread can *differentiate* its behavior is through the `threadIdx` and `blockIdx` variables.
- Each thread can be considered as an element of a *6-D array* with the following definition:
- `Thread t[gridDim.z][gridDim.y][gridDim.x]`

  `[blockDim.z][blockDim.y][blockDim.x];`
- The threadIdx is unique *only* within a block.
- Converting the `threadIdx` and `blockIdx` structures into a unique **scalar** ID can be done with the formula:

What should we do if fewer dimensions were used?

```
int myID = ( blockIdx.z * gridDim.x * gridDim.y  +
             blockIdx.y * gridDim.x +
             blockIdx.x ) * blockDim.x * blockDim.y * blockDim.z +
             threadIdx.z *  blockDim.x * blockDim.y +
             threadIdx.y * blockDim.x +
             threadIdx.x;
```

Hello World version #2

- Modifying the previous formula: for unused dimensions, set their size to 1 and the corresponding coordinates to 0. Example of a 3D grid of 1D blocks:

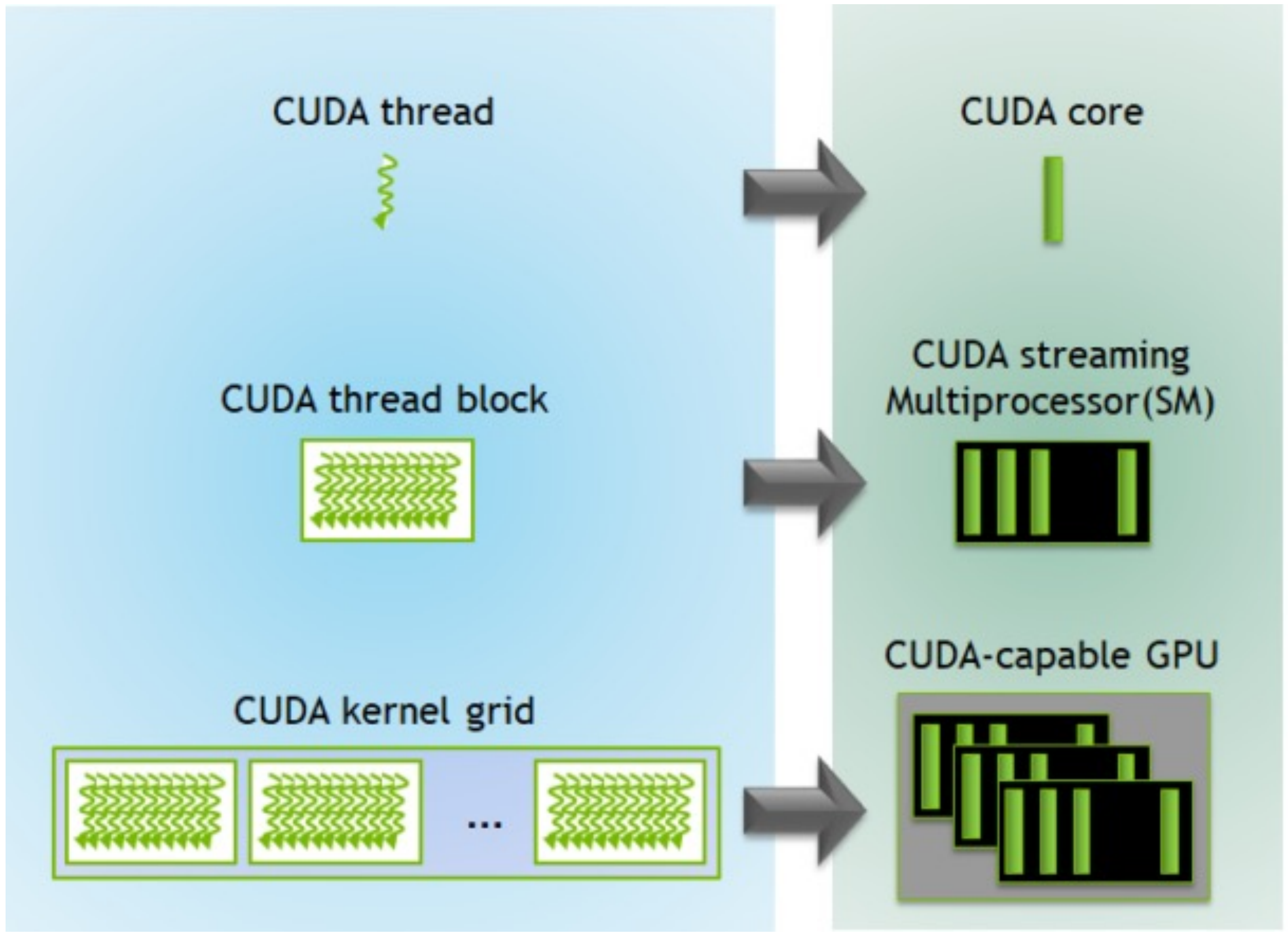
```
__global__ void hello()
{
  int myID = ( blockIdx.z * gridDim.x * gridDim.y  +
               blockIdx.y * gridDim.x +
               blockIdx.x ) * blockDim.x +
               threadIdx.x;

  printf ("Hello world from %i\n", myID);
}
```

CUDA execution model

- GPU cores are *vector processing units*, capable of applying the same instruction on a large collection of operands. So, when a kernel is run on a GPU core, the same instruction sequence is *synchronously* executed by a large collection of processing units called **Streaming Processors** (SPs), or CUDA cores.
- A group of SPs that execute under the control of a *single control unit* is called a **Streaming Multiprocessor** or SM.
- The number of SPs per SM varies with the core architecture. From 8 in early NVidia designs it has gone up to 192 cores/SM in Kepler. For the last GPU generations, NVidia has kept this number at 64 cores/SM.
- A GPU can contain multiple SMs, each running its own kernel. Each thread runs on its own SP.
- NVidia calls this execution model **Single Instruction Multiple Threads** (SIMT).
- SIMT is *analogous* to SIMD. The only major difference is that in SIMT the size of the "vector" is determined by the software, i.e., the block size.

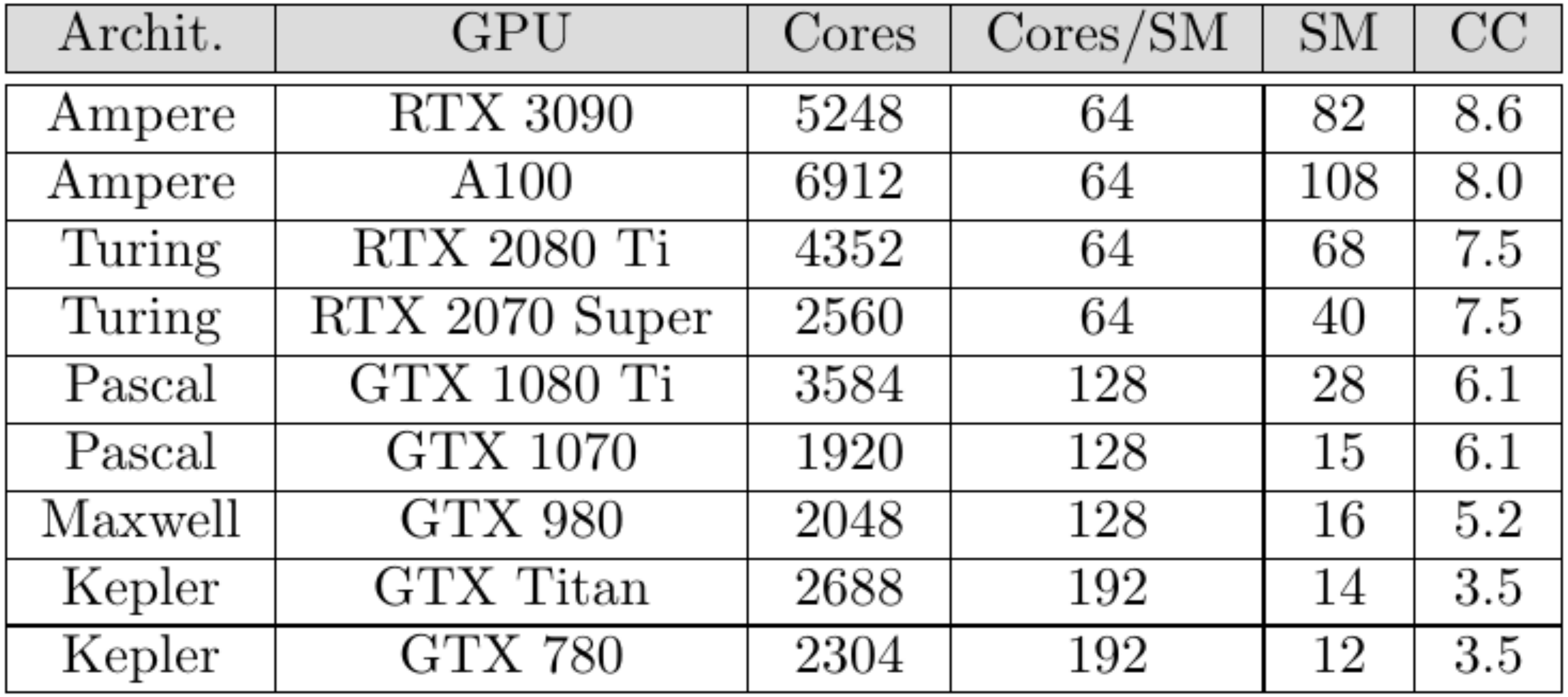
Threads running on cores



Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism).

One SM can run several concurrent CUDA blocks, depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time.

SMs and SPs sample specifications



Archit.	GPU	Cores	Cores/SM	SM	CC
Ampere	RTX 3090	5248	64	82	8.6
Ampere	A100	6912	64	108	8.0
Turing	RTX 2080 Ti	4352	64	68	7.5
Turing	RTX 2070 Super	2560	64	40	7.5
Pascal	GTX 1080 Ti	3584	128	28	6.1
Pascal	GTX 1070	1920	128	15	6.1
Maxwell	GTX 980	2048	128	16	5.2
Kepler	GTX Titan	2688	192	14	3.5
Kepler	GTX 780	2304	192	12	3.5

- In Ampere, each SM has 64 INT32 and 64 FP32 cores that can execute in parallel. For this reason, NVidia actually reports double the core number in official documentation (e.g., 10496 for RTX 3090).

Warps and thread scheduling

- Each block in a grid runs within a single SM.
- Threads in a block are executed “synchronously” in groups called **warps**.
- The intrinsic variable `warpSize` can be *queried* for finding out this number. Currently, it is set to 32.
- The division of threads in warps is based on their intra-block ID as determined by:

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

- In the latest core architectures, each SM is equipped with **four warp schedulers**, which means that **four warps** may be running at the same time, each possibly following a separate execution path.

Thread scheduling (cont.)

- Thread switching in CUDA is “very cheap”.
- Each thread is assigned an “exclusive” part of the SM's resources (e.g., registers) for the entire duration of a block's execution.

	Compute capability						
Item	**3.5**	**5.0**	**6.0**	**6.1**	**7.0**	**7.5**	**8.0**
Concurrent kernels/device	32		128	32	128		
Max. resident blocks/SM	16	32				16	32
Max. resident warps/SM	64					32	64
Max. resident threads/SM	2048					1024	2048
32-bit registers/SM	64k						
Max. registers/thread	255						

Resident threads

Resident threads is the number of threads the GPU can load into its memory.

The GPU has a limited number of cores available. It *cannot* execute millions of threads at the same time. The GPU can only execute as many threads at the same time as it has cores/resources available.

However, sometimes some of these threads may *stall* for different reasons. Therefore, the GPU has some additional resources which is used to store/load additional threads onto the GPU. These threads are *not* yet executed but they are initialized so that they can be executed at a moments notice.

Thread scheduling (cont.)

- The timing of thread execution, or the assignment of blocks to SMs is *unspecified*.
- The relative order of threads' execution is also *unspecified*.
- For all purposes, we can only assume that a thread terminates when the block it belongs to terminates.
- When an SM completes a block, it switches to another one.
- A switch is also possible if the threads of a block stall while waiting. Having other **resident** blocks depends on resource availability (e.g., shared memory, registers).
- **Important:** The existence of multiple warp schedulers in SMs means that at any point in time multiple warps from the same block may be executing. This influences optimization efforts discussed later.

Instruction-level parallelism (ILP)

- Each warp scheduler can issue up to two instruction at the same time.
- The goal is to keep the GPU cores occupied as much as possible, instead of waiting for data to be fetched from RAM (while one instruction is executing the operands of the next one are "prepared").
- Two instructions can be issued only if they are *independent* of each-other.
- Example :

```
a = a * b;
d = b + e;
```

- Counter example:

```
a = a * b;
d = a + e;   // needs the value of a
```

Types of data dependences

Flow (True) Dependence

a= …

…= a

Anti Dependence

…= a

a = …

Output dependence

a = …

a = …

All dependences *must* be respected in parallel execution

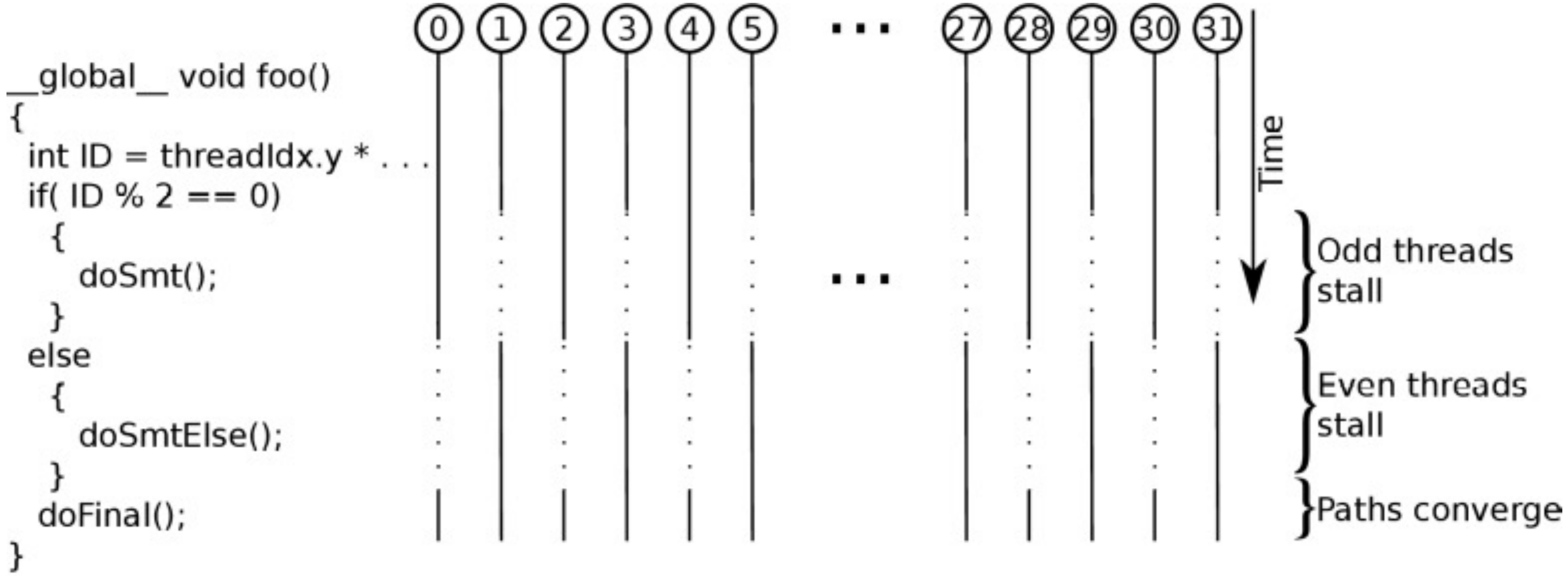
Anti and output dependences are also called **false/pseudo dependences**. Why?

Renaming can be used to eliminate anti and output dependences

Renaming can be easily done by a **compiler**

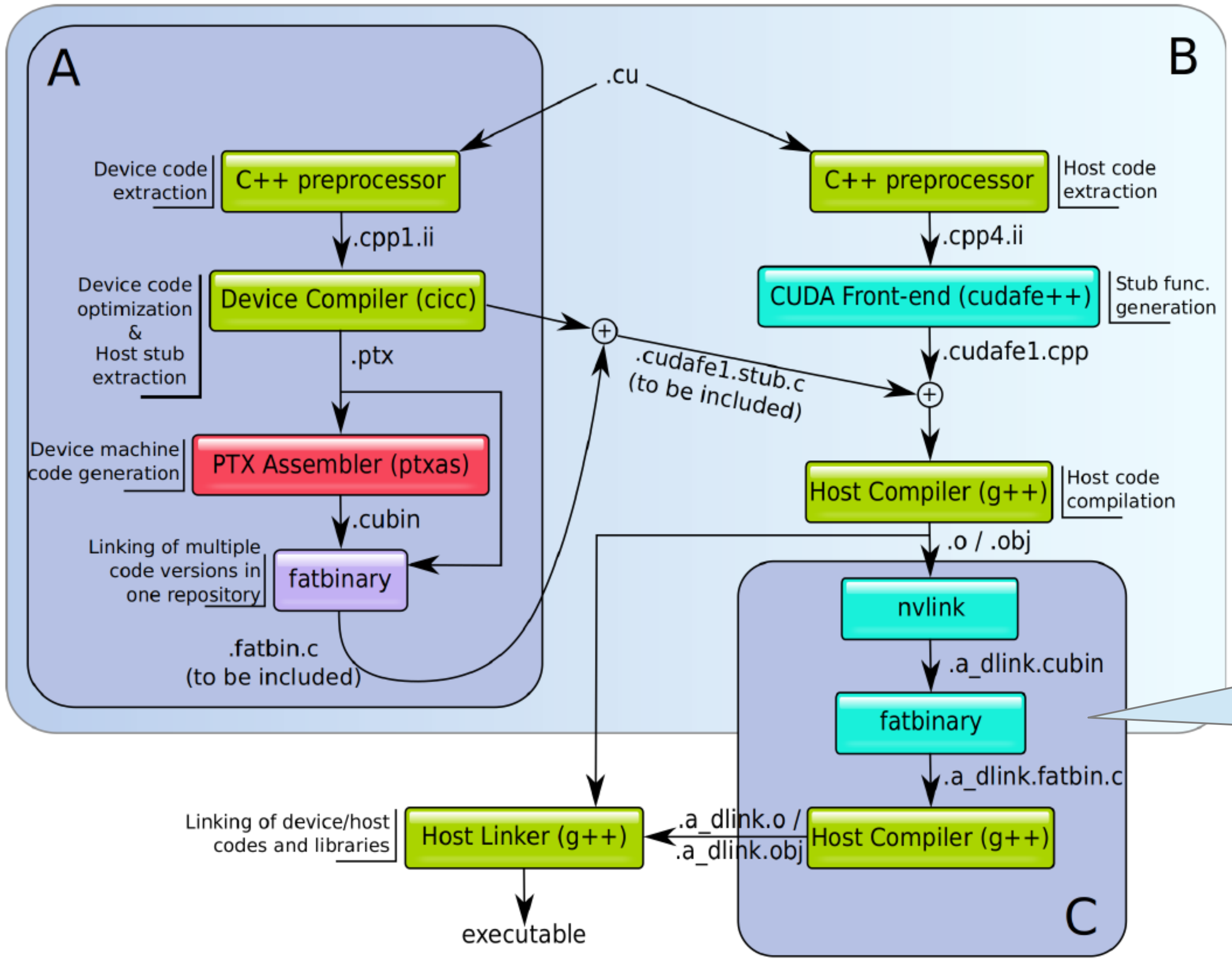
Diverging execution paths

- Along with ILP, another important issue that needs to be addressed at the kernel-level, is **path divergence**.
- A revealing illustration (dotted lines represent a stall):



```
__global__ void foo()
{
 int ID = threadIdx.y * . . .
 if( ID % 2 == 0)
   {
     doSmt();
   }
 else
   {
     doSmtElse();
   }
  doFinal();
}
```

CUDA compilation process



A: device part

B: host part

C: optional part for multi-file projects.

A fat binary can contain multiple executable GPU codes targeting different architectures

GPU memory management

- CPU and GPU memories are typically *disjoint* (AMD's APU's are an exception). Data transfer between them requires crossing the **PCIe bus**.
- GPU memory can be allocated in a similar fashion to using `malloc` on the CPU:

PCIe, or peripheral component interconnect express, is an **interface standard** for connecting **high-speed input output** (HSIO) components. Every high-performance computer motherboard has a number of PCIe slots you can use to add GPUs, RAID cards, WiFi cards, or SSD add-on cards.

```
cudaError_t cudaMalloc ( void** devPtr ,   // Host pointer address ,
                                           // where the address of
                                           // the allocated device
                                           // memory will be stored
                         size_t size )     // Size in bytes of the
                                           // requested memory block
```

- GPU memory has to be freed as well:

```
cudaError_t cudaFree ( void* devPtr );   // Parameter is the host
                                         // pointer address , returned
                                         // by cudaMalloc
```

Moving data between CPU and GPU

- The transfer is done via a call to the `cudaMemcpy` function.

```
cudaError_t cudaMemcpy ( void* dst,           // Destination block address
                         const void* src,     // Source block address
                         size_t count,        // Size in bytes
                         cudaMemcpyKind kind ) // Direction of copy.
```

- The options for the `kind` parameters are:
  - `cudaMemcpyHostToHost` : similar to plain `memcpy`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice` : for multi-GPU configurations
  - `cudaMemcpyDefault`, used when Unified Virtual Address space capability is available

Example : adding two vectors – memory allocation

```
int main (void)
{
  unique_ptr<int[]> ha, hb, hc; // host (h*) and
```

```
int *da, *db, *dc;              // device (d*) pointers
int i;

// host memory allocation
ha = make_unique<int[]>(N);
hb = make_unique<int[]>(N);
hc = make_unique<int[]>(N);

// device memory allocation
CUDA_CHECK_RETURN (cudaMalloc ((void **) &da, sizeof (int) * N));
CUDA_CHECK_RETURN (cudaMalloc ((void **) &db, sizeof (int) * N));
CUDA_CHECK_RETURN (cudaMalloc ((void **) &dc, sizeof (int) * N));

for (i = 0; i < N; i++)
  {
    ha[i] = rand () % 10000;
    hb[i] = rand () % 10000;
  }
```

Example : adding two vectors – data transfer and kernel invocation

```
// data transfer, host -> device
CUDA_CHECK_RETURN (cudaMemcpy (da, ha.get(), sizeof (int) * N, ↩
   cudaMemcpyHostToDevice));
CUDA_CHECK_RETURN (cudaMemcpy (db, hb.get(), sizeof (int) * N, ↩
   cudaMemcpyHostToDevice));

int grid = ceil (N * 1.0 / BLOCK_SIZE);
vadd <<< grid, BLOCK_SIZE >>> (da, db, dc, N);

CUDA_CHECK_RETURN (cudaDeviceSynchronize ());
// Wait for the GPU launched work to complete
CUDA_CHECK_RETURN (cudaGetLastError ());

// data transfer, device -> host
CUDA_CHECK_RETURN (cudaMemcpy (hc.get(), dc, sizeof (int) * N, ↩
   cudaMemcpyDeviceToHost));
```

Example : adding two vectors – releasing memory

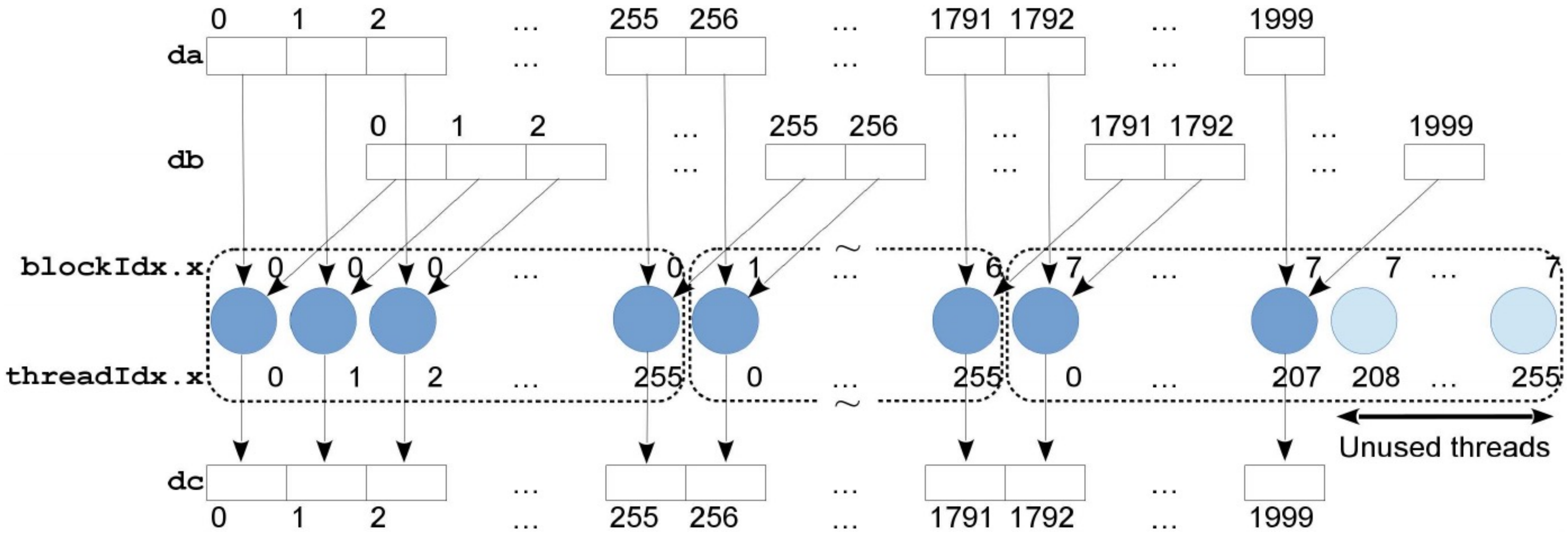
```
// correctness check
for (i = 0; i < N; i++)
  {
    if (hc[i] != ha[i] + hb[i])
      printf ("Error at index %i : %i VS %i\n", i, hc[i], ha[i] + hb[i]);
  }

CUDA_CHECK_RETURN (cudaFree ((void *) da));
CUDA_CHECK_RETURN (cudaFree ((void *) db));
CUDA_CHECK_RETURN (cudaFree ((void *) dc));
CUDA_CHECK_RETURN (cudaDeviceReset ());
```

Kernel code:

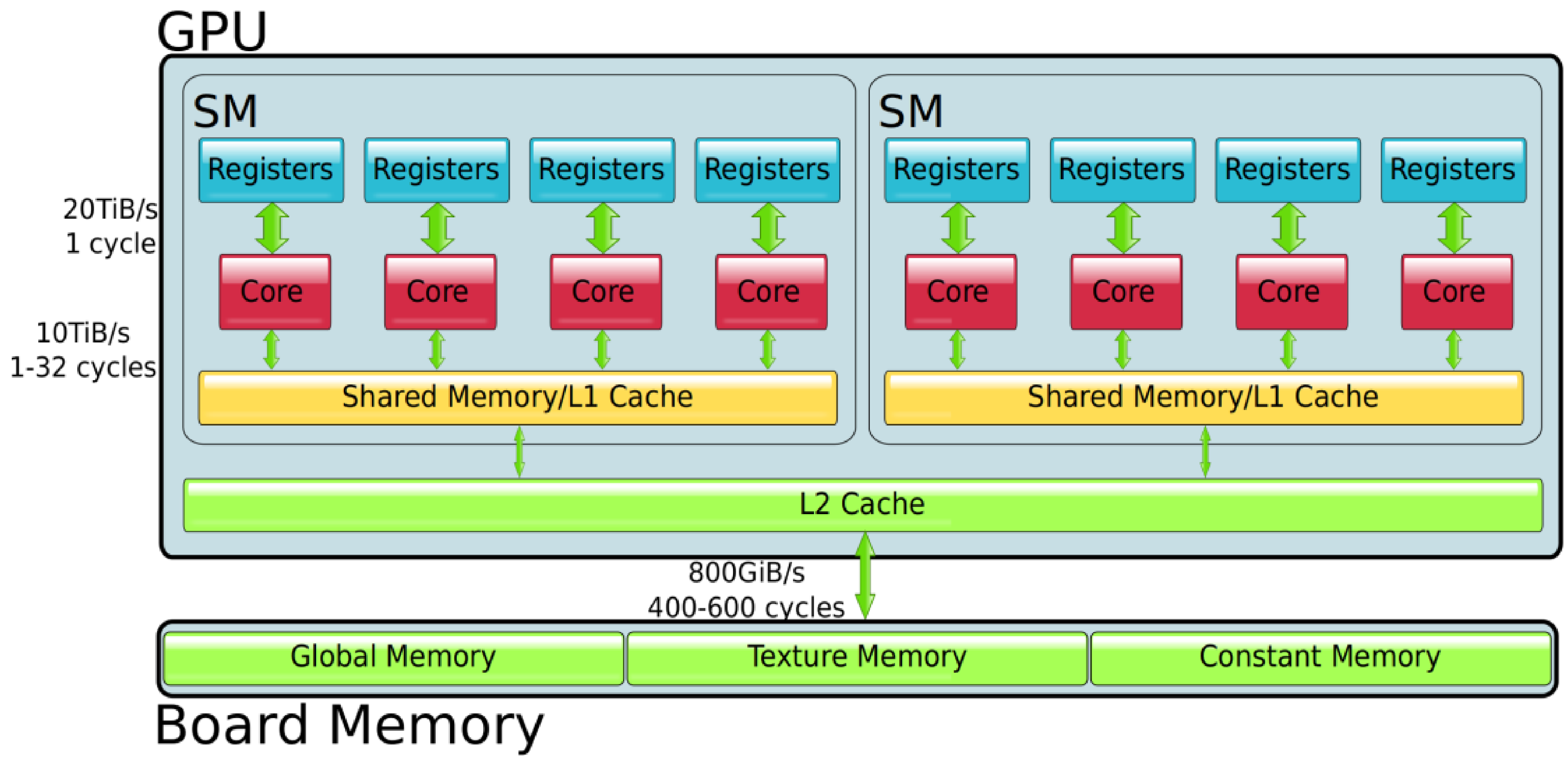
```
__global__ void vadd (int *a, int *b, int *c, int N)
{
  int myID = blockIdx.x * blockDim.x + threadIdx.x;
  if (myID < N)      // in case myID exceeds the array bounds
    c[myID] = a[myID] + b[myID];
}
```

Example : adding two vectors – illustration



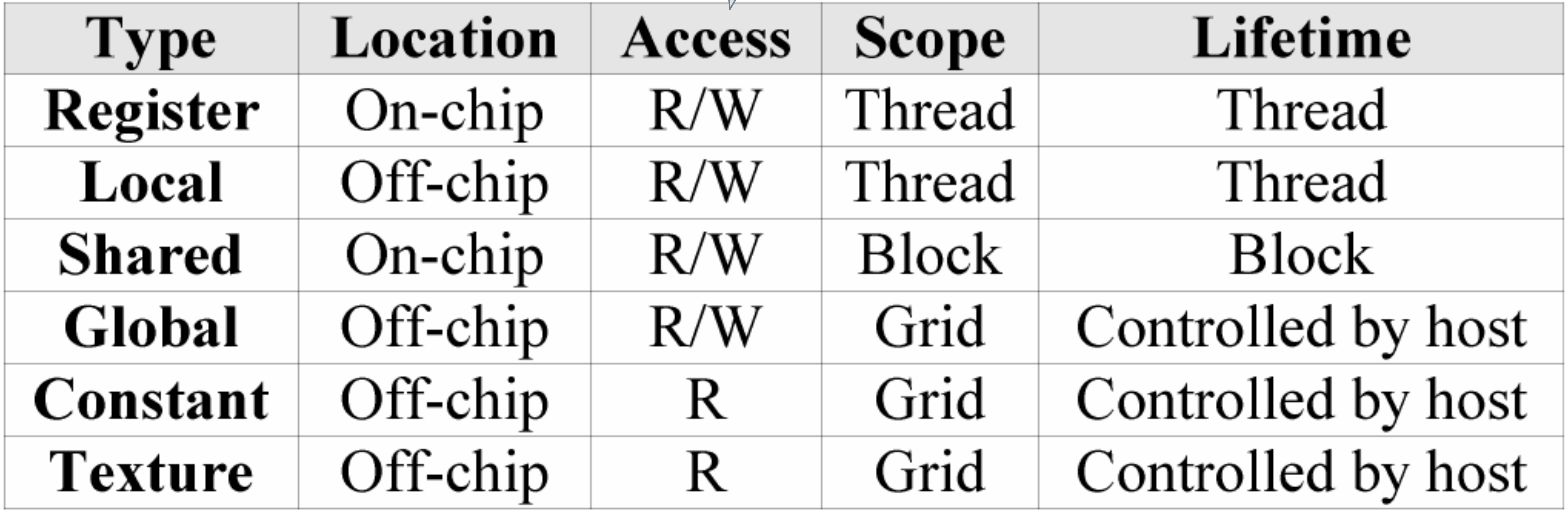
GPU memory hierarchy

Typical memory bandwidth and latency shown.



GPU memory characteristics

By kernel



Type	Location	Access	Scope	Lifetime
Register	On-chip	R/W	Thread	Thread
Local	Off-chip	R/W	Thread	Thread
Shared	On-chip	R/W	Block	Block
Global	Off-chip	R/W	Grid	Controlled by host
Constant	Off-chip	R	Grid	Controlled by host
Texture	Off-chip	R	Grid	Controlled by host

Local memory/registers

- The only common thing between these two is that they are used to hold automatic variables.
- Local memory is actually off-chip. "Local" just refers to the scope of its contents (thread-only).
- Local memory locations can be cached in L1 cache
- GPUs provide large register files to avoid having to use off-chip memory frequently.
- Each thread in a block is assigned its own part of the register file. Hence, the registers/thread and the threads/block are critical numbers.
- The number of registers allocated can be reported with an appropriate switch by `nvcc`:

```
$ nvcc -Xptxas -v warpFixMultiway.cu
ptxas info    : 14 bytes gmem
ptxas info    : Compiling entry function '_Z3foov' for 'sm_30'
ptxas info    : Function properties for _Z3foov
    8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 14 registers, 320 bytes cmem[0]
```

Occupancy

- NVidia defines as device **occupancy**, the ratio of allowed resident warps for a kernel, over the maximum possible resident warps for a device:

$$occupancy = \frac{resident\_warps}{maximum\_warps}$$

- NVidia's guidelines calls for maximizing occupancy, as a means to maximize performance.
- NVidia provides an **occupancy calculator** that allows programmers to experiment with execution configurations.
  - Homework: explore the occupancy calculator
- In practice getting maximum performance is a much more involved affair that requires **experimentation**: actually testing different execution configurations.

Shared memory

- Shared memory provides a fast, on-chip data repository that can be accessed by all the threads in a block.
- Shared among *all cores* of an SM
- The RAM used for shared memory can be also used as L1-cache.
- A programmer can specify the *balance* between cache and shared memory space by using the `cudaFuncSetCacheConfig` function.
- E.g., a 64KB RAM can be partitioned into 16KB shared memory and 48KB L1-cache
- Note: partitioning between shared memory and L1-cache is only a suggestion; CUDE runtime is the ultimate decider

Shared memory

Q: How can we specify that the holding place of some data will be the shared memory of the SM and not the global memory of the device?

A: via the `__shared__` specifier

Shared memory can be *statically* or *dynamically* allocated.

Static allocation can take place if the size of the required arrays is known at compile-time. Dynamic allocation is needed if shared memory requirements can only be calculated at runtime, i.e., upon kernel invocation.

To facilitate dynamic allocation, the execution configuration has an alternative syntax, with a *third parameter* holding the size in bytes of the shared memory to be reserved.

Shared memory (cont.)

- Shared memory can be used as:
  - A holding place for very *frequently-used data* that would otherwise require global memory access (e.g., an array of counters).
  - A fast *mirror of data* that reside in global memory, if they are to be accessed multiple times (e.g., part of the input data or a lookup table).
  - As a fast way for cores within a SM, e.g., to *share* data.
- Shared memory contents *cannot* be accessed from the host.
- Shared memory contents have the scope of a block of threads.

Allocating shared memory

- **Statically** (hard-coded size):

```
__global__ void foo(...) {
    __shared__ int counter[100];
    ...
```

Per block, not per thread

- **Dynamically** (size specified during kernel invocation):

```
__global__ void foo(...) {
extern __shared__ int counter[];
    ...
foo<<<grid, block, 200>>>(...);
```

- Size is in bytes and corresponds to the **bundle** of all shared memory data items.

Initializing shared memory

- As shared memory *cannot* be accessed from the host; the only way to initialize it prior to using it is to have CUDA threads perform *initialization*.
- Example:

```
__global__ void foo(...) {
    __shared__ int counter[BLOCKSIZE];
   int myID = threadIdx.x;
   counter[myID] = 0;
   __syncthreads ();
```

- The `__synchthreads` is a *block-wide barrier*, that ensures that all warps have reached that statement before the threads can resume execution.

Example: Histogram calculation

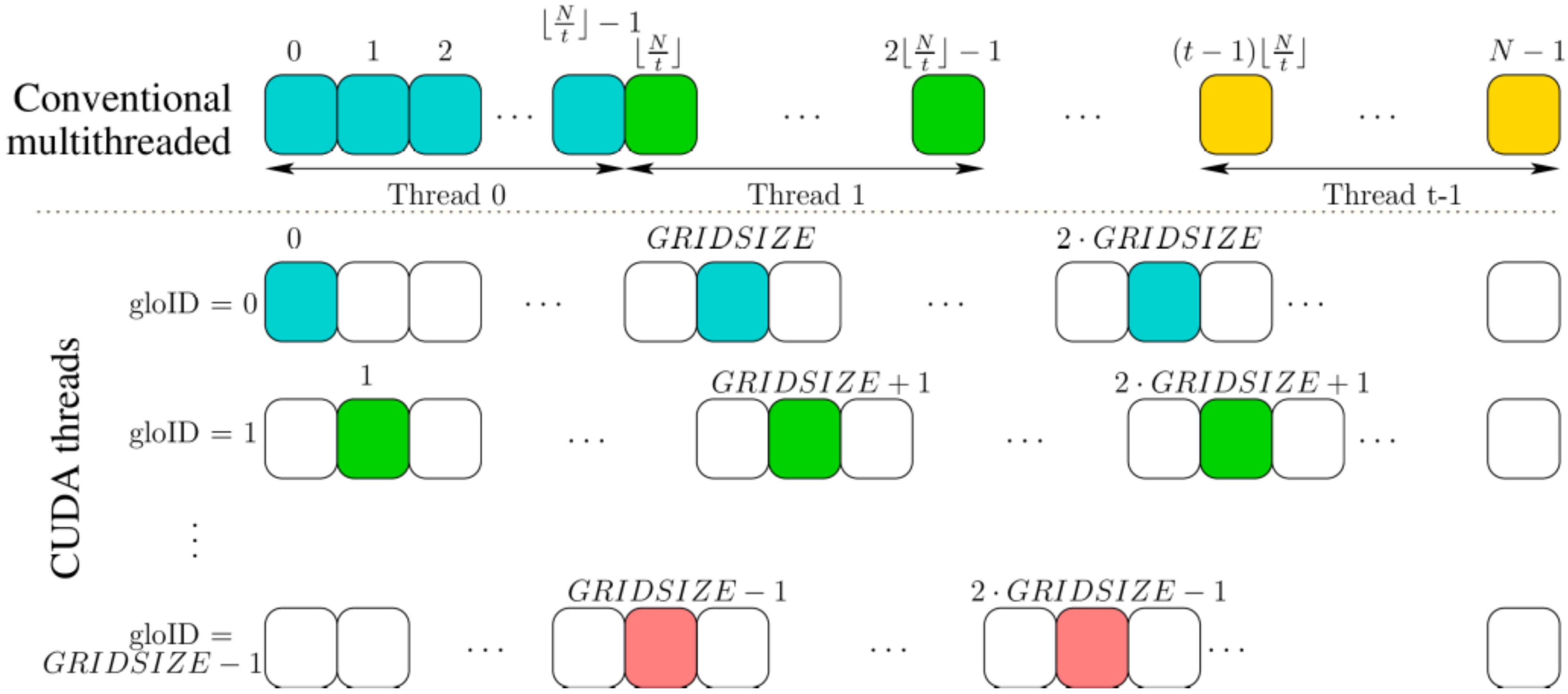
- The CPU code is as follows:

```
void CPU_histogram (unsigned char *in, int N, int *h, int bins)
{
  int i;
  // initialize histogram counts
  for (i = 0; i < bins; i++)
    h[i] = 0;

  // accummulate counts
  for (i = 0; i < N; i++)
    h[in[i]]++;
}
```

- *Why is it imperative to have a CPU version of the code?*

GPU histogram calculation : visualizing the work partitioning



Note the difference between the *conventional multithreaded version* and *CUDA solution* (mostly due to memory coalescing).

GPU histogram calculation – static shared memory allocation

- The idea is to keep local counts in shared memory.
- If the number of bins is known a priori:

```
static const int BINS = 256;

__global__ void GPU_histogram_static (int *in, int N, int *h)
{
  int gloID = blockIdx.x * blockDim.x + threadIdx.x;
  int locID = threadIdx.x;
  int GRIDSIZE = gridDim.x * blockDim.x;   // total number of threads
  __shared__ int localH[BINS];             // shared allocation
  int i;

  // initialize the local, shared-memory bins
  for (i = locID; i < BINS; i += blockDim.x)
    localH[i] = 0;

  // wait for all warps to complete the previous step
  __syncthreads ();
```

- locID is the position in the block the thread belongs to
- gloID is thread’s position in the grid

GPU histogram calculation (pt.2)

```
  //start processing the image data
  for (i = gloID; i < N; i += GRIDSIZE)
    {
      int temp = in[i];
      atomicAdd (localH + (temp & 0xFF), 1);
      atomicAdd (localH + ((temp >> 8) & 0xFF), 1);
      atomicAdd (localH + ((temp >> 16) & 0xFF), 1);
      atomicAdd (localH + ((temp >> 24) & 0xFF), 1);
    }

  // wait for all warps to complete the local calculations, before ↩
      updating the global counts
  __syncthreads ();

  // use atomic operations to add the local findings to the global ↩
      memory bins
  for (i = locID; i < BINS; i += blockDim.x)
    atomicAdd (h + i, localH[i]);
}
```

Atomic operations

- **`atomicAdd`** is overloaded, having an appropriate version for most of the primitive data types.

```
int atomicAdd(int* address, // Location to modify
              int val);     // Value to add
```

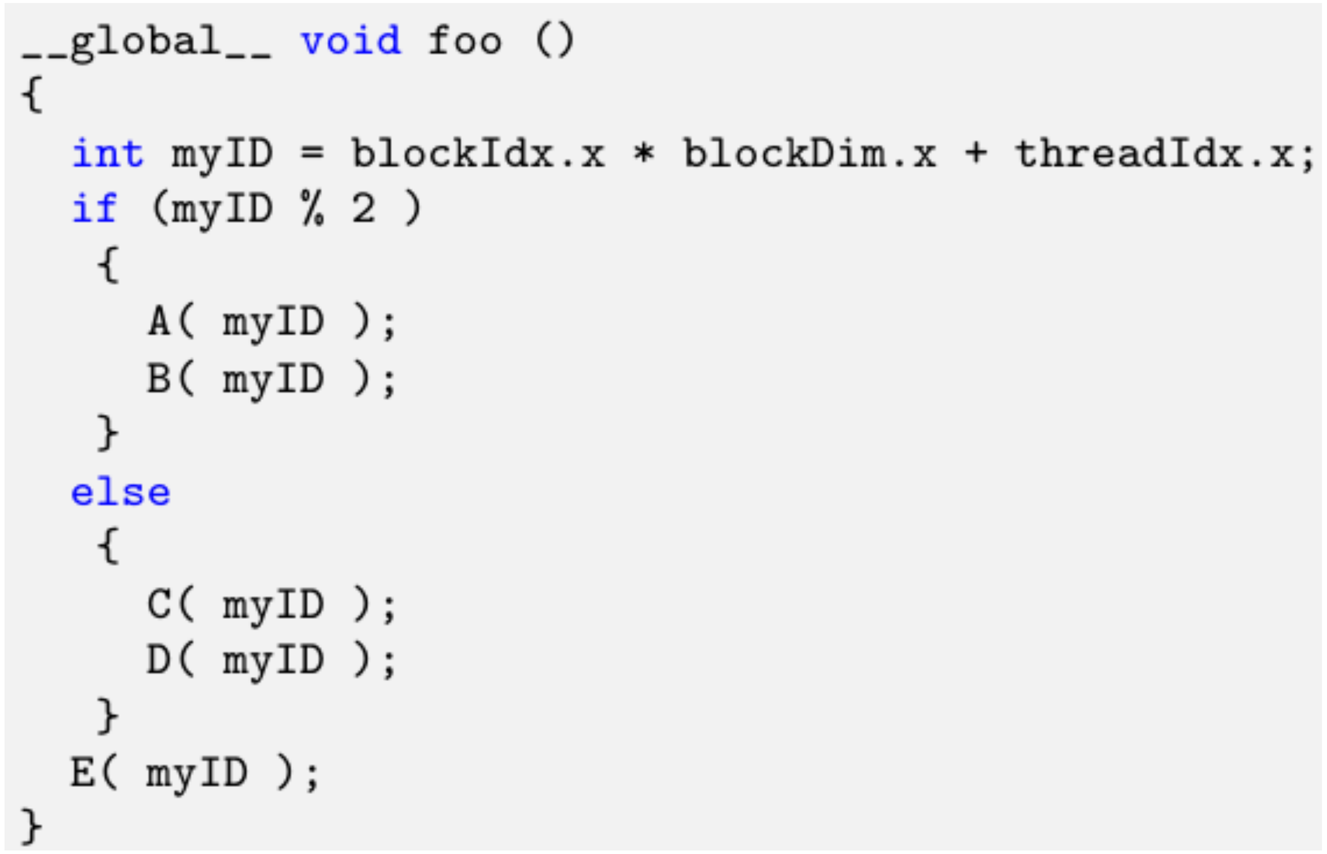
- A rich collection of atomic operations are available, including:
  - `atomicMax`, `atomicMin`
  - `atomicAnd`, `atomicOr`, `atomicXor`

Independent thread scheduling (ITS)

- ITS was introduced in the Volta GPU family.
- ITS affords a separate program counter and a separate call stack for each of the threads of a warp.
- This does *not* eliminate the performance penalty for thread divergence, but it allows divergent threads to continue executing in an interleaved fashion.
- It also permits **intra-warp cooperation** (e.g., signaling or data exchange) between threads on *different execution paths*.
- For example, threads of the same warp can operate on a lock without the possibility of a deadlock.

ITS example

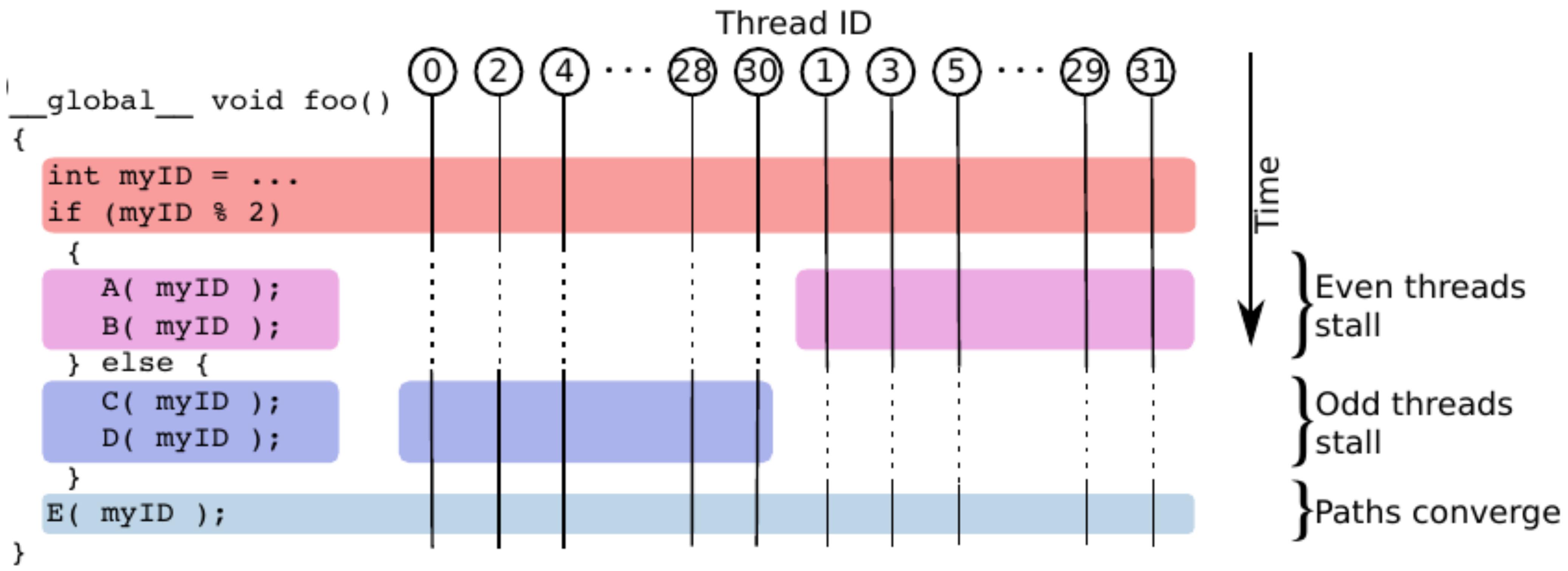
- The following produces a path divergence based on the `if`'s outcome:



```
__global__ void foo ()
{
  int myID = blockIdx.x * blockDim.x + threadIdx.x;
  if (myID % 2 )
   {
     A( myID );
     B( myID );
   }
  else
   {
     C( myID );
     D( myID );
   }
  E( myID );
}
```

ITS example (cont.)

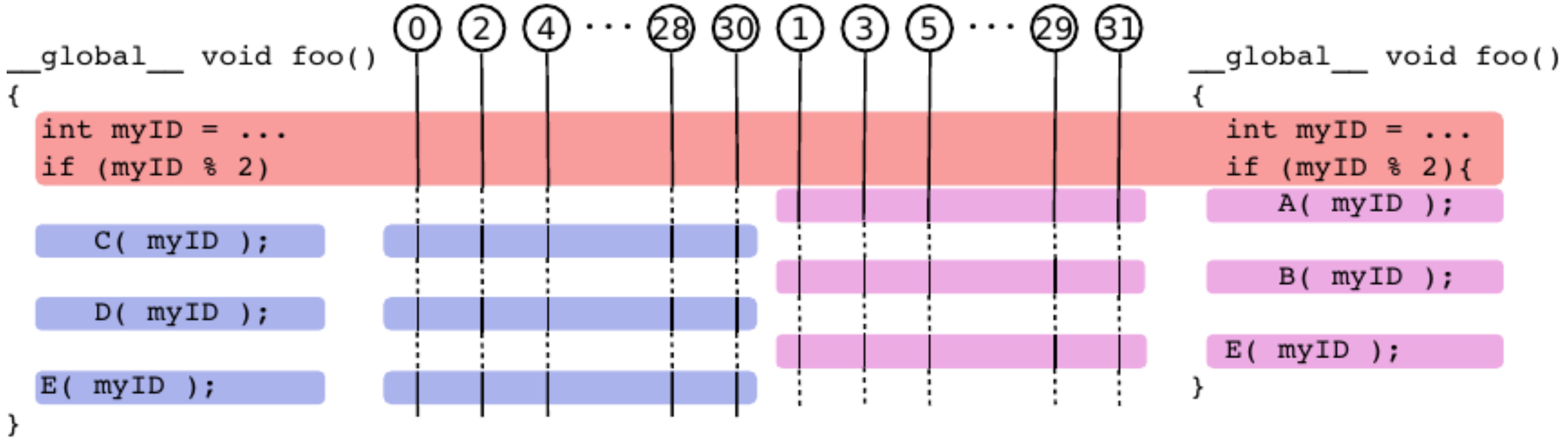
- Pre-ITS execution:



```
__global__ void foo()
{
  int myID = ...
  if (myID % 2)
   {
     A( myID );
     B( myID );
   } else {
     C( myID );
     D( myID );
   }
  E( myID );
}
```

ITS example (cont.)

- ITS based execution:



```
__global__ void foo()
{
  int myID = ...
  if (myID % 2)

        C( myID );

        D( myID );

  E( myID );
}
```

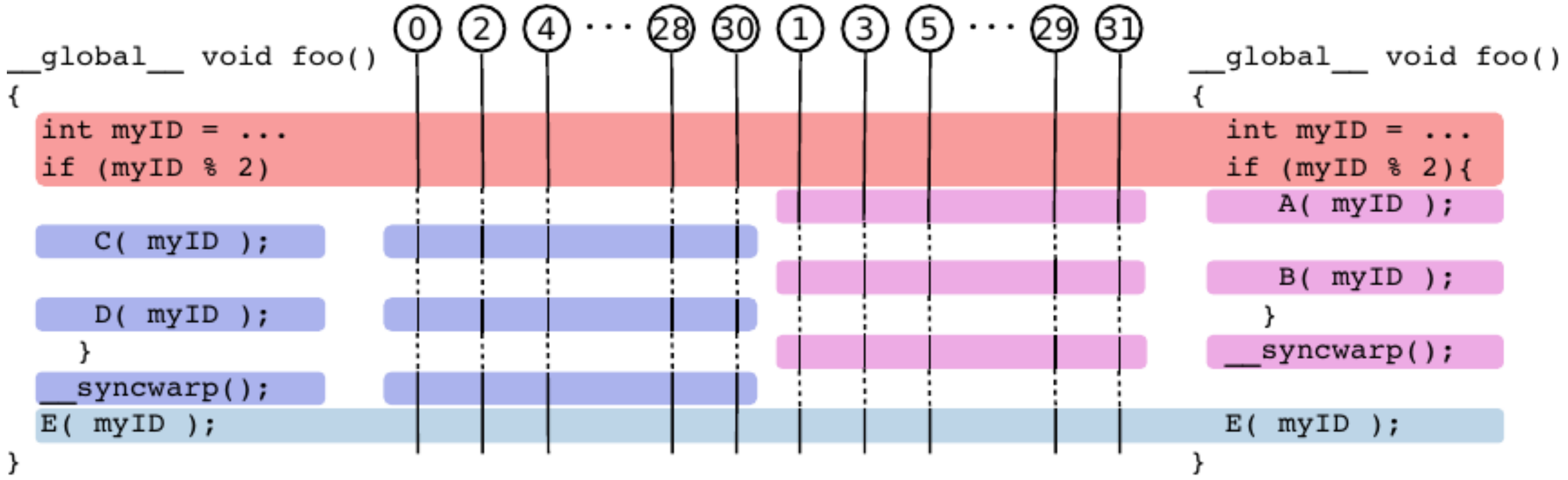
```
__global__ void foo()
{
  int myID = ...
  if (myID % 2){
        A( myID );

        B( myID );

  E( myID );
}
```

ITS example (cont.)

- ITS based execution with forced convergence:



```
__global__ void foo()
{
  int myID = ...
  if (myID % 2){
    A( myID );
    B( myID );
  }
  else {
    C( myID );
    D( myID );
  }
  __syncwarp();
  E( myID );
}
```

- This is accomplished by placing a `__syncwarp()` call following the closing bracket of the else block.

Summary: launching a kernel

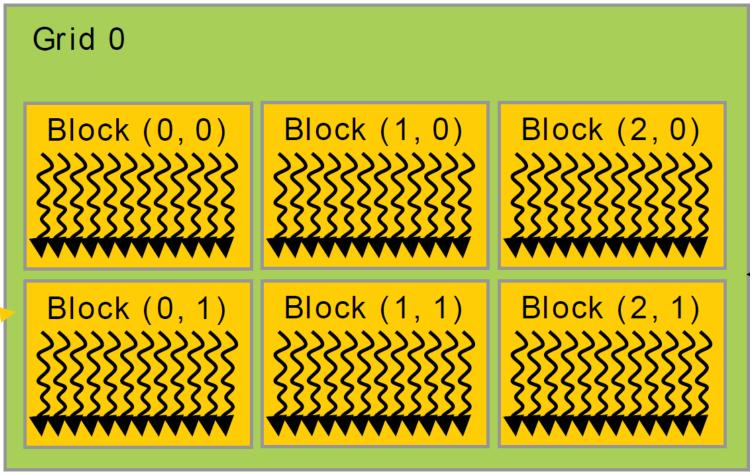
CPU program
(serial code)

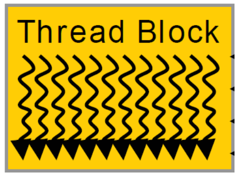
cudaMemcpy (…) — Copy data from CPU memory to GPU memory

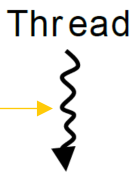
Function <<<nb,nt >>> — Launch a kernel with *nb* blocks, each with *nt* threads

cudaMemcpy (…) — Copy results from GPU memory to CPU memory

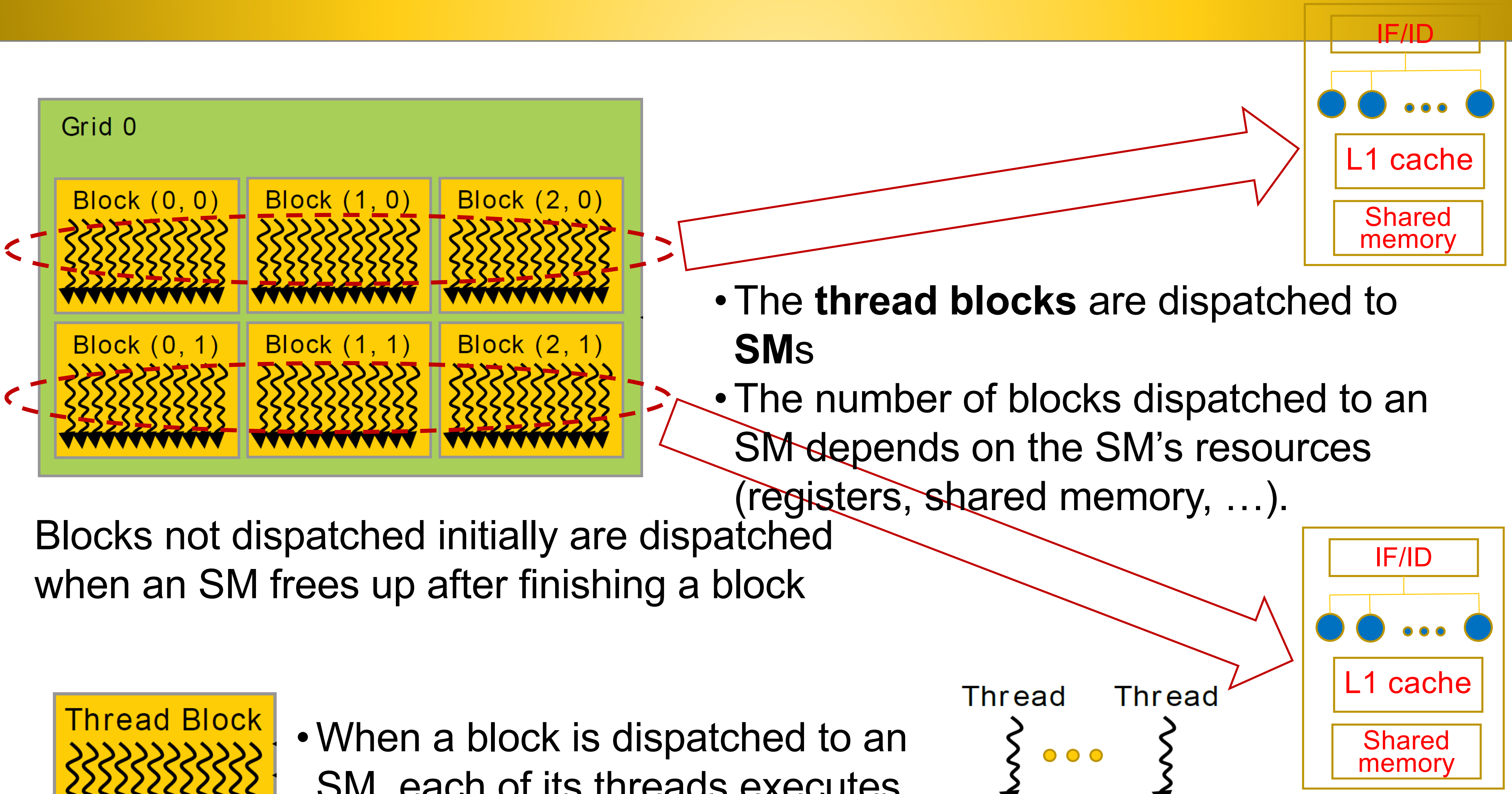
global Function (…) — Implementation of kernel (the function run by each GPU thread)





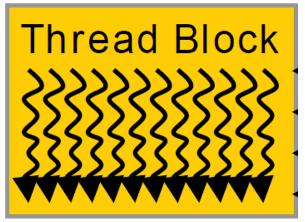


Summary: execution model

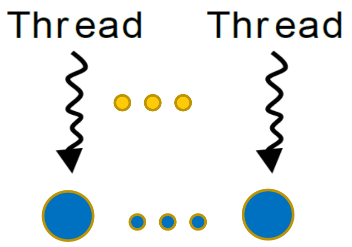


- The **thread blocks** are dispatched to **SMs**
- The number of blocks dispatched to an SM depends on the SM's resources (registers, shared memory, …).

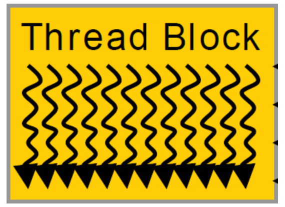
Blocks not dispatched initially are dispatched when an SM frees up after finishing a block



- When a block is dispatched to an SM, each of its threads executes on an SP in the SM.



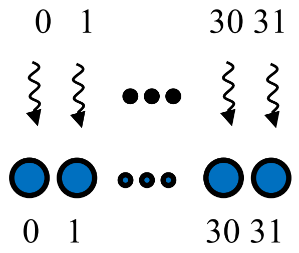
Summary: execution model



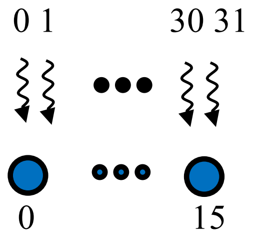
Each block (up to 1K threads) is divided into groups of 32 threads (called **warps**) – empty threads are used as fillers.

A warp executes as a SIMD **vector instruction** on the SM.

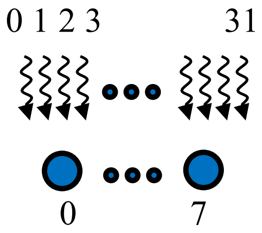
Depending on the number of SPs per SM:



If 32 SP per SM → 1 thread of a warp executes on 1 SP (32 lanes of execution, one thread per lane)



If 16 SP per SM → 2 threads are time multiplexed on 1 SP (16 lanes of execution, 2 threads per lane)

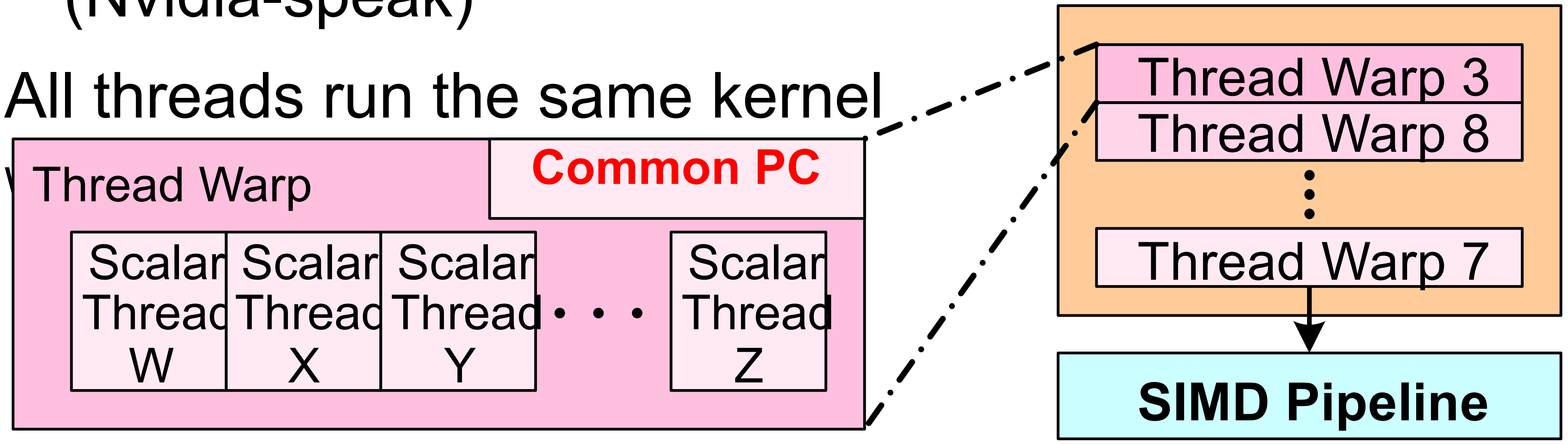


If 8 SP per SM → 4 threads are time multiplexed on 1 SP (8 lanes of execution, 4 threads per lane)

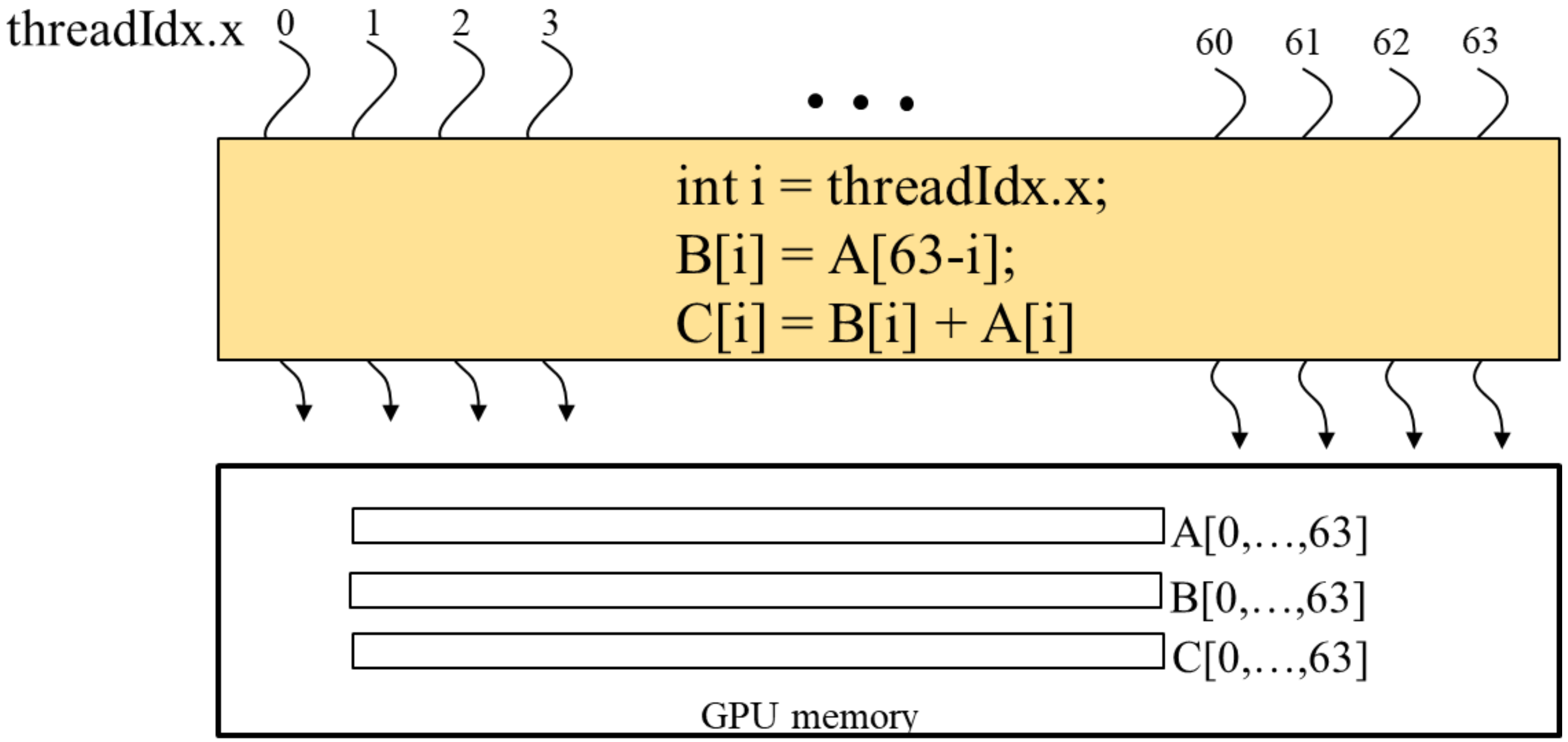
Summary: thread warps and SIMT

Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)

All threads run the same kernel

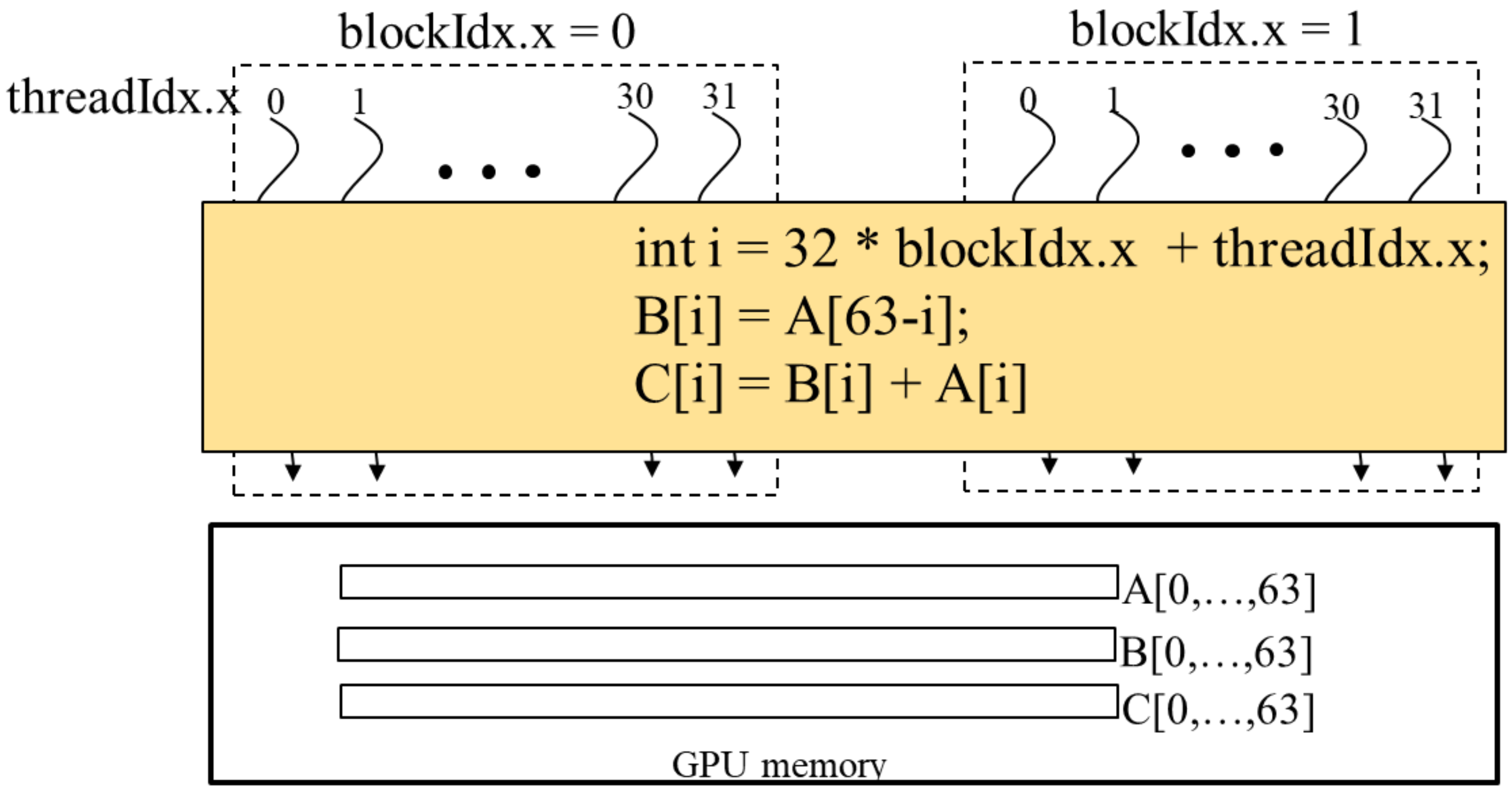


Summary: SIMT execution



Launched using **Kernel <<<1, 64>>>** : 1 block with 64 threads

Summary: SIMT execution



Launched using **Kernel <<<2, 32>>>** : 2 blocks each with 32 threads

Profiling CUDA code

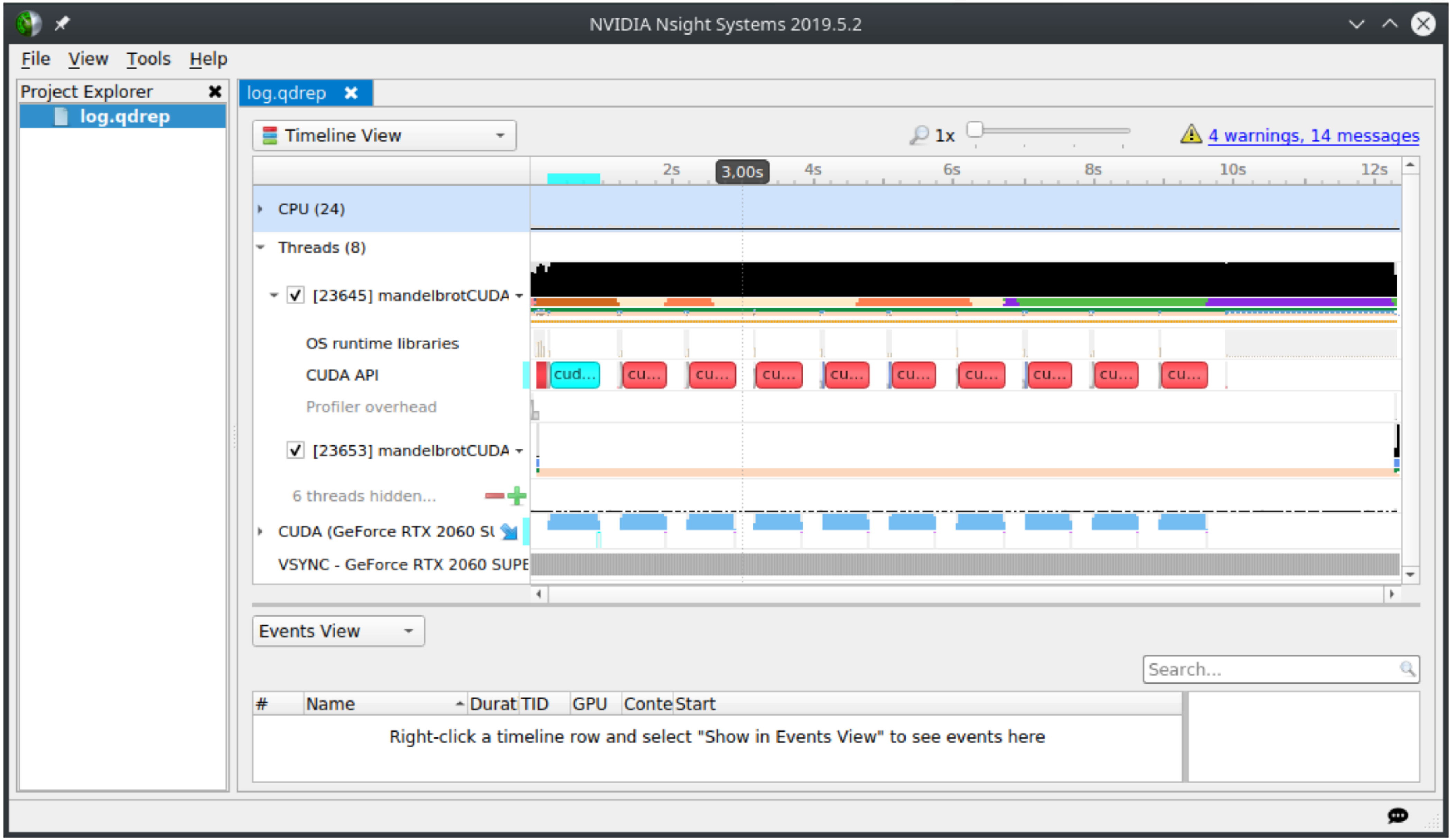
- Nsight Systems is the latest tool in the CUDA toolkit for tracing and profiling GPU execution.
- It is replacing NVidia's Visual Profiler for the current and next generation GPU architectures.
- Nsight Systems comes in two flavors:
  - Command line utility that one can use in a terminal. Example:

```
$ nsys profile -o log ./ffooProg sample.pgm
```

The above results in the creation of a log.qdrep file.

  - A GUI program (nsight-sys) that can be used to visualize the results of a profiling session.

Profiling CUDA code – Nsight-sys



#	Name	Durat	TID	GPU	Conte	Start

Right-click a timeline row and select "Show in Events View" to see events here