UNIVERSITÀ
DEGLI STUDI
FIRENZE

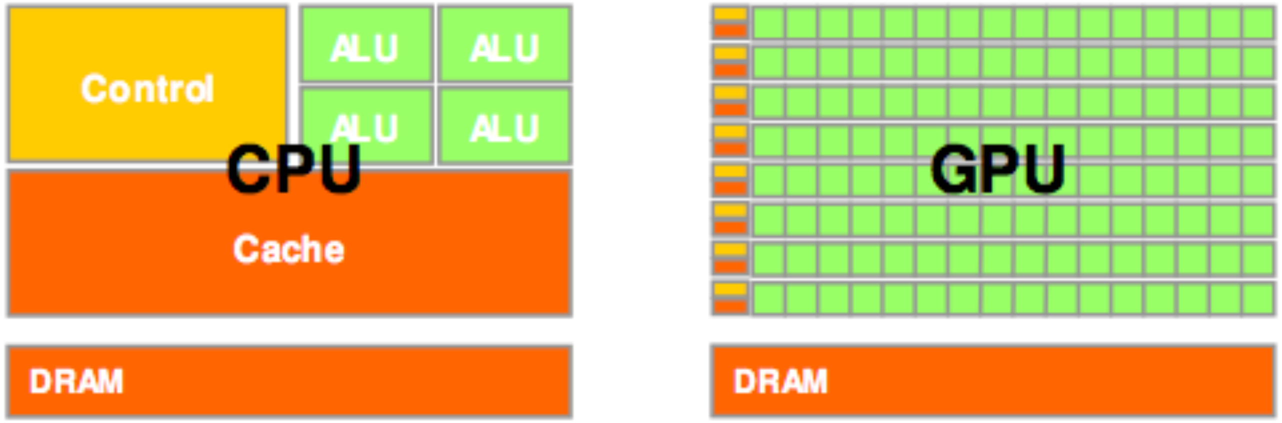
GPU programming basics

Prof. Marco Bertini

Data parallelism: GPU computing

CPUs vs. GPUs

- The design of a CPU is optimized for sequential code performance.
  - out-of-order execution, branch-prediction
  - large cache memories to reduce latency in memory access
  - multi-core
- GPUs:
  - many-core
  - massive floating point computations for video games
  - much larger bandwidth in memory access
  - no branch prediction or too much control logic: just compute



CPUs have latency oriented design:

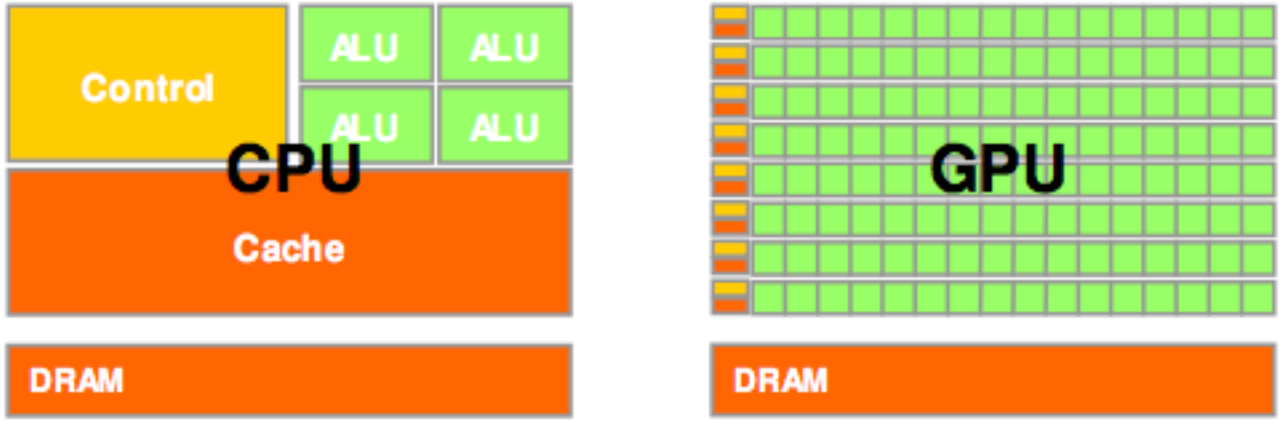
- Large caches convert long latency RAM access to short latency
- Branch pred., OoOE, operand forwarding reduce instructions latency
- Powerful ALU for reduced operation latency

GPUs have a throughput oriented design:

- Small caches to boost RAM throughput
- Simple control (no operand forwarding, branch prediction, etc.)
- Energy efficient ALU (long latency but heavily pipelined for high throughput)
- Require massive # threads to tolerate latencies

    - multi-core

- GPUs:
    - many-core
    - massive floating point computations for video games
    - much larger bandwidth in memory access
    - no branch prediction or too much control logic: just compute



CPUs and GPUs

- GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well;
- One should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.
- We are going to deal with **heterogenous architectures**: CPUs + GPUs.

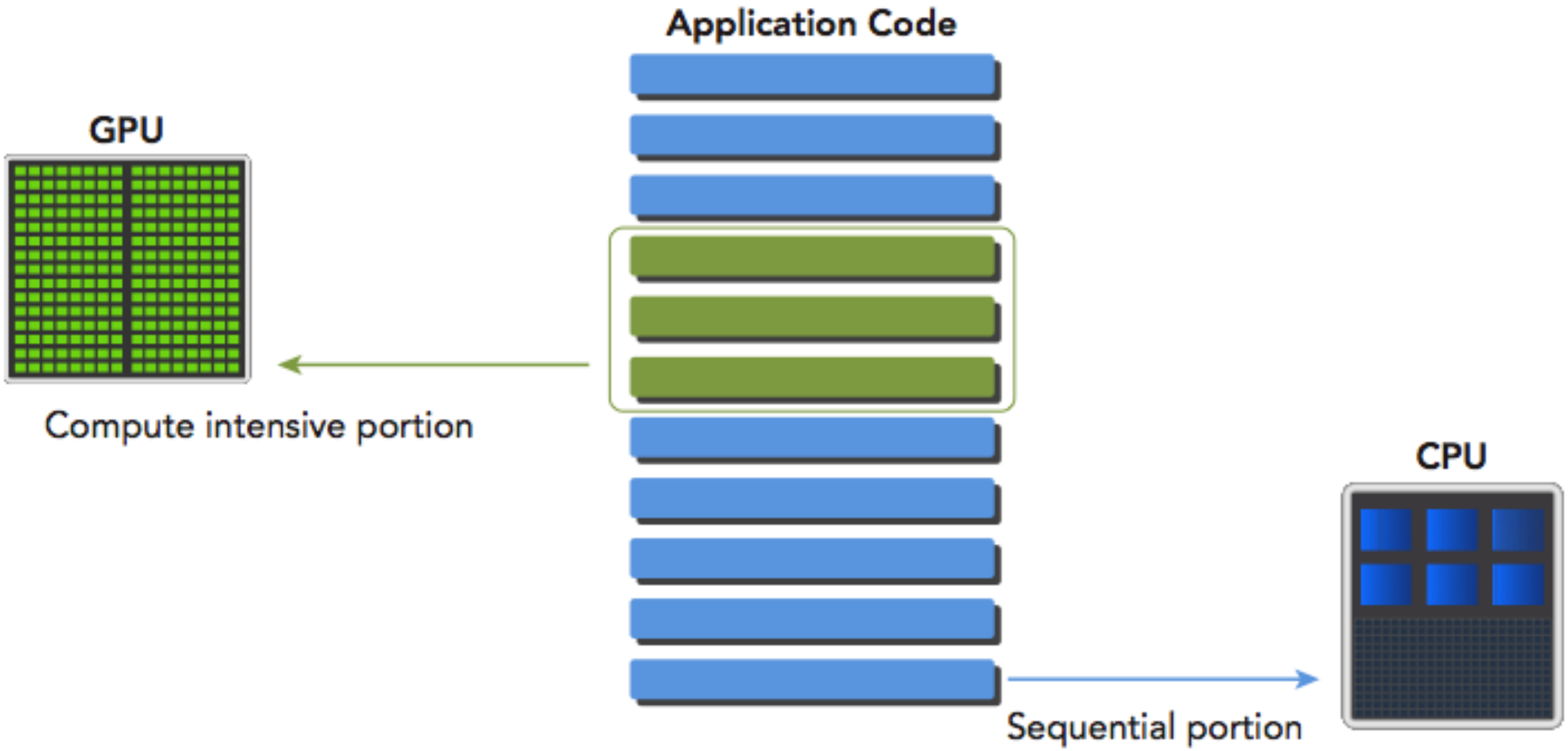
Heterogeneous Computing

- CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks.

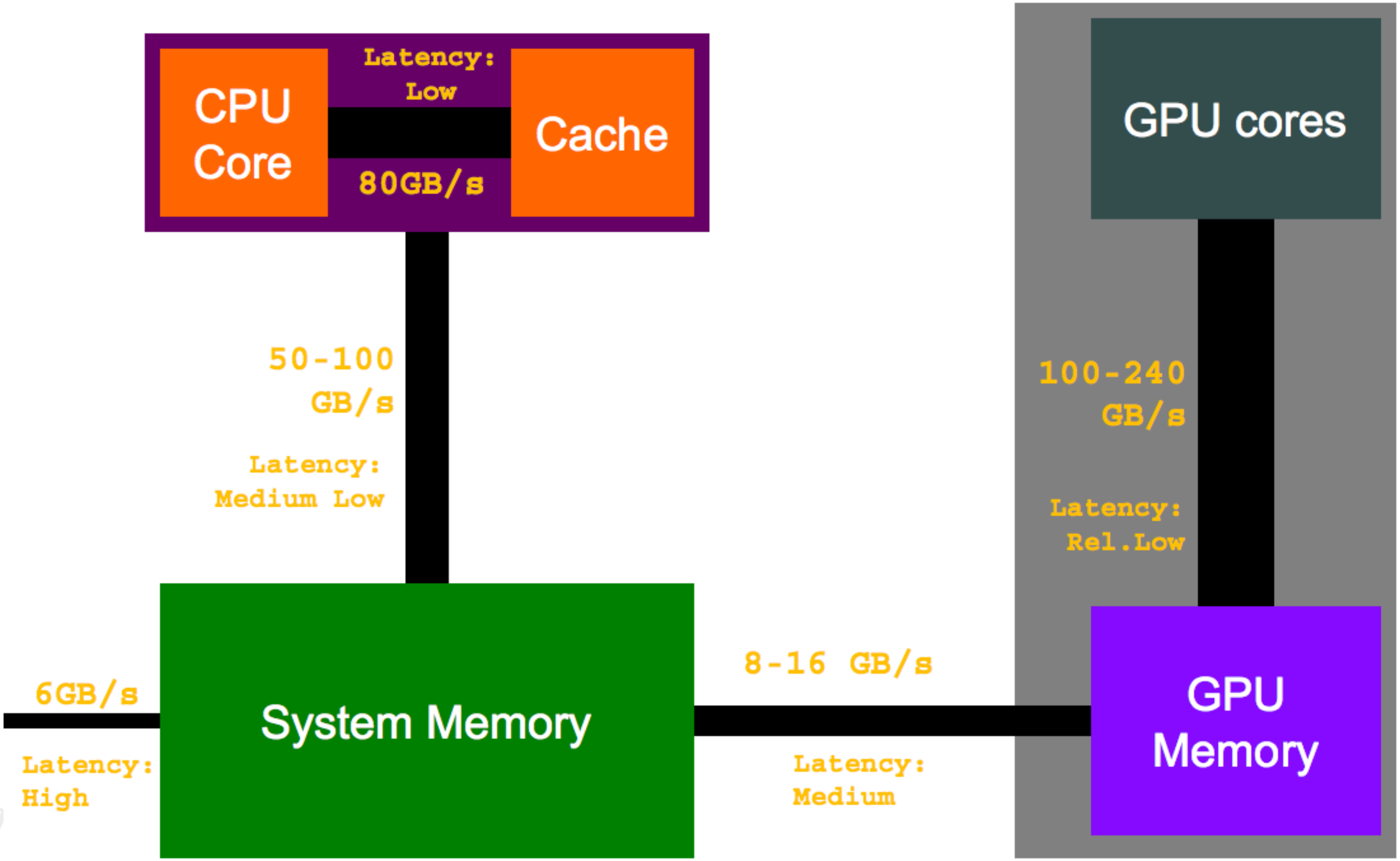
- The CPU is optimized for dynamic workloads marked by short sequences of computational operations and unpredictable control flow;

- GPUs aim at workloads that are dominated by computational tasks with simple control flow.

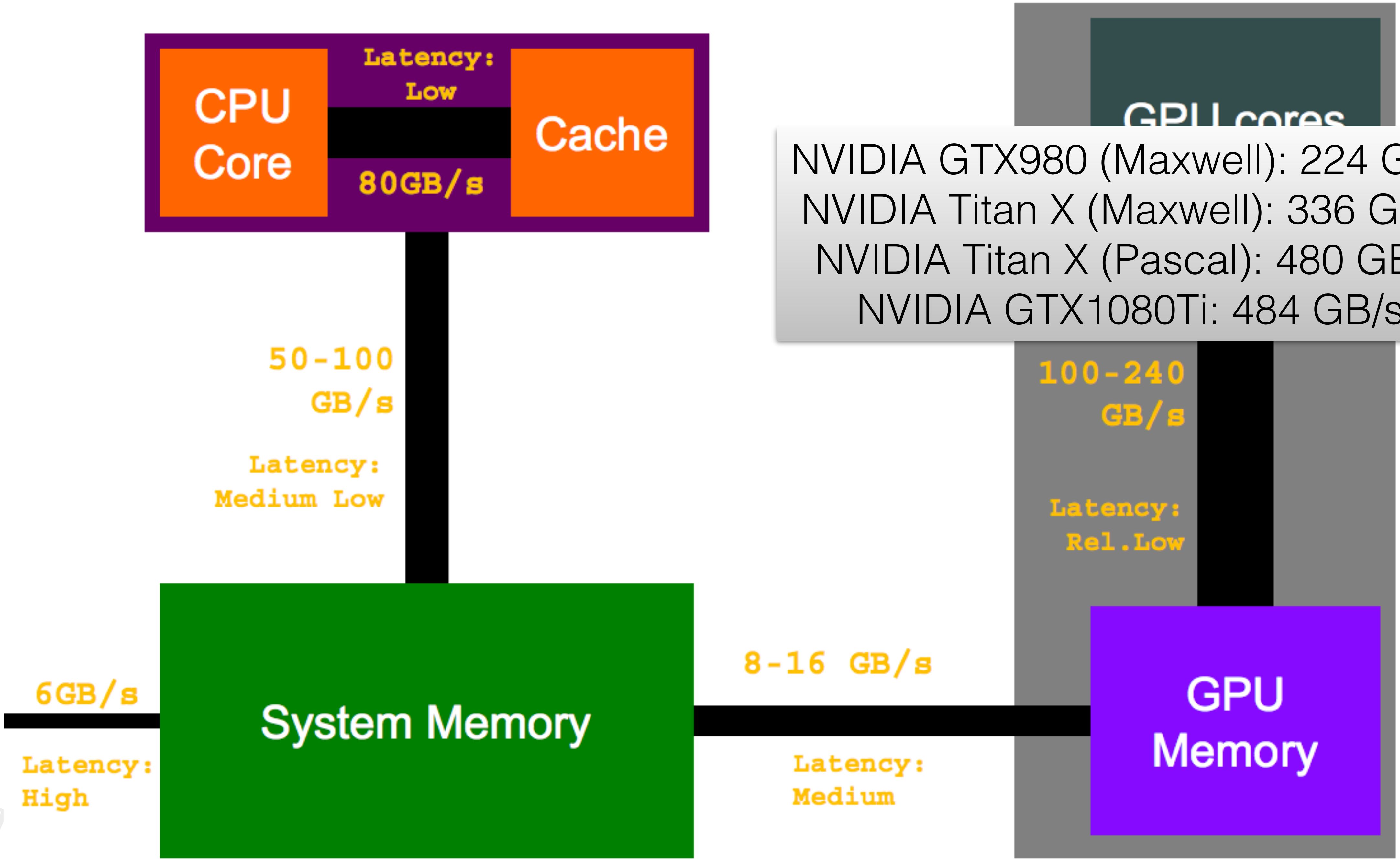
Heterogeneous Computing



Bandwidth in a CPU-GPU System



Bandwidth in a CPU-GPU System



NVIDIA GTX980 (Maxwell): 224 GB/s
NVIDIA Titan X (Maxwell): 336 GB/s
NVIDIA Titan X (Pascal): 480 GB/s
NVIDIA GTX1080Ti: 484 GB/s

Heterogeneous Computing

- A heterogeneous application consists of two parts:
  - Host code
  - Device code
- **Host** code runs on **CPUs** and **device** code runs on **GPUs**.

Threads

- Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.

  We deal with a few tens of threads per CPU, depending on HyperThreading.

- Threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.

  We deal with tens of thousands of threads per GPU.

SIMT

- GPU is a SIMD (Single Instruction, Multiple Data) device → it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that the GPU is assigned to process
  - NVIDIA calls this model SIMT (single instruction multiple thread)
- The SIMT architecture is similar to SIMD. Both implement parallelism by broadcasting the same instruction to multiple execution units.

  A key difference is that SIMD requires that all vector elements in a vector execute together in a unified synchronous group, whereas SIMT allows multiple threads in the same group to execute independently.

SIMT

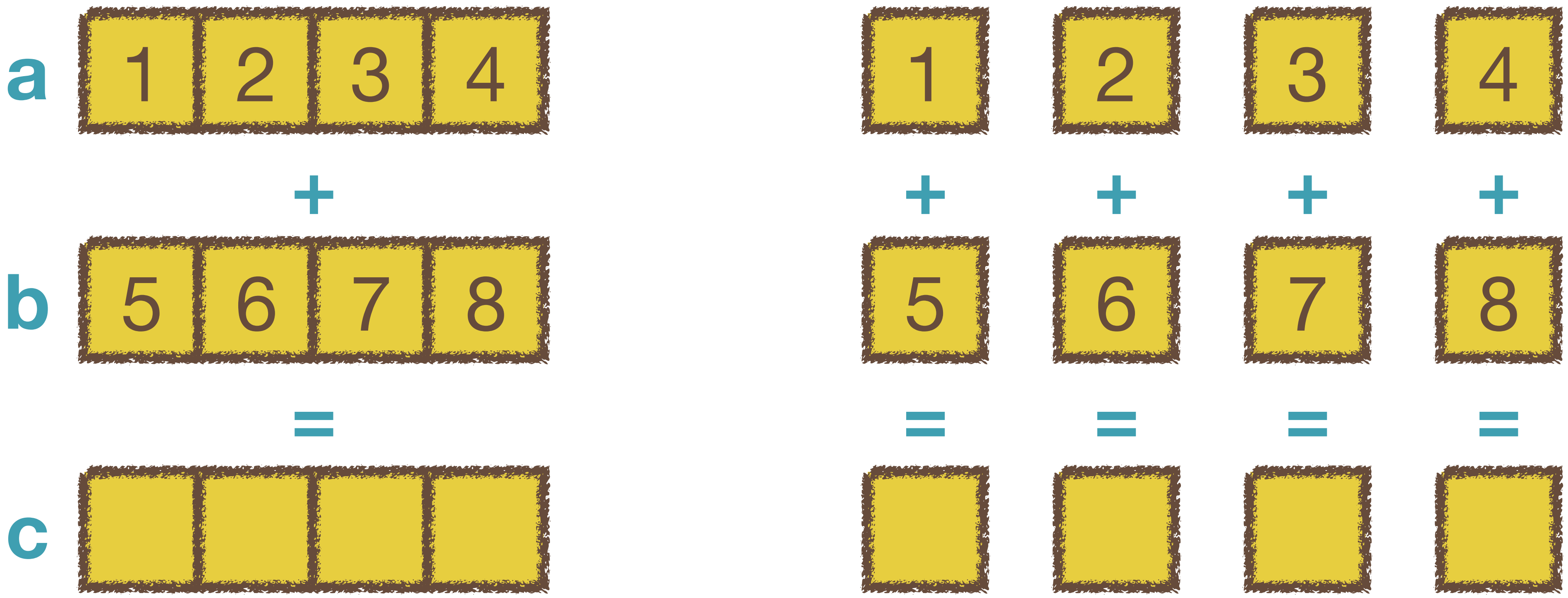
- The SIMT model includes three key features that SIMD does not:

- Each thread has its own instruction address counter.

- Each thread has its own register state, i.e. it has a register set.

- Each thread can have an independent execution path.

SIMD (SSE) view vs. SIMT (CUDA) view



```
__m128 a = _mm_set_ps (4, 3, 2, 1);
__m128 b = _mm_set_ps (8, 7, 6, 5);
__m128 c = _mm_add_ps (a, b);
```

```
float a[4] = {1, 2, 3, 4},
        b[4] = {5, 6, 7, 8}, c[4];

// …
// Define a compute kernel, which
// a fine-grained thread executes.
{
  int id = … ;  // my thread ID
  c[id] = a[id] + b[id];
}
```

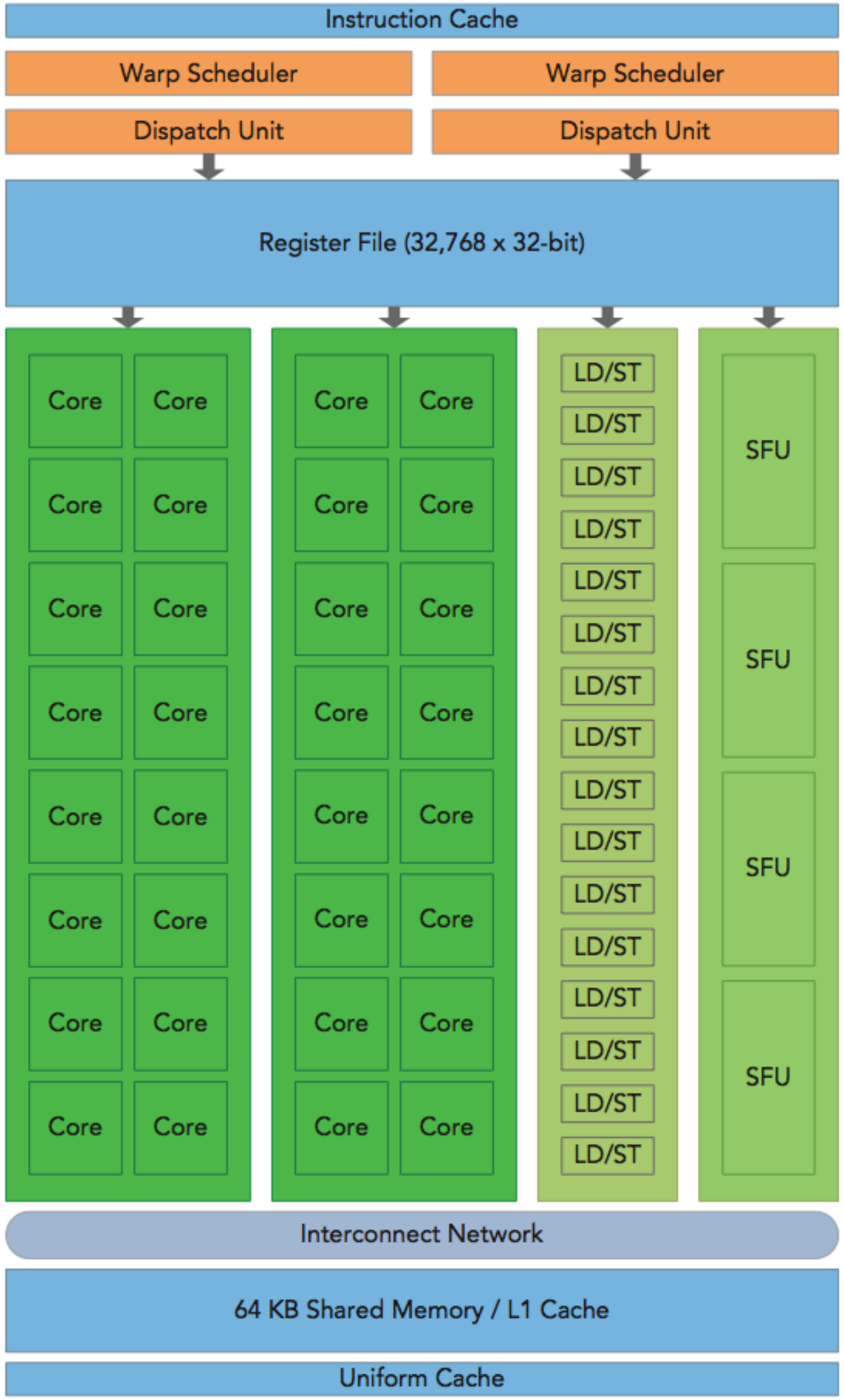
GPU Architecture Overview

- The GPU architecture is built around a scalable array of Streaming Multiprocessors (SM).

- Each SM in a GPU is designed to support concurrent execution of hundreds of threads, and there are multiple SMs per GPU

- NVIDIA GPUs execute threads in groups of 32 called warps. All threads in a warp execute the same instruction at the same time.

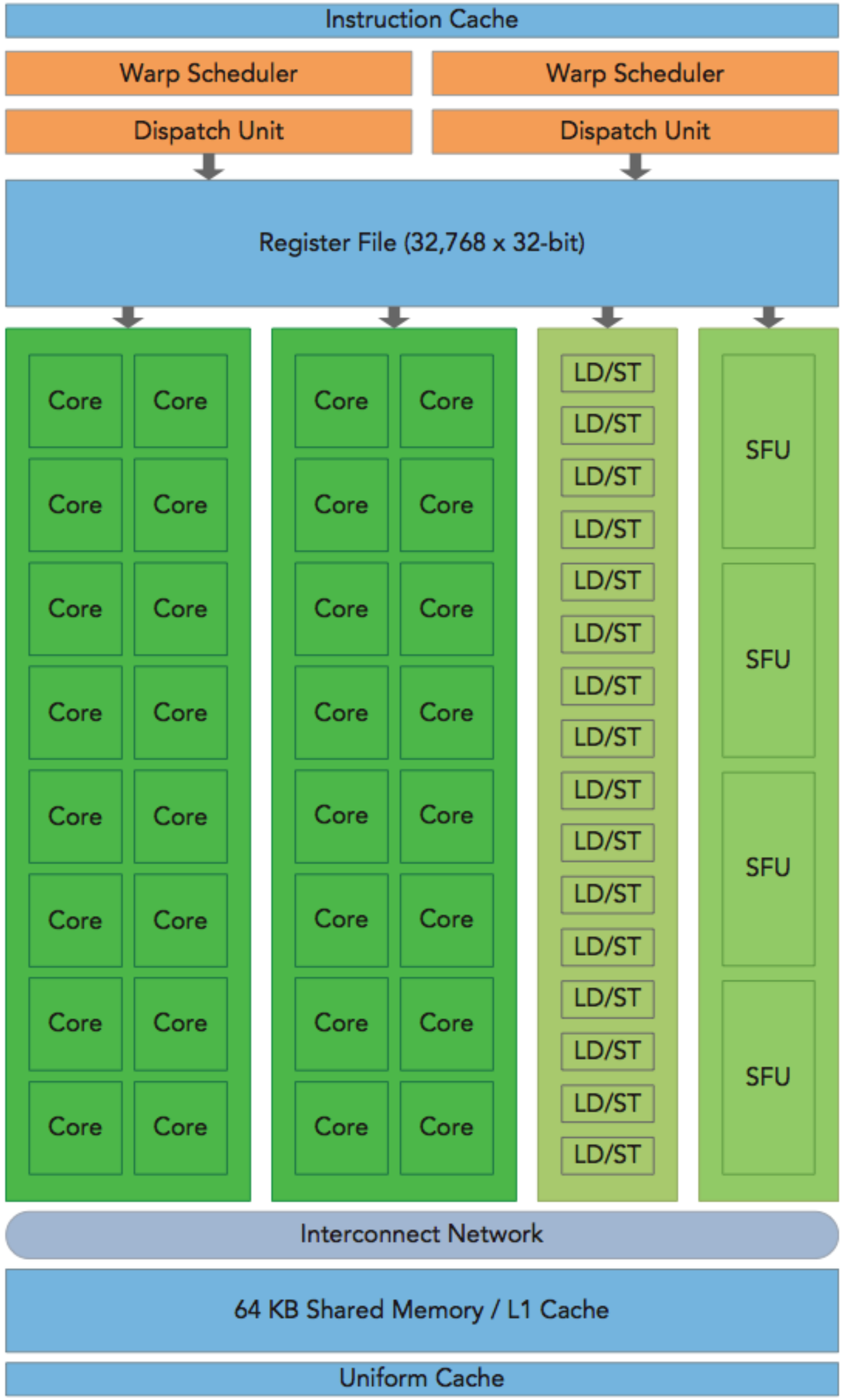
- GPU H/Ws are differentiated based on their "compute capabilities". The higher the better. Maxwell architecture (e.g. GTX980) have 5.2. Pascal architecture (e.g. GTX1080) GPUs has 6.0-6.2. The latest Volta architecture has 7.0.



LD/ST: load/store data from cache and DRAM
SFU: Execute transcendental instructions such as sin, cosine, reciprocal, and square root.

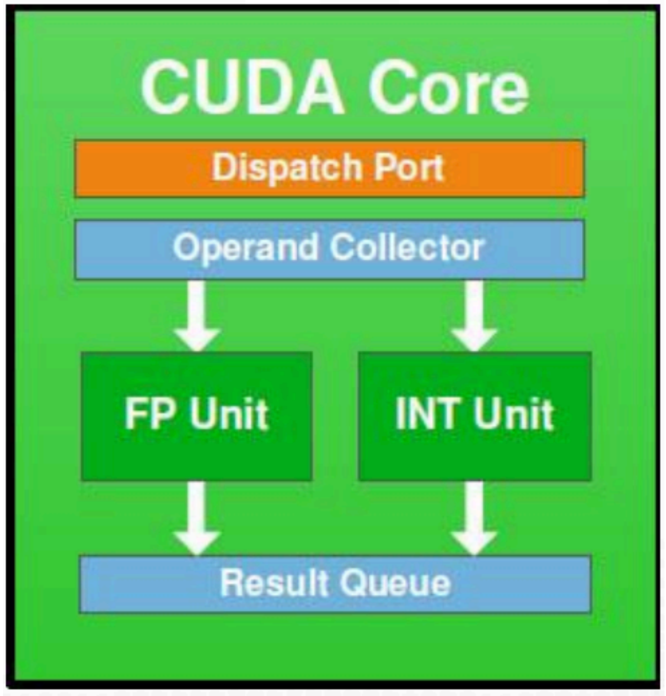
Overview

- The GPU architecture is built around a scalable array of Streaming Multiprocessors (SM).
- Each SM in a GPU is designed to support concurrent execution of hundreds of threads, and there are multiple SMs per GPU
- NVIDIA GPUs execute threads in groups of 32 called warps. All threads in a warp execute the same instruction at the same time.
- GPU H/Ws are differentiated based on their “compute capabilities”. The higher the better. Maxwell architecture (e.g. GTX980) have 5.2. Pascal architecture (e.g. GTX1080) GPUs has 6.0-6.2.
The latest Volta architecture has 7.0.



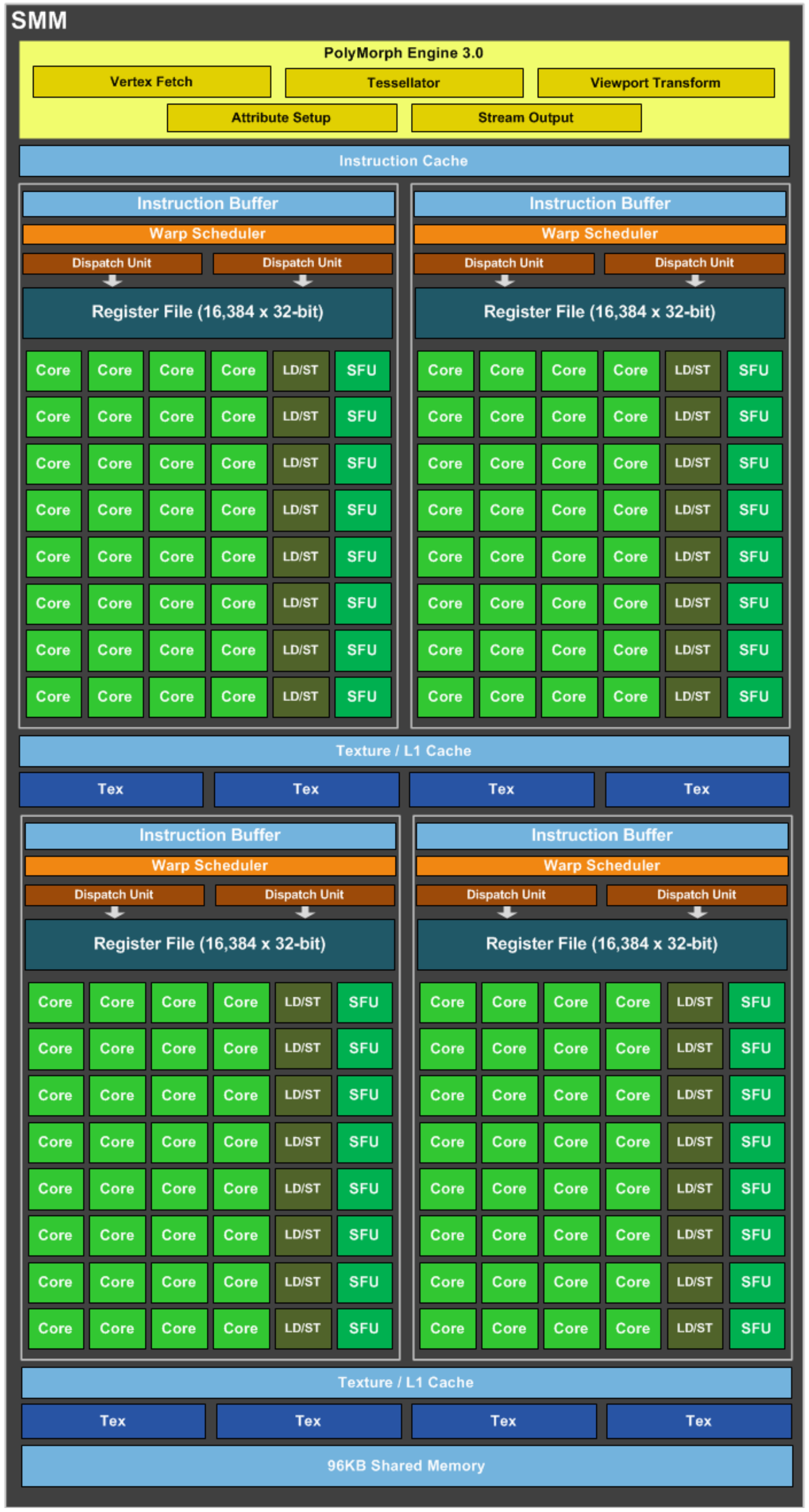
CUDA core

- It’s a vector processing unit
- Works on a single operation
- It’s the building block of SM
- As the process reduces them (e.g. 28nm) they increase in number per SM



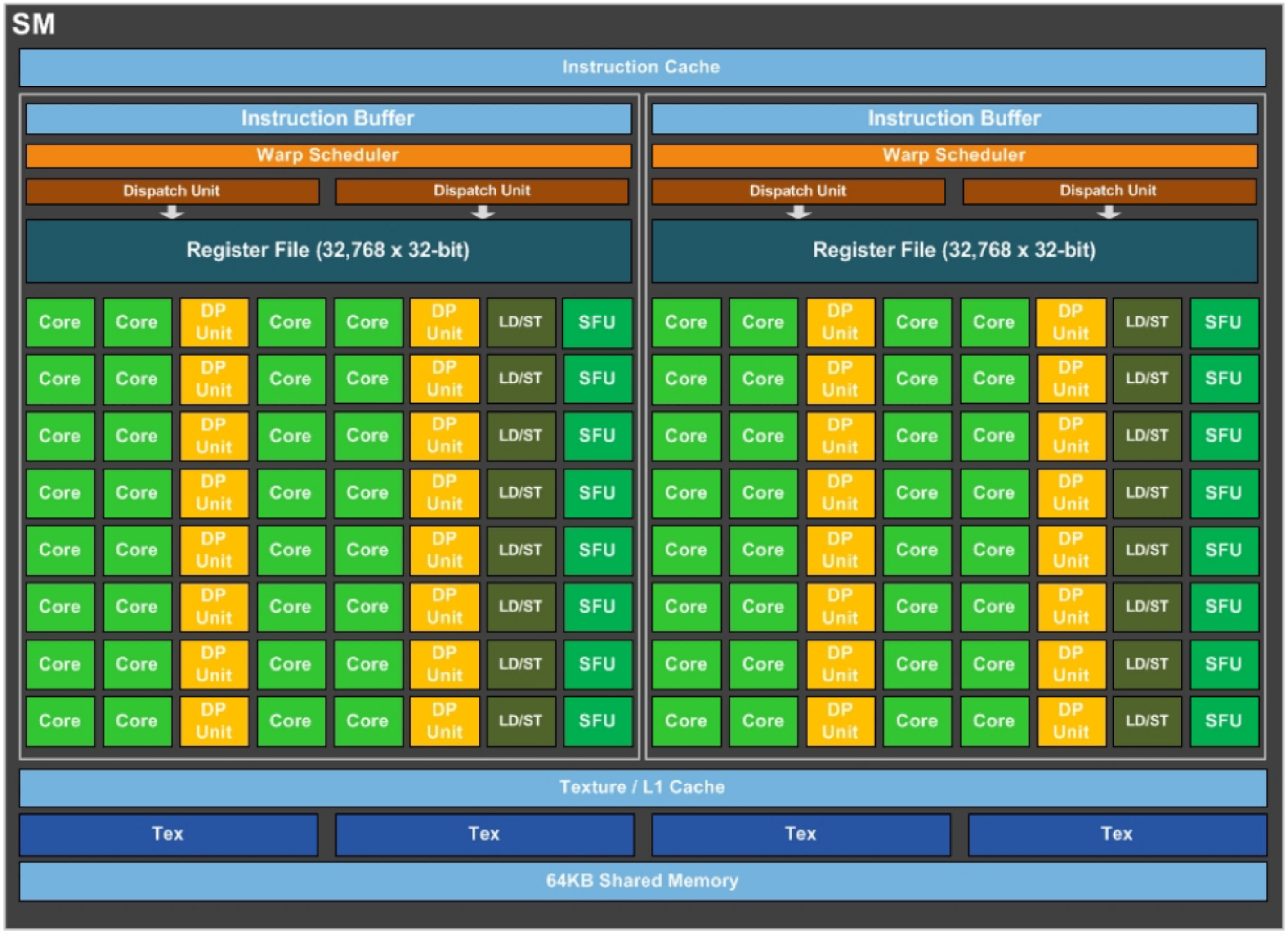
Maxwell SMM

- Four 32-core processing blocks each with a dedicated warp scheduler that can dispatch 2 instructions per clock
- Larger shared memory (dedicated to SM)
- Larger L2 cache (shared by SMs)



Pascal SM

- More SMs per GPU
- FP16 computation (2× faster than FP32)
- Less cores but same # registers: more registers per core
- Fast HBM2 memory interface
- Fast NVLink bus
- Unified memory: programs can access both CPU and GPU RAM



SM

Instruction Cache

Instruction Buffer | Instruction Buffer

Warp Scheduler | Warp Scheduler

Dispatch Unit | Dispatch Unit | Dispatch Unit | Dispatch Unit

Register File (32,768 x 32-bit) | Register File (32,768 x 32-bit)

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

Core Core DP Unit Core Core DP Unit LD/ST SFU | Core Core DP Unit Core Core DP Unit LD/ST SFU

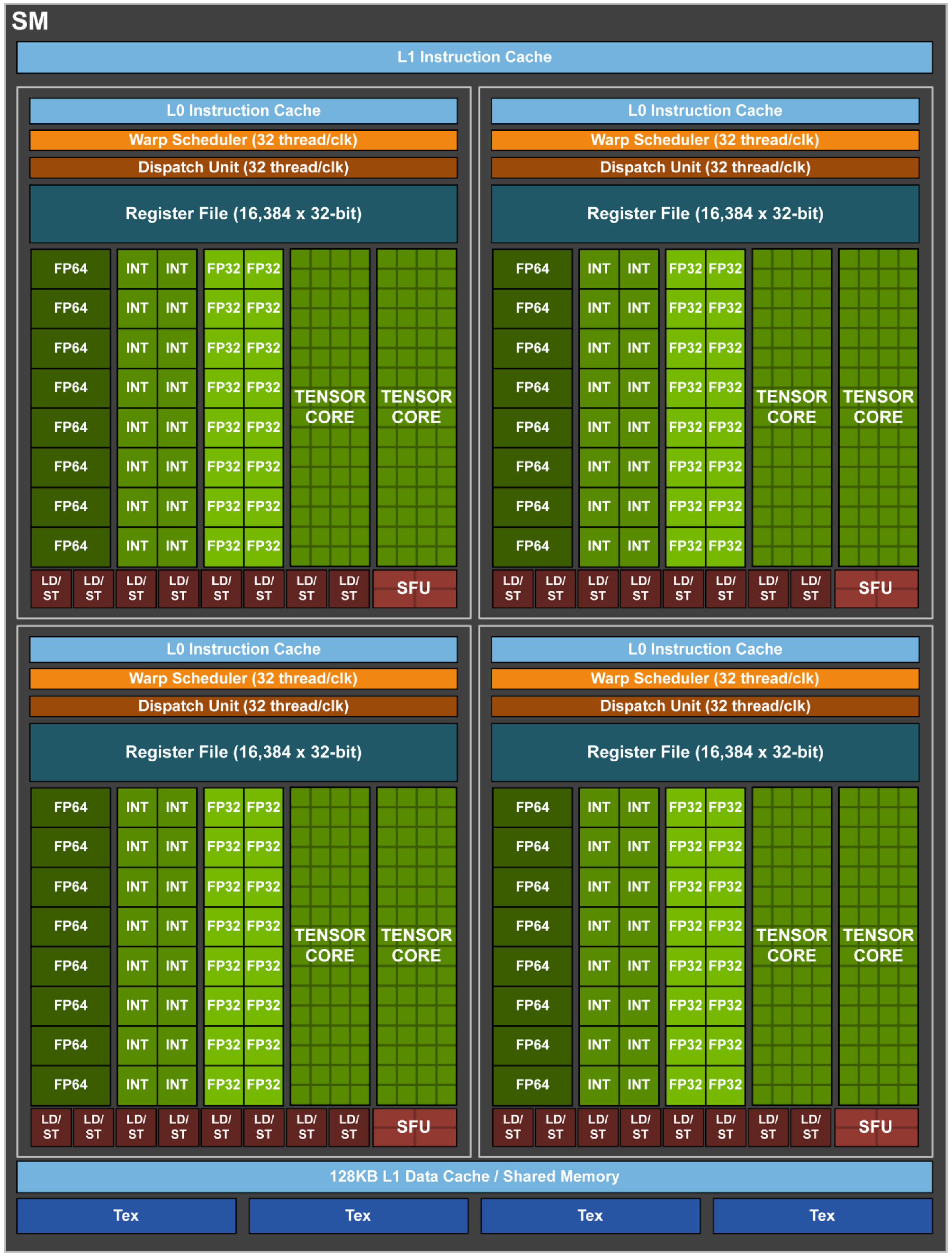
Texture / L1 Cache

Tex | Tex | Tex | Tex

64KB Shared Memory

Volta SM

- New tensor cores
- Unified L1 / shared memory
- Independent FP32 and INT32 cores
- More SMs per GPU
- Larger L2 cache
- New L0 instruction cache (accessed directly from functional units)


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64 INT INT FP32 FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64 INT INT FP32 FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64 INT INT FP32 FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64 INT INT FP32 FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
Tex
Tex
Tex

NVIDIA GPUs

Volta Architecture (compute capabilities 7.x)				Tesla V Series
Pascal Architecture (compute capabilities 6.x)		GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center

Back to bandwidth

- A NVIDIA GTX1080Ti has 28 SM and 3584 CUDA cores (128 cores per SM). It's clocked at 1.48GHz and memory bandwidth is 484GB/s. This means:
- ~327 bytes/cycle for thew whole GPU.
- 11.7 bytes/cycle per SM (~4× of Intel i7-7700K)
- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Back to bandwidth

- A NVIDIA GTX1080Ti has 28 SM and 3584 CUDA cores (128 cores per SM). It’s clocked at 1.48GHz and memory bandwidth is 484GB/s. This means:
- ~327 bytes/cycle for thew whole GPU.
- 11.7 bytes/cycle per SM (~4× of Intel i7-7700K)
- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

Absolute memory bandwidths in consumer devices have gone up by several orders of magnitude from the ~1MB/s of early 80s home computers, but available compute resources have grown much faster still.
The only way to stop bumping into bandwidth limits all the time is to make sure your workloads have reasonable locality of reference so that the caches can do their job.

L2 caches of NVIDIA GPUs are going up from 1536KB in Kepler (K40), to 4096KB in Pascal (GP100) and 6144KB in Volta (GV100)

- ~327 bytes/cycle for thew whole GPU.
- 11.7 bytes/cycle per SM (~4× of Intel i7-7700K)
- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

Absolute memory bandwidths in consumer devices have gone up by several orders of magnitude from the ~1MB/s of early 80s home computers, but available compute resources have grown much faster still.

The only way to stop bumping into bandwidth limits all the time is to make sure your workloads have reasonable locality of reference so that the caches can do their job.

L2 caches of NVIDIA GPUs are going up from 1536KB in Kepler (K40), to 4096KB in Pascal (GP100) and 6144KB in Volta (GV100)

- ~327 bytes/cycle for thew whole GPU.

Pascal and Volta GPUs use High Bandwidth Memory 2 (HBM2) high-perf. RAM interface to achieve higher bandwidth (900 GB/s)

- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Absolute memory bandwidths in consumer devices have gone up by several orders of magnitude from the ~1MB/s of early 80s home computers, but available compute resources have grown much faster still.
The only way to stop bumping into bandwidth limits all the time is to make sure your workloads have reasonable locality of reference so that the caches can do their job.

L2 caches of NVIDIA GPUs are going up from 1536KB in Kepler (K40), to 4096KB in Pascal (GP100) and 6144KB in Volta (GV100)

- ~327 bytes/cycle for thew whole GPU.

Pascal and Volta GPUs use High Bandwidth Memory 2 (HBM2) high-perf. RAM interface to achieve higher bandwidth (900 GB/s)

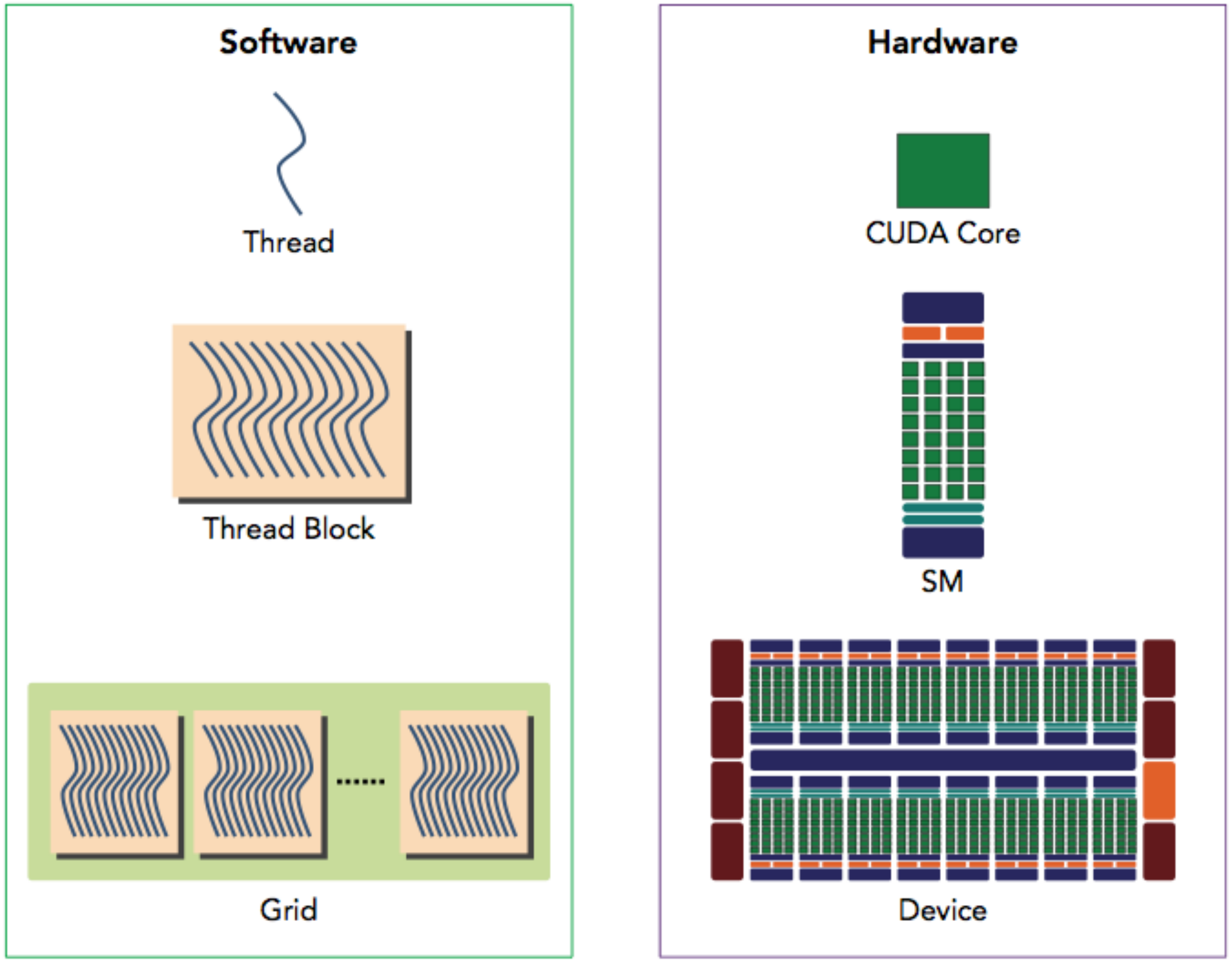
NVLink bus connects CPU and GPU (or multiple GPUs) at 80-200 GB/s - it’s an alternative to PCI Express

- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

Execution

- A thread block is scheduled on only one SM. Once a thread block is scheduled on an SM, it remains there until execution completes. An SM can hold more than one thread block at the same time.



CUDA

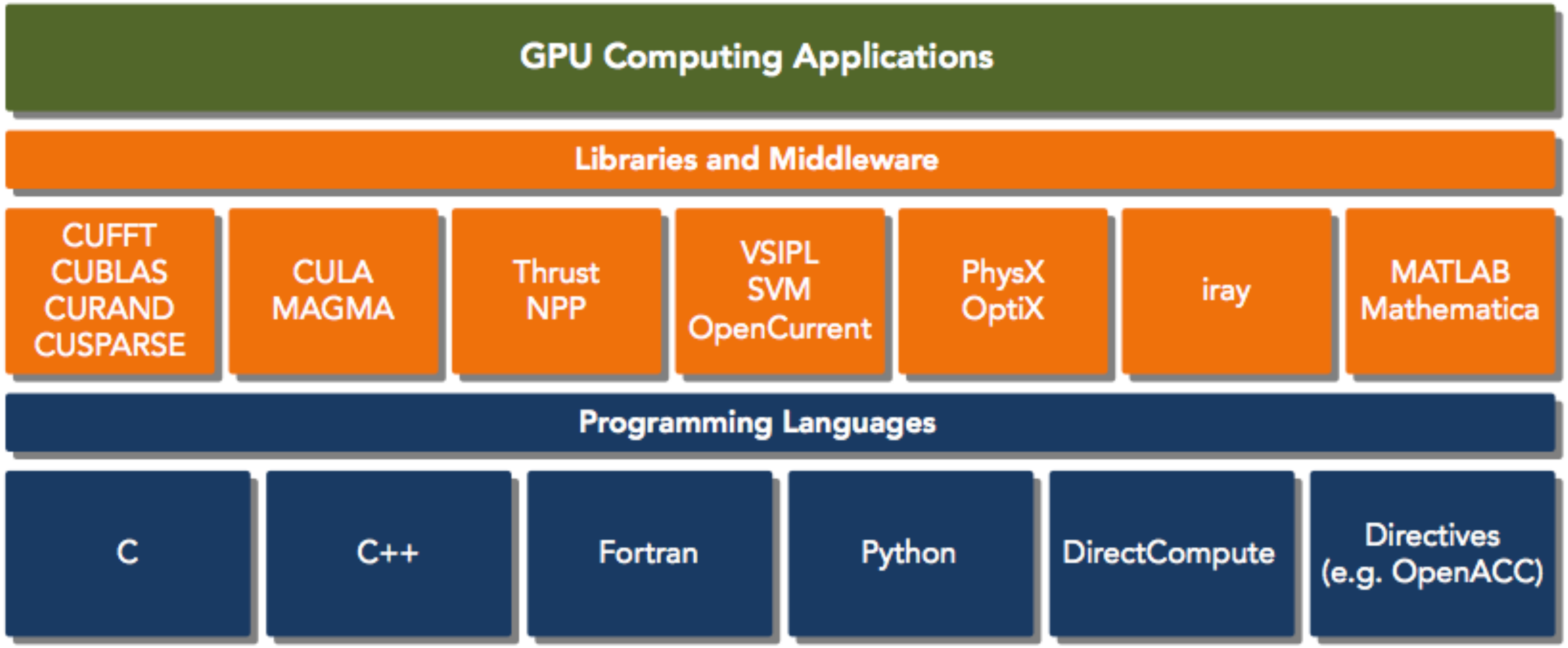
CUDA: Compute Unified Device Architecture

- It enables a general purpose programming model on NVIDIA GPUs. Current CUDA SDK is 9.0.
- Enables explicit GPU memory management
- The GPU is viewed as a compute **device** that:
  - Is a co-processor to the CPU (or **host**)
  - Has its own DRAM (global memory in CUDA parlance)
  - Runs many threads in parallel

The CUDA platform

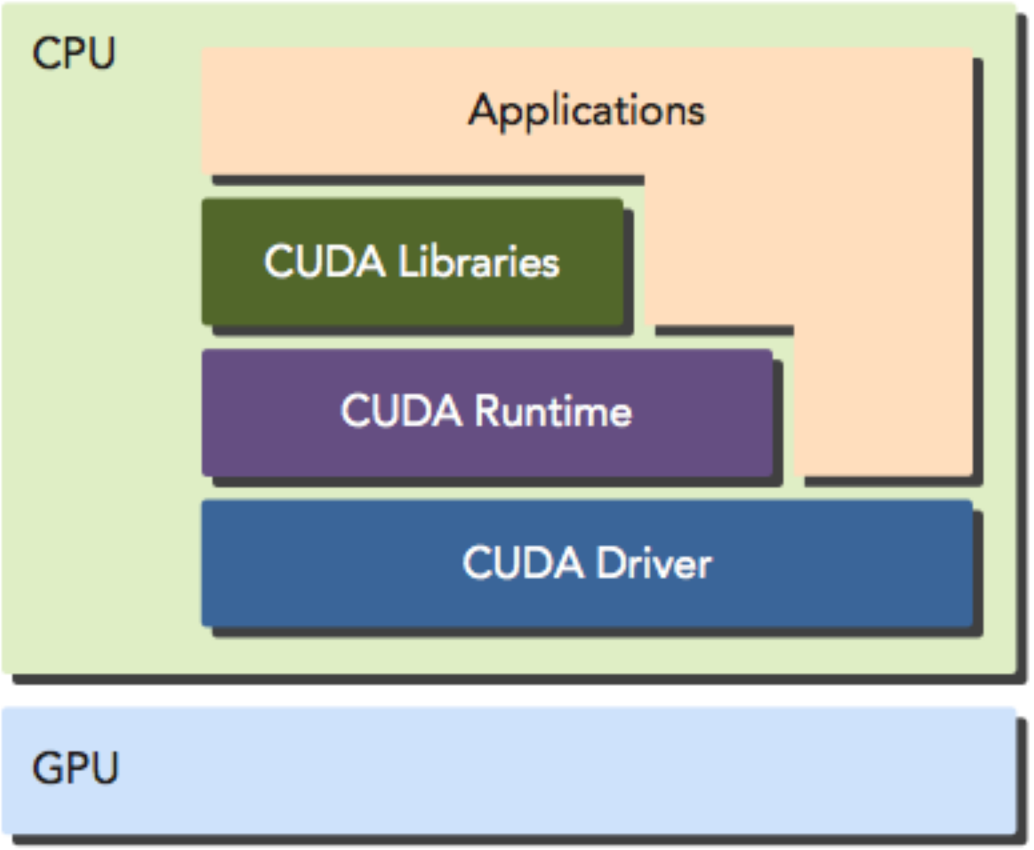
- The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces, and extensions to industry-standard programming languages, including C, C++, Fortran, and Python

- CUDA C is an extension of standard ANSI C with a handful of language extensions to enable heterogeneous programming, and also straightforward APIs to manage devices, memory, and other tasks.



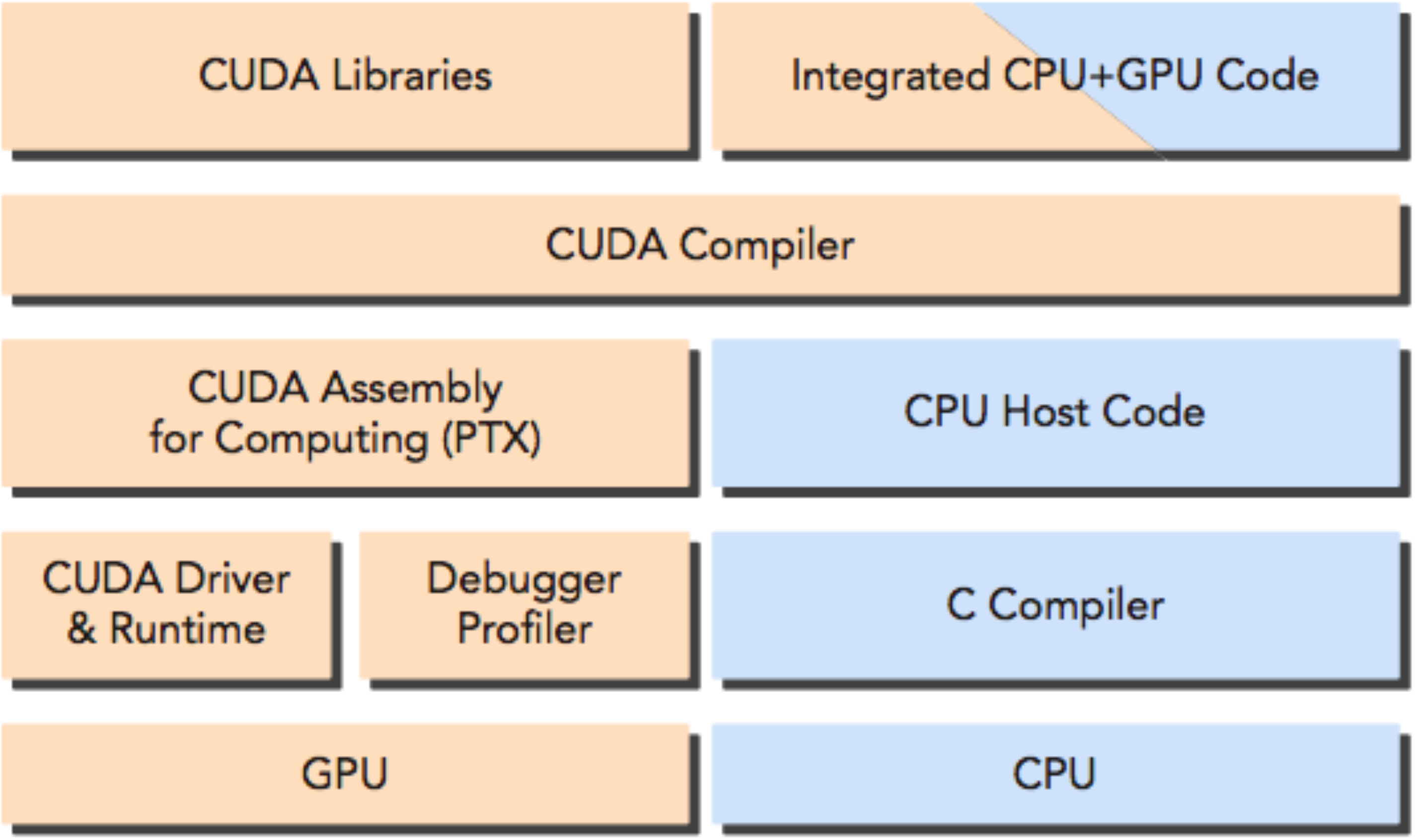
CUDA APIs

- CUDA provides two API levels for managing the GPU device and organizing threads:
  - CUDA Driver API
  - CUDA Runtime API
- The driver API is a low-level API and is relatively hard to program, but it provides more control over how the GPU device is used.
  The runtime API is a higher-level API implemented on top of the driver API. Each function of the runtime API is broken down into more basic operations issued to the driver API.



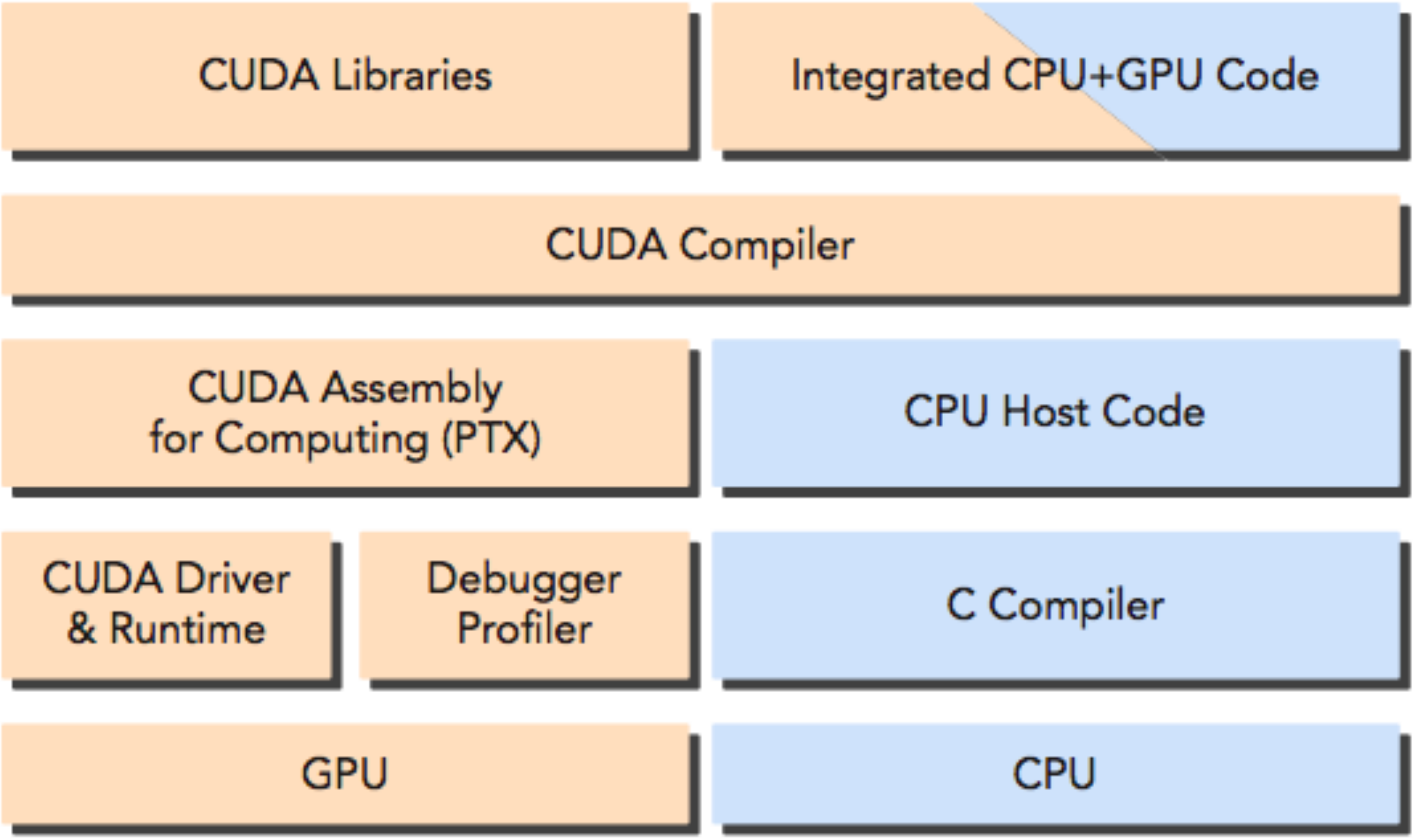
A CUDA program

- A CUDA program consists of a mixture of the following two parts:
  - The host code runs on CPU.
  - The device code runs on GPU.
- NVIDIA's CUDA nvcc compiler separates the device code from the host code during the compilation process.

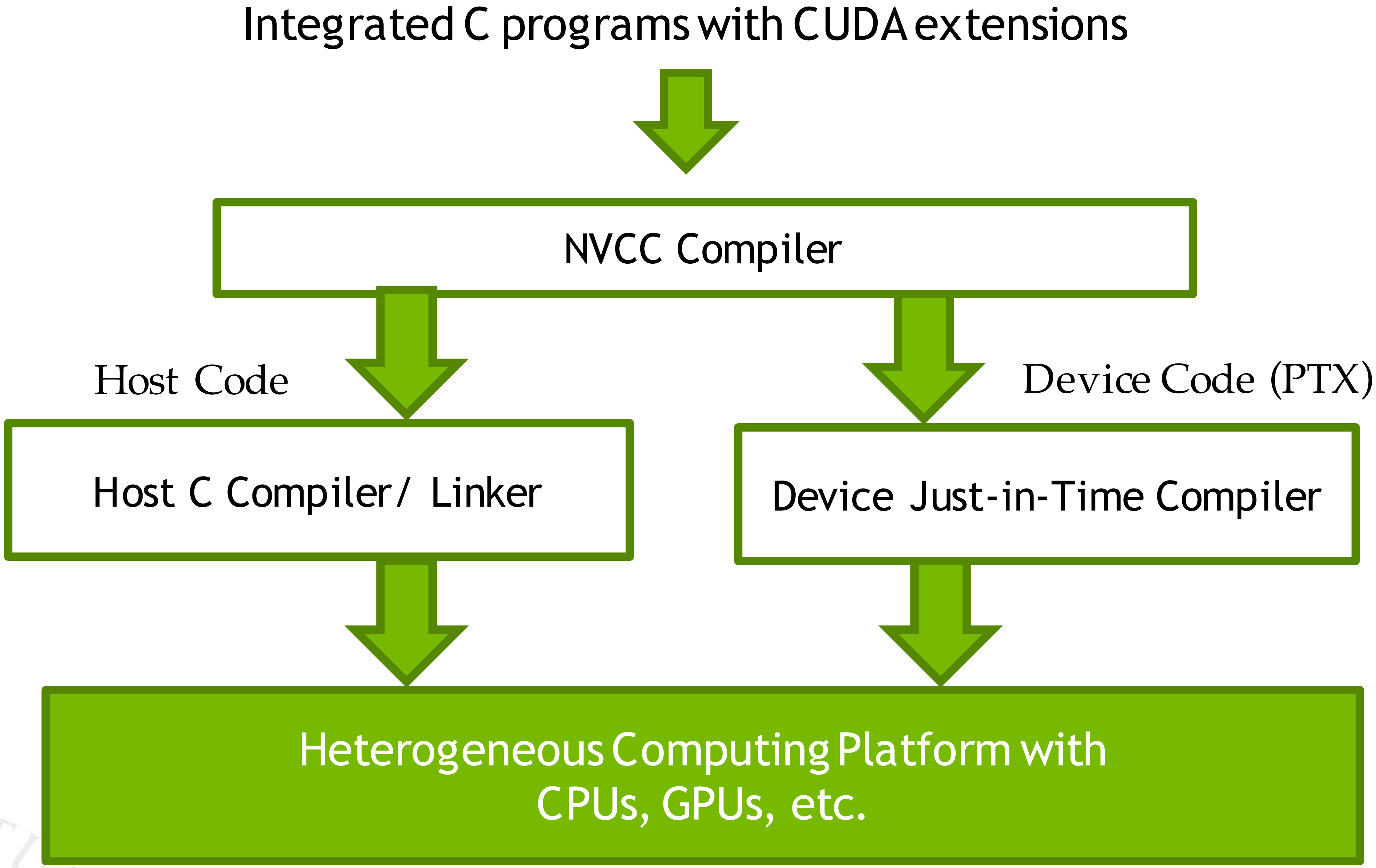


A CUDA program

- A CUDA program consists of a mixture of the following two parts:
  - The host code runs on CPU.
  - The device code runs on GPU.
- NVIDIA provides the NSight IDE (based on Eclipse) to ease development of C/C++/CUDA programming



Compiling a CUDA program



Integrated C programs with CUDA extensions

NVCC Compiler

Host Code

Device Code (PTX)

Host C Compiler/ Linker

Device Just-in-Time Compiler

Heterogeneous Computing Platform with
CPUs, GPUs, etc.

What a programmer expresses in CUDA

- Computation partitioning (where does computation occur?)
  - Declarations on functions `__host__`, `__global__`, `__device__`
  - Mapping of thread programs to device:
    `compute <<<gs, bs>>>(<args>)`
- Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data `__shared__`, `__device__`, `__constant__`, …
- Data management and orchestration
  - Copying to/from host: e.g., `cudaMemcpy(h_obj,d_obj, cudaMemcpyDevicetoHost)`
- Concurrency management
  - E.g. `__synchthreads()`

CUDA C: C extension + API

- Declspecs
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- Runtime API
  - Memory, symbol, execution management
- Function launch

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Languages

- The host code is written in ANSI C, and the device code is written using CUDA C.

- You can put all the code in a single source file, or you can use multiple source files to build your application or libraries.

- The NVIDIA C Compiler (`nvcc`) generates the executable code for both the host and device.

- Typical CUDA C extension is `.cu`

CUDA program structure

- A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA functions (called **kernel**) to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

CUDA program structure

- A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.

2. Copy data from CPU memory to GPU memory.

As the developer, you can express a **kernel** as a sequential program. Behind the scenes, CUDA manages scheduling programmer-written kernels on GPU threads.

From the host, you define how your algorithm is mapped to the device based on application data and GPU device capability.

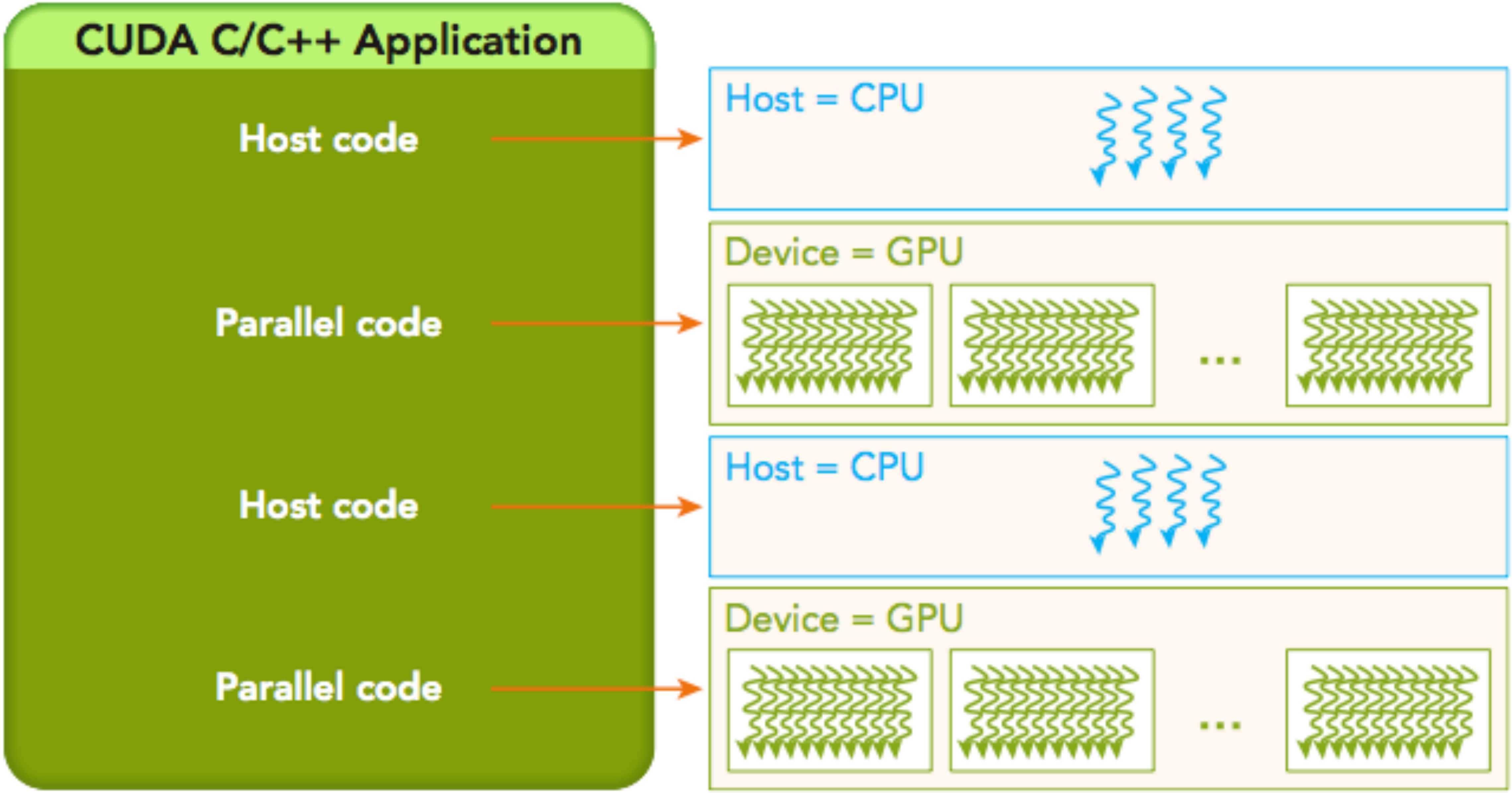
4. Copy data back from GPU memory to CPU memory.

5. Destroy GPU memories.

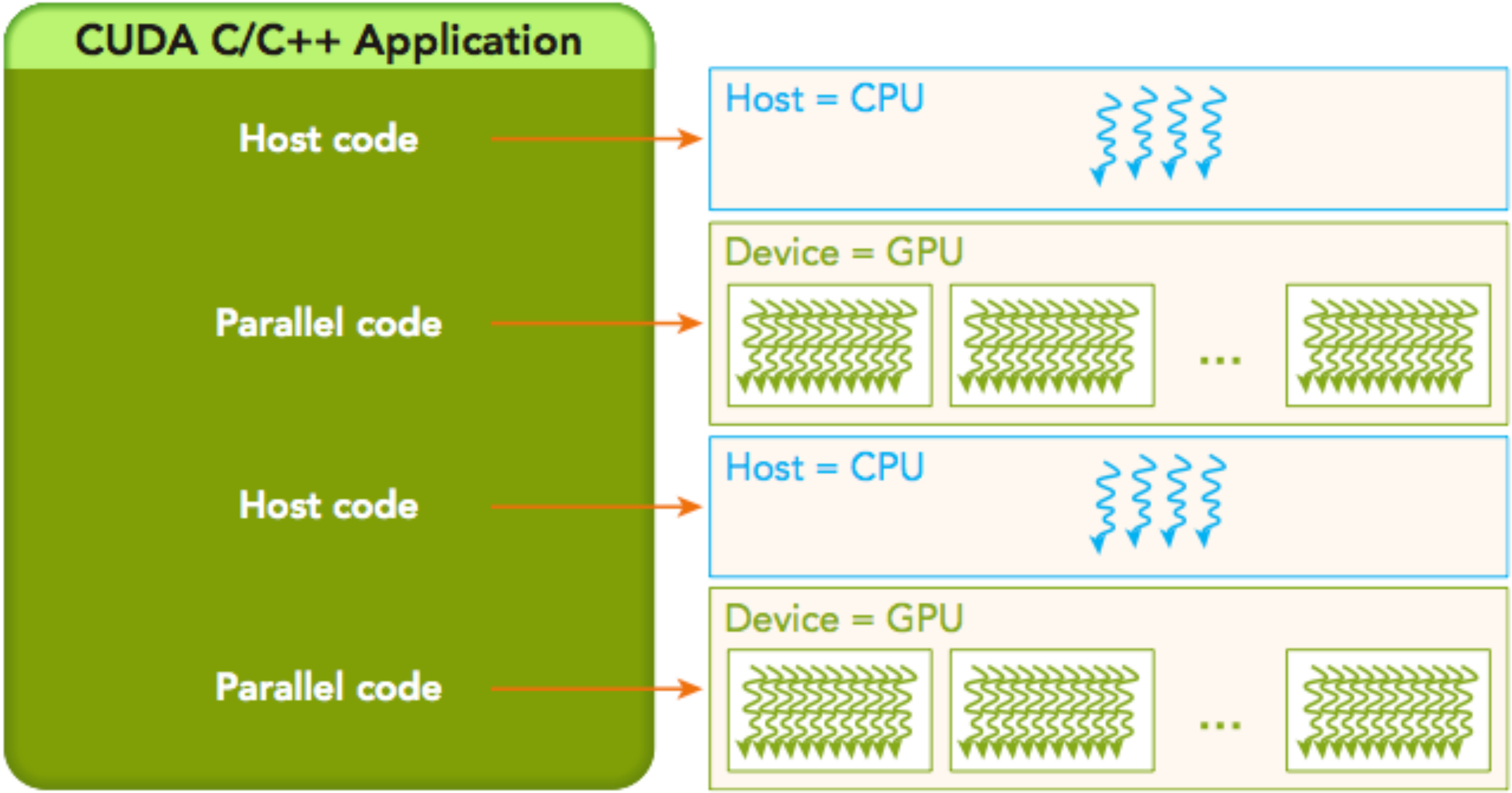
CUDA program structure

- The host can operate independently of the device for most operations. When a kernel has been launched, control is returned immediately to the host, freeing the CPU to perform additional tasks complemented by data parallel code running on the device.
  - The CUDA programming model is primarily asynchronous so that GPU computation performed on the GPU can be overlapped with host-device communication.

CUDA program structure



CUDA program structure



All the threads that are generated by a kernel during an invocation are collectively called a **grid**.

Grid

- The programmer decides how to organize a **grid**, to improve parallelization
- When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.
- Grids are organized into **blocks**.

Hello world

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float myDeviceFunc()`	device	device
`__global__ void myKernelFunc()`	device	host
`__host__ float myHostFunc()`	host	host

- `__global__` defines a kernel function, launched by host, executed on the device
  - Must return `void`
- By default, all functions in a CUDA program are `__host__` functions if they do not have any of the CUDA keywords in their declaration.

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float myDeviceFunc()`	device	device
`__global__ void myKernelFunc()`	device	host
`__host__ float myHostFunc()`	host	host

One can use both `__host__` and `__device__` in a function declaration.

This combination triggers the compilation system to generate two versions of the same function.

One is executed on the host and can only be called from a host function.
The other is executed on the device and can only be called from a device or kernel function.

Hello world

```
#include "greeter.h"

#include <stdio.h>
#include <cuda_runtime_api.h>

__global__ void helloFromGPU() {
    printf("Hello World from GPU thread %d!\n", threadIdx);
}

int main(int argc, char **argv) {
    greet(std::string("Pinco"));

    helloFromGPU<<<1, 10>>>();

    // destroy and clean up all resources associated with current device
    //                                                  + current process.
    cudaDeviceReset(); // CUDA functions are async...
                       // the program would terminate before CUDA kernel prints
    return 0;
}
```

Hello world

Provides declaration of greet(std::string name).
Definition is in greeter.cpp

```
#include "greeter.h"

#include <stdio.h>
#include <cuda_runtime_api.h>

__global__ void helloFromGPU() {
    printf("Hello World from GPU thread %d!\n", threadIdx);
}

int main(int argc, char **argv) {
    greet(std::string("Pinco"));

    helloFromGPU<<<1, 10>>>();

    // destroy and clean up all resources associated with current device
    //                                                  + current process.
    cudaDeviceReset(); // CUDA functions are async...
                       // the program would terminate before CUDA kernel prints
    return 0;
}
```

Hello world

Provides declaration of `greet(std::string name)`.
Definition is in `greeter.cpp`

```
#include "greeter.h"

#include <stdio.h>
#include <cuda_runtime_api.h>

__global__ void helloFromGPU() {
    printf("Hello World from GPU thread %d!\n", threadIdx);
}

int main(int argc, char **argv) {
    greet(std::string("Pinco"));

    helloFromGPU<<<1, 10>>>();

    // destroy and clean up all resources associated with current device
    //                                                  + current process.
    cudaDeviceReset(); // CUDA functions are async...
                       // the program would terminate before CUDA kernel prints
    return 0;
}
```

10 CUDA threads running on the GPU.
This uses is 1 grid.

Managing memory

Memory model

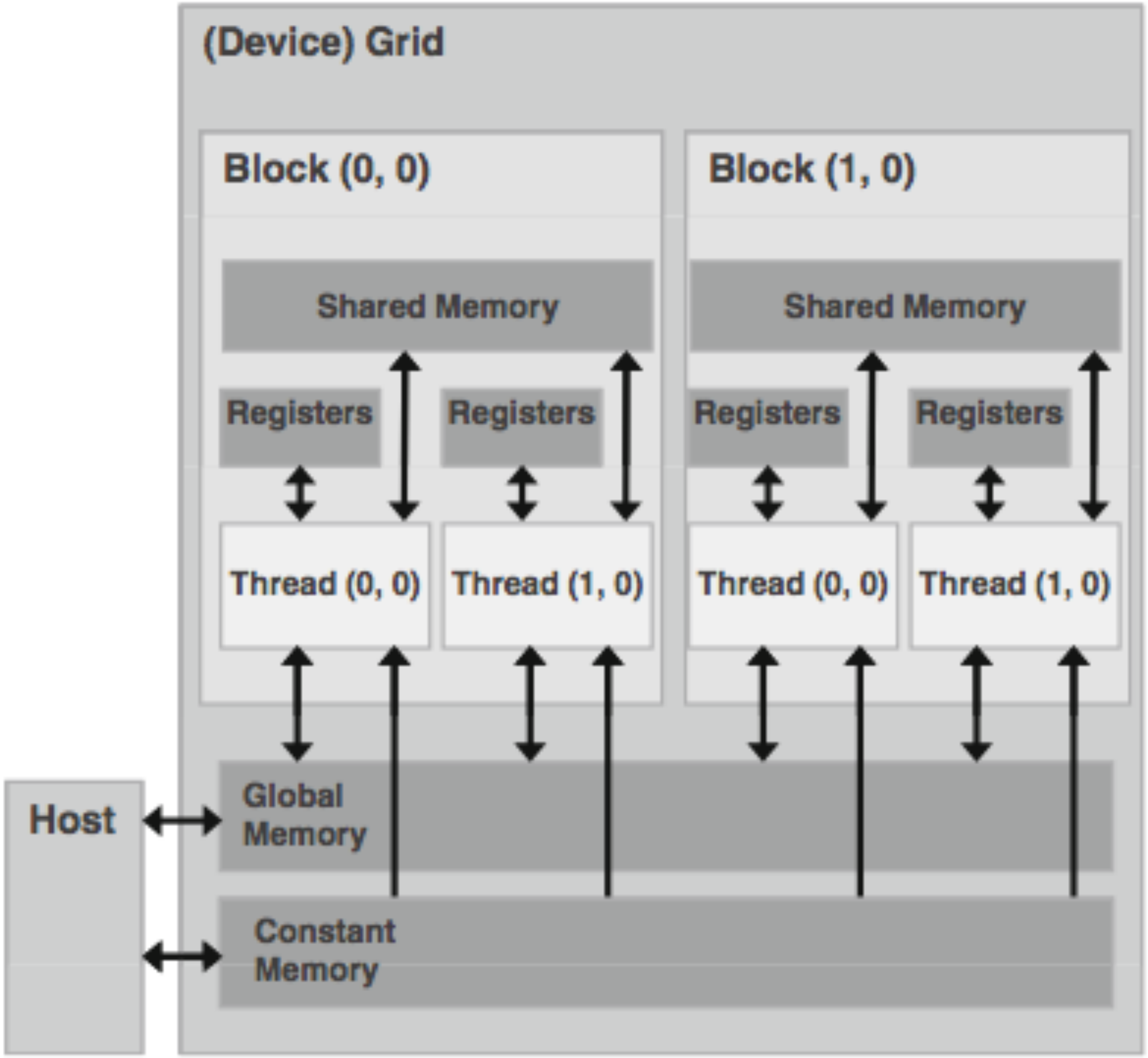
- The CUDA programming model assumes a system composed of a host and a device, each with its own separate memory.

- Kernels operate out of device memory. To allow you to have full control and achieve the best performance, the CUDA runtime provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory.

Standard C function	CUDA C function
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

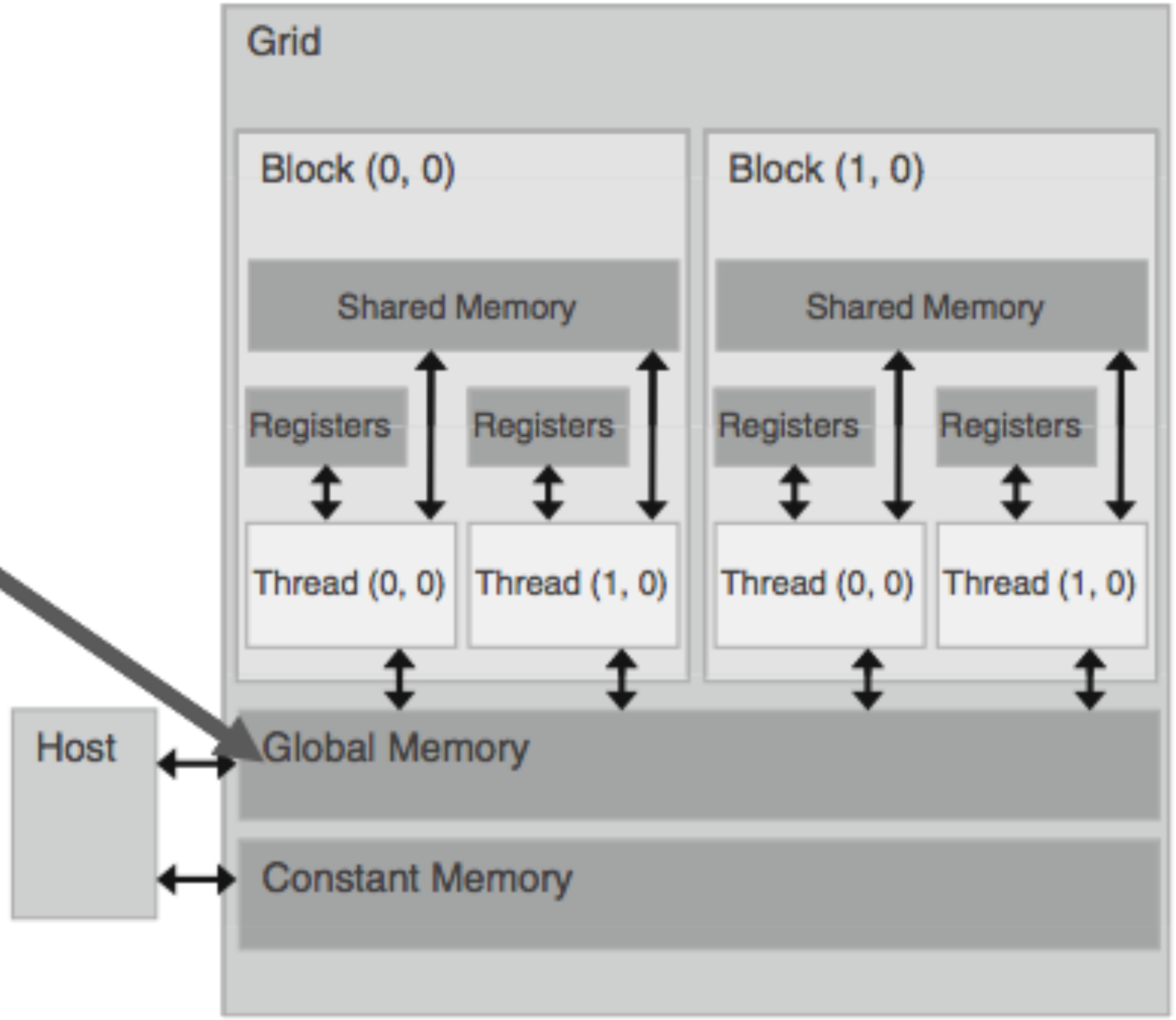
CUDA device memory model

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories



Example of CUDA API

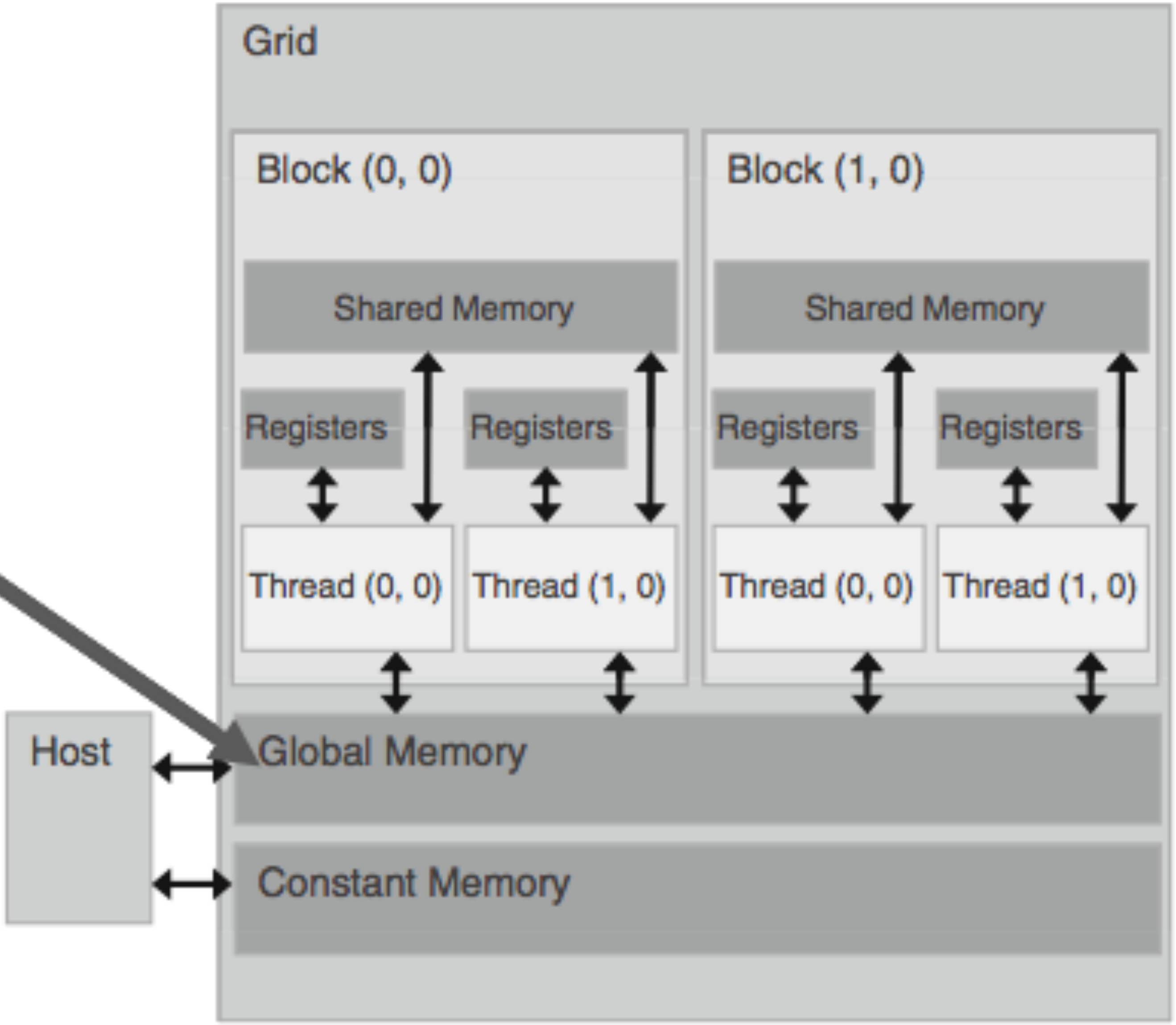
- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object



Exam

```
float *Md
int size = width * width * sizeof(float);
cudaMalloc((void**)&Md, size);
// ...
cudaFree(Md);
```

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object



cudaMalloc

-
  ```
  cudaError_t cudaMalloc ( void** devPtr,
                           size_t size )
  ```

- This function allocates a linear range of device memory with the specified `size` in bytes. The allocated memory is returned through `devPtr`.

- if GPU memory is successfully allocated, it returns:

  - `cudaSuccess`

- Otherwise, it returns:

  - `cudaErrorMemoryAllocation`

CUDA errors

- You can convert an error code to a human-readable error message with the following CUDA run- time function:
- `char* cudaGetErrorString(cudaError_t error)`
- A common practice is to wrap CUDA calls in utility functions that manage the returned error

```
#define CUDA_CHECK_RETURN(value) CheckCudaErrorAux(__FILE__,__LINE__,
#value, value)

static void CheckCudaErrorAux (const char *file, unsigned line, const char
*statement, cudaError_t err) {
    if (err == cudaSuccess)
        return;
    std::cerr << statement<<" returned " << cudaGetErrorString(err) <<
                 "("<<err<< ") at "<<file<<":"<<line << std::endl;
    exit (1);
}
```

Use as: `CUDA_CHECK_RETURN(cudaFunction(parameters));`

- `char* cudaGetErrorString(cudaError_t error)`
- A common practice is to wrap CUDA calls in utility functions that manage the returned error

```
#define CUDA_CHECK_RETURN(value) CheckCudaErrorAux(__FILE__,__LINE__,
#value, value)

static void CheckCudaErrorAux (const char *file, unsigned line, const char
*statement, cudaError_t err) {
    if (err == cudaSuccess)
        return;
    std::cerr << statement<<" returned " << cudaGetErrorString(err) <<
                    "("<<err<< ") at "<<file<<":"<<line << std::endl;
    exit (1);
}
```

Use as: `CUDA_CHECK_RETURN(cudaFunction(parameters));`

```
#define CHECK(call)                                                    \
{                                                                      \
    const cudaError_t error = call;                                    \
    if (error != cudaSuccess)                                          \
    {                                                                  \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);         \
        fprintf(stderr, "code: %d, reason: %s\n", error,               \
                cudaGetErrorString(error));                            \
        exit(1);                                                       \
    }                                                                  \
}
```

cudaMemcpy

-
  ```
  cudaError_t cudaMemcpy ( void* dst, const void* src,
                           size_t count, cudaMemcpyKind kind )
  ```

- This function copies the specified bytes from the source memory area, pointed to by `src`, to the destination memory area, pointed to by `dst`, with the direction specified by kind, where kind takes one of the following types:
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

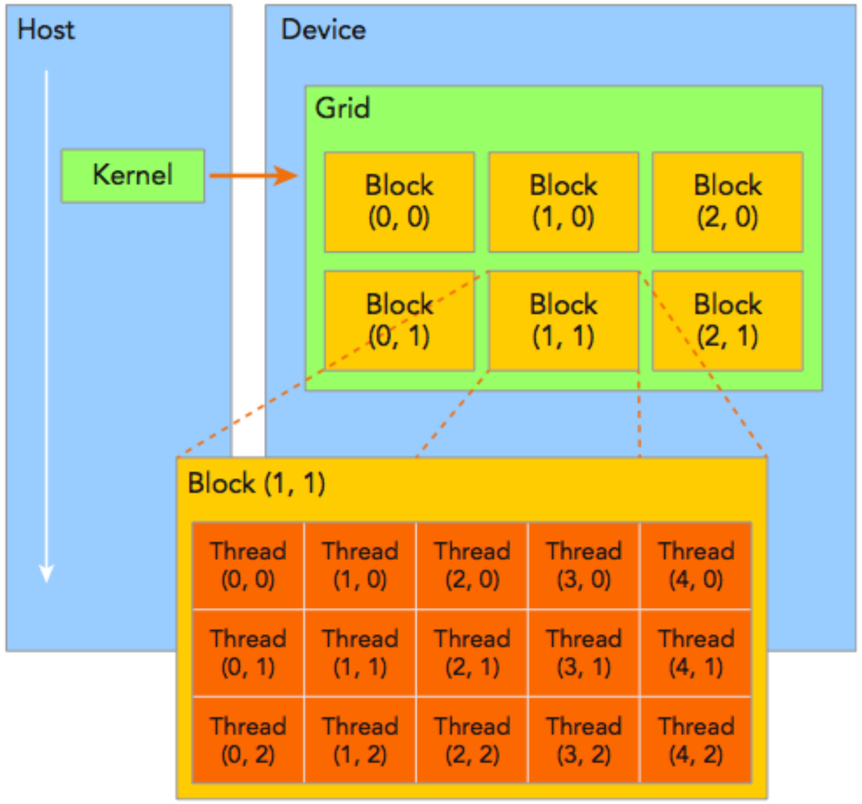
- This function exhibits **synchronous** behavior because the host application blocks until `cudaMemcpy` returns and the transfer is complete.

Organizing threads

Thread hierarchy

- When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function.

- CUDA exposes a thread hierarchy abstraction to enable you to organize your threads. This is a two-level thread hierarchy decomposed into blocks of threads and grids of blocks



Thread memory and coop

- All threads spawned by a single kernel launch are collectively called a **grid**.
  All threads in a grid share the same global memory space.
  A grid is made up of many thread **blocks**. A thread block is a group of threads that can cooperate with each other using:
  - Block-local synchronization
  - Block-local shared memory
- Threads from different blocks cannot cooperate.
- Threads rely on the following two unique coordinates to distinguish themselves from each other:
  - blockIdx (block index within a grid)
  - threadIdx (thread index within a block)

3D threadIdx

- `threadIdx` and `blockIdx` is a 3-component vector (`uint3`), so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block.

- This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

3D threadIdx

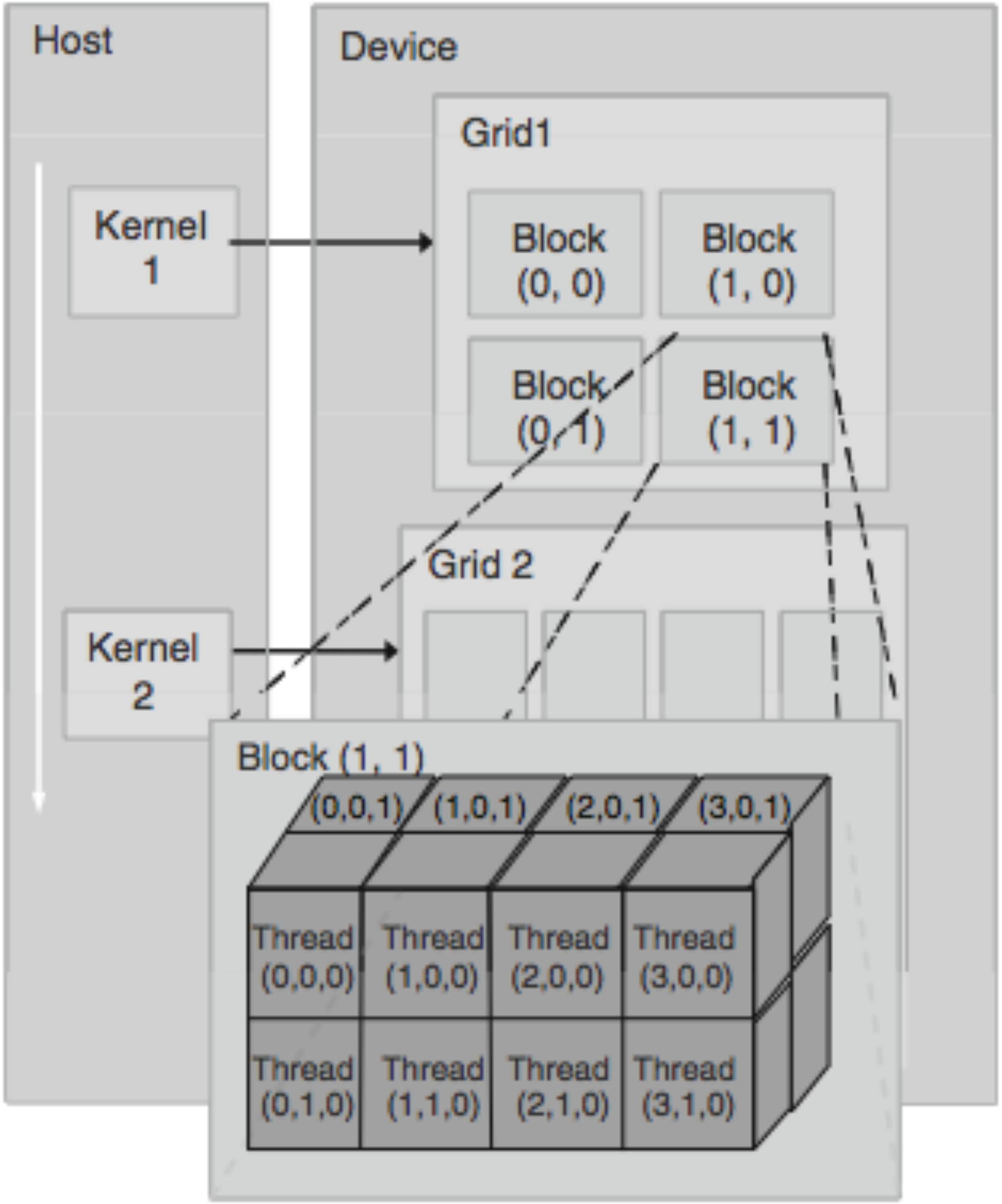
- `threadIdx` and `blockIdx` is a 3-component vector (`uint3`), so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block.

threadIdx and blockIdx are accessible through the fields x, y, and z respectively.

```
blockIdx.x
blockIdx.y
blockIdx.z

threadIdx.x
threadIdx.y
threadIdx.z
```

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate



```
threadIdx.x
threadIdx.y
threadIdx.z
```

Creating threads

- Each CUDA thread grid typically is comprised of thousands to millions of lightweight GPU threads per kernel invocation.
- Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism; for example, each element of a large array might be computed in a separate thread.
- `kernel_name <<<grid, block>>>(argument list);`
- The first value in the execution configuration is `grid`, i.e. the **grid dimension**, the number of blocks to launch. The second value is `block`,i.e. the **block dimension**, the number of threads within each `block`. By specifying the grid and block dimensions, you configure:
  - The total number of threads for a kernel
  - The layout of the threads you want to employ for a kernel

Synchronizing threads

- A kernel call is asynchronous with respect to the host thread. After a kernel is invoked, control returns to the host side immediately.
- You can call the following function to force the host application to wait for all kernels to complete:
- `cudaError_t cudaDeviceSynchronize(void);`

Demo: organizing threads

Summing a vector

- CPU:

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int idx = 0; idx < N; idx++)
        C[idx] = A[idx] + B[idx];
}
```

- GPU

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C,
const int N) {
    int i = threadIdx.x;
    if (i < N)
      C[i] = A[i] + B[i];
}
```

Summing a vector

- CPU:

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int idx = 0; idx < N; idx++)
```

Supposing a vector with the length of 32 elements, you can invoke the kernel with 32 threads as follows:

```
sumArraysOnGPU<<<1, 32>>>(float *A, float *B, float *C, 32);
```

-

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C,
const int N) {
    int i = threadIdx.x;
    if (i < N)
      C[i] = A[i] + B[i];
}
```

Summing a vector

- CPU:

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int idx = 0; idx < N; idx++)
```

Supposing a vector with the length of 32 elements, you can invoke the kernel with 32 threads as follows:

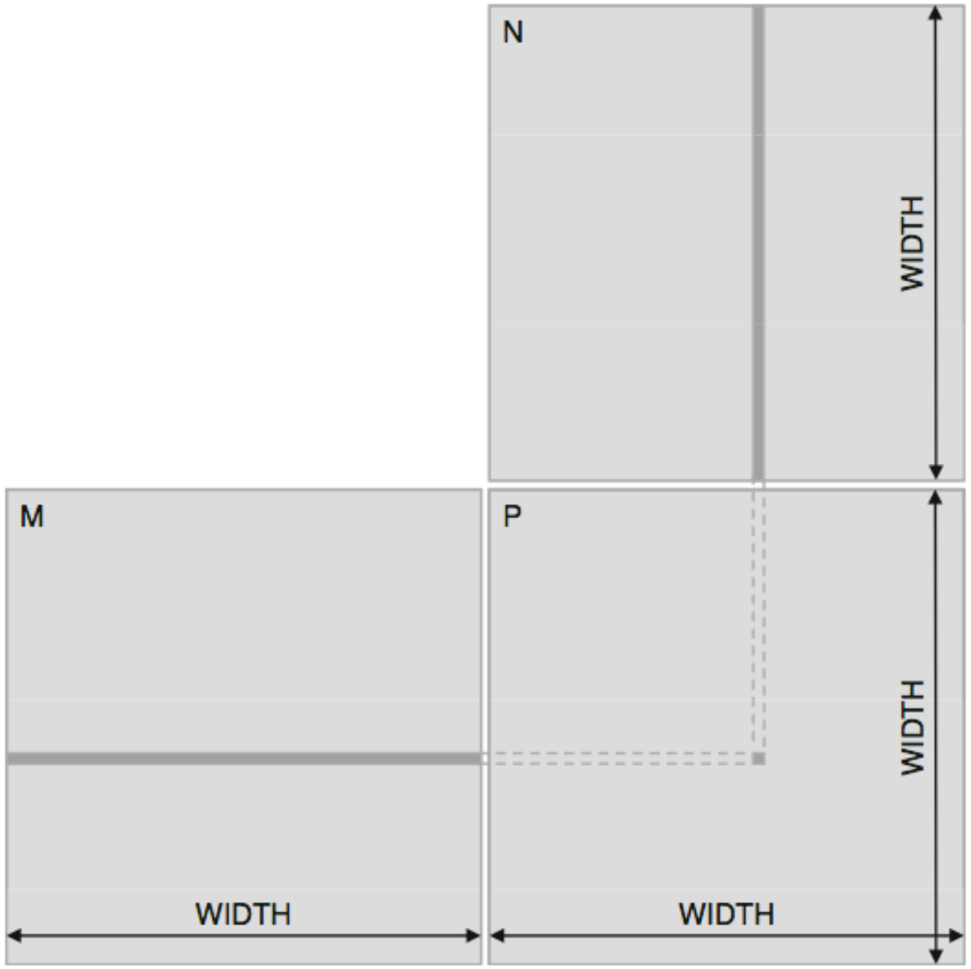
```
sumArraysOnGPU<<<1, 32>>>(float *A, float *B, float *C, 32);
```

Simplified version:

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C) {
       int i = threadIdx.x;
       C[i] = A[i] + B[i];
}
```

Matrix multiplication: CPU

```
void MatrixMultiplication(float* M, float* N, float* P, int width) {
   for (int i = 0; i < width; ++i)
     for (int j = 0; j < width; ++j) {
        float sum = 0;
        for (int k  = 0; k < width; ++k) {
           float a = M[i * width + k];
           float b = N[k * width + j];
           sum += a * b;
         }
       P[i * width + j] = sum;
     }
}
```



Matrix multiplication: GPU

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int width) {

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float PValue = 0;

  for(int k=0; k<width; ++k) {
    float MdElem = Md[ty * width + k];
    float NdElem = Nd[k * width + tx];
    PValue += MdElem * NdElem;
  }

  Pd[ty * width + tx] = PValue;
}
```

Matrix multiplication: GPU

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int width) {

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float PValue = 0;

  for(int k=0; k<width; ++k) {
    float MdElem = Md[ty * width + k];
    float NdElem = Nd[k * width + tx];
    PValue += MdElem * NdElem;
  }

  Pd[ty * width + tx] = PValue;
}
```

Instead of two cycles on i and j, the CUDA threading hardware generates all of the threadIdx.x and threadIdx.y values for each thread.
Each thread uses its threadIdx.x and threadIdx.y to identify the row of Md and the column of Nd to perform the dot product operation.

Matrix multiplication: GPU

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int width) {

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float PValue = 0;

  for(int k=0; k<width; ++k) {
    float MdElem = Md[ty * width + k];
    float NdElem = Nd[k * width + tx];
    PValue += MdElem * NdElem;
  }

  Pd[ty * width + tx] = PValue;
```

Instead of two cycles on i and j, the CUDA threading hardware generates all of the threadIdx.x and threadIdx.y values for each thread.
Each thread uses its threadIdx.x and threadIdx.y to identify the row of **Md** and the column of **Nd** to perform the dot product operation.

Thread$_{2,3}$ will perform a dot product between column 2 of **Nd** and row 3 of **Md** and write the result into element (2,3) of **Pd**.
This way, the threads collectively generate all the elements of the **Pd** matrix.

Memory access

- M and N must be copied to the Md and Nd matrices allocated in the GPU
- Pd must be copied from te device back to the host
- Once all these operations are concluded it's possible to cudaFree Md, Nd and Pd

Limitations

- All these examples use only 1 block, but there's limit on the number of threads per block
- Indexing no longer as simple as using only `threadIdx.x` / `threadIdx.y`
  - One will have to account for the size of the block as well

Credits

- These slides report material from:
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)
  - Prof. Dan Negrut (Univ. Wisconsin - Madison)
  - NVIDIA GPU Teaching Kit

Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufmann - 2nd edition - Chapt. 1 and 3

  or

  Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufmann - 3rd edition - Chapt. 1-2

- Professional CUDA C Programming, J. Cheng, M. Grossman and T. McKercher, Wrox - Chapt. 1-2