COMP 4901Q: High Performance Computing (HPC)

Lecture 7: Introduction to GPU Computing

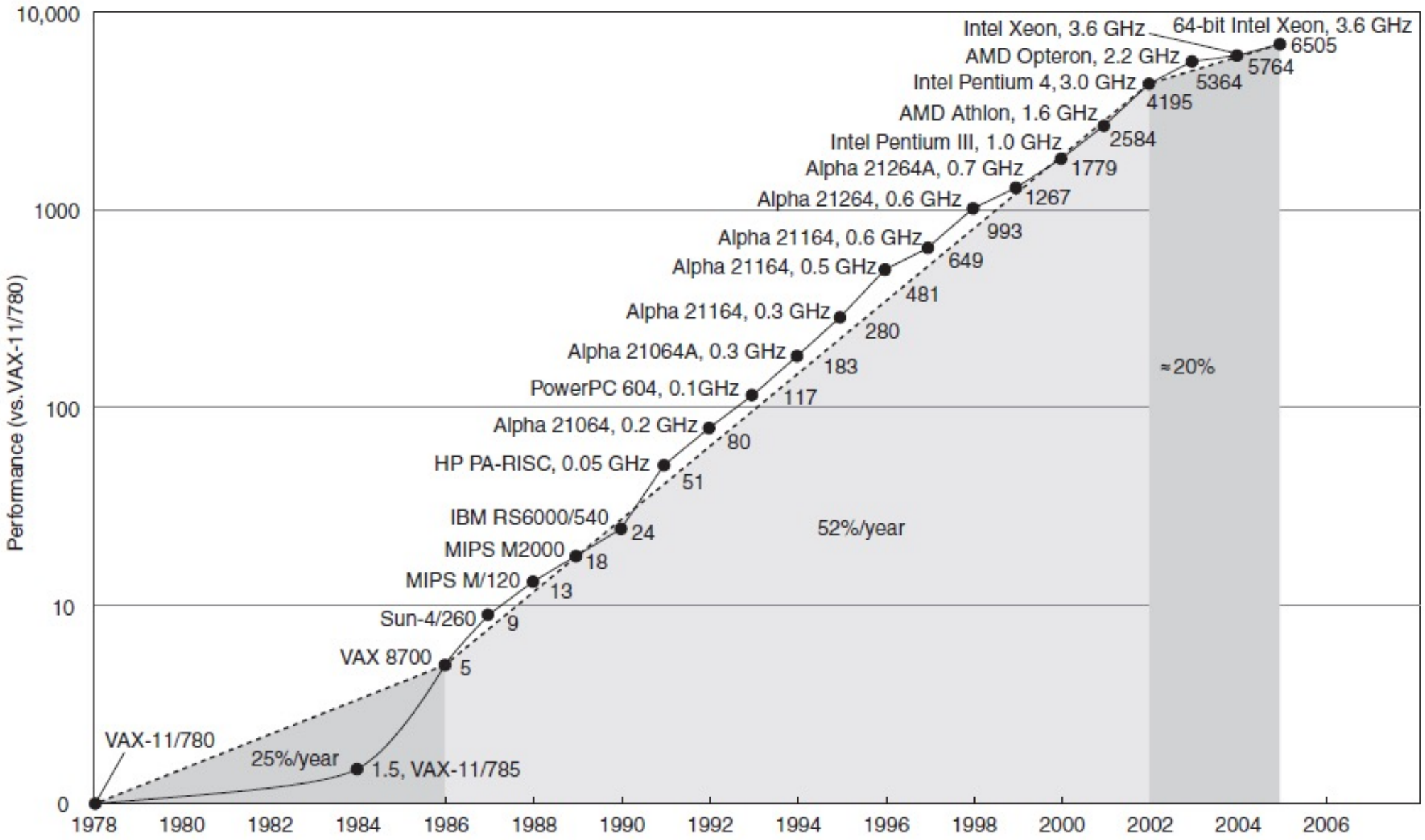
Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)
Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)
Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

Outline

- Introduction to GPU Computing
  - History of processors
  - GPU architecture
- CUDA Programming Model: Basics
  - Workflow
  - Fundamental CUDA API
  - Examples

History of Processor Performance



Observations

- Before 2002, the CPU performance grows by increasing its working frequency

- Afterwards, the CPU frequency remains between 2GHz – 4GHz
  - Challenge of CPU heat dissipation

  - The performance is mainly improved by introducing parallelism, such as multi-core and SIMD (MMX, SSE, AVX, etc.)

Multi-core Processors

- Multi-core CPU combines two or more independent cores into a single package
  - IBM introduced POWER4 in 2001, which places two complete CPU cores on a single chip.
    - POWER5, POWER6, POWER7 (4-8 cores)
  - Sun released Niagara with 8 cores UltraSPARC in 2005, and SPARC T3 with 16 cores in 2010
  - Intel and AMD released their dual-core processors in 2005
    - Intel launched its quad-core processor in 2006, 8-core in 2010, 10-core in 2011, 18-core in 2014
    - AMD released its quad-core processor in 2007, 12-core in 2010, 16-core in 2014

Many-core Processors

- Many-core processors have more processing units than multi-core processors
  - They are usually coprocessors to supplement the CPUs

- Intel's MIC: Many Integrated Core Architecture
  - Market name: Intel Xeon Phi
    - Around 60 cores, each with SIMD unit (vector processing unit)
    - x86-compatible
    - used by Tianhe-II and many other top500 supercomputers

- **GPU: Graphics Processing Unit**
  - Nvidia: GTX980, Tesla K40, RTX3090, Tesla A100
  - AMD: FirePro S10000, MI100
  - Also popular in top500 supercomputers

Multi-core vs. Many-core

Year	Model	# of cores	Peak perf (SP)[1]	Peak perf (DP)[2]	Peak perf (FP16)[3]	Memory Bandwidth	Power
2010	Intel Xeon X5650	6	0.128T	0.064T	N/A	32GB/s	95W
2013	Intel Xeon Phi 7120p	61	2.4T	1.2T	N/A	352GB/s	300W
2013	Nvidia Tesla K40	2880	4.29T	1.43T	N/A	288GB/s	235W
2012	AMD FirePro S10000 [4]	2x1792	5.91T	1.48T	N/A	480GB/s	375W
2020	Nvidia Tesla A100	6912	19.5T	9.7T	312T	1555GB/s	400W
2020	AMD Instinct™ MI100	7680	23.1T	11.5T	184.6T	1228.8GB/s	300W

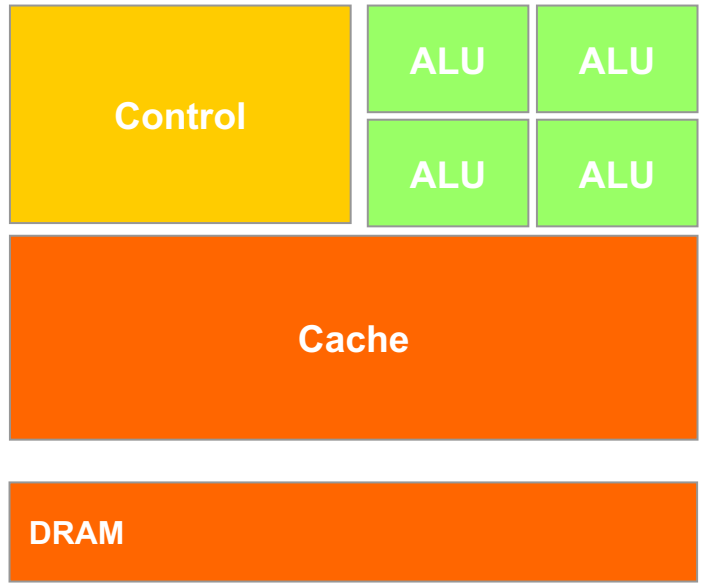
[1]SP: single precision; [2]DP: double precision; [3] Half precision; [4]AMD FirePro S10000 has two GPU chips

GPU: Graphics Processing Unit

- GPU is used to accelerate the creation of images intended for output to a display
  - A display with resolution 1920x1080 has 2 million pixels to process, and tens of frames per second!

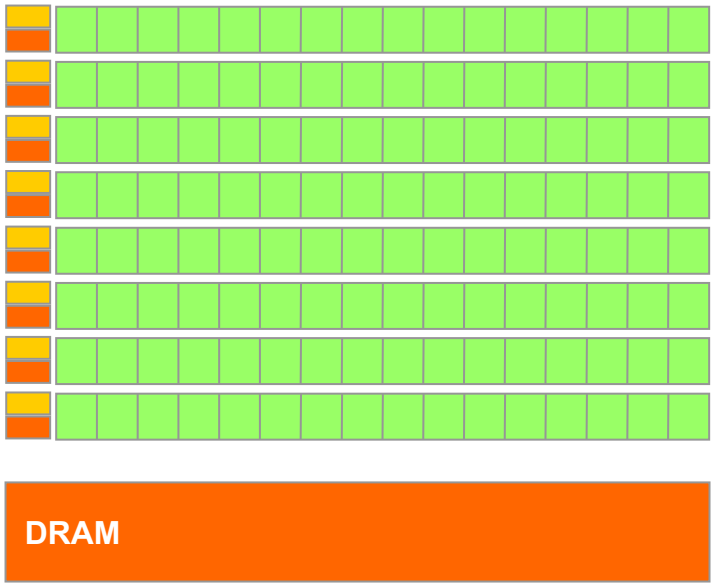
- Today's GPUs use hundreds to thousands of processing cores for calculation
  - Each core is less powerful than a typical CPU core
  - But hundreds to thousands of GPU cores will make a big difference

Different Design Philosophies



CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution



GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

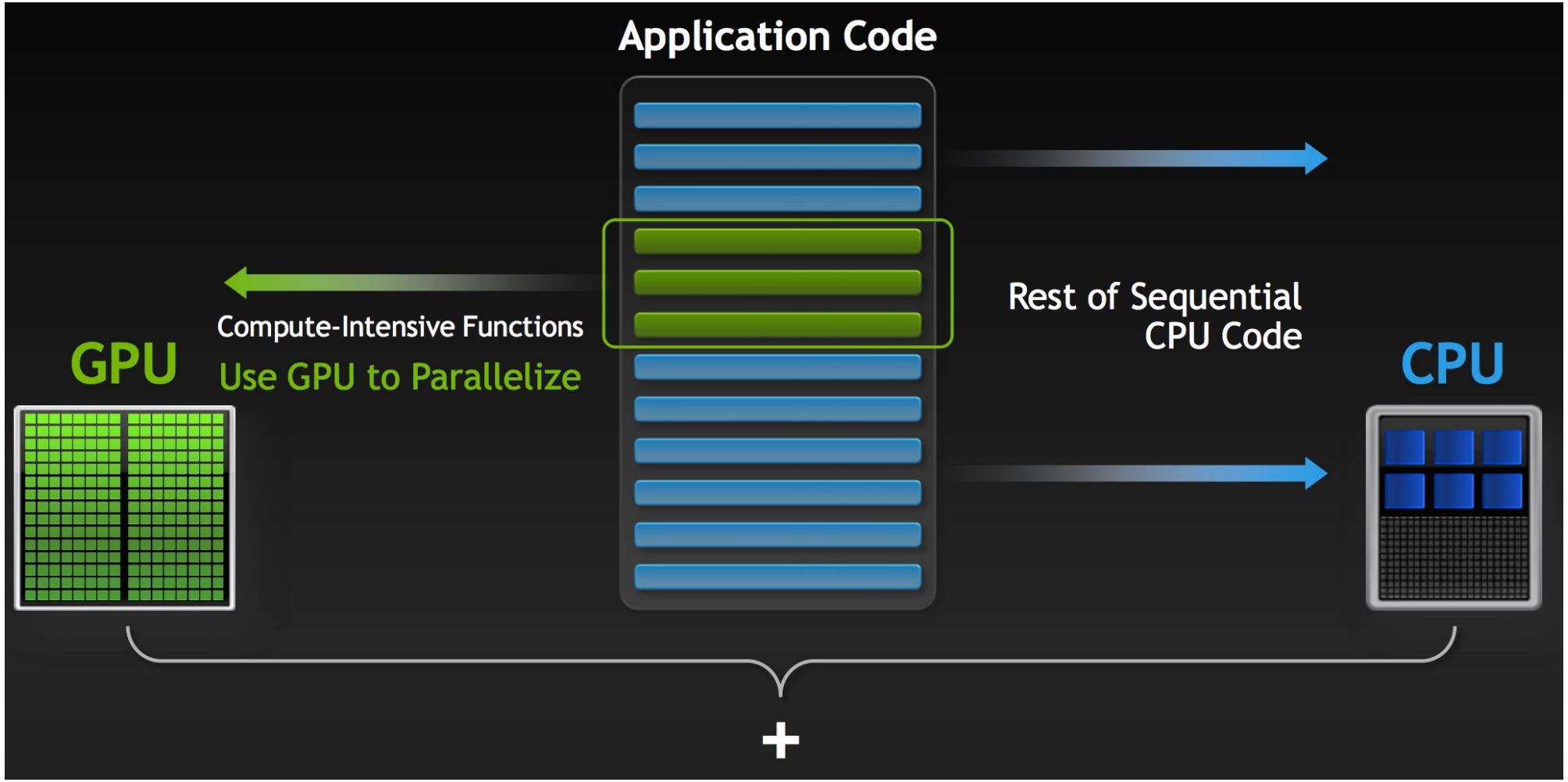
GPGPU: General-Purpose Computing on GPUs

- GPUs are designed for 2D/3D graphics

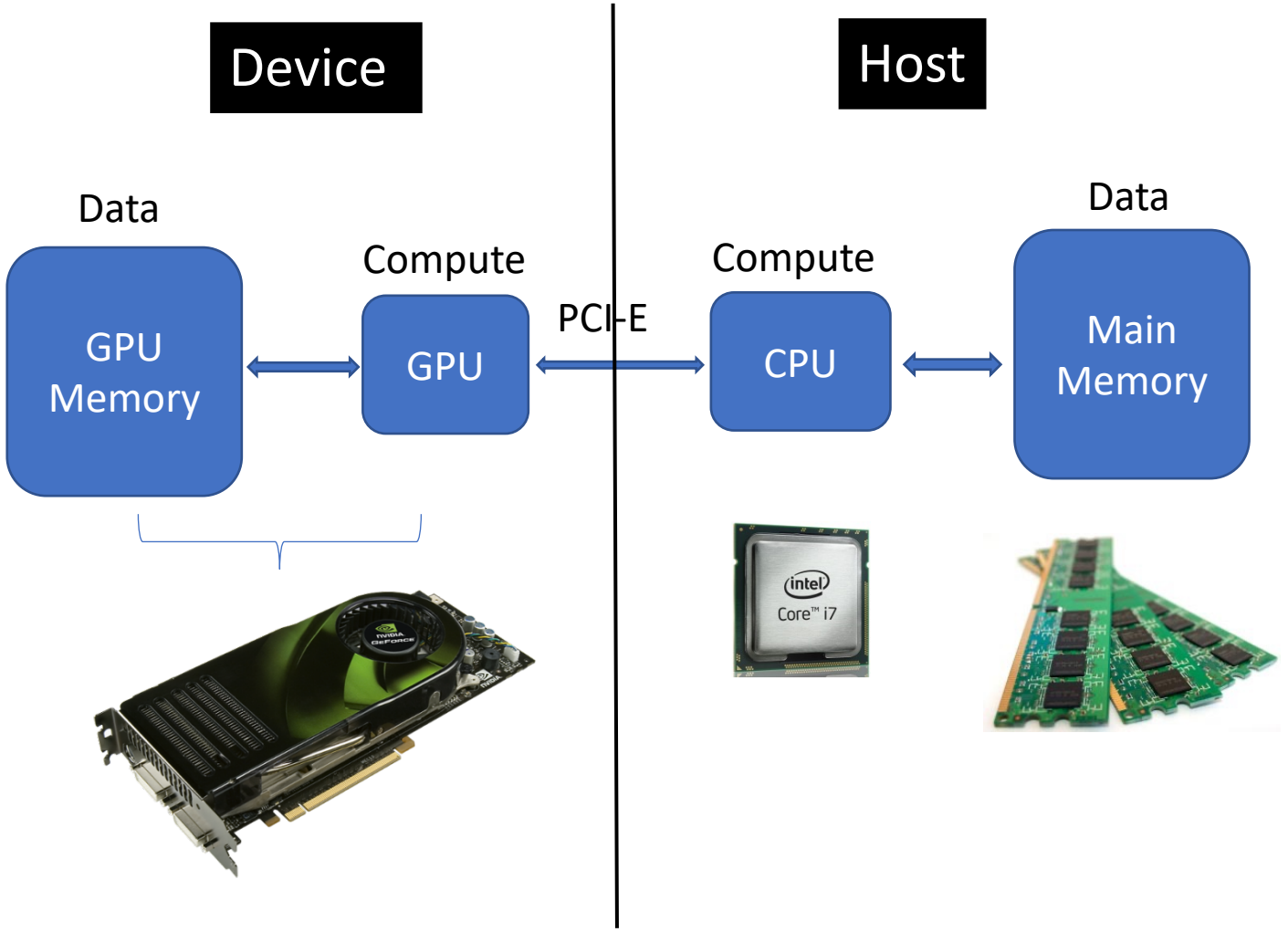
- GPGPU refers to the usage of GPUs for accelerating applications other than graphics
  - Computational finance, data mining, machine learning, data analytics, imaging and vision, bioinformatics, CAD, molecular dynamics, quantum chemistry, etc.

  - Also known as "GPU Computing"

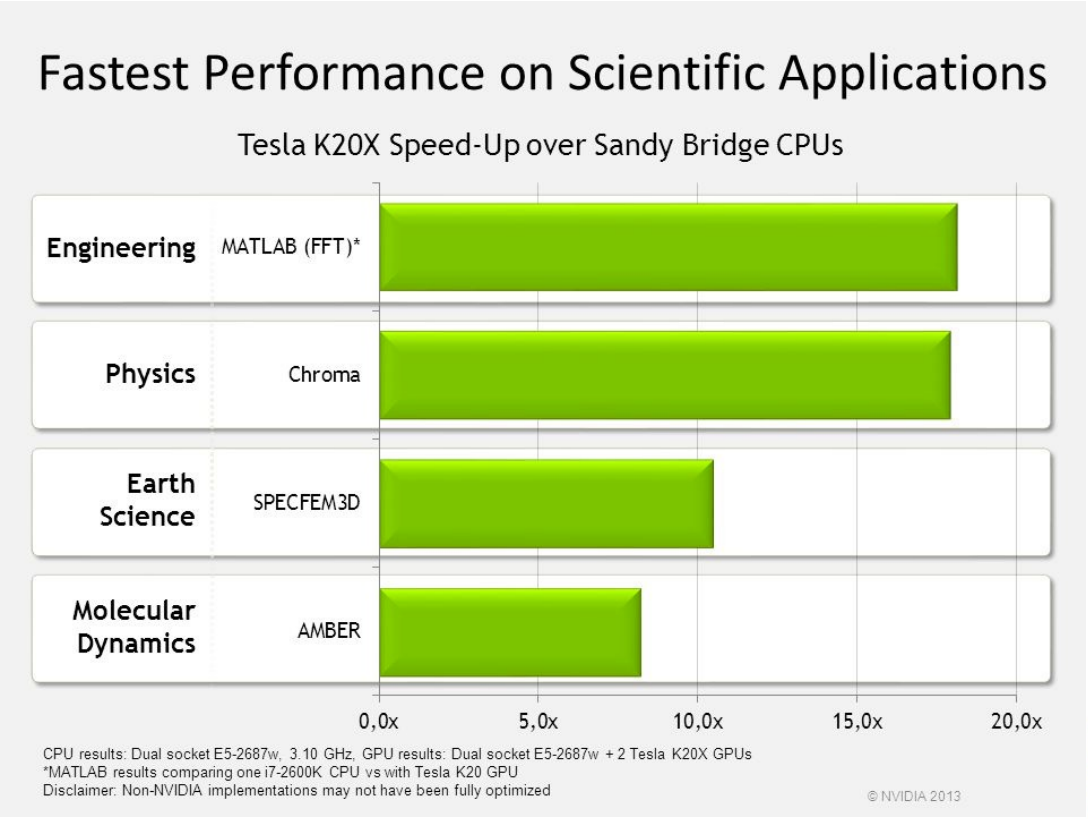
Small Changes, Big Speed-up



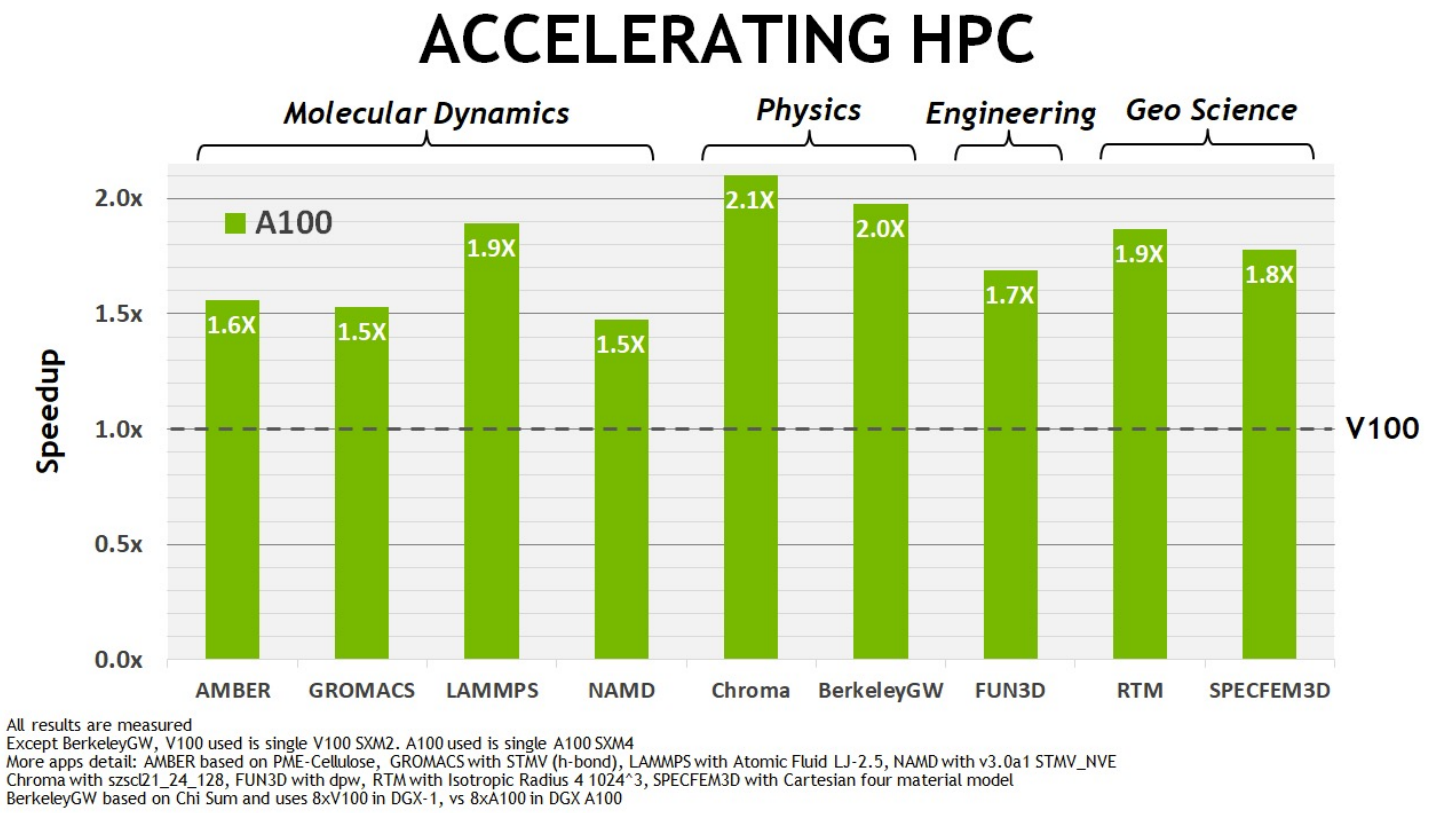
Heterogeneous Computing



GPUs Accelerate Science and Engineering



2013



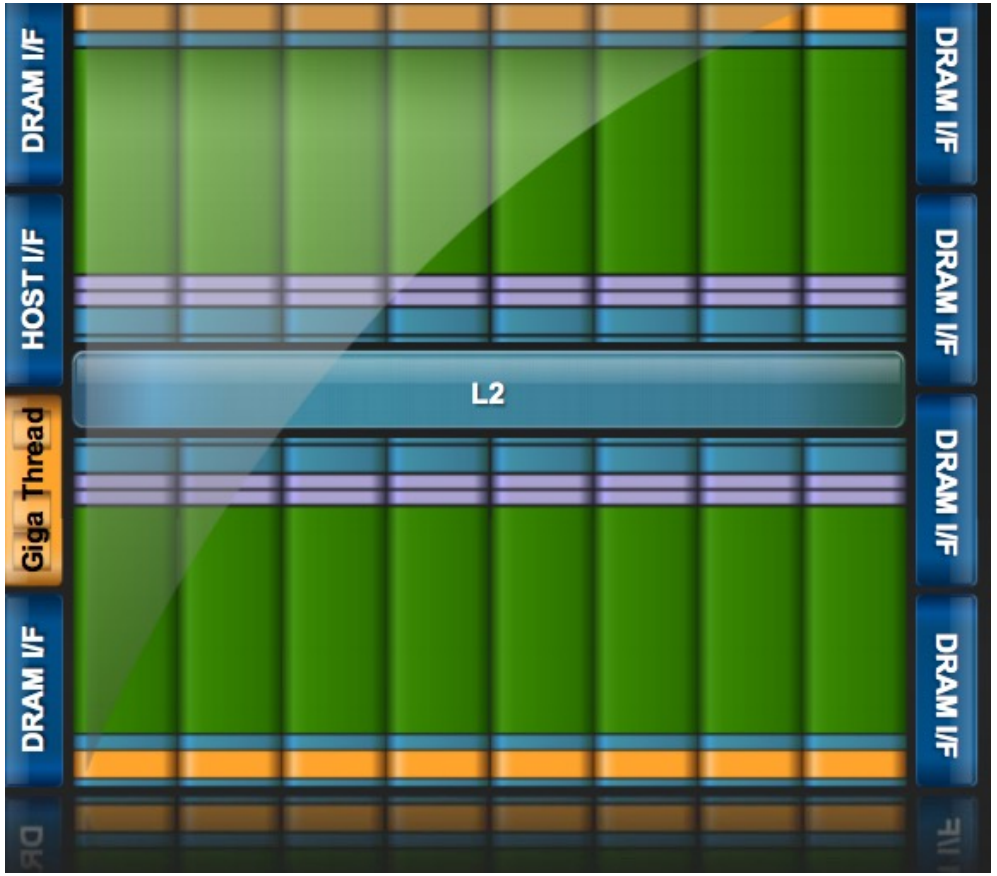
2020

GPUs have almost been a must in deep learning!

https://www.nvidia.com/en-us/gpu-accelerated-applications/

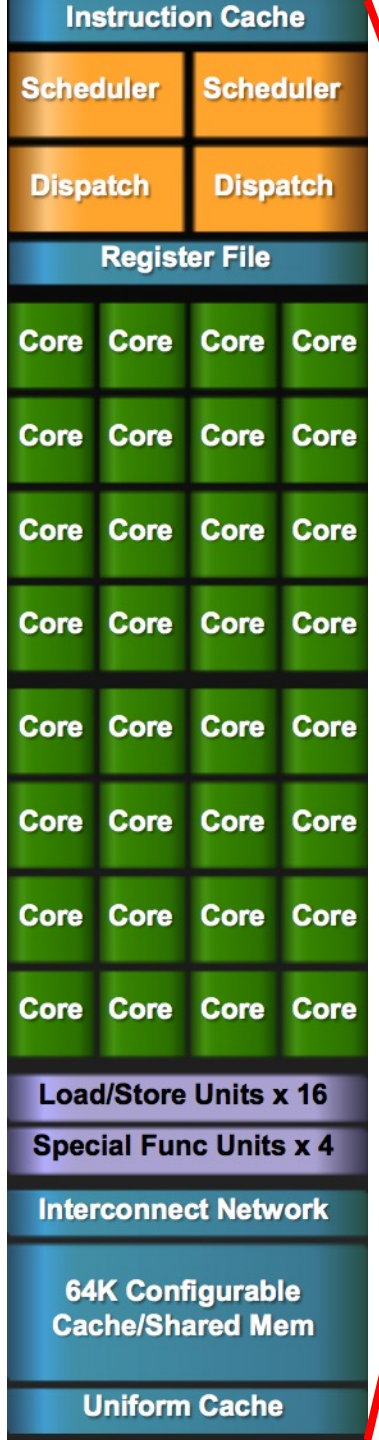
GPU Architecture: Two Main Components

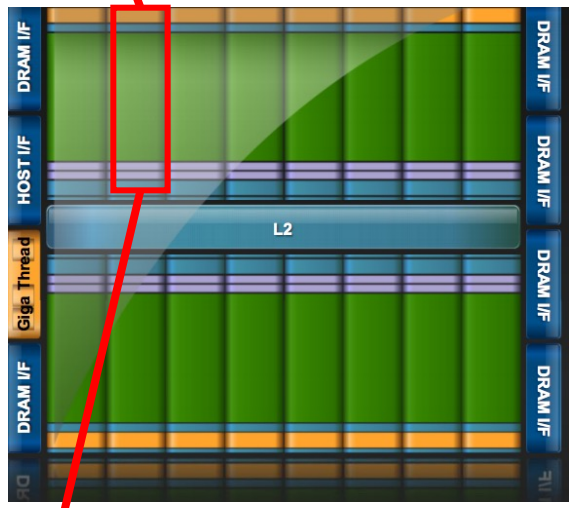
- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 80 GB
  - Bandwidth currently up to 2,039 GB/s for Tesla products
  - ECC on/off option for Quadro and Tesla products
- Streaming Multiprocessors (SMs)
  - Perform the actual computations
  - Each SM has its own:
    - Control units,
    - Registers,
    - Execution pipelines
    - Caches



Fermi:
Streaming Multiprocessor (SM)

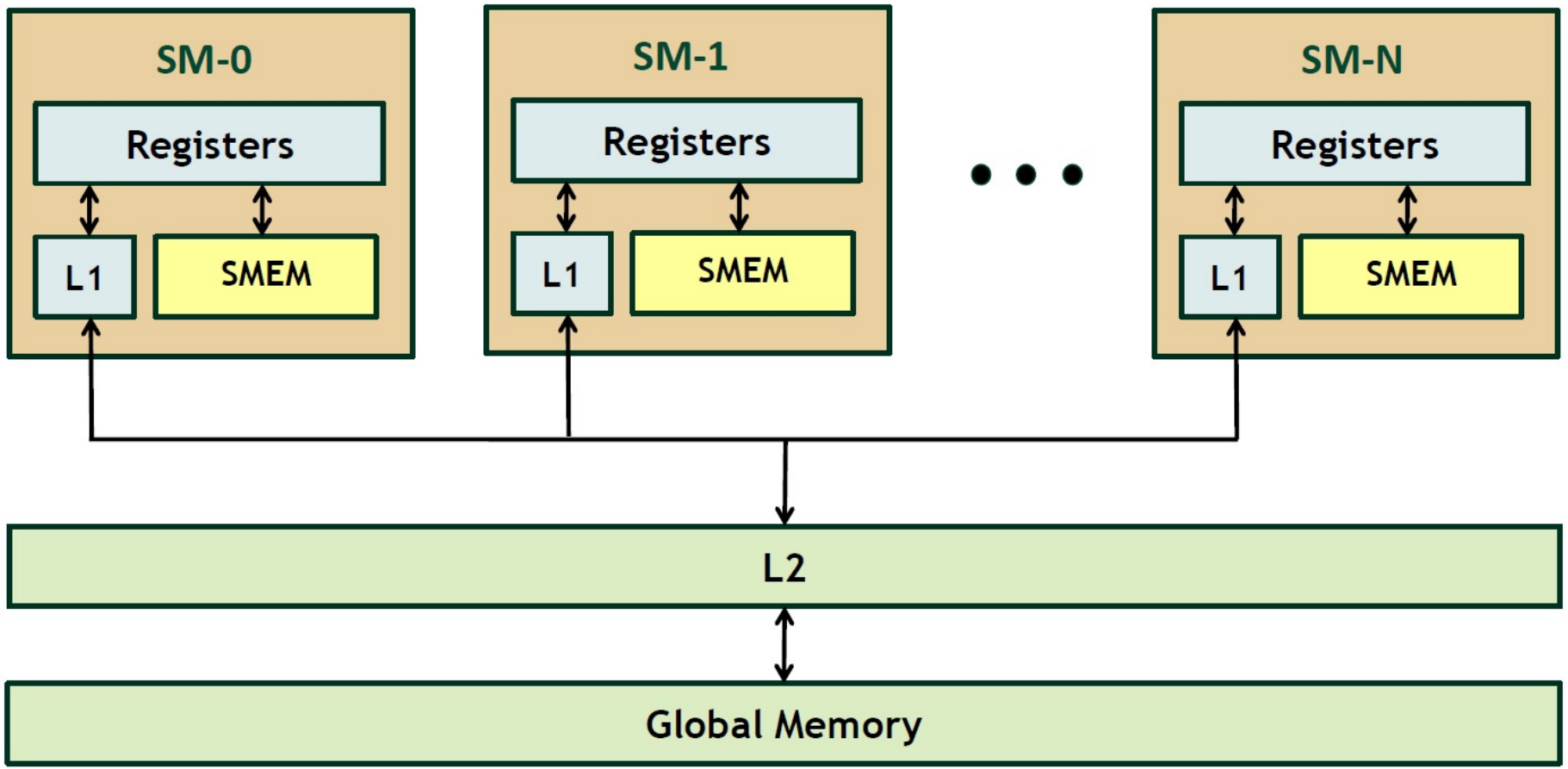
- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



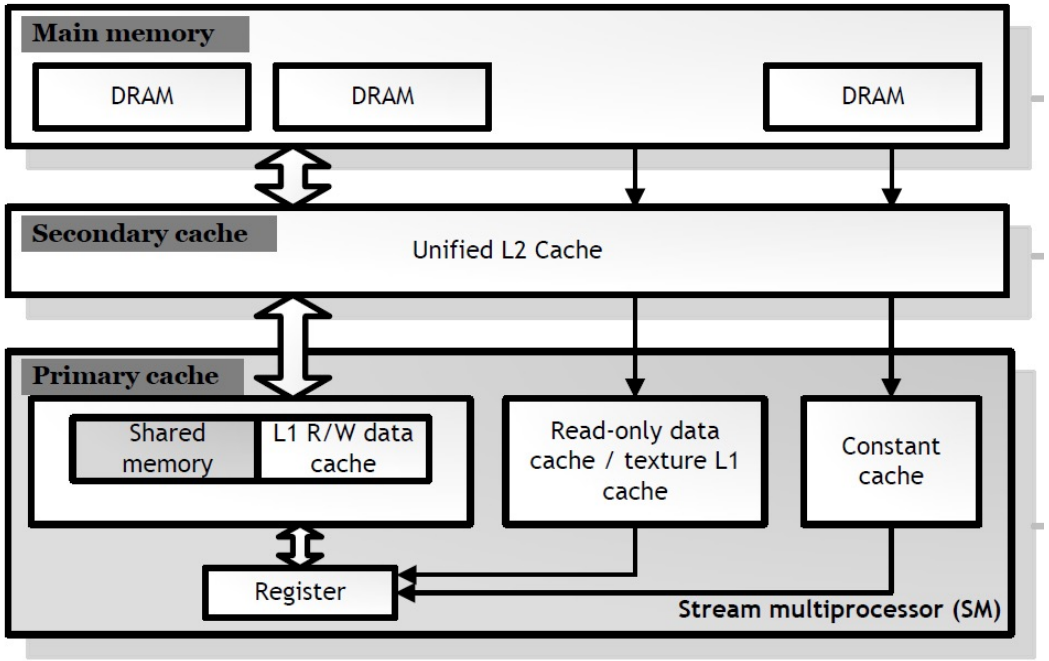


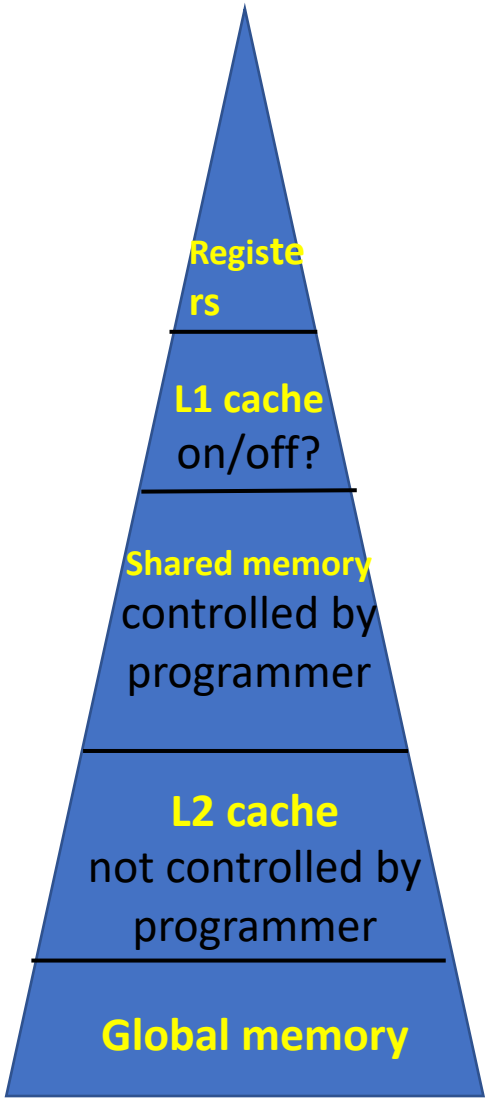
White papers
https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

GPU Memory Hierarchy



GPU Memory Hierarchy

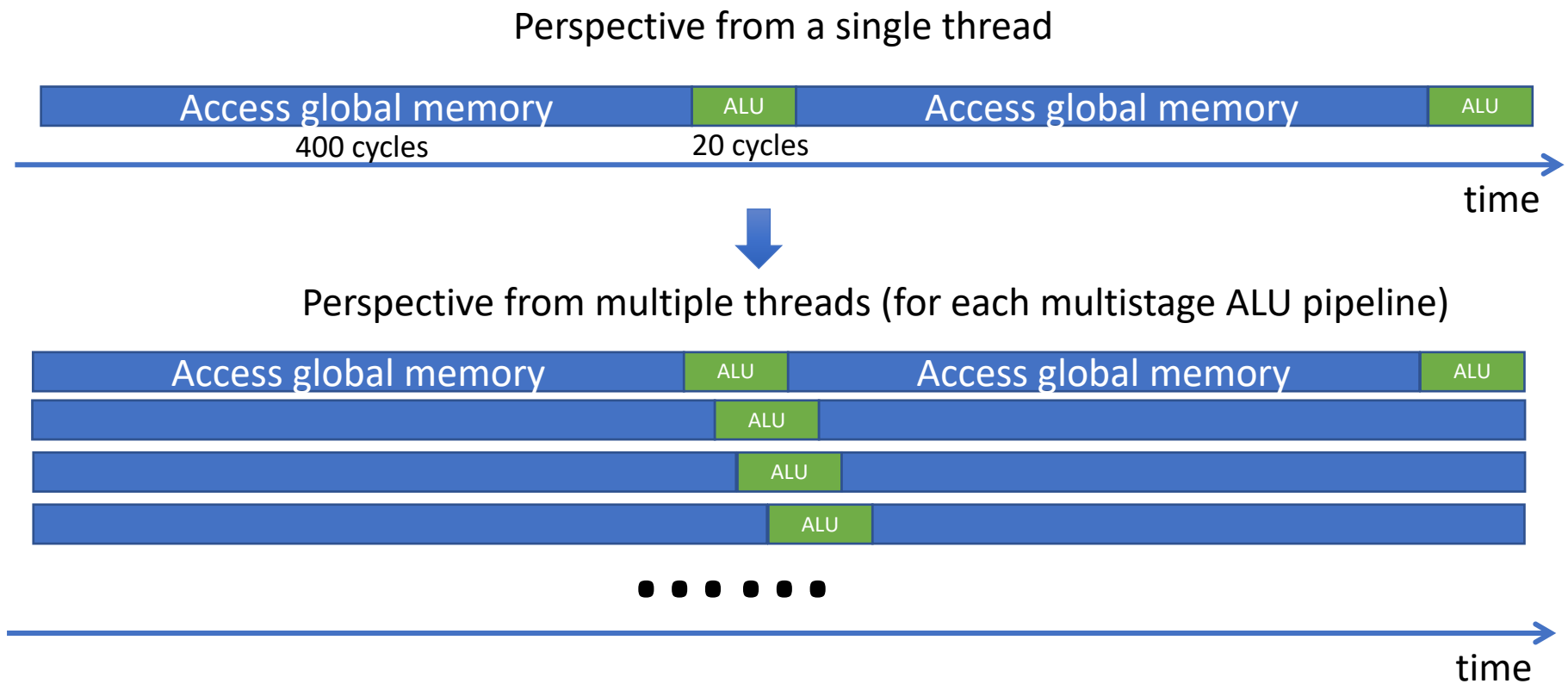




- Example: P100 (PCIe)
  - Registers: 14336 KB
  - L1 cache/Shared mem: 64 KB per SM
  - L2 cache: 4096 KB
  - Global mem: 16 GB
  - Memory bandwidth: 732 GB/s

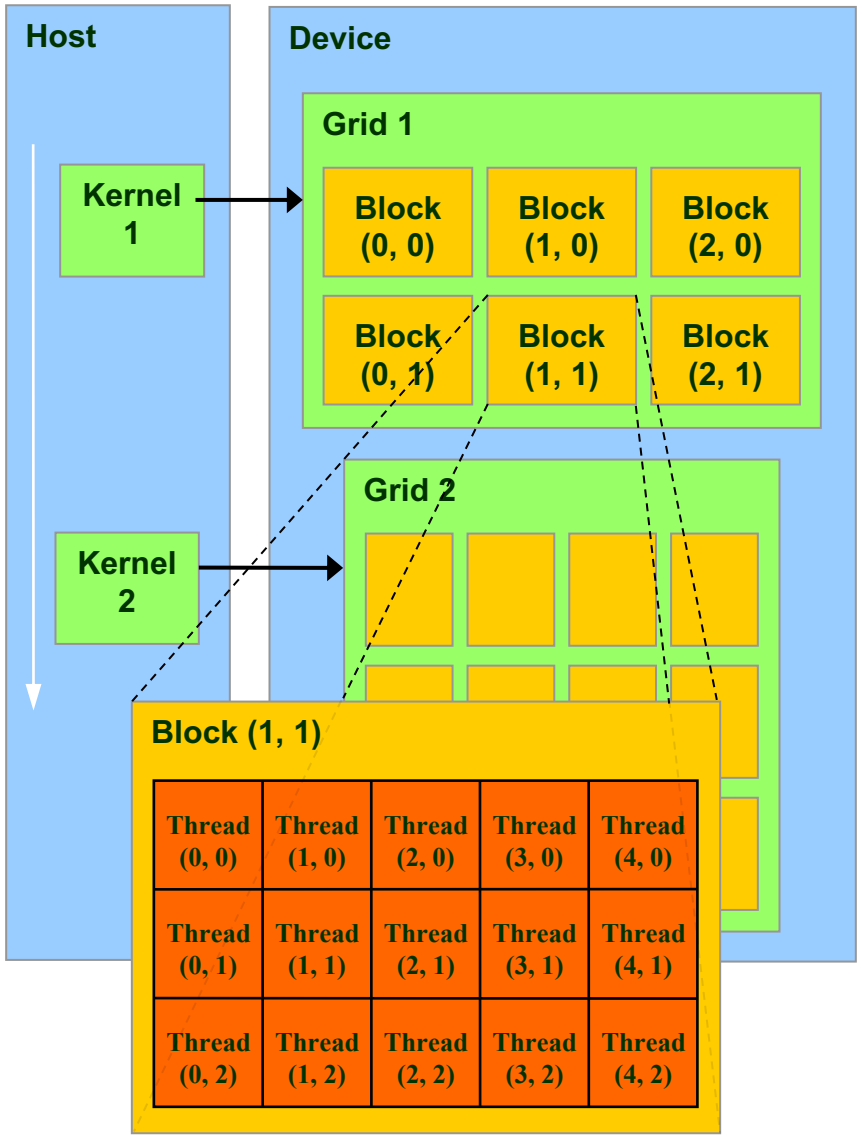
Multithreading to High Memory Latency

- Compute vs Memory
  - E.g., 9340GFLOPS vs 732 GB/s

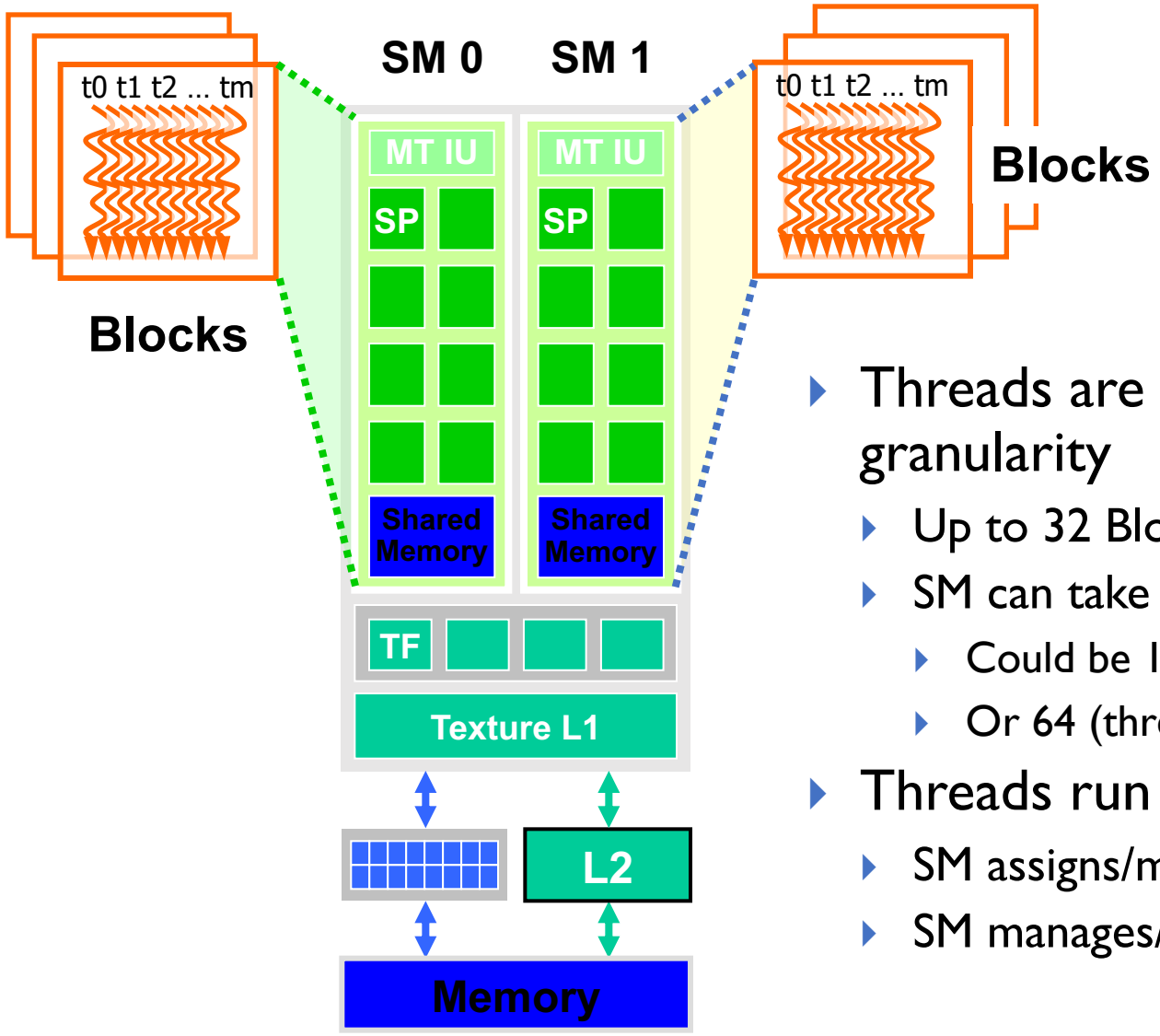


Thread Life Cycle in Hardware

- Grid is launched on GPU
- Thread Blocks are serially distributed to all the SMs
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - GPU can distribute more Thread Blocks



SM Executes Blocks

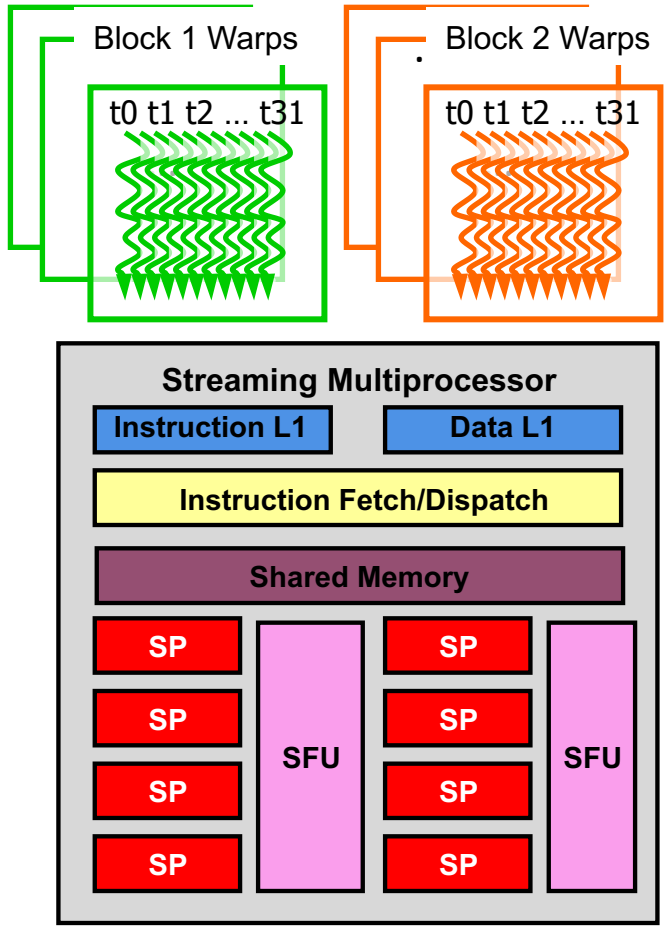


Assume Compute Capability 5.0:

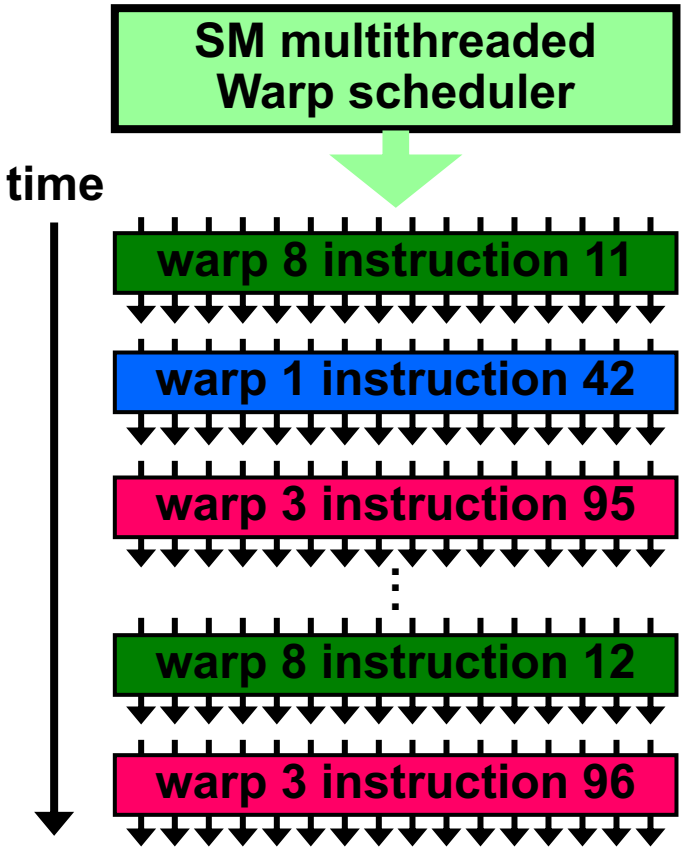
- Threads are assigned to SMs in Block granularity
  - Up to 32 Blocks to each SM as resource allows
  - SM can take up to 2048 threads
    - Could be 1024 (threads/block) * 2 blocks
    - Or 64 (threads/block) * 32 blocks, etc.
- Threads run concurrently
  - SM assigns/maintains thread id
  - SM manages/schedules thread execution

Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 512 threads, how many Warps are there in an SM?
  - Each Block is divided into 512/32= 16 Warps
  - There are 16 * 3 = 48 Warps
  - At any point in time, some of the 48 Warps will be selected for instruction fetch and execution



SM Warp Scheduling

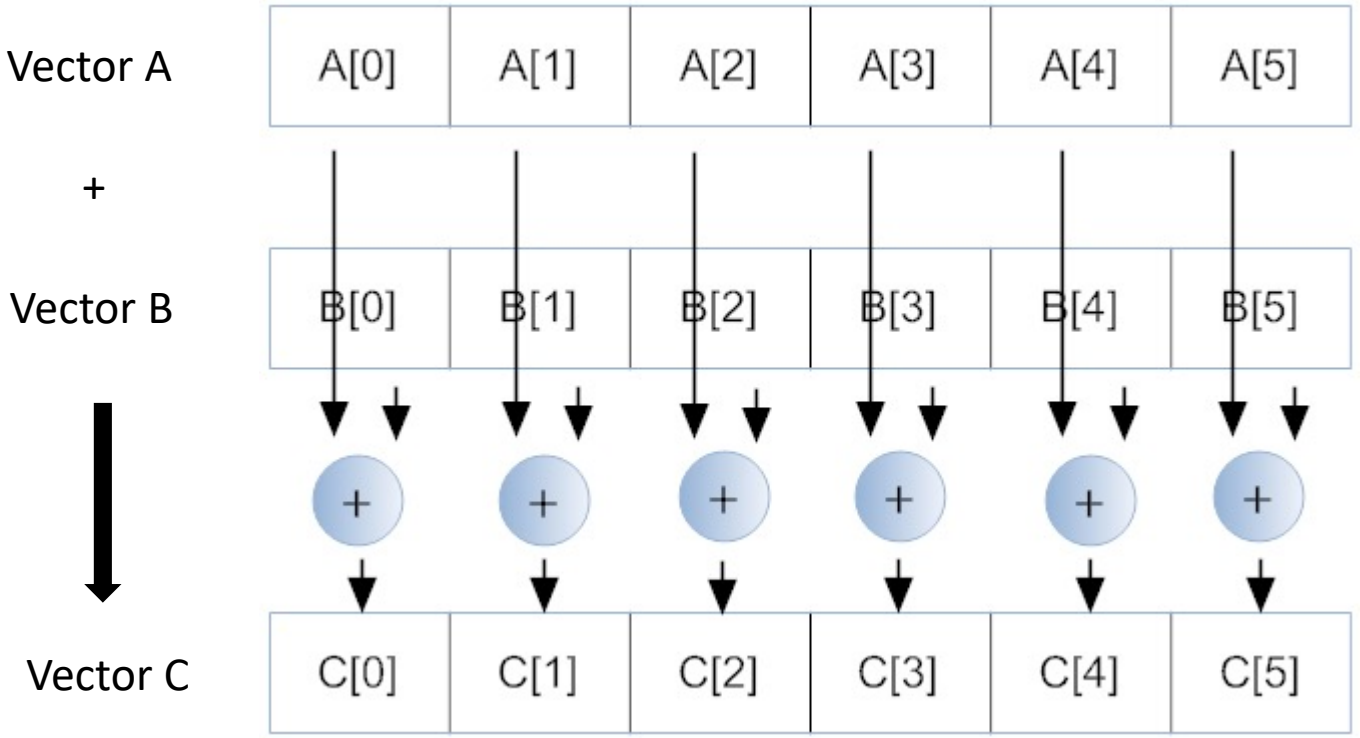


- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

- E.g., in G80, 4 clock cycles needed to dispatch the same instruction for all threads in a Warp
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

Review: Data Parallelism

- In data parallelism, each processor performs the same task on different pieces of data



GPU Programming

- CUDA
  - Particularly for Nvidia GPUs
- OpenCL
  - Supports all popular operating systems across all major platforms
    - CPU, GPUs (Nvidia, AMD, Intel), FPGA,
- OpenACC
  - Directive-based performance-portable parallel programming model

CUDA: Compute Unified Device Architecture

- CUDA is a general-purpose programming model for heterogeneous computing
  - Created by Nvidia to ease GPU computing in 2007
  - User can generate batches of threads on the GPU
  - GPU becomes a dedicated super-threaded, massively data parallel coprocessor
  - CUDA includes libraries, compilers, and extensions to programming languages

CUDA Devices and Threads

- In CUDA, a compute device
  - is a coprocessor to the CPU or host
  - has its own DRAM (device memory)
  - runs many threads in parallel
  - is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Compute Capability

- GPU devices are evolving rapidly, with many new features introduced by each generation of hardware

- "Compute capability" is used to identify the features supported by the GPU hardware

- "Compute capability" comprises a major and a minor version number (x.y), e.g., 1.3, 2.0, 3.5, 5.0, 7.0
  - Devices with the same major version number are of the same core architecture
  - The minor version number corresponds to an incremental improvement to the core architecture

CUDA Programming Model

- SPMD
  - Single Program/Process Multiple Data
  - Tasks are split up and run simultaneously on multiple processors with different input
  - MPI is an example of SPMD using distributed memory

- SPMD on GPUs:
  - CUDA is an example of SPMD using shared memory
  - The GPU processes many elements in parallel using the same program
  - Elements can read data from a shared global memory ("gather"), and also write back to arbitrary locations in memory ("scatter")

CUDA C/C++

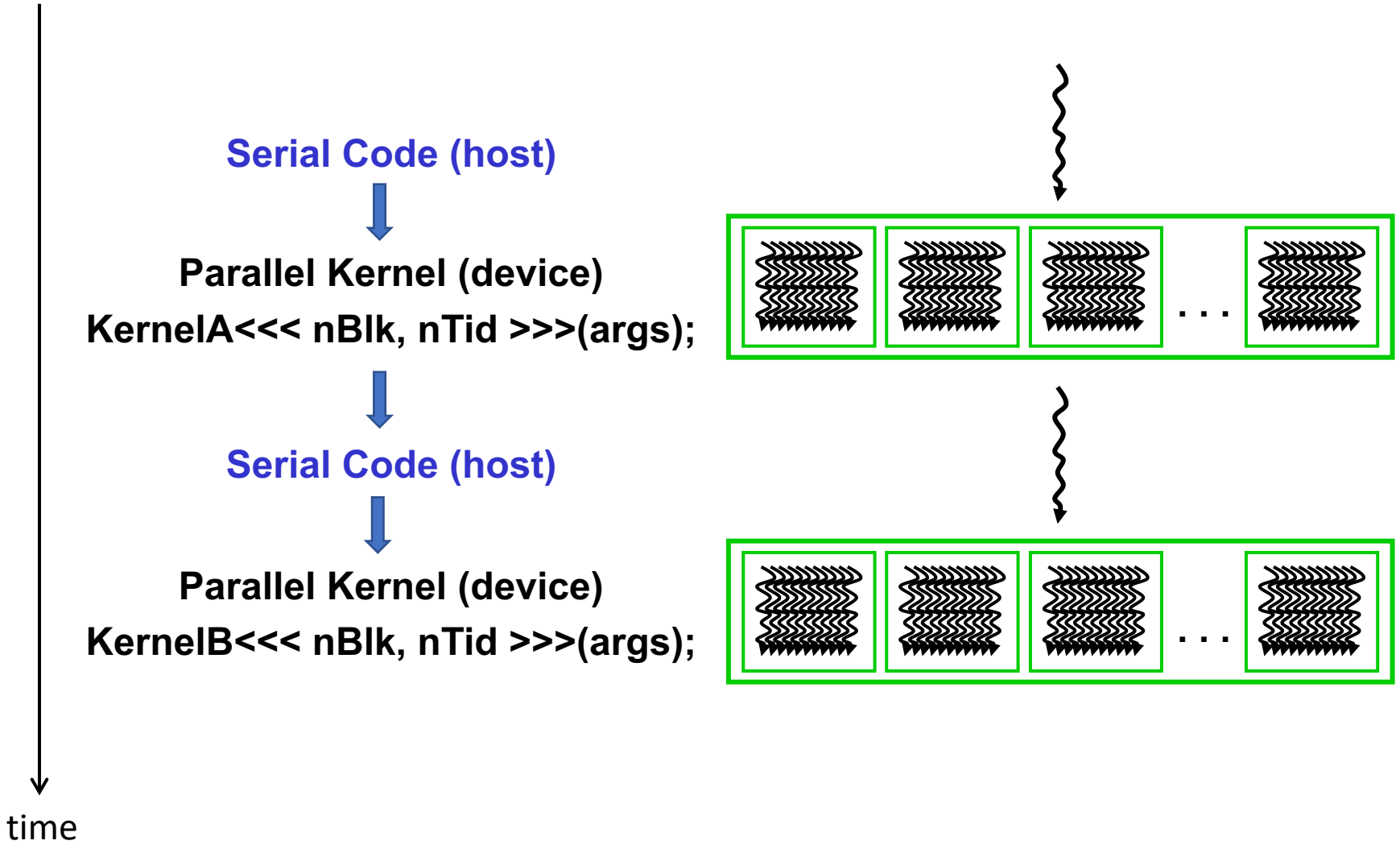
- CUDA supports many programming languages, e.g., Fortran and C/C++
  - We choose CUDA C/C++ in this course

- CUDA C/C++ is an extension of C/C++ language
  - A set of new keywords
  - A set of API functions

- CUDA C/C++: Integrated host+device C/C++ program
  - Serial or modestly parallel parts in host C/C++ code, including the main() function

  - Highly parallel parts in device C/C++ code, called kernels

Execution of a CUDA C Program



Memory Management

- In CUDA, host (i.e., CPU) and device (i.e., GPU) have separate memory spaces

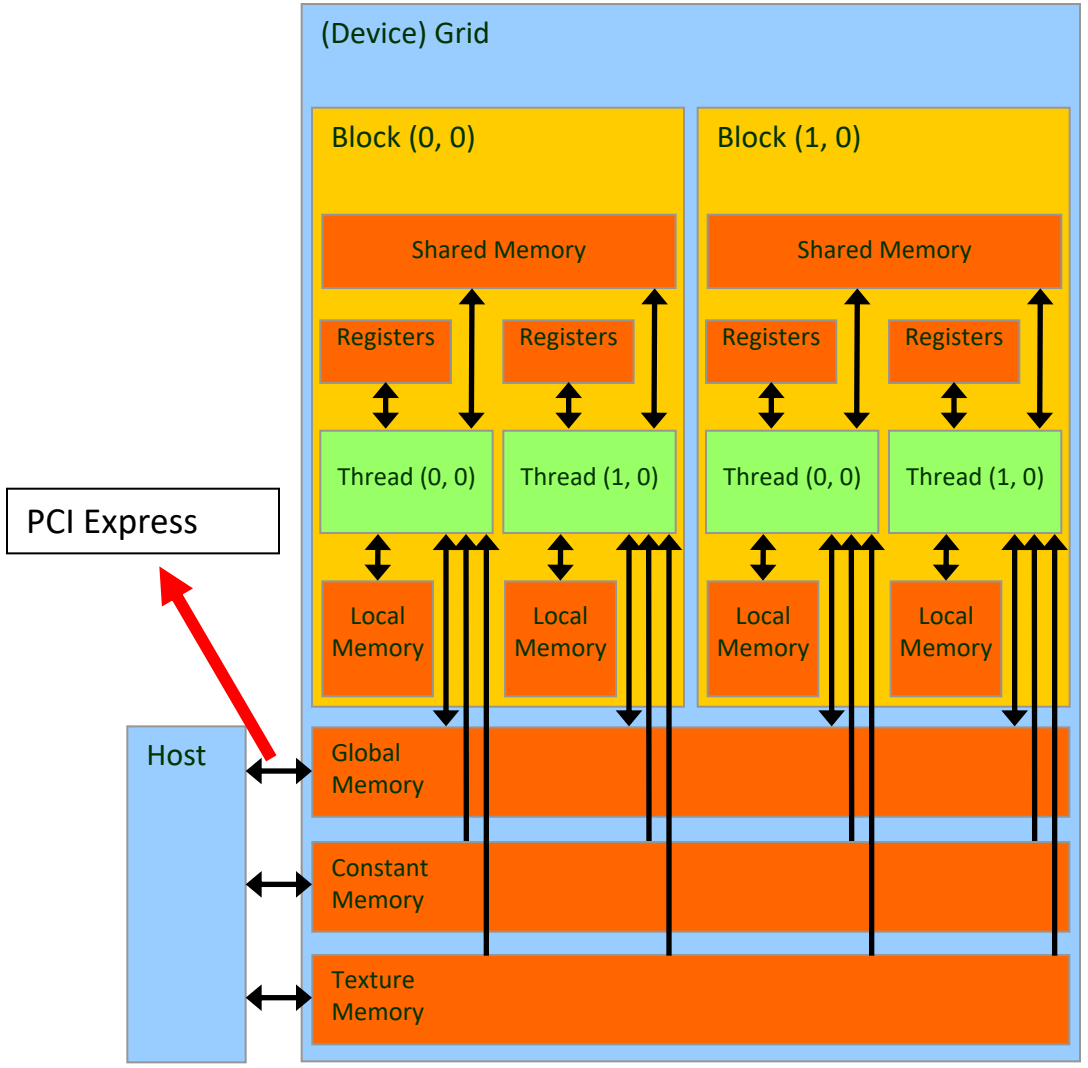
- To execute a kernel on GPU, we need to
  1. allocate memory on the device
  2. transfer data from host memory to allocated device memory

- After device execution, we need to transfer the result data from device memory back to host

CUDA Memory Model

- Each thread 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
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories



Memory Functions

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

cudaMemcpyKind
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

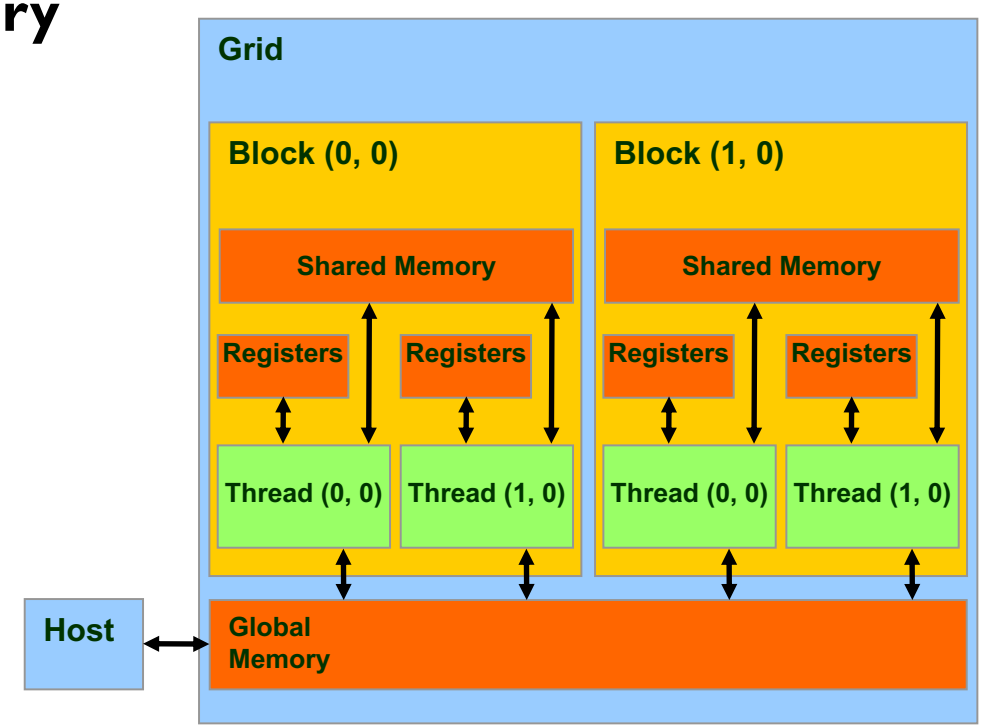
cudaError_t cudaMalloc(void** devPtr, size_t size)
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum **cudaMemcpyKind** kind, cudaStream_t stream = 0)
cudaError_t cudaMemset(void* devPtr, int value, size_t count)
cudaError_t cudaFree(void* depPtr)

cudaError_t: {cudaSuccess, cudaErrorMemoryAllocation, …}

CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device **Global Memory**
  - Requires two parameters
  - Address of a pointer to the allocated object
  - Size of allocated object

- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



Code Example

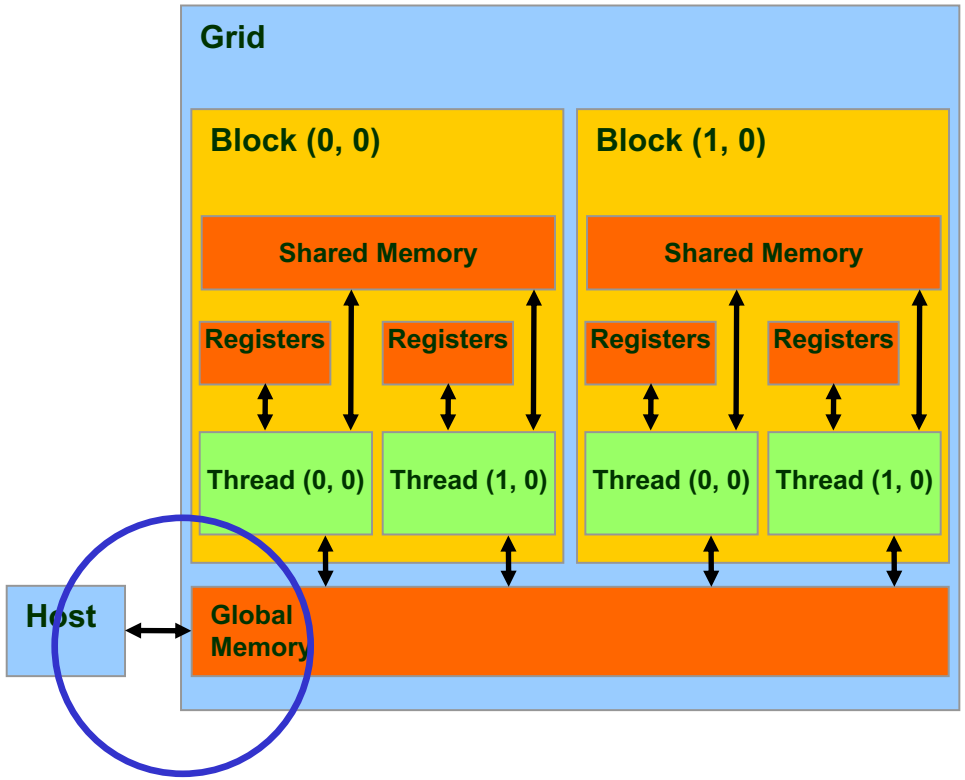
- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to pointer M_d
  - “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;
float* M_d;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&M_d, size);
cudaFree(M_d);
```

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer between host and device
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Cannot be used for GPU-GPU data transfer



Code Example

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and M_d is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(M_d, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, M_d, size, cudaMemcpyDeviceToHost);
```

Synchronous Memory Copy

- cudaMemcpy() is synchronous (or, blocking)
  - It won't begin until all previously issued CUDA calls have completed.
  - Subsequent CUDA calls cannot begin until the synchronous cudaMemcpy() has completed.

Vector Addition on GPU

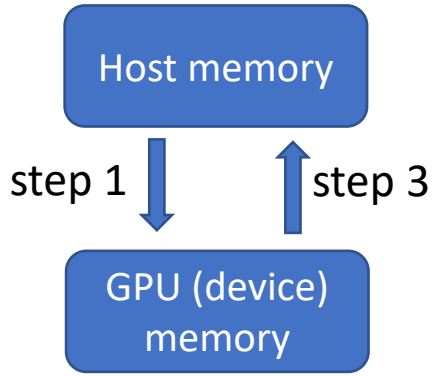
```
#include <cuda.h>

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

  //step 1: allocate device memory for A, B, and C; then copy A and B to device memory

  //step 2: launch the kernel code to perform the actual vector addition on GPU

  //step 3: copy C from device memory and free device memory
}
```



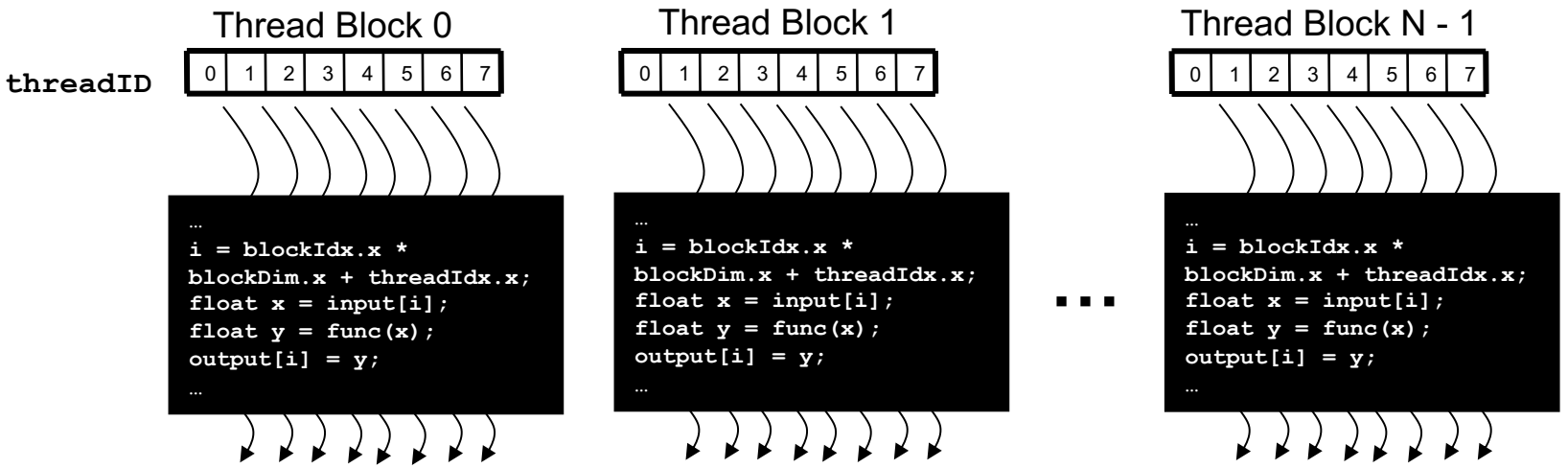
CUDA Kernel Functions

- A kernel function specifies the code to be executed by all threads during a parallel phase.
  - All threads execute the same code: SPMD

- When a host code launches a kernel, a grid of threads will be generated. These threads are organized in a two-level hierarchy:
  - Each grid is organized into many thread blocks
    - Each block has a unique ID within its grid: blockIdx
  - Each block contains many threads
    - Each thread has a unique ID within its block: threadIdx

CUDA Thread Blocks

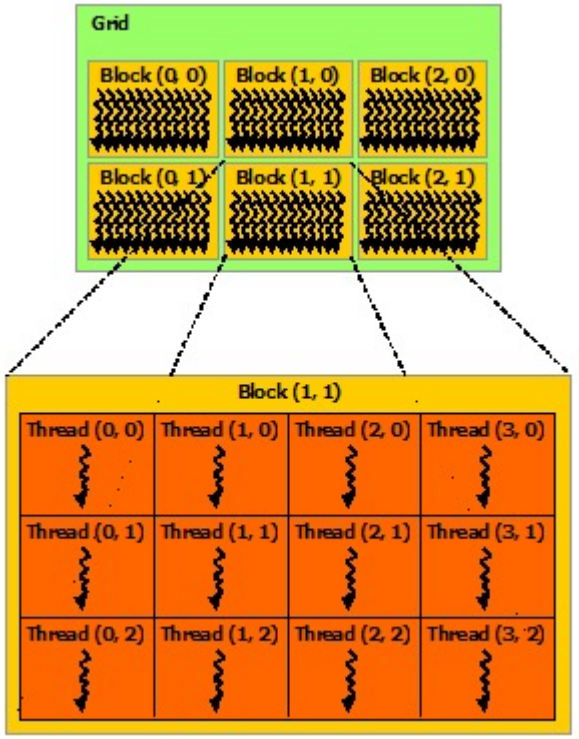
- Threads in a grid are organized as multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate



Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D, 2D, or 3D
  - Thread ID: 1D, 2D, or 3D
  - CUDA introduces dim3 as the data type for 3D vector
    - dim3 is actually a C struct with three unsigned integer fields x, y, and z
- Predefined built-in variables
  - dim3 gridDim: grid dimensions
  - dim3 blockDim: block dimensions
  - dim3 blockIdx: three-dimensional index of a block
  - dim3 threadIdx: three-dimensional index of a thread

Example of
2-D grid and 2-D block



CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- __global__ defines a kernel function
  - Must return void

- __device__ and __host__ can be used together
  - The compiler will generate two versions of the function

Calling a Kernel Function

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Kernel Launch is Asynchronous

- In CUDA, launching a kernel is asynchronous

- The CPU will continue executing the subsequence statements, without waiting for the kernel to finish
  - When the CPU encounters a synchronous CUDA API, such as cudaMemcpy(), it will wait for the previous kernel function to finish

  - A synchronous CUDA API, cudaDeviceSynchronize(), can be used to block the CPU until the GPU has completed all preceding requested tasks

cudaError_t cudaDeviceSynchronize(void);

CUDA Kernel Restrictions

- At present, CUDA kernel function has the following restrictions
  - Access to device memory only
  - Must have void return type
  - No support for a variable number of arguments
  - No support for static variables
  - No support for function pointers

GPU Example 1

- Allocate CPU memory for *n* integers
- Allocate GPU memory for *n* integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values

```
#include <cuda.h>
#include <stdio.h>
int main() {
        int dimx= 16;
        int num_bytes= dimx* sizeof(int);
        int*d_a= 0, *h_a= 0;   // device and host pointers
        h_a= (int*)malloc(num_bytes);
        cudaMalloc((void**)&d_a, num_bytes);
        if (0 == h_a|| 0 == d_a) {
                printf("couldn't allocate memory\n");
                return 1;
        }
        cudaMemset(d_a, 0, num_bytes);
        cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
        for (int i= 0; i< dimx; i++)
                printf("%d\n", h_a[i]);
        free(h_a);
        cudaFree(d_a);
        return 0;
}
```

GPU Example 2

- Use GPU to initialize an array by thread IDs
- Copy the array to CPU
- Print the values

A kernel function:

```
__global__  void  mykernel(int* a)
{
        int idx= blockIdx.x* blockDim.x+ threadIdx.x;
        a[idx] = 7;
}
```

```
#include <cuda.h>
#include <stdio.h>

__global__ void mykernel(int* a) {
        int idx= blockIdx.x* blockDim.x+ threadIdx.x;  // locate the data item handled by this thread
        a[idx] = threadIdx.x;
}

int main() {
        int dimx= 16, num_bytes= dimx*sizeof(int);
        int*d_a= 0, *h_a= 0; // device and host pointers
        h_a= (int*)malloc(num_bytes);
        cudaMalloc((void**)&d_a, num_bytes);
        cudaMemset(d_a, 0, num_bytes);
        dim3  grid, block;
        block.x= 4;                             // each block has 4 threads
        grid.x= dimx / block.x;                 // # of blocks is calculated
        mykernel<<<grid, block>>>(d_a);
        cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
        for(inti= 0; i< dimx; i++)
                printf("%d\n", h_a[i]);
        free(h_a);
        cudaFree(d_a);
        return 0;
}
```

The output will be:

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

GPU Example 3: Vector Addition

A kernel function which will be executed on GPU

```
// compute vector sum C = A + B
// each thread performs one pair-wise addition
__global__ void vecAdd( float *A, float *B, float *C, int n)
{
        // locate the memory
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        // perform the addition
        if(i < n) C[i] = A[i] + B[i];
}
```

Example: Vector Addition (Cont.)

Host code which will be executed on CPU

```
int main ()
{
        int n = 10000;
        // allocate and initialize host (CPU) memory
        float *H_A = …, *H_B = …, *H_C = …;
        // allocate device (GPU) memory
        float *A_d, *B_d, *C_d;
        cudaMalloc(…); …
        // copy host memory to device
        cudaMemcpy(…);…
        // run 16 blocks of 256 threads each
        vecAdd<<< ceil(n/256.0), 256 >>>(d_A, d_B, d_C, n);
        // copy result to host
        cudaMemcpy(…);
        cudaFree(A_d);…
}
```

Error Handling

- It is very common to use an error-handling macro to wrap all CUDA API calls
  - To simplify the error checking process

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

E.g.:

```
CHECK(cudaMemcpy(…));
```

__FILE__ and __LINE__ are standard predefined macros that represent the current input file and input line number, respectively.

Reading List

- **Chapter 2**, David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd Edition, Morgan Kaufmann, 2013. [PDF: https://safari.ethz.ch/architecture/fall2019/lib/exe/fetch.php?media=2013_programming_massively_parallel_processors_a_hands-on_approach_2nd.pdf]

References

1. David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd Edition, Morgan Kaufmann, 2013.

2. CUDA C/C++ Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/