Advanced Topics in Numerical Analysis: High Performance Computing

Intro to GPGPU

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

March 25, 2019

Outline

Organization

Scheduling:

- Homework assignment #3 due next Monday

Topics today:

- Quick review
- GPU architecture
- GPU programming models
- Compiling and running CUDA code

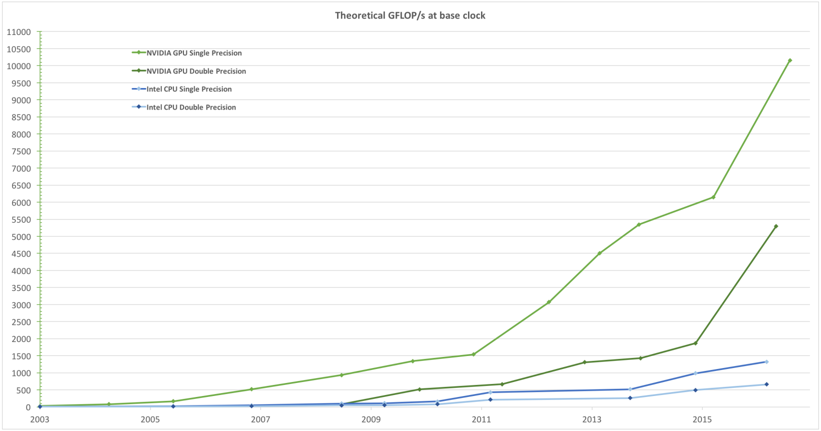
Outline

Summary of Previous Classes

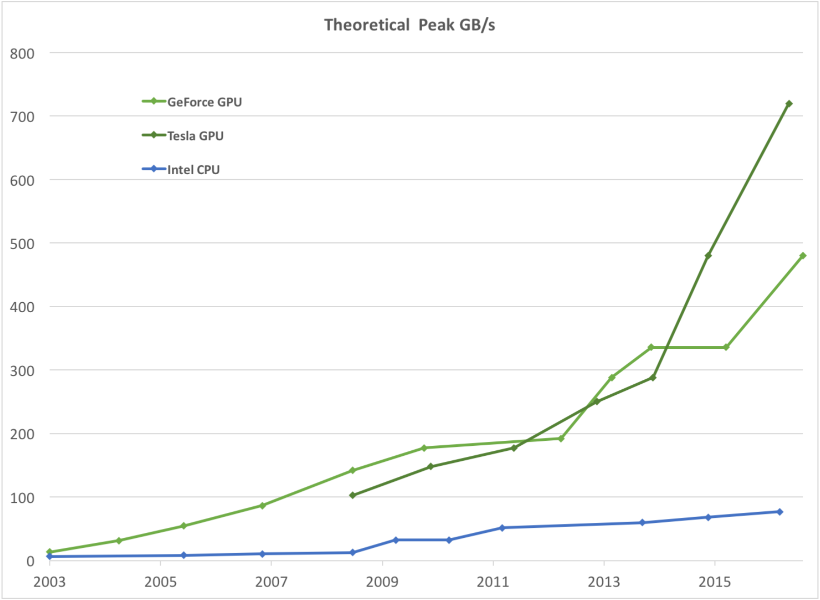
- **Memory Hierarchy:** Minimize latency and maximize bandwidth by keeping frequently used data close to CPU in caches.
- **Instruction Level Parallelism:** Pipelining, Superscalar architecture, Out-of-order execution, Branch prediction
- **Vectorization:** Auto-vectorization (compiler hints, loop unrolling), Explicit (OpenMP SIMD, assembly, intrinsics).
- **Shared Memory Parallelism:** OpenMP, threads, fork-join model, synchronization, atomic-operations, NUMA
- **Parallel Scalability:** Strong and weak scaling, Amdahl's Law (speedup $< (s + (1-s)/p))^{-1}$
- **Libraries:** BLAS, LAPACK, FFTW
- **Tools:** Valgrind, Makefile, Git

Outline

GPUs vs CPUs (FLOP/s)



GPUs vs CPUs (Bandwidth)



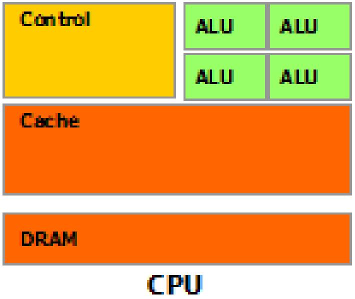
GPUs vs CPUs

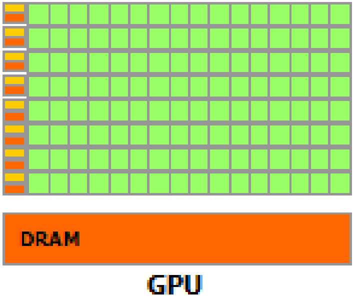
CPUs

- Optimized for sequential performance
- Out-or-order execution, branch-prediction

GPUs

- Optimized for floating-point throughput
- No out-or-order, no branch-prediction





Computing on GPUs

Many different programming models:

NVIDIA

- C for CUDA (Compute Unified Device Architecture)
  - extension of C/C++

Open standards:

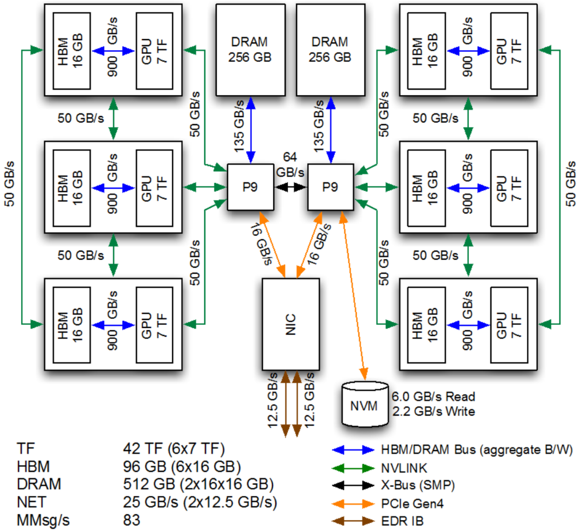
- OpenCL: similar to CUDA in programming style, supports other GPUs (AMD) as well as CPUs, other accelerators (Xeon Phi, FPGAs etc).
- OpenACC: similar to OpenMP in programming style, uses pragmas to offload computations to GPU.

Summit Architecture

CPU (host) connected to one or more GPUs (devices).

Host executes regular C/C++ code, launches CUDA kernels on device, allocates memory on device, and initiates memory transfers between host and device.

Device executes CUDA kernels.



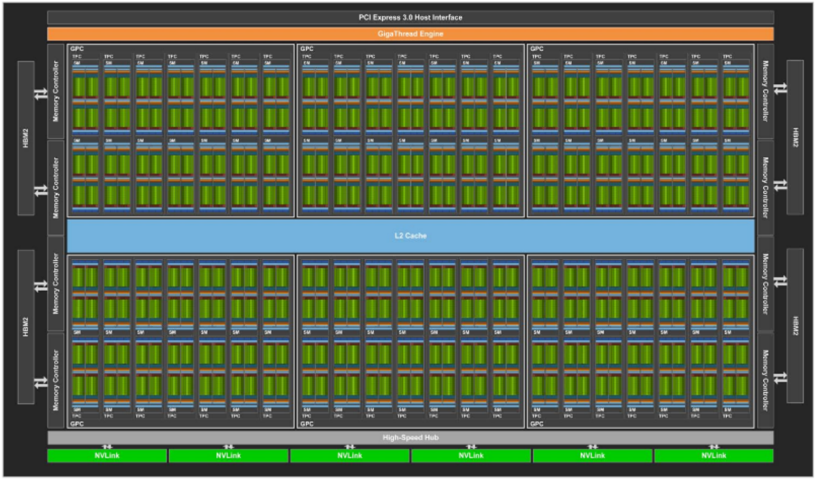
TF	42 TF (6x7 TF)
HBM	96 GB (6x16 GB)
DRAM	512 GB (2x16x16 GB)
NET	25 GB/s (2x12.5 GB/s)
MMsg/s	83

HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) are bi-directional.

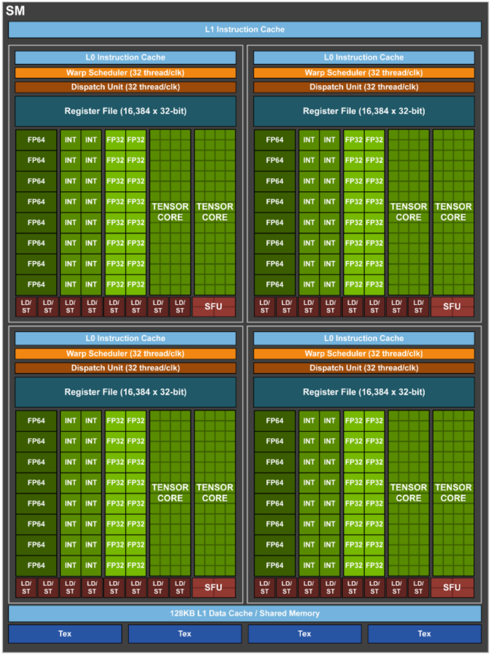
Compute Unified Device Architecture

NVIDIA Volta GV100,

- 6144 KB of L2 cache
- 84 **streaming multiprocessors (SMs)**



Compute Unified Device Architecture



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
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Each SM has

- 4 warp (32-threads) schedulers
- 32 scalar DP cores
- 64 scalar SP cores
- 64 scalar integer cores
- 8 tensor cores (half-precision)
- 128KB L1 cache / shared memory (configurable)

Single Instruction Multiple Threads (SIMT): SIMD + hyper/multi-threading.

- $O(10^3)$ threads / SM
- $O(10^5)$ threads overall

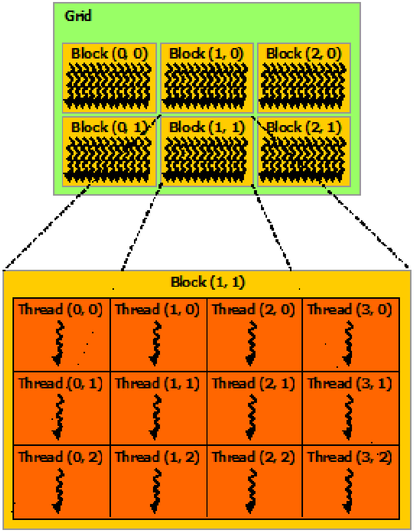
Thread Hierarchy

Grid: collection of blocks.

Block: collection of threads. Scheduled on a single SM. Threads can share data using shared mem.

Warp: group of 32-threads executing together on SM.

Special variables
`gridDim`, `blockDim`, `blockIdx`, `threadIdx`, `warpSize`



Calculate global thread index:

```
idx = threadIdx.x + blockDim.x * blockIdx.x
```

Demo: Hello World

GPU code (kernel function):

```
__global__ // tell compiler this runs on GPU
void print_hello() {
  printf("hello from thread %d of block %d\n",
                              threadIdx.x, blockIdx.x);
}
```

Launching the kernel (from CPU)

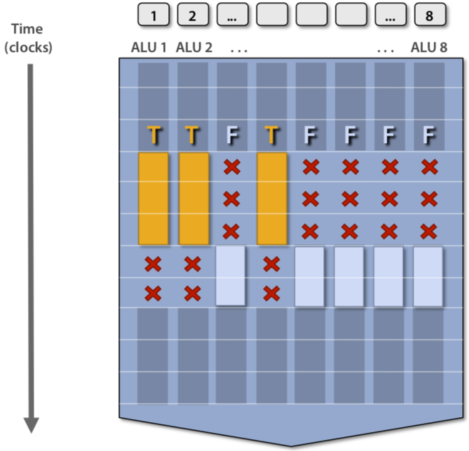
```
dim3 GridDim( nx, ny, nz ); // can be 1D, 2D or 3D
dim3 BlockDim( mx, my, mz ); // can be 1D, 2D or 3D
print_hello<<<GridDim, BlockDim>>>(); // launch kernel
cudaDeviceSynchronize(); // wait for kernel to finish
```

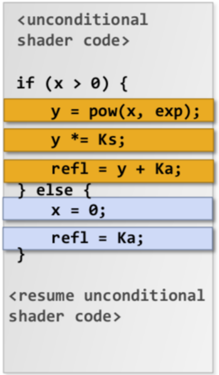
Compiling with NVCC (NVIDIA C Compiler)

```
nvcc hello-world.cu
```

nvcc compiles device functions, host code processed by gcc/g++.

Branches





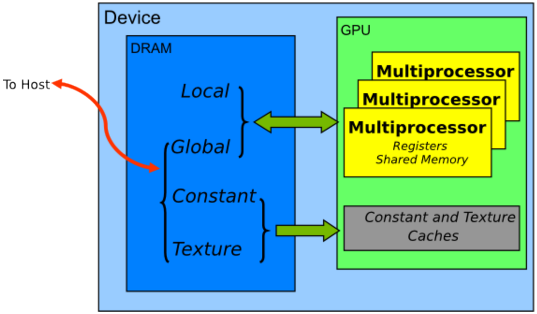
```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

Avoid diverging threads within a warp!

Device Memory Spaces

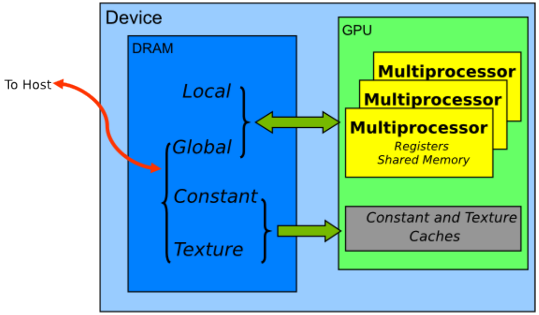
Registers: Thread scope local variables.
Number of 32-bit reg = 64K/SM,   255/thread



```
__global__ void mykernel() {
    int a = 0;
    double x = 1.5;
}
```

Device Memory Spaces

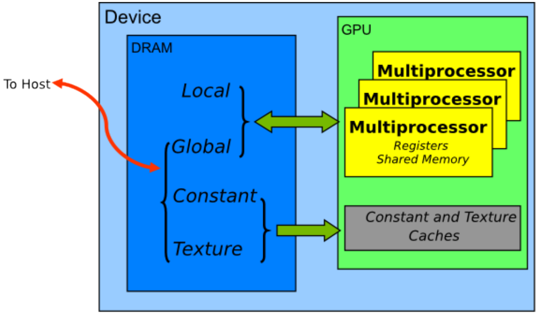
Local memory: Thread scope arrays and spilled registers. Resides in GPU main memory.



```
__global__ void mykernel() {
    int a[100];
    double x[100];
}
```

Device Memory Spaces

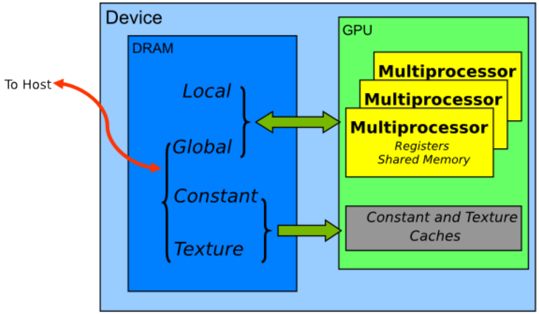
Shared memory:
Variables declared using `__shared__` keyword.
Accessible by all thread in the same thread-block.
size = 96KB/SM



```
__global__ void mykernel() {
    __shared__ double x[1024];
    x[threadIdx.x] = 1.5;
}
```

Device Memory Spaces

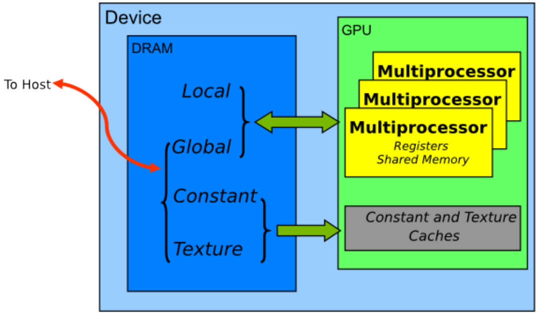
Global memory:
Dynamically allocated GPU main-memory. Managed (allocate, transfer data, free) from CPU. Cached in L2 cache. size = size-of-DRAM



```
int main() {
    double* A;
    cudaMalloc(&A, 512*sizeof(double));
    mykernel<< <15,1028> >>(A);
}
```

Device Memory Spaces

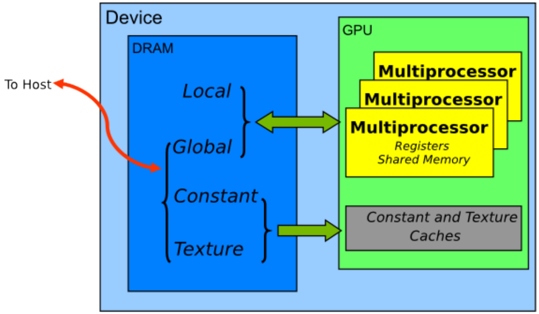
Constant memory:
Global scope, read-only (from GPU) memory declared using `__constant__` keyword.
size = 64KB
cache = 8KB/SM



```
__constant__ double A[512];

__global__ void mykernel(){
    double x = A[threadIdx.x];
}
```

Device Memory Spaces

Texture memory:
Read-only from GPU, cached.



Resource Limits

Limits for compute capability 7.0 (Volta architecture)

Threads per Warp	32
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)
Total amount of constant memory	64KB
Total amount of shared memory per block	48KB
Max threads per block	1024
Warps per SM	64
Threads per SM	2048
Thread Blocks per SM	32
# of 32-bit registers per SM	65536
Shared Memory per SM	96KB

More information:

- compute capabilities (features and limits)
- CUDA Samples:
  `$CUDA_HOME/samples/1_Utilities/deviceQuery`

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
    for (long idx = 0; idx < N; idx++)
        xpy[idx] = x[idx] + y[idx];
}
```

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
    for (long idx = 0; idx < N; idx++)
        xpy[idx] = x[idx] + y[idx];
}
```

GPU version:

```
__global__
void vadd_gpu(double* xpy, double* x, double* y, long N){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) xpy[idx] = x[idx] + y[idx];
}
```

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
    for (long idx = 0; idx < N; idx++)
        xpy[idx] = x[idx] + y[idx];
}
```

GPU version:

```
__global__
void vadd_gpu(double* xpy, double* x, double* y, long N){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) xpy[idx] = x[idx] + y[idx];
}
```

Launching the kernel (from CPU):

```
vadd_gpu<<<GridDim, BlockDim>>>(xpy, x, y, N);
```

Demo: Vector Addition

CPU version:

```
void vadd(double* xpy, double* x, double* y, long N){
  #pragma omp parallel for
    for (long idx = 0; idx < N; idx++)
        xpy[idx] = x[idx] + y[idx];
}
```

GPU version:

```
__global__
void vadd_gpu(double* xpy, double* x, double* y, long N){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) xpy[idx] = x[idx] + y[idx];
}
```

Launching the kernel (from CPU):

```
vadd_gpu<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(xpy, x, y, N);
```

Global Memory Management

CUDA API for Managing Memory:

Memory allocation on device

```
cudaError_t cudaMalloc(void** A, size_t size);
```

Free memory on device

```
cudaError_t cudaFree(void* A);
```

Global Memory Management

CUDA API for Managing Memory:

Memory allocation on device

```
cudaError_t cudaMalloc(void** A, size_t size);
```

Free memory on device

```
cudaError_t cudaFree(void* A);
```

Memory transfer between device and host

```
cudaError_t cudaMemcpy(void* A_d, const void* A,
                size_t N, cudaMemcpyKind kind);
```

kind = `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, `cudaMemcpyDefault`

Global Memory Management (Asynchronous)

Page-locked memory: memory pages on host have fixed mapping (pages cannot be moved). Allows direct memory access (DMA) of host memory from GPU (without CPU involvement).

Allocate and free memory on host (replace malloc, free)

```
cudaError_t cudaMallocHost(void** A, size_t size);
cudaError_t cudaFreeHost(void* A);
```

Global Memory Management (Asynchronous)

Page-locked memory: memory pages on host have fixed mapping (pages cannot be moved). Allows direct memory access (DMA) of host memory from GPU (without CPU involvement).

Allocate and free memory on host (replace malloc, free)

```
cudaError_t cudaMallocHost(void** A, size_t size);
cudaError_t cudaFreeHost(void* A);
```

Asynchronous memory transfers

```
cudaError_t cudaMemcpyAsync(void* A_d, const void* A,
                            size_t N, cudaMemcpyKind kind);
```

Global Memory Management (Asynchronous)

Page-locked memory: memory pages on host have fixed mapping (pages cannot be moved). Allows direct memory access (DMA) of host memory from GPU (without CPU involvement).

Allocate and free memory on host (replace malloc, free)

```
cudaError_t cudaMallocHost(void** A, size_t size);
cudaError_t cudaFreeHost(void* A);
```

Asynchronous memory transfers

```
cudaError_t cudaMemcpyAsync(void* A_d, const void* A,
                size_t N, cudaMemcpyKind kind);
```

Wait / Synchronize

```
cudaError_t cudaDeviceSynchronize();
```

Overlap tasks on CPU and GPU and synchronize at the end!

Global Memory Management (Managed)

Allocate memory on both host and device (accessed using the same pointer). Let CUDA driver manage synchronizing memory between host and device.

Allocate memory on host and device
```
cudaError_t cudaMallocManaged(void** A, size_t size);
```

Free memory on host and device
```
cudaError_t cudaFree(void* A);
```

Global Memory Management (Managed)

Allocate memory on both host and device (accessed using the same pointer). Let CUDA driver manage synchronizing memory between host and device.

Allocate memory on host and device

```
cudaError_t cudaMallocManaged(void** A, size_t size);
```

Free memory on host and device

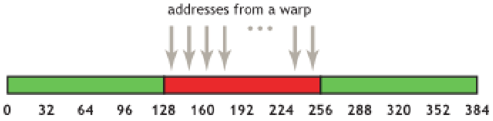
```
cudaError_t cudaFree(void* A);
```

Wait / Synchronize before accessing memory on host

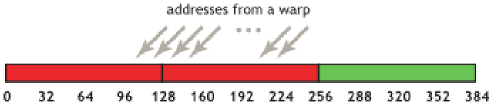
```
cudaError_t cudaDeviceSynchronize();
```

Optimizing Global Memory Accesses

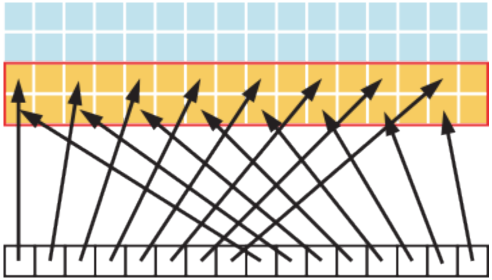
Use aligned, regular memory accesses from each warp



Avoid unaligned access



Avoid strided access



Checking Errors

All CUDA API calls return an error code (`cudaError_t`)

- Error in the API call itself
- Error in an earlier asynchronous operation (e.g. kernel)

Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Summary

CUDA architecture

- Streaming Multiprocessor, CUDA cores, $\mathrm{O}(10^5)$ threads
- Thread hierarchy: grid, block, warp
- Memory hierarchy: registers, local memory, global memory
  - shared memory (more in next class)

CUDA language extensions to C/C++

- **function attributes:** `__global__`, `__device__`, `__host__`
- **variable attributes:** `__shared__`, `__constant__`
- **special variables:** `gridDim`, `blockDim`, `blockIdx`, `threadIdx`, etc.

CUDA API Functions

- Memory Management: `cudaMalloc`, `cudaFree`, `cudaMemcpy`, etc.
  - synchronous vs asynchronous memory transfers (requires page-locked host memory)
  - managed vs un-managed memory

References

NVIDIA Programming Guide

- Programming Guide
- Runtime API
- Best practices

Quick References

- CUDA cheatsheet
- CUDA syntax