CS 623: Introduction to CUDA Programming

Programming NVIDIA GPUs

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

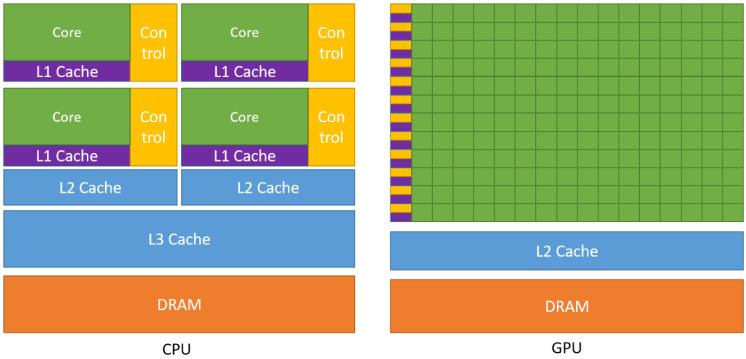
Sem 2026-27-I

GPU Programming Philosophy

Massively parallel The computation can be broken down into hundreds or thousands of independent units of work

Compute intensive The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory

SIMT A warp of threads execute the same instruction



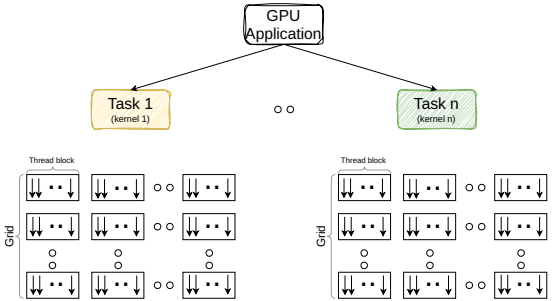
CUDA Programming Model

Allows fine-grained data parallelism and thread parallelism nested within coarse-grained data parallelism and task parallelism

(i) Partition the problem into coarse sub-problems that can be solved independently

(ii) Assign each sub-problem to a “block” of threads to be solved in parallel

(iii) Each sub-problem is also decomposed into finer work items that are solved in parallel by all threads within the block



Hello World with CUDA

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

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

int main() {
  hwkernel<<<1, 1>>>();
}
```

```
$ nvcc hello-world.cu
$ ./a.out

$
```

hello-world.cu

Makefile

Hello World with CUDA

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

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

int main() {
  hwkernel<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

```
$ nvcc hello-world.cu
$ ./a.out
Hello world!

$
```

- CPU thread returns immediately after launching the kernel
- Use `cudaDeviceSynchronize()` (or its variants) to block the caller CPU thread

Function Declarations

	Executed on	Callable from
`__device__ float deviceFunc()`	Device	Device
`__global__ void kernelFunc()`	Device	Host[§]
`__host__ float hostFunc()`	Host	Host

- `__global__` defines a kernel function, must return `void`
- `__device__` functions can have return values
- `__host__` is default, and can be omitted
- Prepending `__host__ __device__` causes the system to compile separate host and device versions of the function

[§]A kernel function can also be called from the device if dynamic parallelism is enabled

Classical Copy-Then-Execute Model

(i) Prepare input data in CPU memory
(ii) Copy data from CPU to GPU memory (e.g., `cudaMemcpy( )`)
(iii) Launch GPU kernel and execute
(iv) Copy results from GPU memory to CPU memory (e.g., `cudaMemcpy( )`)
(v) Use the results on the CPU
(vi) Repeat the above steps based on the application logic

Kernels

- Kernels are special functions that a CPU can call to execute on the GPU
  - Executed N times in parallel by N different CUDA threads
  - Cannot return a value
- Each thread has a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable

```
__global__ void VecAdd(float* A, float* B, float* C) {
  int i = threadIdx.x;
  ...
}
int main() {
  auto *h_A = new float[N];
  ...
  // Kernel invocation with 1 thread block and N threads per block
  VecAdd<<<1, N>>>(d_A, d_B, d_C);
}
```

vector-addition.cu

Kernels

```
KernelName<<<m, n>>>(arg1, arg2, ...)
```

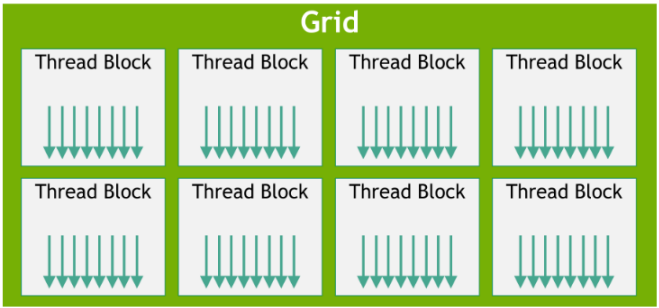
GPU spawns `m` blocks with `n` threads that run a copy of the same function

- CPU can continue processing while GPU runs the kernel
- Kernel call returns when all threads have terminated

```
kernel1<<<X,Y>>>(...);
// kernel starts execution, CPU continues to next statement
kernel2<<<X,Y>>>(...);
// kernel2 placed in queue, will start after kernel1 finishes,
// CPU continues
cudaMemcpy(...);
// CPU blocks until memory is copied, memory copy starts after all
// preceding CUDA calls finish
```

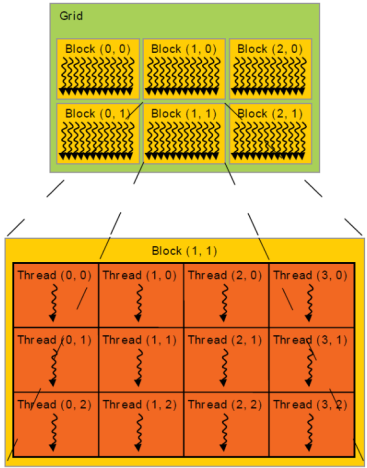
Thread Hierarchy

- A kernel executes in parallel across a set of parallel threads
  - Threads that are generated by a kernel launch are collectively called a **grid**
- Threads are organized in **thread blocks** (aka cooperative thread arrays)
  - A thread block is a set of concurrently executing threads that can communicate among themselves through shared memory and barrier synchronization
- A grid is an array of thread blocks that execute the same kernel



Thread Hierarchy

- Threads in a block reside on the same SM, max 1024 threads in a block
- Thread blocks are organized into 1D, 2D, or 3D grids
  - Grid dimension is given by the `gridDim` variable
- Identify block within a grid with the `blockIdx` variable
- Block dimension is given by the `blockDim` variable



Dimension and Index Variables

Type is `dim3`

Dimension

- `gridDim` specifies the number of blocks in the grid
- `blockDim` specifies the number of threads in each block

Index

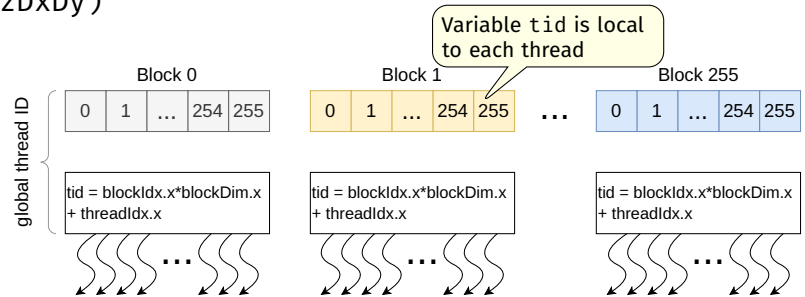
- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

- The keywords are 3-component vectors
  - For example, `threadIdx.[x,y,z]` gives the index of the thread in a block, in the particular direction

Finding Thread IDs

How to find out the linearized thread IDs within a block?

- 1D block: thread ID is `threadIdx.x`
- 2D block of size `(Dx, Dy)`: thread ID of a thread of index `(x, y)` is `(x + yDx)`
- 3D block of size `(Dx, Dy, Dz)`: thread ID of a thread of index `(x, y, z)` is `(x + yDx + zDxDy)`

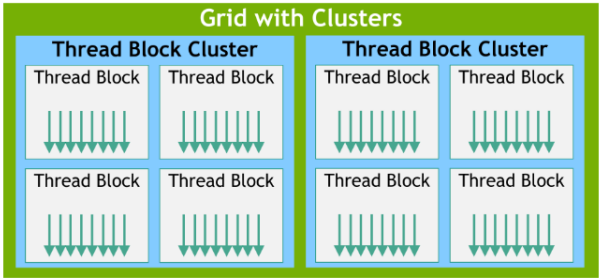


thread-id.cu

Thread Block Clusters

CC 9.0 introduced an **optional** level in the thread hierarchy called Thread Block Clusters

- Thread blocks in a cluster are always co-scheduled on a GPC
  - Thread blocks in a cluster can access the distributed Shared Memory
  - Allows thread blocks in the cluster to synchronize with hardware support
- The number of thread blocks in a cluster can be specified with the compile-time kernel attribute `__cluster_dims__(X,Y,Z)`
- A maximum of 8 thread blocks is guaranteed to be portable



Execution Configuration Uses Integer Arithmetic

- Dimension variables are vectors of integral type
- Assume data is of length `N`, and say the kernel execution configuration is `<<<N/TPB, TPB>>>`
  - Each block has `TPB` threads and there are `N/TPB` blocks

Suppose `N = 64` and `TPB = 32`

Implies there are 2 blocks of 32 threads

Suppose `N = 65` and `TPB = 32`

Implies there are 2 blocks of 32 threads

- Ensure that the grid covers the array length by rounding up the number of blocks from `N/TPB` to `(N+TPB-1)/TPB`
- Use a control statement in the kernel to ensure that the thread index is within the maximum array index

Launching Kernels for 2D Data

```
__global__ void MatAdd(float* A, float* B, float* C) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N && j < N) {
    C[i * N + j] += A[i * N + j] + B[i * N + j];
  }
}
int main() {
  ...
  dim3 threadsPerBlock(32, 32);
  dim3 blocksPerGrid((N + 32 - 1) / 32, (N + 32 - 1) / 32);
  MatAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);
  ...
}
```

matadd-v1.cu

Choosing Optimal Execution Configuration

- The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system
  - It is okay to have a greater number of threads
- Choose number of threads in a block to be a multiple of 32
- No fixed rule, needs exploration and experimentation to find the best grid configuration

matadd-vo.cu

Reporting Errors

- All CUDA API calls return an error code of type `cudaError_t`
  - Error in the API call itself or error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error with `cudaGetLastError()`
- Get a string to describe the error with `char *cudaGetErrorString(cudaError_t)`

What is the canonical way to check for errors using the CUDA runtime API?

Device Management

Query device information

- The application can query and select GPUs
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- Multiple host threads can share a device
- A single host thread can manage multiple devices
  - `cudaSetDevice(i)` to select current device
  - `cudaMemcpy(...)` for peer-to-peer copies

Timing a CUDA Kernel

```
float memsettime;
cudaEvent_t start, stop;
// initialize CUDA timers
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);
// CUDA Kernel
...
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&memsettime,start,stop); // in milliseconds
cout << "Kernel execution time: " << memsettime << "\n";
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

How to Implement Performance Metrics in CUDA C/C++

Warp Scheduling

Understanding SIMT

- In vectorization, users program the SIMD hardware directly with vector ISA, or use auto-vectorization or intrinsics
  - We program with the vector width in mind, excepting auto-vectorization
- GPUs employ SIMD hardware to exploit data-level parallelism
- SIMT can be thought of as “SIMD with multithreading”
  - Software analog compared to the hardware perspective of SIMD
  - For example, we rarely need to know the number of cores with CUDA
- Multiple types of parallelism
  - Independent grids (i.e., kernels) or concurrent thread blocks represent coarse-grained data parallelism or task parallelism
  - Concurrent threads/warps represent fine-grained data parallelism or thread parallelism

Mapping Blocks and Threads

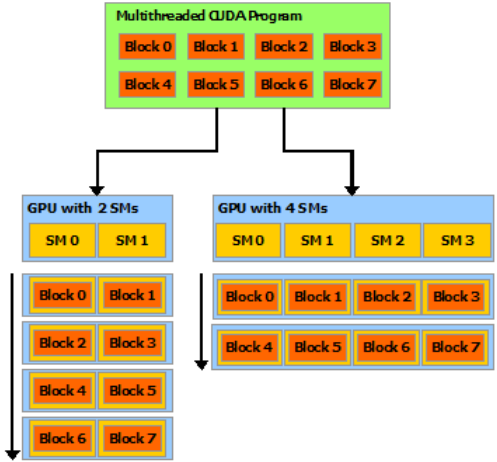
- A GPU executes one or more kernel grids
- A kernel is partitioned into thread blocks that execute independently of each other
- When a CUDA kernel is launched, the thread blocks are distributed to SMs in any order
  - Multiple thread blocks, up to a limit, can execute concurrently on one SM
    - Not all blocks may be resident at the same time
  - For example, a CUDA device may allow up to eight blocks to be assigned to each SM
  - CUDA cores in the SM execute threads of a block
- A block begins execution only when it has secured all execution resources necessary for all the threads
- As thread blocks terminate, new blocks are launched on the vacated SMs

CUDA runtime can execute blocks in any order

- Blocks are mostly **not** supposed to synchronize with each other
  - Allows for simple hardware support for data parallelism

Block Scalability

- A kernel with enough blocks scales across GPUs
- A GPU with more SMs will automatically execute the program faster than a GPU with fewer SMs



Comparing the performance of vector addition kernel

GPU	CC	# SMs	Time (ms)
GeForce GTX 1080	6.1	20	72.5
Quadro RTX 5000	7.5	48	8.6
A40	8.6	84	5.6

vector-addition.cu

Thread Warps

- Conceptually, threads in a block can also execute in any order
- However, sharing a control unit reduces hardware complexity, cost, and power consumption
- A set (currently 32) of consecutive threads that execute in SIMD fashion is called a **warp**
  - Called wavefront (with 64 threads) on AMD GPUs
- Warps are scheduling units in an SM
  - Implementation detail, not part of the programming model

Scheduling Thread Warps

- Each SM launches warps of threads, and executes warps on a time-sharing basis
- SM schedules and executes warps that are ready to run
  - Warps run for fixed-length time slices like processes
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - If more than one warp is ready for execution, a greedy-then-oldest priority mechanism is used to select one for execution
    - Suppose an instruction to be executed by a warp has to wait for the result of a previously initiated long-latency operation (e.g., `ld.global`)
    - The warp is not selected for execution, another warp that is not waiting for results is selected for execution
  - Selection of ready warps for execution does not introduce any idle time into the execution timeline (zero-overhead scheduling)

Scheduling Thread Warps

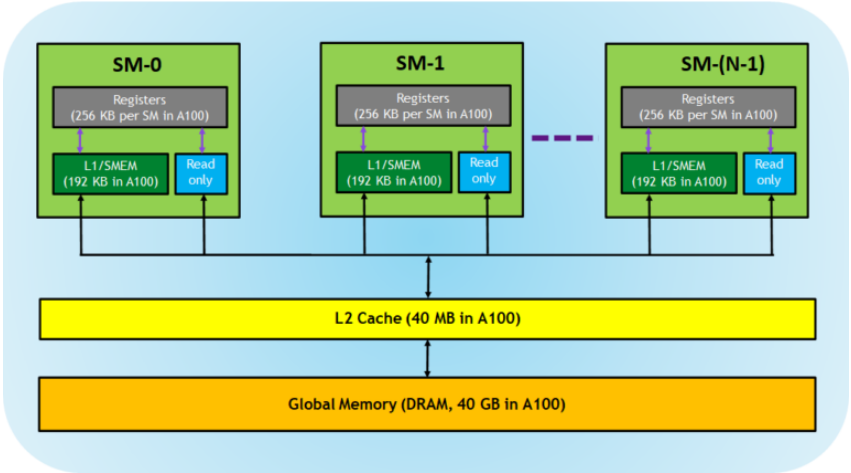
- Goal is to have enough warps so that the GPU scheduler is likely to always find a warp ready to execute
  - Hides latency of long memory operations with work from other threads (called latency tolerance or hiding)
- Thread blocks execute on an SM, thread instructions execute on a core
  - CUDA virtualizes the physical hardware
  - Thread is a virtualized scalar processor (registers, PC, state)
  - Block is a virtualized multiprocessor (threads, shared memory)
- As warps and thread blocks complete, resources are freed

Warp Scheduling

- For an ideal memory system, increasing the number of warps can increase the throughput
  - If the number of warps in a core multiplied by the issue time of each warp exceeds the memory latency, the execution units in the core will always remain busy
  - Round-robin scheduling of warps would suffice
  - However, locality property either favors or discourages round-robin scheduling
- Increasing the number of warps is impractical because of the need to maintain execution state in hardware (i.e., to achieve zero-overhead scheduling)
  - Number of threads that can be simultaneously tracked and scheduled in hardware is bounded
  - Requires resources for an SM to maintain execution status of threads
- Up to 2048 threads can be assigned to each SM on recent CUDA devices
  - For example, 8 blocks of 256 threads, or 4 blocks of 512 threads
- Assume a CUDA device with 28 SMs
  - Each SM can accommodate up to 2048 threads
  - The device can have up to 57344 threads simultaneously **residing** in the device for execution

Memory Hierarchy

Memory Hierarchy in NVIDIA A100



CUDA Refresher: The CUDA Programming Model

Memory Access Efficiency

Compute-to-global-memory access ratio is the number of floating-point operations performed for each access to global memory

```
for (int i = 0; i < N; i++)
  tmp += (A[i*N+K]*B[k*N+j]);
```

Assume a GPU device with 1 TB/s global memory bandwidth and peak single-precision performance of 12 TFLOPS

- What is the performance we expect with an access ratio of 0.5?
- We can do 1000/(4+4) GFLOPS, which is only $\sim$ 1% of the peak performance

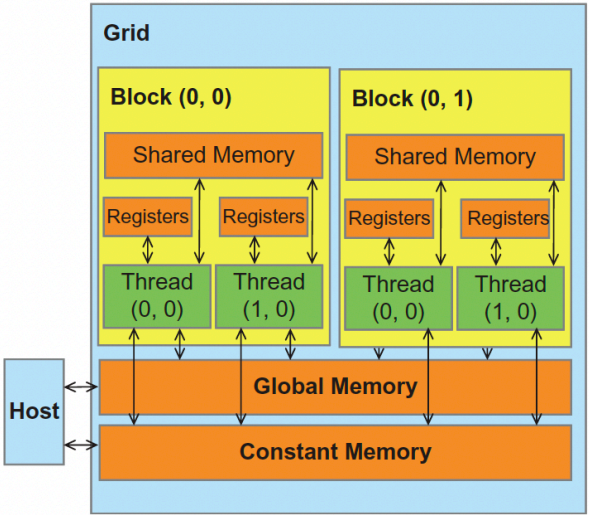
Memory Hierarchy in CUDA

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



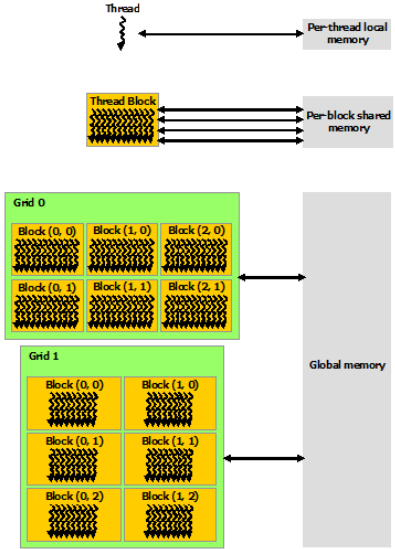
Variable Type Qualifiers in CUDA

	Memory	Scope	Lifetime
`int localVar`	Register	Thread	Kernel
`__device__ __local__ int localVar`	Local	Thread	Kernel
`__device__ __shared__ int sharedVar`	Shared	Block	Kernel
`__device__ int globalVar`	Global	Grid	Application
`__device__ __constant__ int constVar`	Constant	Grid	Application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Pointers can only point to memory allocated or declared in global memory

Memory Organization

- Host and device maintain their own separate memory spaces
  - A variable in CPU memory may not be accessed directly in a GPU kernel
- It is the programmer's responsibility to keep them in sync
  - A programmer needs to maintain copies of variables



Registers

- 64K 32-bit registers (or 256 KB register file) per SM
  - A CPU in contrast has a few (1–2 KB) per core
- Up to 255 registers per thread (compute capability 3.5+)
- If a code uses the maximum number of registers per thread (255) and an SM has 64K registers, then the SM can support a maximum of 256 threads
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread

What if each thread uses 33 registers?

Shared Memory

- Each SM contains a single shared memory structure that acts as a software-managed cache
- Shared memory (also called scratchpad memory) is used for efficient communication among threads in a block
  - Usually 16–100 KB of storage that can be accessed efficiently by all threads in a block
  - The space is shared among all blocks running on that SM

Variable in shared memory is allocated using the `__shared__` specifier

- Latency is comparable to accessing registers

Say an SM with 4 thread blocks has 16 KB of shared memory

```
__shared__ float min[256];
__shared__ float max[256];
__shared__ float avg[256];
__shared__ float stdev[256];
```

Several things like grid configuration, number of registers, and amount of shared memory per block limits occupancy

Global Memory

- Variable `lock` can be accessed by both kernels
  - Resides in global memory space
  - Can be both read and modified by all threads

```
__device__ int lock=0;
__global__ void kernel1(...) {
  // Kernel code
}
__global__ void kernel2(...) {
  // Kernel code
}
```

- On-device memory accessed via 32, 64, or 128 B transactions
- A warp executes an instruction that accesses global memory
  - The addresses are coalesced into transactions
  - Number of transactions depend on the access size and distribution of memory addresses
  - More transactions mean less throughput
    - For example, if 32 B transaction is needed for a thread's 4 B access, throughput is essentially 1/8th

Constant Memory

- Used for data that will not change during kernel execution
  - Accessible from all threads within a grid
  - Constant memory is 64 KB

```
// Global scope
__constant__ float d_filter[FILTER_WIDTH];
// Initialize array in constant memory
cudaMemcpyToSymbol(d_filter, h_filter, FILTER_WIDTH * sizeof(float));
```

- Constant memory is cached
  - Each SM has a read-only constant cache that is shared by all cores in the SM
  - Speeds up reads from the constant memory space which resides in GPU global memory
  - Read from constant memory will incurs a global memory latency on the first miss
  - Subsequent reads will hit in the constant cache, which is almost as fast as registers

Local Memory

- Local memory is off-chip per-thread memory
  - More like thread-local global memory, so it requires memory transactions and consumes bandwidth
- Automatic variables are placed in local memory
  - (i) Arrays when it is not known whether indices are constant quantities
  - (ii) Large structures or arrays that consume too much register space
  - (iii) In case of register spilling
- Inspect PTX assembly code (compile with –`ptx`)
  - Check for `ld.local` and `st.local` mnemonic

Device Memory Management

- Global device memory can be allocated with `cudaMalloc()`
- Freed by `cudaFree()`
- Data transfer between host and device is with `cudaMemcpy()`
- Initialize memory with `cudaMemset()`
- There are asynchronous versions (e.g., `cudaMemcpyAsync()`)

GPU Caches

- GPUs have L1 and L2 data caches on devices with CC 2.x and higher
  - Texture and constant cache are available on all devices
- Accesses to global memory accesses are cached
  - Use compile time option `-dlcm` to control the cache location
    - Default is to cache in both L1 and L2 (`-Xptxas -dlcm=ca`)
    - Cache only in L2 with `-Xptxas -dlcm=cg`
- L1 cache is write-through, and per SM
  - Shared memory is partitioned out of unified data cache and its size can be configured, remaining portion is the L1 cache
- L2 cache is shared by all SMs
- L1 cache lines are 128 B wide in Fermi onward, while L2 lines are 32 B
- L1 cache in GPUs is **not** coherent, L2 cache is coherent

More on GPU Caches

- GPU caches are sectored, where each sector is 32 B
- There is one tag for the entire cache line, but each sector has its own status bits (e.g., valid and dirty)
- If only one 32 B sector is modified, only that sector is written back
- On a cache miss, not all 4 sectors in the cache line have to be filled, just the ones that need updating

References


D. Kirk and W. M. Hwu. Programming Massively Parallel Processors. Chapters 1–5, 13, 20, 3rd edition, Morgan Kaufmann.


NVIDIA Corporation. CUDA C++ Programming Guide, Release 12.9.2.


NVIDIA Corporation. CUDA C++ Best Practices Guide, Release 12.9.


NVIDIA Corporation. CUDA Runtime API: API Reference Manual, Release 12.9.