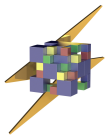
Lecture 20

GPU Memory Hierarchy

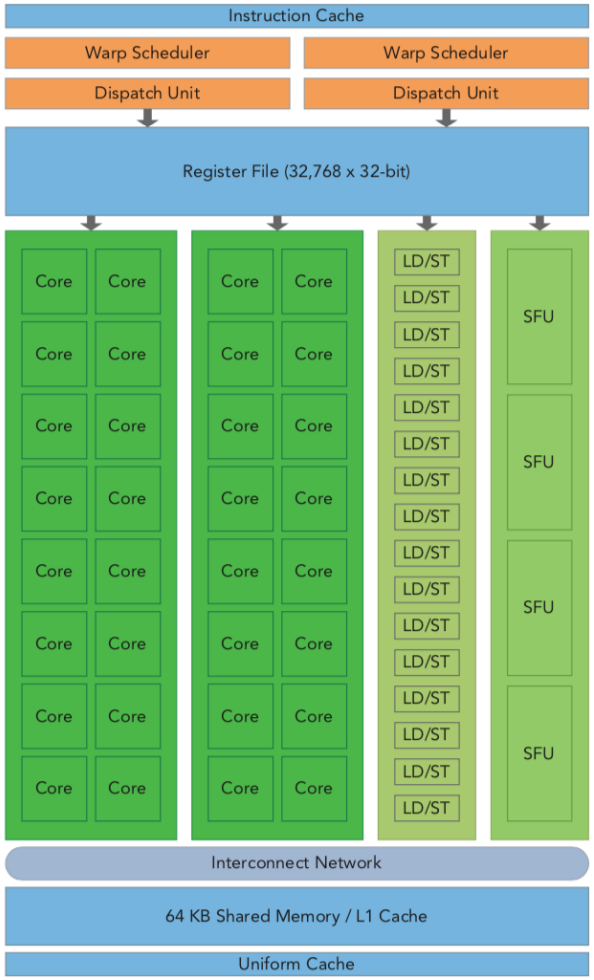
이재진
서울대학교 데이터사이언스대학원
서울대학교 공과대학 컴퓨터공학부
http://aces.snu.ac.kr/~jlee



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실

An SM in NVIDIA Fermi Architecture

- Each SM supports concurrent execution of hundreds of threads
- Multiple SMs per GPU
  - Thousands of threads executing concurrently on a single GPU
- Multiple thread blocks may be assigned to the same SM at once
  - Scheduled based on the availability of SM resources



SIMT

- NVIDIA terminology
- Single Instruction Multiple Thread (SIMT) architecture
  - To manage and execute threads in groups of 32 called warps
- All threads in a warp execute the same instruction at the same time
- Each SM partitions the assigned thread blocks into warps
  - The SM schedules the warps for execution on available hardware resources
  - All threads in a thread block run logically in parallel
    - Not all threads can execute physically at the same time
    - Different threads in a thread block may make progress at a different pace

Thread Block Scheduling

- A thread block is scheduled on only one SM
  - The block remains there until execution completes
- An SM can hold more than one thread block at the same time
- Shared memory is partitioned among thread blocks resident on the SM
- Registers are partitioned among threads

Thread Block Scheduling (cont'd)

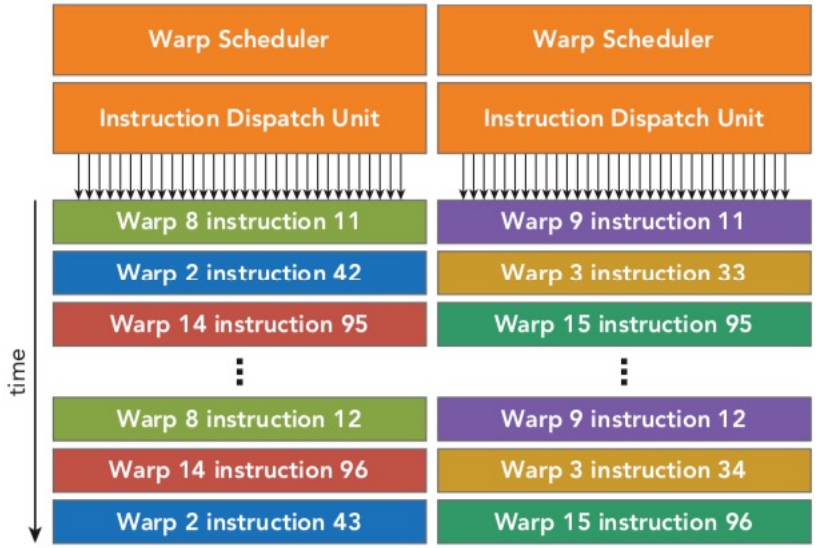
- Sharing data among parallel threads may cause a race condition
  - Multiple threads accessing the same data with an undefined ordering, which results in unpredictable program behavior
  - CUDA provides a means to synchronize threads within a thread block
  - No primitives are provided for inter-block synchronization

Warp Scheduling

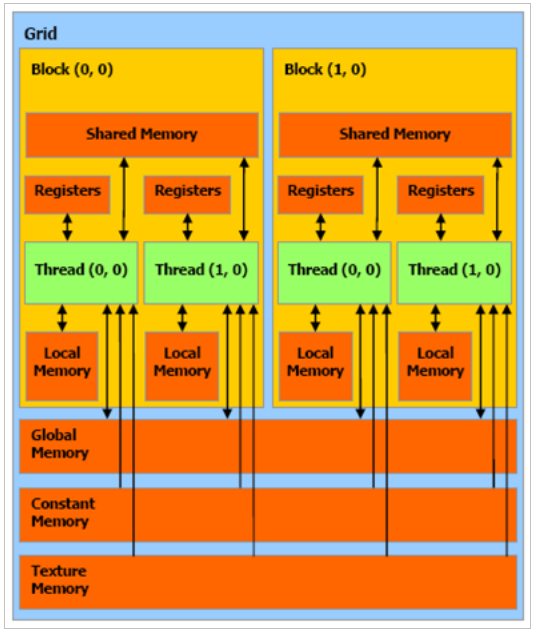
- Warps within a thread block may be scheduled in any order
- The number of active warps is limited by SM resources
- When a warp idles for any reason (e.g., waiting for values to be read from device memory), the SM is free to schedule another available warp from any thread block that is resident on the same SM
- Switching between concurrent warps has no overhead
  - Hardware resources are partitioned among all threads and blocks on an SM

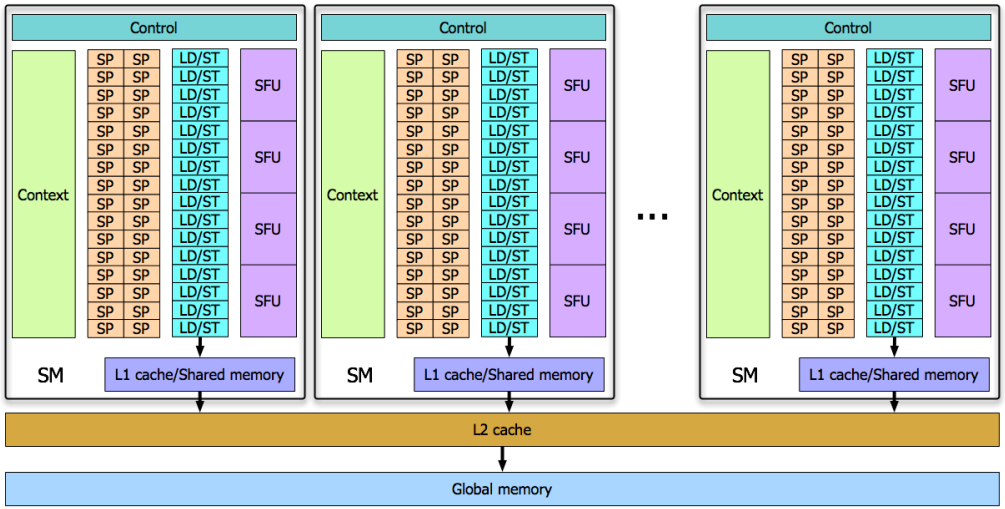
Warp Scheduling (cont'd)

- Each SM features two warp schedulers and two instruction dispatch units
- When a thread block is assigned to an SM, all threads in a thread block are divided into warps
  - The two warp schedulers select two warps and issue one instruction from each warp to a group of 16 CUDA cores, 16 load/store units, or 4 special function units
- The Fermi architecture, compute capability $2.x$, can simultaneously handle 48 warps per SM for a total of $1{,}536$ threads resident in a single SM at a time



CUDA Memory Hierarchy





Type	Read/write	Speed
Global memory	Read/write	slow, but cached
Texture memory	read only	cache optimized for 2D/3D access pattern
Constant memory	read only	where constants and kernel arguments are stored
Shared memory	read/write	fast
Local memory	read/write	used when it does not fit in to registers, just a part of global memory, slow but cached
Registers	read/write	fast

Access Speed

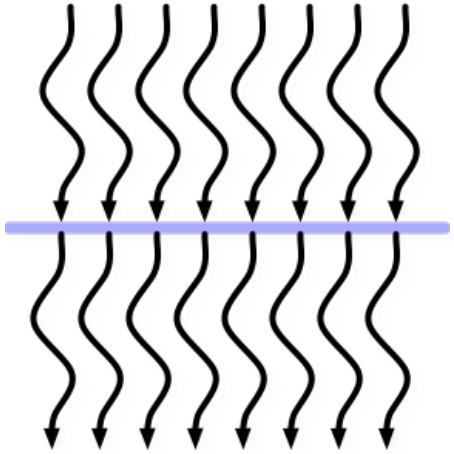
Declaration	Memory	Scope	Lifetime
int v	register	thread	thread
int v[10]	local	thread	thread
__shared__ int v	shared	block	block
__device__ int v	global	grid	application
__constant__ int v	constant	grid	application

Shared Memory

- On-chip
  - User-managed data caches (scratchpad memory)
- Much faster than local and global memory
  - Shared memory latency is roughly 100× lower than uncached global memory latency
  - Threads can access data in shared memory loaded from global memory by other threads within the same thread block
- Memory access can be controlled by thread synchronization to avoid race condition
  - `__syncthreads()`

Barrier

- `__syncthreads()` in CUDA
  - A block level synchronization barrier
  - Ensures all threads have completed before continue
- Barrier가 있는 지점에 도달한 스레드는 실행을 중단하고 다른 모든 스레드가 같은 배리어에 도달하기를 기다린 다음 실행을 계속함
- 같은 함수(코드)를 동시에 실행할 때 흔하게 이용됨
  - SPMD




```
work_on_my_problem();
Barrier();
get_result_from_others();
Barrier();
```

```
if (my_id == 0) {
    work();
    barrier();
} else {
    barrier();
}
…
```

CUDA Memory Consistency Model

- CUDA adopts a relaxed memory consistency model to enable more aggressive compiler optimizations
- To explicitly force a certain ordering for program correctness, memory fences and barriers must be inserted
  - The only way to guarantee the correct behavior of a kernel
- Barriers
  - **`void __syncthreads()`**

CUDA Memory Consistency Model (cont'd)

- Memory fences
  - Ensure that any memory write before the fence is visible to other threads after the fence
  - Do not perform any thread synchronization
    - It is not necessary for all threads in a block to actually execute the fence
  - There are three variants of memory fences depending on the desired scope: block, grid, or system

CUDA Memory Consistency Model (cont'd)

- **`void __threadfence_block();`**
  - Ensures that all writes to shared memory and global memory made by a calling thread before the fence are visible to other threads in the same block after the fence
- **`void __threadfence();`**
  - Stalls the calling thread until all of its writes to global memory are visible to all threads in the same grid
- **`void __threadfence_system();`**
  - Stalls the calling thread to ensure all its writes to global memory, page-locked host memory, and the memory of other devices are visible to all threads in all devices and host threads

Using Shared Memory

- On devices of compute capability 2.x and 3.x, each SM has 64KB of on-chip memory
  - Can be partitioned between L1 cache and shared memory
- For devices of compute capability 2.x, there are two settings:
  - 48KB shared memory and 16KB L1 cache, (default)
  - 16KB shared memory and 48KB L1 cache
- Can be configured at runtime
  - API from the host for all kernels using

    `cudaDeviceSetCacheConfig()`
  - per-kernel basis using **`cudaFuncSetCacheConfig()`**

Using Shared Memory (cont'd)

```
__global__ void reverse(int *d, int n) {
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

int main(void) {
    const int n = 64;
    int a[n], r[n], d[n];
    for (int i = 0; i < n; i++) {
        a[i] = i; r[i] = n-i-1; d[i] = 0;
    }
    int *d_d; cudaMalloc(&d_d, n * sizeof(int));
    cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
    reverse<<<1,n>>>(d_d, n);
    cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
       if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n",
                                i, i, d[i], r[i]);
}
```

Using Shared Memory (cont'd)

```
__global__ void reverse(int *d, int n) {
    // Dynamic shared memory
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
int main(void) {
    const int n = 64;
    int a[n], r[n], d[n];
    for (int i = 0; i < n; i++) {
        a[i] = i; r[i] = n-i-1; d[i] = 0;
    }
    int *d_d; cudaMalloc(&d_d, n * sizeof(int));
    cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
    reverse<<<1,n,n*sizeof(int)>>>(d_d, n);
    cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
       if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n",
                                i, i, d[i], r[i]);
}
```