Samsung DS
System Architect/Expert
2022
© Jaejin Lee

Lecture 23

Optimizations for GPUs

(Reference: John Cheng, Max Grossman, and Ty McKercher, Professional CUDA C Programming, John Wiley & Sons, 2014)

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

THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실

Warps and Thread Blocks

- Warps are the basic unit of execution in an SM
- The thread blocks in the grid are distributed among SMs
  - Thread blocks can be configured to be one-, two-, or three-dimensional
  - However, from the hard- ware perspective, all threads are arranged one-dimensionally

Warps and Thread Blocks (cont'd)

- Threads in the thread block are further partitioned into warps
    - A warp consists of 32 consecutive threads
    - All threads in a warp are executed SIMT fashion
    - All threads execute the same instruction
    - Each thread carries out that operation on its own private data
- Threads with consecutive values for threadIdx.x are grouped into warps
- For example, a one-dimensional thread block with 128 threads will be organized into 4 warps as follows:
    - Warp 0: thread 0, thread 1, thread 2, ... thread 31
    - Warp 1: thread 32, thread 33, thread 34, ... thread 63
    - Warp 3: thread 64, thread 65, thread 66, ... thread 95
    - Warp 4: thread 96, thread 97, thread 98, ... thread 127

Warps and Thread Blocks (cont'd)

- The logical layout of a two or three-dimensional thread block can be converted into its one-dimensional physical layout
  - Using the x dimension as the innermost dimension, the y dimension as the second dimension, and the z dimension as the outermost
- For example, given a 2D thread block, a unique identifier for each thread in a block can be calculated using the built-in **`threadIdx`** and **`blockDim`** variables:
  - **`threadIdx.y * blockDim.x + threadIdx.x`**
- The same calculation for a 3D thread block is as follows:

```
threadIdx.z * blockDim.y * blockDim.x
    + threadIdx.y * blockDim.x + threadIdx.x
```

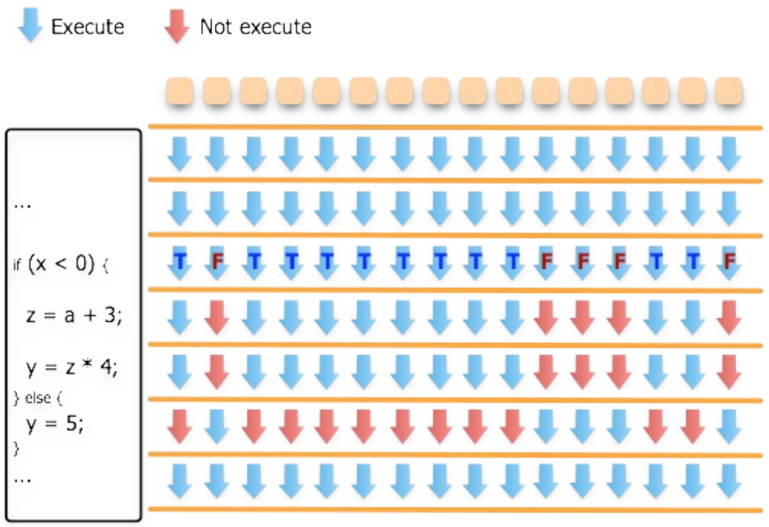
Branch Divergence

- GPUs do not have complex branch prediction mechanisms
- All threads in a warp must execute identical instructions on the same cycle
  - If one thread executes an instruction, all threads in the warp must execute that instruction
  - This could become a problem if threads in the same warp take different paths through an application

```
if (cond) { ...
} else { ...
}
```

Branch Divergence (cont'd)

- To obtain the best performance, avoid different execution paths within the same warp



Resources for Warps

- The local execution context of a warp mainly consists of the following resources:
  - Program counters
  - Registers
    - Each SM has a set of 32-bit registers stored in a register file that are partitioned among threads
    - A few thousands of registers
  - Shared memory
    - A fixed amount of shared memory that is partitioned among thread blocks
- The execution context of each warp processed by an SM is maintained on-chip during the entire lifetime of the warp
  - Switching from one execution context to another has no cost

Resources for Warps (cont'd)

- The number of thread blocks and warps that can simultaneously reside on an SM for a given kernel depends on the number of registers and amount of shared memory available on the SM and required by the kernel
  - When each thread consumes more registers, fewer warps can be placed on an SM
  - When a thread block consumes more shared memory, fewer thread blocks are processed simultaneously by an SM

Occupancy

- A thread block is called an active block when compute resources, such as registers and shared memory, have been allocated to it
  - The warps it contains are called active warps
  - The warp schedulers on an SM select active warps on every cycle and dispatch them to execution units
- Active warps can be further classified into the following three types:
  - Selected warp: an active warp that is actively executing
  - Stalled warp: an active warp that is ready for execution but not currently executing
  - Eligible warp: an active warp that is not ready for execution

Occupancy (cont'd)

- A warp is eligible for execution if both of the following two conditions are met:
  - Thirty-two CUDA cores are available for execution
  - All arguments to the current instruction are ready
- You need to keep a large number of warps active to hide the latency caused by warps stalling
- Occupancy is the ratio of active warps to maximum number of warps, per SM
  - $occupancy = \frac{\#\ of\ active\ warps}{\#\ of\ maximum\ warps}$
  - Less than or equal to 1, the bigger, the better

Occupancy (cont'd)

- To enhance occupancy,
    - Resize the thread block configuration or re-adjust resource usage to permit more simultaneously active warps
- Small thread blocks: too few threads per block leads to hardware limits on the number of warps per SM to be reached before all resources are fully utilized
- Large thread blocks: too many threads per block leads to fewer per-SM hardware resources available to each thread

Grid and Block Size

- Keep the number of threads per block a multiple of warp size (32)
- Avoid small block sizes
    - Start with at least 128 or 256 threads per block
- Adjust block size up or down according to kernel resource requirements
- Keep the number of blocks much greater than the number of SMs to expose sufficient parallelism
- Conduct experiments to discover the best execution configuration and resource usage

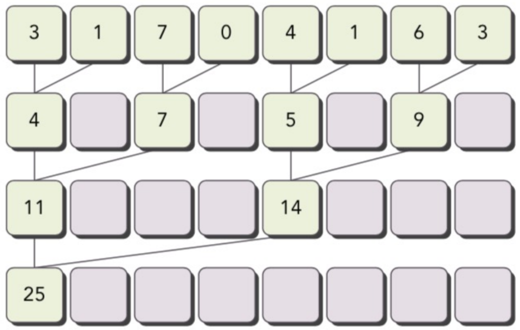
Synchronization

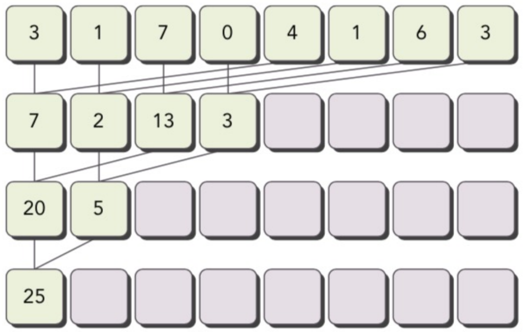
- Two levels:
  - System-level: wait for all work on both the host and the device to complete
    - **`cudaDeviceSynchronize()`** can be used to block the host until all CUDA operations have completed
  - Block-level: wait for all threads in a thread block to reach the same point in execution
    - **`__syncthreads()`**

Parallel Reduction

- Partition the input vector into smaller chunks
- Have a thread calculate the partial sum for each chunk
- Add the partial results from each chunk into a final sum

```
int sum = 0;
for (int i = 0; i < N; i++)
    sum += array[i];
```





Global Memory Accesses

- Global memory loads/stores are cached
  - All accesses to global memory go through the L2 cache
  - Many accesses also pass through the L1 cache, depending on the type of access and the GPU's architecture
- Kernel memory requests are typically served between the device DRAM and SM on-chip memory using either 128-byte or 32-byte memory transactions
  - If both L1 and L2 caches are used, a memory access is serviced by a 128-byte memory transaction
  - If only the L2 cache is used, a memory access is serviced by a 32-byte memory transaction

Global Memory Accesses (cont'd)

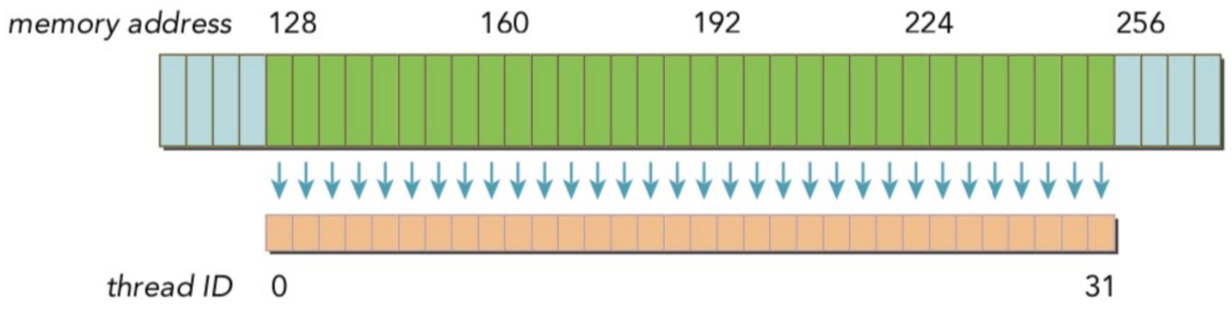
- On architectures that allow the L1 cache to be used for global memory caching, the L1 cache can be explicitly enabled or disabled at compile time
- An L1 cache line is 128 bytes, and it maps to a 128-byte aligned segment in device memory
  - If each thread in a warp requests one 4-byte value, that results in 128 bytes of data per request, which maps perfectly to the cache line size and device memory segment size

Aligned and Coalesced Memory Accesses

- Aligned memory accesses
    - When the first address of a device memory transaction is an even multiple of the cache granularity being used to service the transaction
    - Either 32 bytes for L2 cache or 128 bytes for L1 cache
    - Performing a misaligned load will cause wasted bandwidth
- Coalesced memory accesses
    - When all 32 threads in a warp access a contiguous chunk of memory

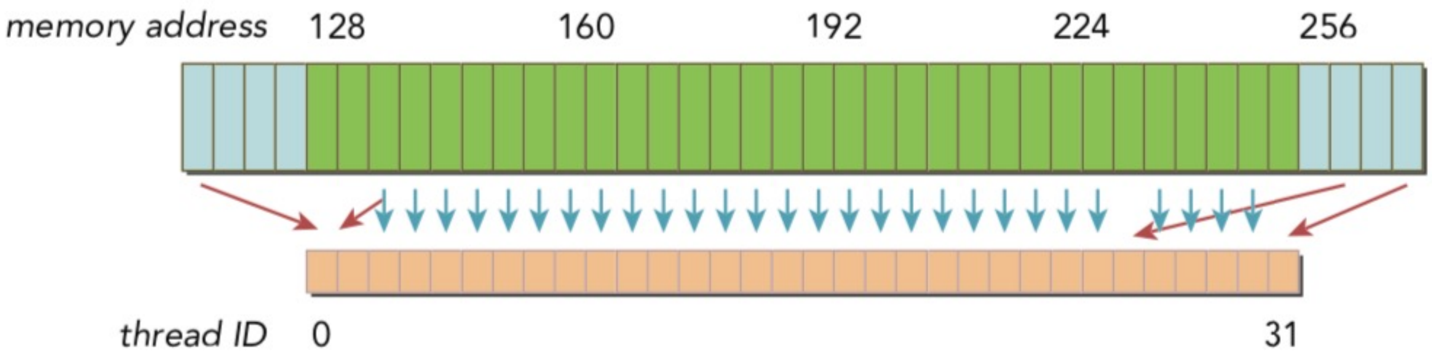
Aligned and Coalesced Memory Accesses (cont'd)

- To maximize global memory throughput, aligned coalesced memory accesses are ideal
  - A wrap accessing a contiguous chunk of memory starting at an aligned memory address
- For example, aligned and coalesced memory load operations require only a single 128-byte memory transaction to read the data from device memory



Aligned and Coalesced Memory Accesses (cont'd)

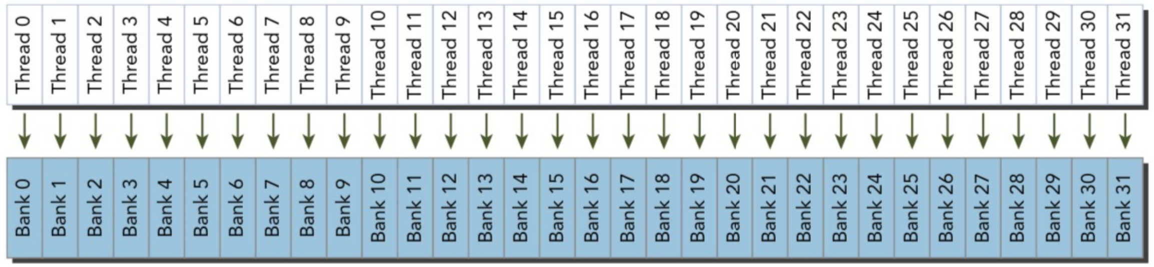
- For example, misaligned and uncoalesced memory accesses require as many as three 128-byte memory transactions to read the data from device memory
  - One starting at offset 0
  - One starting at offset 128
  - One starting at offset 256



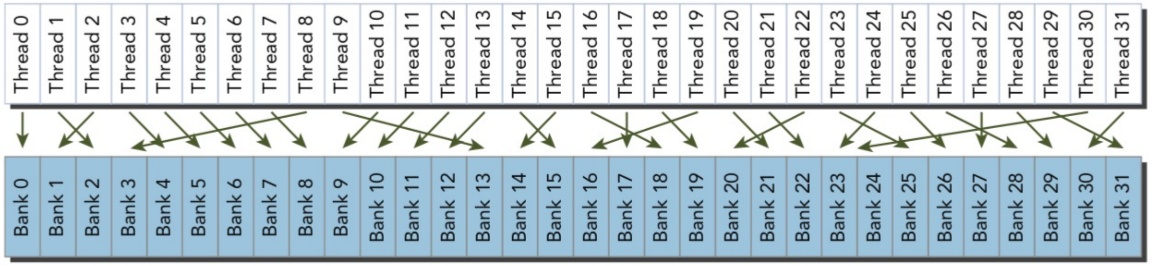
Bank Conflicts in Shared Memory

- To achieve high memory bandwidth, shared memory is divided into 32 equally-sized memory modules, called banks
    - Can be accessed simultaneously
    - There are 32 banks because there are 32 threads in a warp
    - Depending on the compute capability of a GPU, the addresses of shared memory are mapped to different banks in different patterns
- If a shared memory load or store operation issued by a warp does not access more than one memory location per bank, the operation can be serviced by one memory transaction
- Otherwise, the operation is serviced by multiple memory transactions

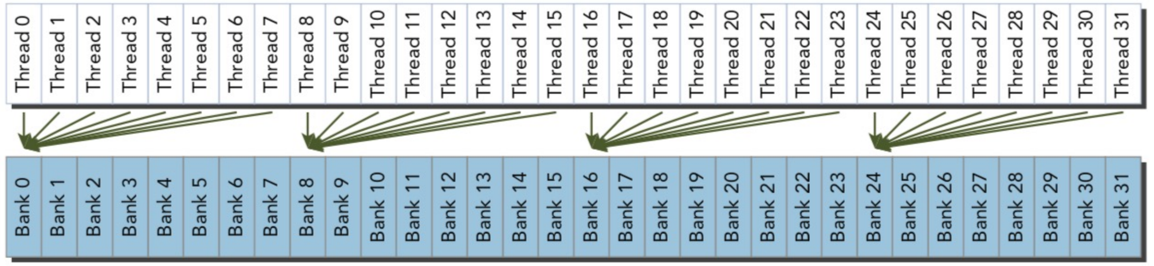
Bank Conflicts in Shared Memory (cont'd)



No conflict



No conflict



- No conflict (conflict-free broadcast access if threads access the same address within a bank)
- Bank conflicts (if threads access different addresses within a bank)