LLM Infrastructure Review (Ongoing Update)

Shengjie Liu

December 2023

Abstract

This note reviews the learning from CUDA C++ programming tutorial, CUDA programming masterclass with C++, and CSC 766 code optimization (advanced compiler optimization) from NCSU. The PyTorch review version will be revealed after the C++ version.

1 CUDA execution model

1.1 CUDA hardware architecture

Each version of device has different compute capability. (Check before use). For *Fermi* architecture, SM includes instruction cache, warp scheduler, register (fastest, temporary statements stored in there), core (computation), L2 cache (persistent data access), shared memory (per thread block), L/S unit (unit of loading and storing data), and special function units. Simply to put, the CUDA code is initialized through *__kernel__* and the dimension is defined through *dim3*.

1.2 All about Warps

SM is divided into thread blocks. The thread blocks are further divided into warps (containing 32 threads). Each warp carries out the SIMT fashion.

From software point view, thread block has 3D dimension; from hardware view, it is 1D. Considering an example, from application, set thread block with $X = 40, Y = 2$, under the view of 1D, CUDA runtime (context) assigns 4 warps to support this block since all threads in a single warp must reside in the same block.

1.3 Warp Divergence

All threads in a single warp should implement same instruction. Condition check to enforce divergence. CUDA context will disable other half to wait for the completion of the other half $\rightarrow$ performance penalty. (Not every conditional check causes divergence)

1.4 Resource Partitioning and latency hiding

1.4.1 Resource Partitioning

Local execution context of warp consider (1) *Program Counter* (2)*Shared Memory* (3) *Register.* Zero-cost warp context switch since on-chip operation ($\rightarrow$ why core is lightweight and compute limited).

set of registers (32 *bits*) on register file partitioned among threads and fixed amount of shared memory is partitioned among thread blocks.

- Fewer warps with more register per thread.
- Fewer blocks with more shared memory per block.

1.4.2 Categorization of blocks and warps

A block is called active if resources have been allocated. A warp in it is called active warps. An active warp is further classified into selected warp, stalled warp, and eligible warp. The warp scheduler dispatch active warp to be execution unit (a.k.a selected warp); an eligible warp is ready for execution but not concurrently executing; if not ready, call it stalled warp. Conditions to be an eligible warps:

- 32 CUDA cores are available for execution
- arguments to the current instruction are ready

If a warp is stalled, switching to eligible warps is easy due to zero-cost context switch.

1.4.3 Latency Hiding

This highlights the needs of the active warps to utilize the computing resource optimally. (*instruction is the number of clock cycles between instruction issued (Arithmetic or memory) and completion*).

CUDA hiding latency mechanism requires enough eligible warps. If a warp is stalled, warp scheduler dispatch to other warp to fill in the execution cycle. The question, *given a kernel, how many warps we need to hide the latency of instructions.*

For an example, SM runs 4 parallel warps, to hide arithmetic latency, each instruction in a single warp takes 20 clock cycles, then the total would need 80 eligible warps. If there are 13 SMs, then need $13 * 80 = 1040$ eligible warps.

To hide memory latency, for example, for Maxwell architecture, the DRAM latency is 350 cycles. $\rightarrow$ find how many bytes of data are transferred within 350 clock cycles. The device has bandwidth 196 GB/S. Next to find how many clock ticking per second (max clock speed) is 3.6 GHz. Then $194/3.6 = 54$ Bytes/cycle. Therefore, the total data transfered inside these cycles is 18900 Bytes between DRAM and SM. If each thread costs 4 bytes, then there needs to be $18900/4 = 4725$ threads and 148 warps.

1.4.4 Categorization of CUDA applications

Depending on the latency, can be divided into bandwidth bounded and computation bound.

1.5 Occupancy

It is defined as

$$\frac{\text{Active Warps}}{\text{Maximum Warps}}$$

One example for calculating occupancy: One kernel use 48 registers per thread and 4096 bytes of Smem per block. The block size is 128. Based on these resources, calculating is as following (per SM)

- $48 * 32 = 1536$ registers per warp. total registers are 65535. Then the allowed warps are $\frac{65536}{1536} = 42.67$ warps.
- Due to the setting of warp allocation granularity, the allowed warp needs to be the multiple of 4, which is 40.
- The total Smem is 98304. Then the active blocks are $\frac{98304}{4096} = 24$, and then the active warps are $24 * 4 = 96$. There is a hardware warp maximum amount of warps which are 64 warps per SM. Therefore, Smem is not an issue.
- Then the occupancy is $\frac{40}{64} = 63\%$.

One can use the *occupancy calculator* provided by NVIDIA to maximize the occupancy of a given kernel. *Maximzing occupancy is not always a good idea since it may increases the size of instruction, register spilling.*

1.6 Profiling with Nvprof

One can utilize Profile driven optimization to improvde the code (The old version: *Nvprof*; the new version: *Nsight*)

1.7 CUDA synchronization

Many operations such as kernel launches are asynchronous in CUDA programming. *cudaDeviceSynchronize* introduces a global synchronize point in host code. *_syncthreads* provides a synchronization point within a block (should call from device code). Consider there are 4 warps in the block, note that not all warps carry the same instruction. Suppose warp 2 first finishes, it needs to wait for all other warps to finish to perform any further instruction.

1.8 Dynamic Parallelism (Compute capability ≥ 3.5)

Allows new GPU kernels to be created and synchronized directly on GPU without from the host code. Pros of dynamic parallelism:

- postpone the decision of amount of blocks and grids during runtime

- reduce the need of transferring execution control and data between host and device (kernel fusion).

Kernel launched by the host is Parent, and launched by Parent is the Child (fork in C). Note that

- Grid launches in a device thread are visible across block
- Execution of a thread block is not complete until all Child grid created by all threads in that block have completed (If all thread blocks exit, then implicit synchronization over child grids is triggered)
- Child block is not guaranteed to begin execution until the parent thread block explicitly synchronizes on the child.
- They share same global and constant memory but distinct local and shared memory; shared and local memory are not coherent between parent and child

2 CUDA memory model

2.1 Memory Utilization Performance Metric for Kernel

Use Nsight to check these metrics

- *gld_efficiency*: global memory load efficiency
- *gld_throughput*: global memory load throughput
- *gld_transactions*: global memory load transactions
- *gld_transactions_per_request*: global memory load transactions needed for one memory request

How to leverage locality with CUDA memory hierarchy is the key to improve these metrics, and most CUDA applications are memory bound applications.

2.2 Different Memory Types in CUDA

In general, there are two memory types in CUDA (SRAM and DRAM). There are On-chip (SM) memory such as Registers, Smem, L1 Cache and Constant Memory, and On-device memory including L2 Cache, Global Memory, Texture Memory.

- *Register*: Fastest; frequently accessed thread private variables, or array with indices predetermined during the compile time; share life time with kernel (*Remark: Register spilling* to not on-chip local memory)
- *Local Memory*: store variables eligible for registers but cannot fit into registers
- *Shared Memory*: declare through *__shared__* accessed modifier and per block on chip memory

2.3 Memory Management and Pinned Memory

2.3.1 Memory Management

There are two sources of memory

- *CPU Main Memory*: Malloc, Free
- *GPU Memory*: CudaMalloc, CudaFree, CudaMemCpy

The GPU memory bandwidth is around 484GB/s, but the CPU-GPU memory bandwidth is 15.75GB/s. Therefore, we need to try to avoid the memory transfer between CPU and GPU.

2.3.2 Pinned Memory

To explain in detail, to have the memory transferred from host to device, since GPU has no control over the pageable host memory (it is controlled by the operating system), the CUDA driver first allocates (page-locked) pinned memory and copies the host pageable data into that pinned memory and then transfer the pinned memory to the DRAM. CUDA runtime provides access to directly create pinned memory (cudaMallocHost and cudaFreeHost).

2.4 Zero Copy Memory (Zero++ DeepSpeed)

It is a type of pinned memory that is mapped into device address space (both device and host have direct access). Pros:

- Leveraging host memory when there is insufficient device memory
- avoid explicit data transfer between host and device
- improving PCIe transfer rates

Use cudaHostAlloc and cudaFreeHost to allocate and free zero copy memory. There are four options of it

- cudaHostAllocDefault: same as pinned memory
- cudaHostAllocPortable: can be used for all CUDA contexts
- cudaHostAllocWriteCombined: written by host and read by device
- cudaHostAllocMapped: most common and mapped into device address space

Frequently write and read memory operations does not fit for the zero copy memory, since it pass the PCIe bus, which will bring larger latency to the application.

2.5 Unified (Managed) Memory

It creates a pool of managed memory, where each allocation from this pool can be accessed from CPU and GPU using the same address. (Automatically managed by the underlying system)

2.6 Global Memory Access Pattern

Memory instruction is also warp-based. Depending on distribution of memory address within the request, memory access can be classified into different patterns. Global memory access can happen via 2 paths:

- L2 Cache and Global Memory (access served by 32 Bytes transactions)
- L1, L2 Cache and Global Memory (128 Bytes transactions)

If each thread in a warp request 4 Bytes variable, then this results in 128 Bytes of data per request. There are two memory access patterns which determine how many transactions required to service a single memory request.

- *Aligned memory access*: First address is an even multiple of the cache granularity
- *Coalesced memory access*: 32 threads in a warp access a contiguous chunk of memory

The ideal case would be both aligned and coalesced (bus utilization 100%) if the goal is to maximize global memory throughput (even if the access is randomized within 128 bytes range). If all threads request same memory address, then the bus utilization is $\frac{\text{Request Memory}}{\text{Memory load}} = \frac{4}{128} = 3.125\%$. When using L1 and L2 Cache line, if it missed align and un-coalesced memory, then $\frac{128}{384} = 33.3\%$. Memory loads does not utilize L1 cache refer as uncache loads, and uncached memory serving are more fine-grained and can lead to better bus utilization for misaligned or un-coalesced memory access (L1 $\rightarrow$ L2 $\rightarrow$ DRAM).

2.7 Global Memory Writes

Here no L1 cache involves. Store operations are cached in L2 cache before being set to global memory, therefore granularity of 32 Bytes segments. Global memory write transactions can be $1 - 4$ segments at a time. If memory access is aligned and consecutive 128 Bytes, then one 4-segments transaction is better than 4 1-segment transactions.

3 CUDA Shared Memory Model

3.1 Introduction to Shared Memory

In case of misaligned and un-coalasced memory, use shared memory to improve kernel's performance.

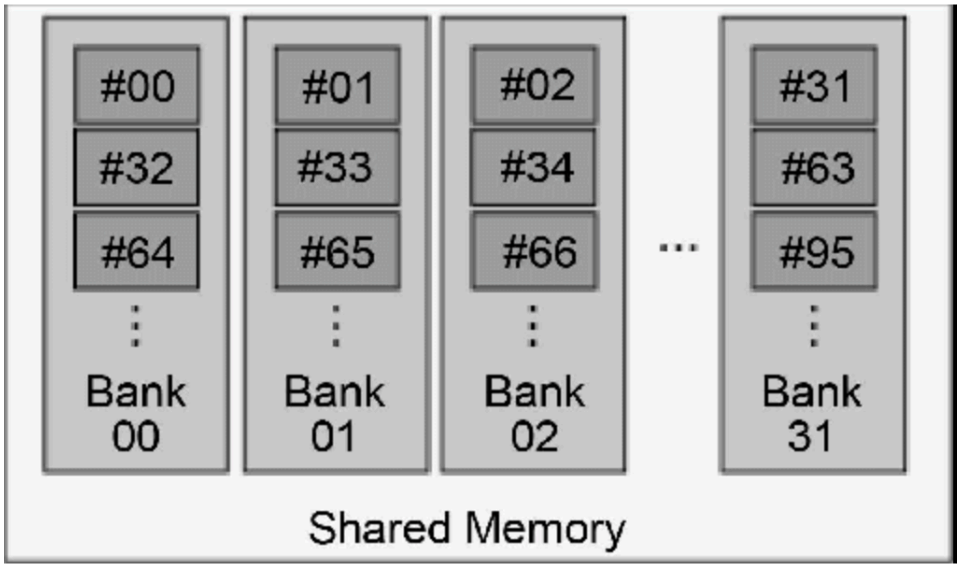
- intra-block thread communication channel
- program-managed cache for global memory data
- fixed amount of Smem is allocated to each block when starting executing and share same lifetime as the thread block

Smem access are issued per warps. Ideally request is served by one transaction. Worst is served by 32 unique transactions sequentially. If multiple threads access the same word in Smem, one thread fetches the word and send it other threads via multicast. There are two ways of restoring repeated memory access

- L1 cache → not programmable and if it is exhausted, the common memory access will be L2 Cache / DRAM.
- Smem → programmable, using *__shared__*.

3.2 Shared Memory Banks and Access Modes

To achieve higher memory bandwidth, shared memory is divided into 32 equally sized memory modules called banks which can be assessed simultaneously.



There are 32 banks since there are 32 threads in a warp. Three access patterns

- multiple accesses issued by a warp fall into different banks, ideally all 32 banks then parallel access (maybe randomized) of shared memory occurs (one single memory transaction).
- multiple accesses fall into same bank then serial access to memory bank occurs (maybe 32 single transactions → Bank Conflict).
- multiple accesses fall into same word (broadcast happens, although one single transaction but the bandwidth utilization is low).

There are two Smem access modes: 32 bit and 64 bit depending on the Smem bank width (32 bit or 64 bit). If it is 64 bit width bank, and the access mode is 32, then the same word address access may becomes to the situation of bank conflict. It places the Smem as Bank0 $\rightarrow 1 \rightarrow \cdots \rightarrow 31$.