GPU Microarchitecture

Agenda

- Introduction
- Architecture overview
- CUDA thread hierarchy
- Control divergence
- Warp scheduling
- Memory hierarchy
- Synchronization
- Structure of a GPU program
- Compute capability

Introduction

- Graphics processing units (GPUs) are historically designed for photorealistic scene rendering
  - Takes the scene geometry as input and produces color for each pixel in the scene frame
  - Lot of parallelism as individual pixel can be operated on in parallel with very little synchronization
  - To take advantage of this inherent massive parallelism, GPUs have evolved to have a large array of simple processing units
  - Exploiting this massive parallel architecture for accelerating general-purpose computing is referred to as general-purpose computing on GPU or GPGPU for short

Introduction

- General-purpose computing on GPUs
  - Early attempts at GPGPU required mapping input data to an equivalent input geometry and mapping output data to pixel colors in the frame buffer
    - Required good knowledge of graphics APIs such as OpenGL or DirectX and a good understanding of the scene rendering pipeline in the GPU
  - Recognizing the large potential for performance improvement, more programmer-friendly APIs were developed for programming a GPU
    - Initial APIs were specific to GPUs
    - More portable universal APIs came into being later
  - Today GPGPU is almost like writing C/C++ programs with a few API calls to assign parallel tasks and data to the GPUs

Architecture overview

GB202 architecture (taken from Blackwell architecture whitepaper)

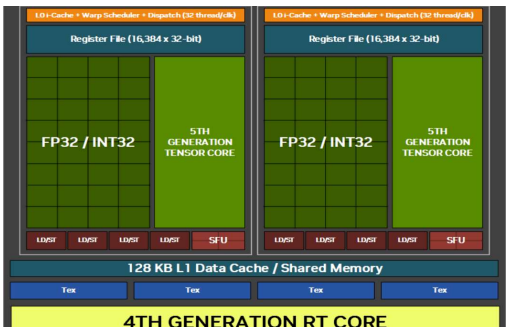
Architecture overview

GB202 architecture (taken from Blackwell architecture whitepaper)

Architecture overview

GB202 architecture (taken from Blackwell architecture whitepaper)

Architecture overview



GB202 architecture (taken from Blackwell architecture whitepaper)

Architecture overview

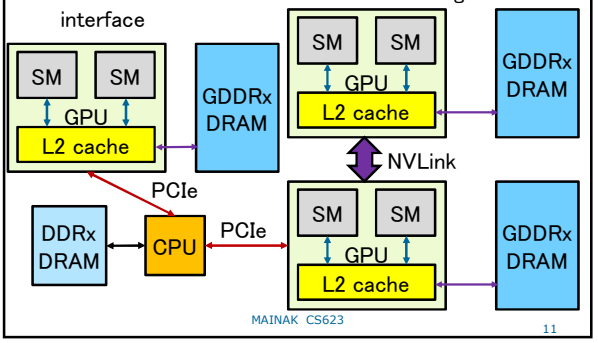
- The complete GB202 microarchitecture has
  - 12 graphics processing clusters (GPCs) clocked at 2.4 GHz
  - 8 texture processing clusters (TPCs) per GPC
  - 2 streaming multiprocessors (SMs) per TPC
  - 16 single-channel memory controllers each having a 32-wide GDDR7 DRAM channel
    - Each wire can transfer effectively 8 bits every half cycle using a 3-level pulse amplitude modulation encoding scheme
    - The channels operating at 1.75 GHz clock frequency have aggregate bandwidth of 1.75 Gcycle/s x 16 bits/cycle/wire x 32 wires/channel x 16 channels or 1.792 TB/s
    - Compare this to the bandwidth of a 16-way banked L3 cache operating at 2 GHz and pumping out a 64-byte block from each bank every cycle: 2 Gcycle/s x 64 B/cycle/bank x 16 banks or 2 TB/s
  - 128 MB L2 cache shared by all GPCs

Architecture overview

- Each streaming multiprocessor of GB202 has
  - 128 CUDA cores partitioned into four clusters
  - Each cluster has
    - One L0 instruction cache producing one instruction per cycle
    - One warp scheduler and dispatcher that dispatches one warp instruction per cycle
    - 16384 32-bit registers
    - One tensor core
    - One special function unit for computing transcendental functions
    - Four load/store address generators
  - 128 KB L1 data cache + shared memory with configurable partitions (shared memory is software managed)
  - Four texture units
  - One ray tracing core

Architecture overview

- A GPU device interfaces with the host CPU through the PCIe interface and other GPUs through NVLink interface



CUDA thread hierarchy

- The CUDA threads are arranged hierarchically
  - Purely logical view with no physical correspondence
  - Threads are partitioned into blocks of threads
  - The collection of thread blocks is referred to as a grid
  - A grid is a three-dimensional collection of thread blocks
  - Each thread block is a three-dimensional collection of threads
  - The three-dimensional arrangement of thread blocks and threads in a block has no relationship with how threads are mapped on CUDA cores
  - All threads in a thread block are scheduled on the same SM
  - There is no context switch of thread blocks
    - If all SMs are exhausted, a new thread block can be scheduled only after some thread block completes
    - Grid-wide global barriers may deadlock

CUDA thread hierarchy

- The CUDA threads are arranged hierarchically
  - Threads in a block are further grouped into physical entities called warps each containing 32 threads matching the number of CUDA cores in one of the four partitions of an SM
  - A warp is a scheduling unit i.e., a ready warp from a thread block is picked up by the scheduler (each SM partition has a scheduler) and issues it to the CUDA cores of the partition
  - The threads in a warp execute in lock-step i.e., all threads execute the same instruction but operate on different data points
    - Creates problem with branches because the "if block" and the "else block" may get executed by different subsets of threads
    - Referred to as intra-warp control divergence

Control divergence

- Intra-warp control divergence

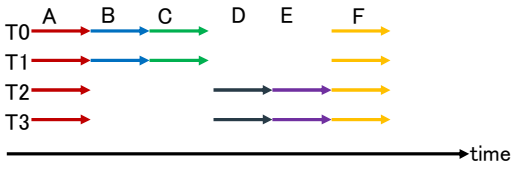
```
if (x[i] >= 0) x[i]++;
else x[i]--;
```

  - Suppose x[i] is operated on by thread i
  - Lock-step execution is no longer possible
  - The threads for which x[i] is non-negative will execute first while the remaining threads in the warp will sit idle
  - Next the other subset of threads will execute while the first subset will sit idle
  - The diverging branches are essentially serialized within a warp
    - Leads to loss in speedup
  - First subset must wait for the second subset to reconverge before resuming lock-step execution: recipe for deadlock

Control divergence

- Example of intra-warp control divergence (assume four threads in a warp)

```
A;
if (thread_id < 2) { B; C; }
else { D; E; }
F;
```



Control divergence

- Example of intra-warp control diveregence
  - Lock using atomicCAS (int *addr, expectedVal, newVal)
  - Atomic compare and set: atomically compares the value at addr with expectedVal and if the comparison passes, sets the value at addr to newVal; returns oldVal at addr

```
while (atomicCAS (&lock, 0, 1));
x++;  // Critical section code
lock = 0;
```

  - Requirement to reconverge after the while loop before proceeding to critical section makes this implementation deadlock

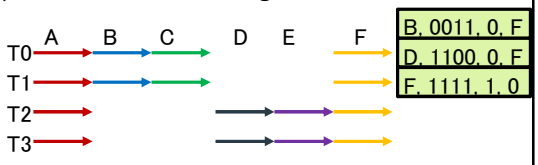
Control divergence

- Following is a deadlock-free implementation

```
int done = 0;
while (!done) { // Threads already done diverge and skip
  if (!atomicCAS(&lock, 0, 1)) { // Lock holder diverges
    x++; // Critical section code
    lock = 0;
    done = 1;
  }
  // All threads reconverge and proceed to next iteration
}
```


Control divergence

- Per warp active mask, program counter (PC), reconvergence PC
- On encountering a branch, the active masks and start PCs of the two paths and the reconvergence PC are pushed onto a "SIMT divergence stack"



- Implementation up to Pascal architecture

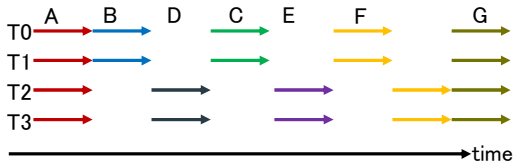
Control divergence

- From Volta architecture onward, per thread program counter and stack are maintained
  - 32x state
  - Any group of threads *with the same PC* can be scheduled together in a cycle
  - Independent thread scheduling
  - Lock-step execution of threads in a warp is no longer guaranteed
  - Intra-warp synchronization instruction must be used to force reconvergence of threads within a warp
  - Paths from SIMT divergence stack independently participate in scheduling
    - These sub-warps become independent scheduling entities

Control divergence

- Example of independent thread scheduling

```
A;
if (thread_id < 2) { B; C; }
else { D; E; }
F;  __syncwarp();  G;
```



Control divergence

- Example of independent thread scheduling
  - Lock using atomicCAS (int *addr, expectedVal, newVal)
  - Atomic compare and set: atomically compares the value at addr with expectedVal and if the comparison passes, sets the value at addr to newVal; returns oldVal at addr

```
while (atomicCAS (&lock, 0, 1));
x++;  // Critical section code
lock = 0;
```

  - Lock holder can execute the critical section and release the lock, while the remaining threads in a warp loop on atomicCAS
  - Independent thread scheduling makes it deadlock-free

Warp scheduling

- Warp scheduling
  - The threads in a warp may suffer from long delays due to memory access or some long-latency floating-point operation
  - Delay due to memory access is particularly large
  - If the threads in a warp access contiguous addresses all falling on a cache block, these accesses are coalesced and a single L2 cache access is made in the case of an L1 cache miss
    - The L1 cache block size is usually 128 bytes matching this good case
  - If the threads in a warp have irregular accesses, coalescing may not work at its peak efficiency generating 32 L2 cache accesses in the worst case
    - These accesses may take different amounts of time

Warp scheduling

- Warp scheduling
  - If any thread in a warp is waiting for a memory operation to complete, the warp is de-scheduled and a new ready warp is scheduled in its place
    - There is no register saving or restoring during context switch; hence scheduling overhead is low
  - If different threads in a warp take different amounts of time to complete their memory accesses, it leads to what is referred to as memory divergence
    - The longest delay dictates when the warp will be ready for scheduling
  - An SM can schedule warps from multiple different thread blocks simultaneously to maximize utilization of the CUDA cores
    - Full warp scheduling can be preferred over sub-warps (unfair)

Warp scheduling

- Which warps should be scheduled in a given cycle?
  - Decided by a warp scheduling algorithm
  - Only active (or ready) warps (or sub-warps) are eligible to be scheduled
    - Definition of ready: not stalled due to branch, memory access, computation, or structural hazard
- Loose round-robin (LRR) scheduling [Rogers et al., 2012]
  - Schedule the next eligible round-robin warp in this cycle
  - All eligible warps make equal progress
  - Warps enjoy spatial locality if that is how the program data is partitioned
  - All warps may stall on long-latency operations together exhibiting periodic high-utilization and no-utilization phases

Warp scheduling

- Loose round-robin scheduling
  - Consider an array A that is double the size of the L1 data cache (aggregate across all SMs)
  - The following code snippet shows the advantages and disadvantages of round-robin scheduling

```
for (i=0; i<N; i++) A[i] = f(i);    // Initialize
for (i=N-1; i>=0; i--) A[i] = g(A[i]);  // Compute
```

  - If g is just one instruction, all intra-warp locality is exploited
  - If g has more instructions, there is a danger of L1 cache thrashing if the number of ready warps is very large

Warp scheduling

- Two-level round-robin scheduling [Narasiman et al., 2011]
  - Form warp groups consisting of consecutive warps preserving inter-warp DRAM locality
  - Keep scheduling the warps from a warp group in round-robin fashion until all warps in the group stall at which point the next round-robin warp group is scheduled
  - Attempts to stagger the stalls across the warp groups minimizing the no-utilization time spans

Warp scheduling

- Greedy then oldest (GTO) [Rogers et al., 2012]
  - Keep the oldest warp scheduled until it stalls at which point the oldest warp among the ready ones is scheduled
  - Age of a warp is measured from its creation time
    - Oldest warp is the one with the smallest creation time
    - Ties are broken in favor of the smallest warp id
  - GTO improves intra-warp L1 cache locality by not bringing in too much of data simultaneously
  - A case for scheduler-controlled cache thrashing
- Two-level GTO [Rogers et al., 2012]
  - GTO within a warp group and across warp groups
  - The oldest warp group is the one that has the oldest warp

Warp scheduling

- Cache-conscious warp scheduling (CCWS) [Rogers et al., 2012]
  - What is the optimal number of warps to be considered for scheduling during a time window in a streaming multiprocessor?
  - Idea is to keep scheduling the warps that could have enjoyed L1 cache hits if other interfering warps were not scheduled
  - Each warp is assigned a base score x at the beginning, the score is incremented by y whenever the hardware detects a lost L1 cache hit, and the score is decremented by 1 each cycle otherwise as long as the score is more than x
  - Warps are sorted in descending order of score and the maximum number of warps are selected from top such that the sum of their scores does not exceed a threshold T

Warp scheduling

- Cache-conscious warp scheduling (CCWS) [Rogers et al., 2012]
  - A warp that is not selected for scheduling is descheduled at its next load instruction
  - How does a warp detect a lost L1 cache hit due to inter-warp interference?
  - When a block is inserted by a warp into the L1 cache, the warp id is recorded in the extended tag of the block
  - When a block is evicted from the L1 cache, its tag is copied into the filling warp's victim tag array
    - A per-warp extra storage
  - When a warp suffers an L1 cache miss, it looks up its victim tag array and if the tag is found there, one lost hit is detected and the tag is invalidated from the victim tag array

Warp scheduling

- Divergence-aware warp scheduling (DAWS) [Rogers et al., 2013]
  - A warp with threads accessing different cache blocks in the same instruction is often referred to as a warp having memory divergence
    - A memory divergent warp has a very high L1 cache footprint per instruction, but may have a lot of reuse across instructions
  - A warp with control divergence gets its total footprint divided over time
    - A scheduler that is aware of this would allow other warps to be scheduled along with a sub-warp of a control divergent warp

Warp scheduling

- Divergence-aware warp scheduling (DAWS) [Rogers et al., 2013]
  - CCWS favors warps that have suffered from L1 cache interference
    - Does not act until interference is detected
  - CCWS assigns the same base score to all warps
    - Effectively assumes that all warps have equal volume of reuses
  - DAWS estimates footprint and divergence of a warp by either profiling or estimating dynamically through L1 cache and victim tag array hit counts
    - Target loops are marked by compiler
    - Uses one sample warp to do this estimation in the target loops
  - DAWS schedules only those warps whose aggregate footprint can be accommodated in the L1 data cache

Warp scheduling

- Take-away regarding warp scheduler design
  - Large number of proposals based on L1 data cache locality
  - Few others combine L1 data cache locality with warp criticality
  - Some proposals bring out synergy between L1 data cache replacement policies and warp scheduling
- In the context of Blackwell
  - Each SM has four independent warp schedulers
  - All scheduled warps share 128 KB L1 data cache
  - Without coordination among the four schedulers, it is easy to thrash the L1 data cache
  - Problem of optimal warp scheduler design that jointly maximizes the utilization of CUDA cores and L1 data cache space is still wide open

Memory hierarchy

- Threads within a thread block can share data through the L1 data cache and/or the software-managed shared memory
  - 128 KB can be partitioned into L1 data cache and shared memory
  - Whatever is not used as shared memory can be used as L1 data cache
- Sharing data across thread blocks can happen through "global memory" only
  - "Global memory" variables are allocated in L2 cache
  - Such sharing is slow
  - Implies that synchronization across thread blocks is also slow (implemented using atomic instructions)
  - Best performance when thread blocks don't share any data

Accessing CPU data

- Primarily two ways
  - Explicit copy from CPU memory to GPU memory
    - Allocate space in GPU memory and then invoke DMA to copy the data
  - No explicit copy
    - Method-1: keep data pinned on CPU and access from GPU
      - Slow due to thread stalls arising from data fetch
    - Method-2: use unified virtual addressing (UVA) to enable transparent migration of pages between CPU and GPU
      - First-touch allocation followed by migration depending on access pattern
      - For pre-Pascal GPUs, the migration happens at the time of kernel launch
      - From Pascal onward, GPU execution can generate page faults and at the time of page fault, page is migrated
      - Migration cost can be partially hidden by hardware prefetching

Synchronization

- A large array of atomic instructions to synthesize lock-free as well as lock-based mutual exclusion
  - atomicAnd, atomicOr, atomicXor, atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS
  - Each atomic instruction comes with a scope flavor
    - Visibility within thread block, within entire GPU, within entire system
- Capability to mark load/store instructions (including atomics) as acquire or release
- Warp-wide and thread block-wide hardwired barrier instructions (no grid-wide barrier)
- Scope-flavored order enforcing fence instructions

A typical GPU program

- The GPU threads are launched by the CPU (referred to as the host)
  - The task executed by the GPU is referred to as a kernel
  - CPU specifies the thread grid structure when launching the kernel
  - Two ways for the kernel to access input data and return results to CPU
    - Explicit copy between CPU memory and GPU memory
    - Use of shared virtual memory between CPU and GPU
      - Referred to as unified virtual addressing (UVA) in CUDA
  - The kernel specifies the work of a thread in a manner very similar to how POSIX thread functions are written
    - Need a way to identify individual threads in a grid of threads

Compute capability

- The features supported by a GPU depends on its compute capability
  - Compute capability of a GPU is represented as X.Y where X is a major revision number and Y is a minor revision number
  - The major revision number is also known as the SM version
  - Tesla GPU: major revision 1
  - Fermi GPU: major revision 2
  - Kepler GPU: major revision 3
  - Maxwell GPU: major revision 5
  - Pascal GPU: major revision 6
  - Volta GPU: major revision 7
  - Turing GPU: compute capability 7.5
  - Ampere GPU: major revision 8

Compute capability

- The features supported by a GPU depends on its compute capability
  - Compute capability can be found from the output of deviceQuery
  - Certain features are available for all compute capabilities while some others are selectively available
  - Thread hierarchy features available to all compute capabilities
    - Maximum grid dimensions: x=$2^{31}-1$, y=65535, z=65535
    - Maximum thread block dimensions: x=1024, y=1024, z=64
    - Maximum number of threads per block in one dimension: 1024
    - Warp size: 32 threads

Compute capability

- The features supported by a GPU depends on its compute capability
  - Thread hierarchy features that vary across compute capabilities
    - Maximum number of simultaneous thread blocks per SM
      - 16 for compute capabilities 3.5, 3.7 (up to Kepler)
      - 32 from 5.0 to 7.2 (Maxwell, Pascal, Volta)
      - 16 for 7.5 (Turing)
      - 32 for 8.0 (early Ampere)
      - 16 for 8.6, 8.7 (more recent Ampere)
    - Maximum number of resident warps per SM at any time
      - 64 for compute capabilities up to 7.2
      - 32 for 7.5
      - 64 for 8.0
      - 48 for 8.6, 8.7
      - Can compute the maximum number of threads per SM