Graphics Processing Units (GPUs)

Mengjia Yan
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on slides from Daniel Sanchez
Why Study GPUs?

• Very successful commodity accelerator/co-processor

• GPUs combine two strategies to increase efficiency
  – Massive parallelism
  – Specialization

• Illustrates tension between performance and programmability in accelerators
Graphics Processors Timeline

• Until mid-90s
  – Most graphics processing in CPU
  – VGA controllers used to accelerate some display functions

• Mid-90s to mid-2000s
  – Fixed-function accelerators for 2D and 3D graphics
    • triangle setup & rasterization, texture mapping & shading
  – Programming:
    • OpenGL and DirectX APIs
Contemporary GPUs

- Modern GPUs
  - Some fixed-function hardware (texture, raster ops, ...)
  - Plus programmable data-parallel multiprocessors
  - Programming:
    - OpenGL/DirectX
    - Plus more general purpose languages (CUDA, OpenCL, ...)

Luebke and Humphreys, 2007
GPUs in Modern Systems

- **Discrete GPUs**
  - PCIe-based accelerator
  - Separate GPU memory

- **Integrated GPUs**
  - CPU and GPU on same die
  - Shared main memory and last-level cache

- **Pros/cons?**

April 30, 2020

Intel Ivy Bridge, 22nm 160mm²

Nvidia Kepler

Apple A7, 28nm TSMC, 102mm²
SIMT

- Many threads, each with private architectural state, e.g., registers
- Group of threads that issue together called a warp
- All threads that issue together (in the same warp) execute same instruction
- Entire pipeline is an SM or streaming multiprocessor

green-> Nvidia terminology
Multithreading + SIMT

All threads in one thread block are assigned to one SM
Streaming Multiprocessor Overview

- Each SM supports 10s of warps (e.g., 64 in Kepler) with 32 threads/warp
- Fetch 1 instr/cycle
- Issue 1 ready instr/cycle
  - Simple scoreboard: all warp elements must be ready
- Instruction broadcast to all lanes
- Multithreading is the main latency-hiding mechanism

What average latency is needed to keep busy? 64
Context Size vs Number of Contexts

- SMs support a variable number of contexts based on required registers (and shared memory)
  - Few large contexts → Fewer register spills
  - Many small contexts → More latency tolerance
  - Choice left to the compiler

- Example: Kepler supports up to 64 warps
  - Max: 64 warps @ <=32 registers/thread
  - Min: 8 warps @ 256 registers/thread
Many Memory Types

- Per Thread Memory
- Scratchpad Shared Memory
- Global Memory

Thread 0
Thread 1
Thread 2
Private Per Thread Memory

- Private memory
  - No cross-thread sharing
  - Small, fixed size memory
    - Can be used for constants
  - Multi-bank implementation (can be in global memory)
Shared Scratchpad Memory

- Shared scratchpad memory (share data within a thread block)
  - Small, fixed size memory (16K-64K/SM)
  - Banked for high bandwidth
  - Fed with address coalescing unit (ACU) + crossbar
    - ACU can buffer/coalesce requests
Memory Access Divergence

- All loads are gathers, all stores are scatters

- Address coalescing unit detects sequential and strided patterns, coalesces memory requests, but complex patterns can result in multiple lower bandwidth requests (memory divergence)

- Writing efficient GPU code requires most accesses to not conflict, even though programming model allows arbitrary patterns!
Shared Global Memory

- Shared global memory
  - Large shared memory
  - Will suffer also from memory divergence
• Memory hierarchy with caches
  – Cache to save memory bandwidth
  – Caches also enable compression/decompression of data
Serialized cache access

- Trade latency for power/flexibility
  - Only access data bank that contains data
  - Facilitate more sophisticated cache organizations
    - e.g., greater associativity
Handling Branch Divergence

• Similar to vector processors, but masks are handled internally
  – Per-warp stack stores PCs and masks of non-taken paths

• On a conditional branch
  – Push the current mask onto the stack
  – Push the mask and PC for the non-taken path
  – Set the mask for the taken path

• At the end of the taken path
  – Pop mask and PC for the non-taken path and execute

• At the end of the non-taken path
  – Pop the original mask before the branch instruction

• If a mask is all zeros, skip the block
Example: Branch Divergence

Assume 4 threads/warp,
initial mask 1111

if (m[i] != 0) {
  if (a[i] > b[i]) {
    y[i] = a[i] - b[i];
  } else {
    y[i] = b[i] - a[i];
  }
} else {
  y[i] = 0;
}

Optimization for branches that all go same way?
Branch divergence and locking

• Consider the following executing in multiple threads in a warp:

```c
if (condition[i]) {
    while (locked(map0[i])){}
    lock(locks[map0[i]]);
} else {
    unlock(locks[map1[i]]);
}
```

where \( i \) is a thread id and \( \text{map0}[i], \text{map1}[i] \) are permutations of thread ids.

What can go wrong here?

Warp-based implementation can cause deadlock
CUDA GPU Thread Model

- Single-program multiple data (SPMD) model
- Each context is a thread
  - Threads have registers
  - Threads have local memory
- Parallel threads packed in blocks
  - Blocks have shared memory
  - Threads synchronize with barrier
  - Blocks run to completion (or abort)
- Grids include independent blocks
  - May execute concurrently
  - Share global memory, but
  - Have limited inter-block synchronization
Code Example: DAXPY

**C Code**

```c
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

**CUDA Code**

```c
// Invoke DAXPY with 256 threads per block
__host__
int nbblocks = (n + 255) / 256;
daxpy<<<nbblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- CUDA code launches 256 threads per block
- CUDA vs vector terminology:
  - Thread = 1 iteration of scalar loop (1 element in vector loop)
  - Block = Body of vectorized loop (VL=256 in this example)
  - Grid = Vectorizable loop
GPU Kernel Execution

1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space
  \[\rightarrow\] no copies, but CPU & GPU contend for memory
Hardware Scheduling

- Grids can be launched by CPU or GPU
  - Work from multiple CPU threads and processes

- HW unit schedules grids on SMs
  - Priority-based scheduling

- Multi-level scheduling
  - Limited number of active grids
  - More queued/paused

The redesigned Kepler HOST to GPU workflow shows the new Grid Management Unit, which allows it to manage the actively dispatching grids, pause dispatch, and hold pending and suspended grids.

NVIDIA GPUDirect™ when working with a large amount of data, increasing the data throughput and reducing latency is vital to increasing compute performance. Kepler GK110 supports the RDMA feature in NVIDIA GPUDirect, which is designed to improve performance by allowing direct access to GPU memory by third-party devices such as IB adapters, NICs, and SSDs. When using CUDA 5.0, GPUDirect provides the following important features:

- Direct memory access (DMA) between NIC and GPU without the need for CPU-side data buffering.
- Significantly improved MPISend/MPIRecv efficiency between GPU and other nodes in a network.
- Eliminates CPU bandwidth and latency bottlenecks.
- Works with variety of 3rd-party network, capture, and storage devices.
Synchronization

- Barrier synchronization within a thread block (\texttt{__syncthreads()})
  - Tracking simplified by grouping threads into warps
  - Counter tracks number of warps that have arrived barrier

- Atomic operations to global memory
  - Read-modify-write operations (add, exchange, compare-and-swap, ...)
  - Performed at the memory controller or at the L2

- Limited inter-block synchronization!
  - Can’t wait for other blocks to finish
GPU ISA and Compilation

• GPU microarchitecture and instruction set change very frequently

• To achieve compatibility:
  – Compiler produces intermediate pseudo-assembler language (e.g., Nvidia PTX)
  – GPU driver JITs kernel, tailoring it to specific microarchitecture

• In practice, little performance portability
  – Code is often tuned to specific GPU architecture
System-Level Issues

- **Instruction semantics**
  - Exceptions

- **Scheduling**
  - Each kernel is non-preemptive (but can be aborted)
  - Resource management and scheduling left to GPU driver, opaque to OS

- **Memory management**
  - First GPUs had no virtual memory
  - Recent support for basic virtual memory (protection among grids, no paging)
  - Host-to-device copies with separate memories (discrete GPUs)
GPU: Multithreaded Multicore Chip

- Example: Nvidia Pascal GP100 (2016)
  - 60 streaming multiprocessors (SMs)
  - 4MB Shared L2 cache
  - 8 memory controllers
    - 720 GB/s (HBM2)
  - Fixed-function logic for graphics (texture units, raster ops, ...)
  - Scalability \(\rightarrow\) change number of cores and memory channels
  - Scheduling mostly controlled by hardware
Pascal Streaming Multiprocessor (SM)

- Execution units
  - 64 FUs (int and FP)
  - 16 load-store FUs
  - 16 special FUs (e.g., sqrt, sin, cos, ...)

- Memory structures
  - 64K 32-bit registers
  - 64KB shared memory

- Contexts
  - 2048 threads
  - 32 blocks
Vector vs GPU Terminology

<table>
<thead>
<tr>
<th>Type</th>
<th>More descriptive name</th>
<th>Closest old term outside of GPUs</th>
<th>Official CUDA/NVIDIA GPU term</th>
<th>Book definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorizable Loop</td>
<td>Vectorizable Loop</td>
<td>Grid</td>
<td>A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.</td>
<td></td>
</tr>
<tr>
<td>Body of Vectorized Loop</td>
<td>Body of a (Strip-Mined) Vectorized Loop</td>
<td>Thread Block</td>
<td>A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.</td>
<td></td>
</tr>
<tr>
<td>Sequence of SIMD Lane Operations</td>
<td>One iteration of a Scalar Loop</td>
<td>CUDA Thread</td>
<td>A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.</td>
<td></td>
</tr>
<tr>
<td>A Thread of SIMD Instructions</td>
<td>Thread of Vector Instructions</td>
<td>Warp</td>
<td>A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.</td>
<td></td>
</tr>
<tr>
<td>SIMD Instruction</td>
<td>Vector Instruction</td>
<td>PTX Instruction</td>
<td>A single SIMD instruction executed across SIMD Lanes.</td>
<td></td>
</tr>
<tr>
<td>Multithreaded SIMD Processor</td>
<td>(Multithreaded) Vector Processor</td>
<td>Streaming Multithread Processor</td>
<td>A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.</td>
<td></td>
</tr>
<tr>
<td>Thread Block Scheduler</td>
<td>Scalar Processor</td>
<td>Giga Thread Engine</td>
<td>Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.</td>
<td></td>
</tr>
<tr>
<td>SIMD Thread Scheduler</td>
<td>Thread scheduler in a Multithreaded CPU</td>
<td>Warp Scheduler</td>
<td>Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.</td>
<td></td>
</tr>
<tr>
<td>SIMD Lane</td>
<td>Vector Lane</td>
<td>Thread Processor</td>
<td>A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.</td>
<td></td>
</tr>
<tr>
<td>GPU Memory</td>
<td>Main Memory</td>
<td>Global Memory</td>
<td>DRAM memory accessible by all multithreaded SIMD Processors in a GPU.</td>
<td></td>
</tr>
<tr>
<td>Private Memory</td>
<td>Stack or Thread Local Storage (OS)</td>
<td>Local Memory</td>
<td>Portion of DRAM memory private to each SIMD Lane.</td>
<td></td>
</tr>
<tr>
<td>Local Memory</td>
<td>Local Memory</td>
<td>Shared Memory</td>
<td>Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.</td>
<td></td>
</tr>
<tr>
<td>SIMD Lane Registers</td>
<td>Vector Lane Registers</td>
<td>Thread Processor Registers</td>
<td>Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).</td>
<td></td>
</tr>
</tbody>
</table>

[H&P5, Fig 4.25]
Thank you!

Next Lecture: Transactional Memory