GPU Performance

CS6969 – Spring 2026

Sreepathi Pai
University of Rochester
sree@cs.rochester.edu

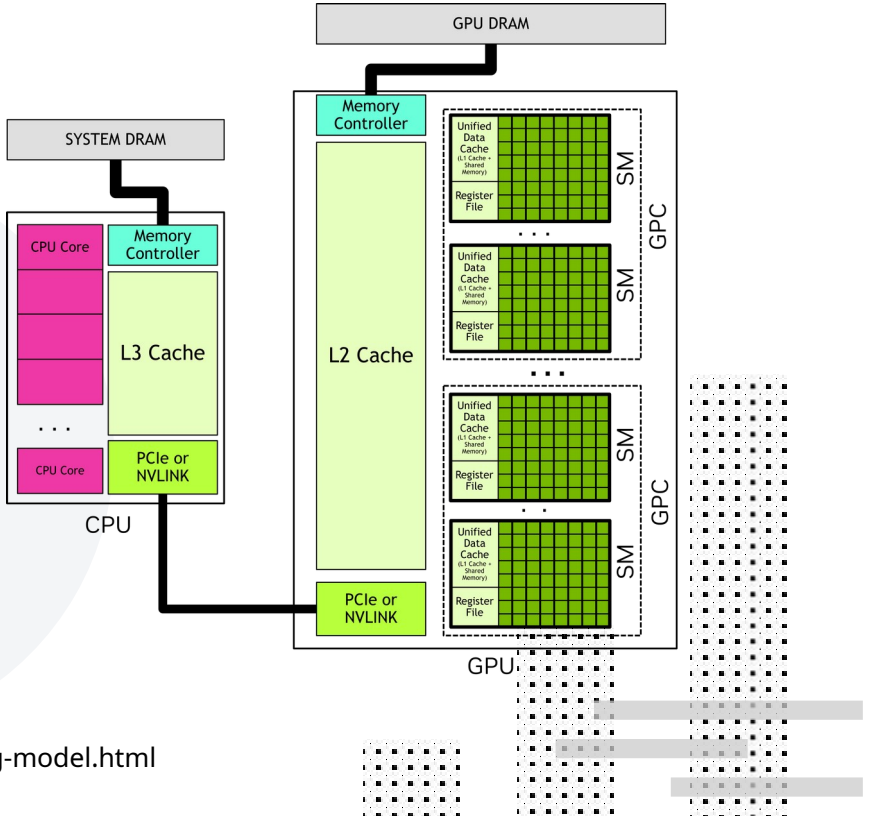
Disclaimers

- Not intended to teach GPU programming
- Mostly biased towards NVIDIA GPUs
  - AMD should be similar
  - Intel, Qualcomm, etc., are not

GPU Organization

Cores

- Aka SM or streaming multiprocessors
- Not to be confused with CUDA “Core”



https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html

Hyperthreading

- Multiple *hardware* thread contexts on same core
- CPUs = 2, GPUs = 48 to 64

“Thread” Instructions

- Each thread’s instructions are 32-wide (perform 32 operations, also known as vector instructions)
- Newer GPUs also support scalar instructions
- Known as a *warp,* with each of the 32 individual lanes known as a (CUDA) thread.

Warps

- Similar to SIMD instructions on CPUs, except not visible to programmers in early GPUs
- Were an implementation detail, but manifested as “lock-step” execution of CUDA threads
- Programmers took advantage of it for performance, known as “warp-synchronous” programming, never a part of CUDA
- NVIDIA later relented, releasing warp-centric communication primitives, allowing CUDA threads to communicate with each other without going through memory

Warp Instructions

- Most lanes of a warp perform the same operation replicated to each lane (e.g., addition)
- Some operations are truly warp-based
  - Shuffling/Permuting values from each lane

Branches

- Even branches are 32-wide
- Two approaches:
  - Predication
  - “Control Divergence”

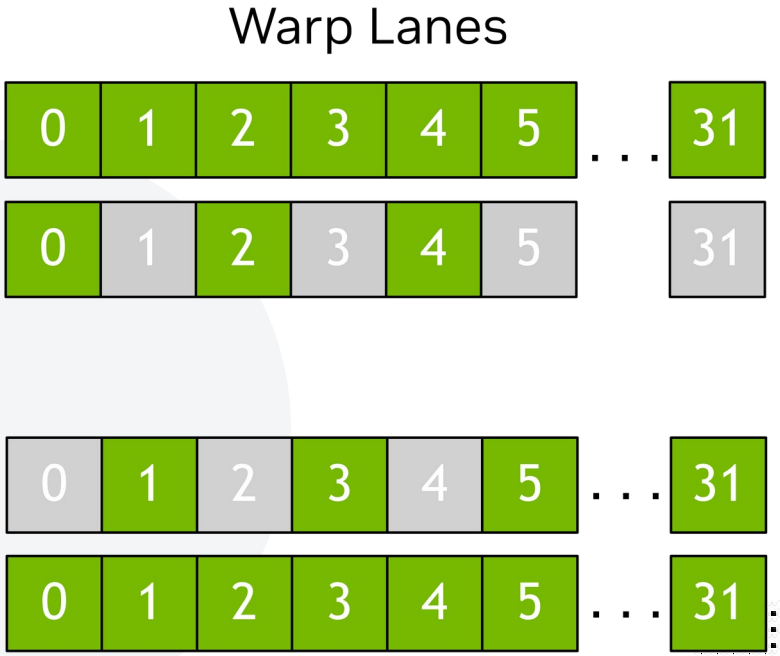
```
val = A[thread_id]
if(val > 5)
   out[thread_id] = val
else
   out[thread_id] = 0
```

Predication, Control Divergence

- Predication is a compiler approach
  - Each lane’s execution is enabled/disabled by a bit in a predicate register
  - P1 = val > 5 ; @P1 out[...] = val ; @!P1 out[...] = 0
- Control Divergence is a hardware mechanism
  - Uses an internal *active mask*, which functions as a predicate register
  - Has a mechanism to wait for reconvergence at a predetermined point (usually decided by compiler)

A Picture

```
if(threadIdx.x%2 == 0)
   {a = r(t); }
else
   {a = q(t); }
y = f(a);
```



https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html

Loads/Stores on GPUs

- Are also 32-wide
  - Each lane supplies an address
- Typically called “Gather” (load) or ”Scatter” (store)
- Can suffer from “memory divergence”
  - More than four (varies) memory *transactions*
  - Each transaction is to a cache line

Caches on GPUs

- Each SM has a L1 cache (private, non-coherent), and the GPU has a shared L2 cache
- Number of threads: 8M (A100)
- Size of cache: 40M
- Bytes / Thread: 5
- Spatial locality? Temporal locality?

On-chip Memories

- Each SM has an on-chip memory known as “shared memory”
  - term varies by programming model
  - sharing is restricted to CUDA threads belonging to a “thread block” (later)
- Programmer managed

GPU Performance

CPU-GPU Interface

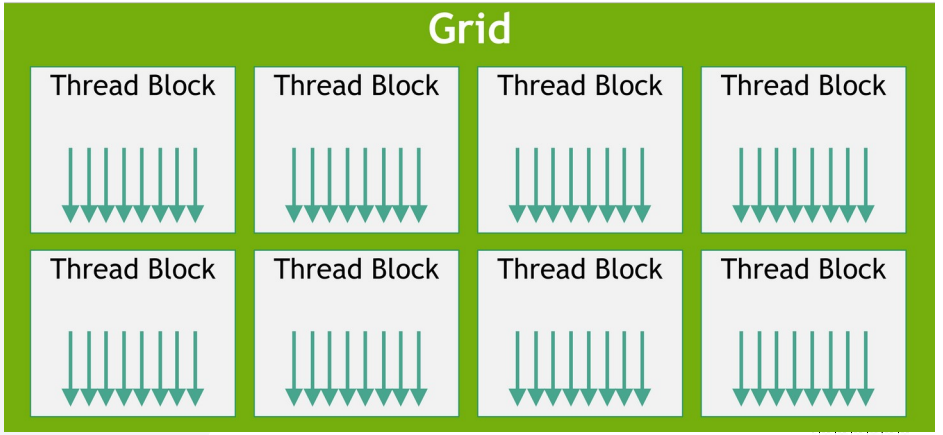
- CPU “launches” GPU “kernels” which then run asynchronously
  - i.e. CPU code doesn’t wait for completion
- CPU orchestrates data transfers between itself and the GPU

Is the work large enough?

- Recall: 8M threads
- Data transfer overhead?
- Computation overhead?
- Is it faster than the CPU?
- Does it fully utilize the GPU?

Hierarchical Threading Model

- GPU programming models organize threads as a hierarchy:
  - 3-D grids consisting of
  - 3-D same-sized thread blocks consisting of
  - Threads
- Note: 3-D can be (256,1,1) which is actually 1-D and is very common in compute kernels



https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html

Scheduling thread blocks to SMs

- When number of thread blocks is fewer than the number of SMs

Scheduling thread blocks to SMs

- When number of thread blocks is greater than number of SMs

(Thread Block) Occupancy

- Each thread block consumes a number of an SM’s resources:
  - threads
  - registers
  - shared memory
- The number of thread blocks resident on an SM (“occupancy”) is limited by the resource that runs out the fastest

Is higher occupancy better?

- Simple answer: only if it enables the kernel to reach peak performance
- Low occupancy – where no resource runs out during kernel execution – is underutilization of the GPU

Peak Performance

- GPU hardware has maximum achievable performance
  - Compute: *x* TFLOPs/s, Memory: *y* TB/s
- Your code achieves some fraction of this performance
- How to get closer to GPU peak?

Heuristic 1: GPU underutilized?

- A GPU has 8 TFLOPs/s as a *whole*
- But split into individual SMs
- Roughly speaking, if you're using half of all SMs, then you're getting half of 8 TFLOPs/s
  - "roughly" because each SM must also achieve peak

Heuristic 2: Enough parallelism?

- Silly example: Run 1 thread on the GPU
- Not-so-silly example: Each thread runs

  `atomicAdd(&finalSum, mySum)`
- where `atomicAdd` is a Read-Modify-Write instruction and `finalSum` is a global.

Heuristic 3: Compute/Memory

- Most operations can be classified into compute or memory
- Ratio of compute to memory operations is sometimes called “arithmetic intensity”
- If each compute operation requires 1 byte of memory, and the memory can only deliver 256MB/s, then you can only perform 256M ops/s even if the machine is capable of more
- “Roofline”, most useful for regular code

Performance

$$T = (W \times t) / P$$

- Minimize time T where
  - W: number of work items
  - t: average time per work item
  - P: average parallelism

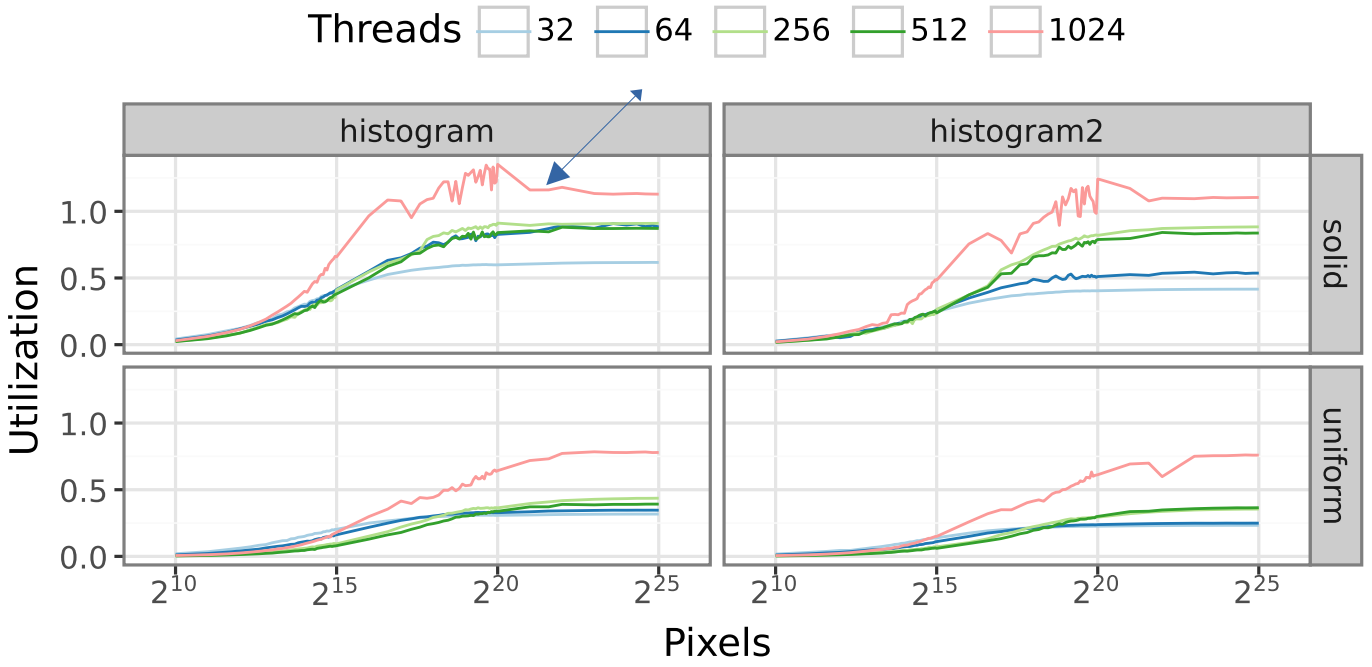
Focus on the bottleneck

- The previous equation can be written for each subsystem of the GPU
- If a subsystem is 100% utilized, it is the bottleneck
- Explore substitutions to move the bottleneck to another subsystem with higher performance
  - reading from DRAM → reading from cache → reading from shared memory → reading from registers

Challenges

- Need *t* and *P*
- Moreover *t* is load dependent, *t(n)*
  - *n* (the load) is unknown, as is *t(n)*
- Needs tedious micro-benchmarking

But when done...



Rongcui Dong and Sreepathi Pai, GPGPU 2025

GPU Performance Tools

Tools

- Profiler [e.g. nsight compute / ncu]
  - Timelines (at kernel level), CPU-GPU interaction
- Performance counters [e.g., ncu]
  - Work, Parallelism, some utilization
  - Kernel level, or sampling uniformly over time
- Instrumentation tools [e.g., nvbit]
  - Examine program values: data and control
  - e.g. count how many times a particular instruction gets executed
- APIs (CUPTI, nvtx, etc.)