GPU Memory

Mark Greenstreet and Ian M. Mitchell

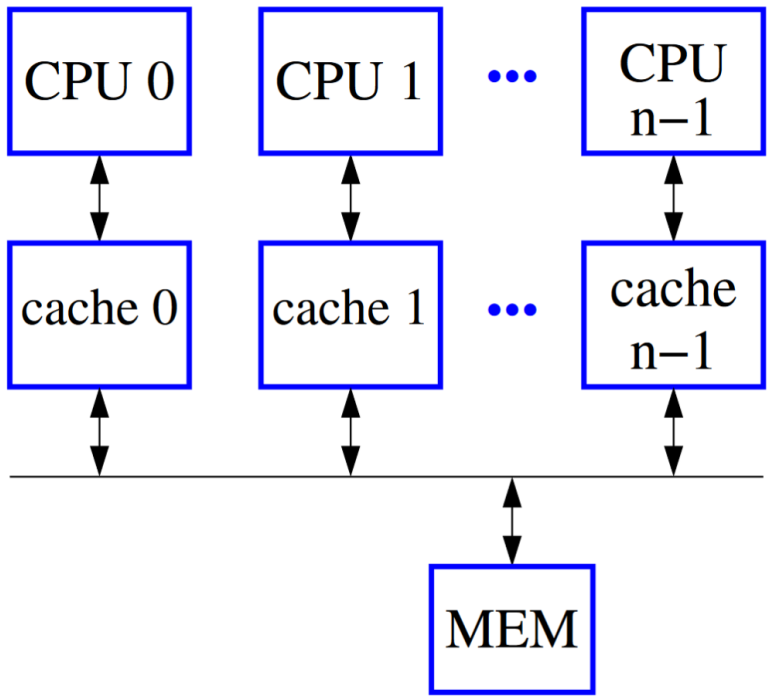
CpSc 418 – March 7, 2018

Table of Contents

Outline

CPU Memory Hierarchy

From "Shared Memory Multiprocessors" slides (Jan. 19).



- What about current multicore CPU designs?
- What are relative speeds and sizes of memories?

GPU Design Considerations

Sell chips / cards by being *much* faster than a CPU for rendering and other data parallel applications (scientific computing, machine learning, . . .).

- Have thousands of floating point ALUs (lots of compute).
- Use SIMD design (fewer resources spent on control).
- Limit (ignore?) dependency between and within threads (avoid moving data far and/or fast).
- Have very large register file (swap between many active threads).
- Use fastest available DRAM memory.
- Provide explicitly managed specialized memory.

GPU Capabilities

Using lab GPUs (GTX 1060) as an example:

- Peak 3470 single precision GFLOPS from 1152 single precision SPs running at 1.5 GHz.
- Memory bandwidth 192 GB/s.

How many times must we reuse each data value from global memory to avoid being memory bound?

Exposing (abstract) hardware details to programmer enables design of software which can make better use of massive parallelism available and thereby dramatically improve performance.

or

If it isn't fast, it's your fault.

Aside: Counting FLOPS

FLOPS = Floating Point Operations per Second.

- What is an operation? Mostly add / subtract / multiply single precision floating point numbers.
- Standard integer ops similarly fast.
- Some hardware available for other common functions: sin, cos, tan, exp, log, pow, sqrt.
- *But:* Divide (accurate version) is very slow.

Marketing trick: “Fused multiply-add” (FMA or MAD).

- Very common numerical operation: $w = x * y + z$.
- Easily implemented with little extra space and no extra time in a hardware multiplier.
- Counts as *two* operations!

Outline

Abstract GPU Memory Types

Type	Scope	Speed	Size	Explicitly Managed
Registers	thread	+++	256 KB / SM	
Shared	SM	+	64 KB / SM	✓
Constant	global	+	10 KB cached / SM	✓
Global	global	–	2–12 GB	∼

See K&H Figure 4.6.

- Like standard von Neumann architecture, there is a register file (managed by the compiler) and "global" memory (shared management).
- CUDA abstract architecture provides two additional categories of programmer managed memory.
  - Slower than registers but faster than global memory.
  - Smaller than registers but shared between threads.
- Global memory is still far too slow.

Hardware GPU Memory Types

Type	Latency (cycles)	Bandwidth	Size	Explicitly Managed
Registers	1–6	8–16 B / cycle / SP	256 KB / SM	
Shared	30–50	4 B / cycle / SP	64 KB / SM	✓
Constant	30–50	?	10 KB cached / SM	✓
L1 cache	30–50	?	48 KB / SM	
L2 cache	∼200	?	1–2 MB	
Global	∼400	48–540 GB / s	2–12 GB	∼

Details taken from Pascal microarchitecture / GeForce 10 series cards (GTX 1060 3GB / GP106 chip).

- 1.2–1.6 GHz core clock (1.5 GHz).
- 3–30 SMs, each with 4 warp schedulers handling 128 single precision SPs and 4 double precision SPs (9 SMs).
- 2–12 GB global (3 GB) at 48–540 GB/s (192 GB/s).
- 942–10790 single precision GFLOPS (3470).

Note: Nvidia publishes throughput / bandwidth numbers, but latency numbers have to be determined experimentally.

Outline

Revisit Matrix Multiply

Compute $C = AB$ with $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$ and $C \in \mathbb{R}^{m \times p}$.

- Total memory read:
- Total memory written:
- Total computation:

Compute to memory access (CMA):

Why are we achieving such poor performance despite high CMA for $m = n = p \approx 3000$?

What is the Memory Access Pattern?

Examine our basic implementation of matrix multiply

```
__global__ void s418mm_kernel(float *A, float *B, float *C, uint m
  // Determine our location in the matrix using nvcc built-ins.
  uint i = blockIdx.x*blockDim.x + threadIdx.x;
  uint k = blockIdx.y*blockDim.y + threadIdx.y;
  // If we are not an extra thread.
  if((i < m) && (k < p)) {
    // Calculate a single element of the output matrix.
    float sum = 0.0;
    for(uint j = 0; j < n; j++)
      sum += A[IDX2F(i,j,m,n)] * B[IDX2F(j,k,n,p)];
    C[IDX2F(i,k,m,p)] = sum;
  }
}
```

Compare to K&H figure 4.3.

- Consider the loop performed by each thread: How many computations per global memory access (CGMA)?

Outline

Shared Memory to Reduce Global Memory Traffic

- Observe that
  - Threads for $c_{i,*}$ read the same elements $a_{i,*}$.
  - Threads for $c_{*,j}$ read the same elements $b_{*,j}$.
  - See K&H figure 4.5.
- If we can avoid reloading these shared elements multiple times from global memory, we can increase CGMA.
- Where to store?
  - Registers?
  - L1 cache?
  - Constant memory?
  - Shared memory?

Laying Out the Data

How should we organize the data in shared memory?

Assume we can store $s \gg 1$ matrix entries in shared memory.

- Work on just i^{th} row of output matrix C (elements $c_{i,*}$).
  - Can share s elements of matrix A.
  - Cannot share any elements of matrix B.
  - Change in CGMA:
- Work on just k^{th} column of output matrix C (elements $c_{*,k}$).
  - Cannot share any elements of matrix A.
  - Can share s elements of matrix B.
  - Change in CGMA:
- Work on $r = \sqrt{s/2}$ rows and r columns of output matrix C.
  - Can share $r^2 = s/2$ elements of matrix A.
  - Can share $r^2 = s/2$ elements of matrix B.
  - See K&H figures 4.14 and 4.17.
  - Change in CGMA:

This common design pattern is called “tiling” the data.

Implementing Tiled Matrix Multiply

See kernel in K&H figure 4.16

- Why are there two loops?
- What might happen if we omitted the first `__syncthreads()`?
- What might happen if we omitted the second `__syncthreads()`?
- How should you size your tiles and blocks?