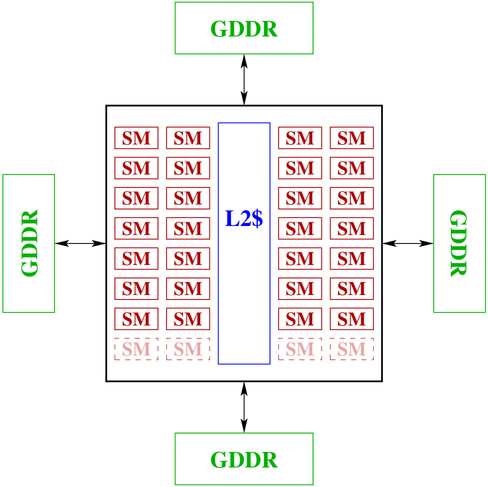
CUDA: Memory

Mark Greenstreet

CpSc 418 – November 14, 2018

- Architecture Snapshot
- Registers
- Shared Memory
- Global Memory
- Other Memory: texture memory, constant memory, caches
- Summary, preview review

First, GPU Architecture Review

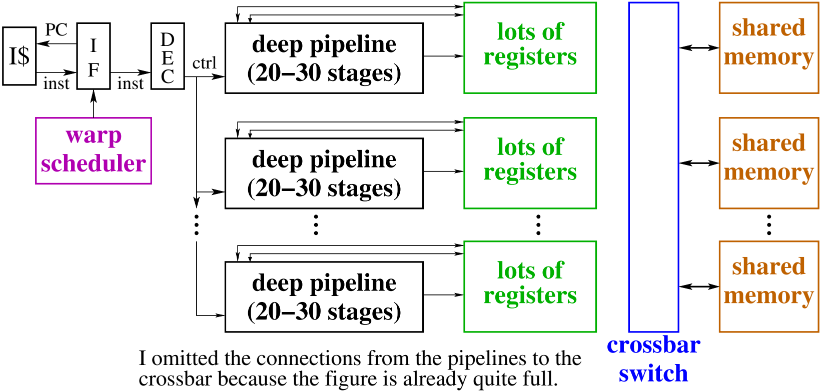


GTX 1080 Architecture
(high–end, "Pascal" generation)

GPUs in the the `linXX.ugrad.cs.ubc.ca` machines

- GeForce GTX 1060
- 9 SMs (10 on chip, 9 reported by the device):
  - 128 SPs/SM.
  - That's 1152 SPs on the chip.
  - Each SM can schedule 4 warps in a single cycle.
- $\sim$ 1.6GHz clock frequency.
- 3 GBytes of GDDR5 memory, $\sim$ 192GBytes/sec. memory bandwidth.

A Streaming Multiprocessor (SM)



I omitted the connections from the pipelines to the crossbar because the figure is already quite full.

- Each of the pipelines is an SP (streaming processor)
- Lots of deep pipelines.
- Lots of threads: when we encounter an architectural challenge:
  - Raising throughput is easy, lowering latency is hard.
  - Solve problems by increasing latency and adding threads.
  - Make the programmer deal with it.

Why do we need a memory hierarchy

```
__global__ void saxpy(uint n, float a, float *x, float *y) {
  uint myId = blockDim.x*blockIdx.x + threadIdx.x;
  if(myId < n)
    y[myId] = a*x[myId] + y[myId];
}
```

- A GPU with 1152 SPs, and a 1.7GHz clock rate (see slide 2 can perform over 3900 single-precision GFlops.
  - With a main memory bandwidth of 192 GBytes/sec., and 4 bytes per `float`, a CUDA kernel needs to perform $\frac{1152*1.7*2*4}{192} \approx 82$ floating point operations per memory read or write.
  - Otherwise, memory bandwidth becomes the bottleneck.
- Registers and shared memory let us use a value many times without going to the off-chip, GDDR memory.
  - But, we need to program carefully to make this work.
- Is `saxpy` a good candidate for GPU execution?

CGMA – calculation details

- CGMA: Compute-to-Global-Memory-Access ratio
  - Compute: number of floating point operations
  - Global memory access: number of 32-bit words read and/or written to/from the GPU's DRAM
- The example from the previous slide:

$$\begin{aligned}\mathrm{CGMA} &= \frac{1152\mathrm{SPs} * 1.7 \times 10^9 \frac{\mathrm{instructions}}{\mathrm{SP}\cdot\mathrm{sec}} * 2 \frac{\mathrm{flops}}{\mathrm{instruction}}}{192 \frac{\mathrm{bytes}}{\mathrm{sec}} * 1 \frac{32-\mathrm{bitword}}{\mathrm{byte}}} \\ &\approx 82 \frac{\mathrm{flops}}{\mathrm{memoryaccess}(32-\mathrm{bitword})}\end{aligned}$$

where

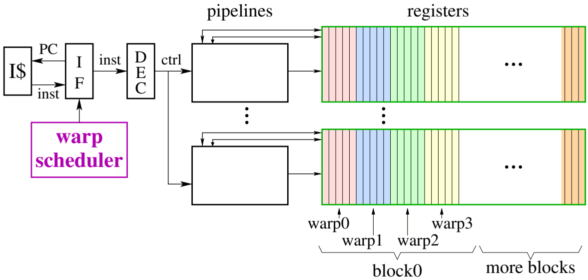
- 1152 SPs: GTX 1060 architecture details
- $1.7 \times 10^9 \frac{\mathrm{instructions}}{\mathrm{SP}\cdot\mathrm{sec}}$: GTX 1060 clock frequency
- $2 \frac{\mathrm{flops}}{\mathrm{instruction}}$: fused multiply-add
- $192 \frac{\mathrm{bytes}}{\mathrm{sec}}$: GTX 1060 off-chip memory bandwidth
- $1 \frac{32-\mathrm{bitword}}{\mathrm{byte}}$: `sizeof(float)`

Matrix Multiplication and Memory

```
for(int i = 0; i < M; i++)
  for(int j = 0; j < N; j++)
    for(int k = 0; k < L; k++)
      c[i,j] += a[i,k]*b[k,j];
```

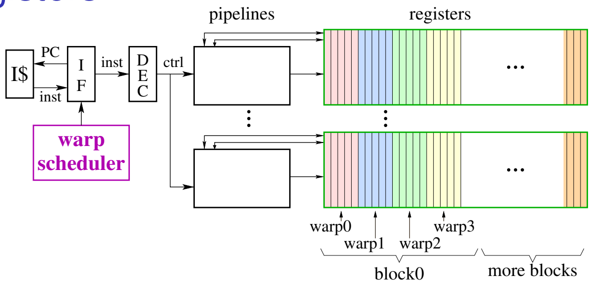
- Focus on the innermost loop: `for(k ...)`
  - Why?
- How many floating point operations per iteration?
- How many memory reads?
- How many memory writes?
- What is the “Compute-to-Global-Memory-Access” ratio (CGMA)?

Registers



- Each SP has its own register file.
- The register file is partitioned between threads executing on the SP.
- Local variables are placed in registers.
  - The compiler in-lines functions when it can
    - A kernel with recursive functions or deeply nested calls can cause register spills to main memory – this is **slow**.
  - Local array variables are mapped to global memory – **watch out**.

More Registers



- In recent versions of CUDA, threads in the same warp can swap registers.
  - Provides very efficient intra-warp communication.
  - For example: to implement the various strides of for compare-and-swap in bitonic sort.
- Performance trade-offs
  - A thread can avoid slow, global memory accesses by keeping data in registers.
  - But, using too many registers reduces the number of threads that can run at the same time.

Registers and Memory Bandwidth

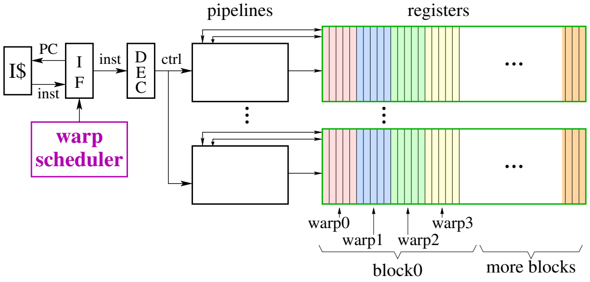
The GPU on slide 2 has 9 SMs, each with 128 SPs.

- Each SP has access to a register file.
- I'll guess two register reads and one write per clock cycle, per SP.
- I'll assume 4-byte registers.
- We get

$$9\mathrm{SM} * 128 \frac{\mathrm{SP}}{\mathrm{SM}} * 3 \frac{\mathrm{RW}}{\mathrm{SP} * \mathrm{cycle}} * 1.7 \times 10^9 \frac{\mathrm{cycle}}{\mathrm{sec.}} * 4 \frac{\mathit{Byte}}{\mathit{RW}} \approx 23500 \frac{\mathit{GByte}}{\mathrm{sec.}}$$

- 122 times faster than main memory bandwidth!

Registers and Thread Scheduling

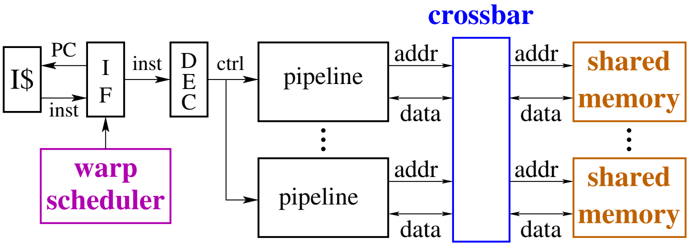


- Each SM has a 256Kbyte register file, and 64 active warps, with 32 threads/warp.
  - That's 32 4-byte registers per thread.
- If a thread uses more registers
  - The SM cannot fully use its warp scheduler, or
  - Registers will spill to main memory – **slow**
- The numbers are smaller for older GPUs.

Registers and Matrix Multiply

- Matrix multiplication can be broken into blocks.
- We can load blocks of `A` and `B` into registers and compute the result block for `C`.
  - E.g. if we load 2×2 blocks of `A`, `B` into registers, we can compute (part of) a 2×2 block of `C`.
  - This involves 8 loads, 4 stores, 8 multiplies, and 4 adds.
  - CGMA = 1. Better than the brute-force algorithm, but we need to take this idea further.

Shared Memory



- On-chip, one bank per SP.
- Banks are interleaved by:
  - Early CUDA GPUs: 4-byte word
  - Later GPUs: programmer configurable 4-byte or 8-byte words
  - Why?
- Shared memory is a limited resource: 48KBytes to 96Kbytes/SM.
  - Each SM has more registers than shared-memory.
  - Shared memory demands limit how many blocks can execute concurrently on a SM.

Shared Memory Example: Matrix Multiply

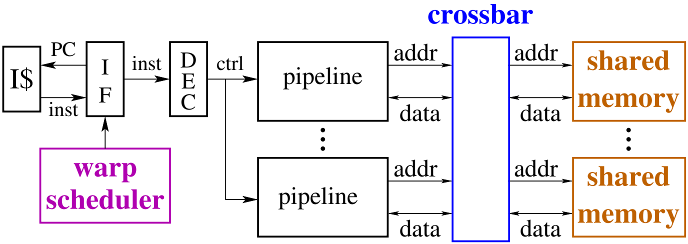
Running example from the textbook: $C = A\,B$

- Each thread-block loads a 16×16 block from A and B.
  - The threads to these loads "cooperatively":
  - Read $A_{I,K}$ and $B_{K,J}$ from global memory with "coalesced" loads.
  - Write these blocks to shared-memory in a way that avoids bank conflicts.
- Compute: $C_{I,J} \mathrel{+}= A_{I,K} B_{K,J}$.
  - This takes $16^3 = 4096$ fused multiply-adds.
  - Loading $A_{I,K}$ fetches $16^2 = 256$ floats from global memory.
  - Likewise for $B_{K,J}$. Total of 512 floats fetched.
  - $\mathrm{CGMA} \;=\; 2 * 4096/512 \;=\; 16$.
- Note: the L2 cache may help here: A and B are read-only.
  - Need to try more experiments.

Matrix Multiply: Notes

- CGMA of 16 is much better, but still not the 82 that we need.
- Many graphics and machine learning applications work with low-precision arithmetic.
  - Use 16-bit floating point numbers.
  - Can load twice as many float16's per second.
  - Can do twice as many float16 operations.
  - Can hold more in shared memory. We've improved CGMA some.
- Tune the algorithm: the blocks we use for `A` and `B` don't have to be square.
- nVidia GPUs seem to be designed to the point that matrix multiplication balances floating point throughput with memory bandwidth for a carefully optimized implementation.

Shared Memory: Collisions



- When one thread in a warp accesses shared memory, **all** active threads in the warp access shared memory.
- If each thread accesses a different bank, then all accesses are performed in a single cycle.
  - Otherwise, the load or store can take multiple cycles.
  - Multiple accesses to the same bank are called **collisions**.
  - The **worst-case** occurs when **all threads access different locations in the same bank**.
- The programmer needs to think about the index calculations to avoid collisions.
  - When programming GPUs, the programmer needs to think about index calculations a lot.

Global Memory

- Off-chip DRAM
  - GDDR supports higher-bandwidth than than regular DDR.
  - A GPU can have multiple memory interfaces.
  - Total bandwidth 80 to 484$+$ GBytes/sec
- Memory accesses can be a big bottleneck.
  - CGMA: compute to global memory access ratio

Now for a word about DRAM

The memory that you plug into your computer is mounted on DIMMs (dual-inline memory modules).

- A DIMM typically has 16 or 18 chips
- E.g. each chip of an 8Gbyte DIMM holds 512MBytes $=$ 4Gbits.
- Each chip consists of many “tiles”,
  - a typical chip has 1Mbit/tile
  - that’s 4096 tiles for a 4Gbit chip.
- Each tile is an array of capacitors.
  - each capacitor holds 1 bit.
  - a typical tile could have 1024 rows and 1024 columns.

Writing and reading DRAM

- Writing: easy
  - drive all 1024 column-lines to the values you want to write.
  - open up all the valves for one row.
  - the drinking cups for each column in that row get filled or emptied.
  - note: you end up writing **every** column in the row; so writes are often preceded by reads.
- Reading: hard
  - drive all 1024 column-lines to "half-way", and let them "float".
  - open up all the valves for one row.
  - if the level in the pipe goes up a tiny amount, that cup held a 1.
  - if the level in the pipe goes down a tiny amount, that cup held a 0.
  - it's a delicate measurement – it takes time to set it up.
  - This is why DRAM is slow.
- But: we just read 1024 bits, from each chip of the DIMM.
  - That's 16Kbits $=$ 2Kbytes total.
  - Conclusion: DRAM has awful latency, but we can get very good bandwidth.
    - ★ The bandwidth bottleneck is the wires from the DIMM to the CPU or GPU.
    - ★ But I'm pretty sure that Ian won't let me give a lecture on transmission lines, phase-locked loops, equalizers, and all the other cool stuff in the DDR (or GDDR) interface.

GPUs meet DRAM

- DRAM summary: terrible latency (60-200ns or more), fairly high bandwidth.
- The GPU lets the program take advantage of high bandwidth.
  - If the 32 loads from a warp access 32 consecutive memory location,
    - The GPU does **one** GDDR access,
    - and it transfers a large block of data.
  - The same optimization is applied to stores, and to loads from the on-chip caches.
- In CUDA-speak, if the loads from a warp access consecutive locations, we say that the memory accesses are **coalesced**.
- It's a big deal to make sure that your memory accesses are coalesced.
  - Note that the memory optimizations are exposed to the programmer.
  - You can get the performance by considering the memory model.
  - But, it's not automatic.

Example: Matrix Multiplication

- In C, matrices are usually stored in row-major order.
  - `A[i,k]` and `A[i,k+1]` are at adjacent locations, but
  - `B[k,j]` and `B[k+1,j]` are N words apart (for $N \times N$ matrices).
- For matrix multiplication, accesses to `A` are naturally coalesced, but accesses to `B`.
- The optimized code loads a block of `B` into shared memory.
  - This allows accesses to be coalesced.
  - But we need to be careful about how we store the data in the shared memory to avoid bank conflicts.

Other Memory

- Constant memory: cached, read-only access of global memory.
- Texture memory: global memory with special access operations.
- L1 and L2 caches: only for memory reads.

Summary

- GPUs can have thousands of execution units, but only a few off-chip memory interfaces.
  - This means that the GPU can perform 10-50 floating point operations for every memory read or write.
  - Arithmetic operations are very cheap compared with memory operations
- To mitigate the off-chip memory bottleneck
  - GPUs have, limited on-chip memory
  - Registers and the per-block, shared-memory will be our main concerns in this class.
- Moving data between different kinds of storage is the programmer's responsibility.
  - The programmer explicitly declares variables to be stored in shared memory.
  - The programmer needs to be aware of the per-thread register usage to achieve good SM utilization.
  - The only way to communicate between thread blocks is to write to global memory, end the kernel, and start a new kernel (ouch!)

Preview

November 16: GPU Mamory: Part 2
November 19: GPU Performance: Part 1 Reading: Kirk & Hwu – Chapter 5
November 21: GPU Performance: Part 2
November 23: GPU Performance: Part 3

Review

- What is CGMA?
- On slide 13 we computed the CGMA for matrix-multiplication using 16×16 blocks of the A, B, and C matrices.
  - How many such thread-blocks can execute concurrently on an SM with 48KBytes of memory?
  - How does the CGMA change if we use 32×32 blocks?
  - If we use the larger matrix-blocks, how many thread blocks can execute concurrently on an SM with 48Kbytes of memory?
  - If we use the larger matrix-blocks, how many thread blocks can execute concurrently on an SM with 96Kbytes of memory?
- What are bank conflicts?
- How can increasing the number of registers used by a thread improve performance?
- How can increasing the number of registers used by a thread degrade performance?
- What is a “coalesced memory access”?