GPU Performance Idiosyncracies

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – March 12, 2018

Unless otherwise noted or cited, these slides are copyright 2018 by Mark Greenstreet & Ian M. Mitchell and are made available under the terms of the Creative Commons Attribution 4.0 International license
`http://creativecommons.org/licenses/by/4.0/`

Table of Contents

Outline

Memory Layout

Memories are arranged in 2D arrays of identical cells.

- Each cell stores one (or a small number) of bits.
  - Size of cell determines density of memory (and hence cost).
  - SRAM cells (cache memory) typically use 6 transitors / bit.
  - DRAM cells (main / global memory) require 1 transistor and 1 capacitor / bit.
- Extra logic arranged around edges of array.
  - Demultiplexors to activate appropriate row and column.
  - Amplifiers to make small signals bigger.
  - Address and data I/O lines.
- Array size is limited by length of row and column wires.
  - Longer wires have more capacitance.
  - More capacitance requires more power and/or time to change voltage and hence signal value.
- Bandwidth of each array is limited by time to read / write a cell.

Additional DRAM Headaches

SRAM arrays: Access any one cell per cycle.

DRAM arrays: It's complicated…

- Precharge two "bit-lines" for each column to $1/2$ operating voltage.
- Isolate bit-lines and charge appropriate row access line.
- All cells on selected row connect capacitors to one pre-charged bit-line.
  - If capacitor is empty ("0"), bit-line voltage drops a tiny amount.
  - If capacitor is full ("1"), bit-line voltage rises a tiny amount.
- Sense-amplifiers at bottom of columns use positive feedback to drive this slight imbalance toward a complete imbalance.
- Latches attached to bit-lines capture result.
- Bit-lines refresh capacitors.
- Data from one or more columns is sent from latches.

See Wikipedia's DRAM entry and basic structure diagram

Yet Another Memory Hierarchy

Implications of DRAM design on access:

- Limited data I/O from any single memory array.
- Large delay between requesting a memory location from a DRAM array and receiving the data.
- We can use the I/O lines for other accesses while we wait.

To achieve large, high bandwidth memory build a hierarchy.

- Memory array.
- Banks: One or more memory arrays that share address lines.
- Chips: Multiple banks that share address lines.
- Ranks: Multiple chips that share address lines.
- Module (DIMM): One or more ranks
- Channel: One or more modules.

Why So Many Layers?

Bandwidth = frequency $\times$ (occupied) bus width.

- High frequency limits number of elements on each wire.
- Use multiple channels to access more modules in parallel.
- Use multiple ranks to increase capacity of each module.
- Use multiple chips to increase width of each transfer.
- Use multiple banks to allow overlapping accesses.
- Use multiple arrays to provide multiple bits.

The good news: Efficient memory access patterns can be designed based only on properties of channels and banks (see K&H figure 5.7).

Implications of a Read on the Memory Subsystem

```
uint j = blockDim.x * blockIdx.x + threadIdx.x;
if(j < n) {
  uint x_less2 = x0[j];
    ⋮
```

- At least 32 threads (one warp) execute read at the same time.
- Cache(s) identify missed cache lines.
- If memory locations are consecutive, 1–2 cache lines needed.
    - Called a “coalesced” memory access.
- If memory locations are **not** consecutive:
    - More cache lines must be loaded (requires more memory bus transactions).
    - Some cache line data may not be used (wasted memory bandwidth).

Implications of Coalescing on GPU Programming

Arrange thread and data layout so that consecutive threads access consecutive data.

- Threads that differ in `threadIdx.x` should access consecutively stored data.

Use shared memory to rearrange inconvenient access patterns ("corner turning").

- Consider matrix-matrix multiply $C = AB$ (row-major storage).
  - Thread for element $c_{i,j}$ assigned indexes by:

    ```
    uint i = blockIdx.x*blockDim.x + threadIdx.x;
    uint k = blockIdx.y*blockDim.y + threadIdx.y;
    ```
  - Consecutive threads will access one column of A: not good (K&H figure 5.4).
  - Consecutive threads will access one row of B: good (K&H figure 5.3).
  - To allow coalescing, make sure that both A and B are loaded into shared memory one row at a time (K&H figure 5.5 and 5.6).

Distributing Memory Accesses

Banks and channels must fulfill memory requests (K&H figures 5.9 and 5.8).

- In Pascal architecture (CC 6.x) consecutive 32B L1 cache “sectors” (1/4 of a cache line) are mapped to different channels.
  - Ensuring that consecutive sectors are accessed together allows simultaneous use of multiple channels.
  - GTX 1060 3GB has 6 channels each with 32 GB/s bandwidth.
- After all channels are assigned consecutive sectors, next set of sectors is assigned to different banks in the same channels.
  - DRAM access delay for different banks can be overlapped.
  - To keep data bus fully utilized requires many overlapping requests.
- Moral of the story: Coalesced access is good.
  - Reduces number of cache lines to load.
  - Increases number of channels in use simultaneously.
  - Increases overlapping of DRAM access delay to different banks.

Collisions

- Multiple simultaneous accesses to a single channel.
- Multiple simultaneous or consecutive accesses within the DRAM access delay to a single bank.

Global memory (off-chip DRAM):

- Coalesced memory accesses will avoid collisions.
- Many banks reduce likelihood of bank collision.
- Each channel needs many active requests to fully utilize bandwidth, so channel collisions are good(?)

Shared memory (on-chip SRAM):

- Divided into 32 “banks” (more like global memory’s channels).
- No need to account for DRAM access delay.
- Each bank stores 4 consecutive bytes.
- Each bank has bandwidth 4B / cycle.
- Each thread can access any bank.

Avoiding Shared Memory Collisions

- Simple: Consecutive threads access consecutive data.
- Also efficient: Consecutive threads access exactly the same data.
- Also efficient: Consecutive threads access data in a pattern which hits different banks modulo 32; for example, any odd stride.
- Avoid: Consecutive threads access data with a power of two stride.

See Figure 17 and Figure 18 from Nvidia programming guide.

Outline

SIMD, Warps and Thread Divergence

All CUDA threads are scheduled in batches (“warps”) of 32 based on thread index (always `threadIdx.x` first, see K&H figure 5.12).

- One instruction sent to all threads in the warp each cycle; each thread may either perform that instruction or a no-op.
    - If one or more threads in a warp need to execute an instruction, that whole warp will consume resources.
    - If zero threads in a warp need to execute an instruction, that warp will not consume resources.
- Ensure that blocks always contain a multiple of 32 threads.
- If branching (“control divergence”) is necessary, try to reduce the number of warps executing different branches.
    - Relatively easy if branching is based on thread index.
- Example: Reduce
    - If neighboring threads handle neighboring elements, every warp has divergence (K&H figure 5.14).
    - If neighboring threads handle maximally separated elements, no more than one warp has divergence (K&H figure 5.16).
    - Aside: Which has better global memory access pattern?

How to Choose Block Size?

Threads within a block can share memory and synchronize, so bigger is better? Not necessarily!

- Each block must be scheduled onto a single SM.
- Each block has limits (limits for CC 6.1):
  - Number of resident threads (1024).
  - Number of registers (65536).
  - Amount of shared memory (48 KB).
- Each SM has limits (limits for CC 6.1):
  - Number of resident blocks (32).
  - Number of resident warps (64).
  - Number of resident threads (2048).
  - Number of registers (65536).
  - Amount of shared memory (96 KB).
  - Cache working set for constant memory (10 KB).
  - Number of ALUs (128).
- Poorly chosen block size can result in idle SM resources.

How Much Work per Thread?

More threads gives more options for scheduling, so more threads is better? Not necessarily!

- Can more work / thread increase CGMA?
  - Tiled matrix multiply: Have each thread compute two output elements in different output tiles.
  - 200% computation, 150% global memory accesses (one input tile is reused).
- Can more work / thread increase independent operations?
  - Tiled matrix multiply: Have each thread compute one or four output elements in the same tile.
  - 32×32 tiles: 8 KB shared memory and 1024 or 256 threads / block.
  - SMs can host either 2 blocks or 8 blocks.
- But more work per thread may increase resource use per thread and hence constrain block size and/or reduce residency of SMs.
  - More work often requires more registers, which can limit number of active threads / SM.