EE382N (20): Computer Architecture - Parallelism and Locality
Fall 2011
Lecture 20 – GPUs (V)

Mattan Erez

The University of Texas at Austin
Multiply Using Several Blocks

- One **block** computes one square sub-matrix $P_{\text{sub}}$ of size $\text{BLOCK\_SIZE}$
- One **thread** computes one element of $P_{\text{sub}}$
- Assume that the dimensions of $M$ and $N$ are multiples of $\text{BLOCK\_SIZE}$ and square shape
How addresses map to banks on G80

• Each bank has a bandwidth of 32 bits per clock cycle
• Successive 32-bit words are assigned to successive banks
• G80 has 16 banks
  – So bank = address % 16
  – Same as the size of a half-warp
    • No bank conflicts between different half-warps, only within a single half-warp
Shared memory bank conflicts

• Shared memory is as fast as registers if there are no bank conflicts

• The fast case:
  – If all threads of a half-warp access different banks, there is no bank conflict
  – If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)

• The slow case:
  – Bank Conflict: multiple threads in the same half-warp access the same bank
  – Must serialize the accesses
  – Cost = max # of simultaneous accesses to a single bank
Data types and bank conflicts

- This has no conflicts if type of shared is 32-bits:

  \[
  \text{foo} = \text{shared}[\text{baseIndex} + \text{threadIdx.x}]
  \]

- But not if the data type is smaller:
  - 4-way bank conflicts:
    __shared__ char shared[];
    \[
    \text{foo} = \text{shared}[\text{baseIndex} + \text{threadIdx.x}]
    \]
  - 2-way bank conflicts:
    __shared__ short shared[];
    \[
    \text{foo} = \text{shared}[\text{baseIndex} + \text{threadIdx.x}]
    \]
Structs and Bank Conflicts

- Struct assignments compile into as many memory accesses as there are struct members:

  ```
  struct vector { float x, y, z; };
  struct myType {
    float f;
    int c;
  };
  __shared__ struct vector vectors[64];
  __shared__ struct myType myTypes[64];
  ```

- This has no bank conflicts for vector; struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 16)

  ```
  struct vector v = vectors[baseIndex + threadIdx.x];
  ```

- This has 2-way bank conflicts for my Type; (2 accesses per thread)

  ```
  struct myType m = myTypes[baseIndex + threadIdx.x];
  ```
Common Array Bank Conflict Patterns 1D

- Each thread loads 2 elements into shared mem:
  - 2-way-interleaved loads result in 2-way bank conflicts:

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.
  - Not in shared memory usage where there is no cache line effects but banking effects.
A Better Array Access Pattern

- Each thread loads one element in every consecutive group of blockDim elements.

\[
\text{shared}[\text{tid}] = \text{global}[\text{tid}]; \\
\text{shared}[\text{tid} + \text{blockDim.x}] = \text{global}[\text{tid} + \text{blockDim.x}];
\]
Common Bank Conflict Patterns (2D)

- Operating on 2D array of floats in shared memory
  - e.g. image processing
- Example: 16x16 block
  - Each thread processes a row
  - So threads in a block access the elements in each column simultaneously (example: row 1 in purple)
  - 16-way bank conflicts: rows all start at bank 0

- Solution 1) pad the rows
  - Add one float to the end of each row
- Solution 2) transpose before processing
  - Suffer bank conflicts during transpose
  - But possibly save them later
Load/Store (Memory read/write) Clustering/Batching

- Use LD to hide LD latency (non-dependent LD ops only)
  - Use same thread to help hide own latency

- Instead of:
  - LD 0 (long latency)
  - Dependent MATH 0
  - LD 1 (long latency)
  - Dependent MATH 1

- Do:
  - LD 0 (long latency)
  - LD 1 (long latency - hidden)
  - MATH 0
  - MATH 1

- Compiler handles this!
  - But, you must have enough non-dependent LDs and Math
Memory Coalescing

• Modern DRAM channels return a large burst of data on each access
  – 64B in the case of FERMI

• Unused data in a burst degrades effective (and scarce) memory throughput

• Reduce this waste in two ways:
  – Use caches and hope they are effective
    • Cache lines are 128B in Fermi
  – Explicitly optimize for spatial locality
    • Memory coalescing hardware minimizes the number of DRAM transactions from a single warp
    • Essentially per-warp MSHRs when no cache available

• Memory coalescing deprecated in Fermi?
  – Use caches for coalescing
    • 128B if L1 cacheable, but only 32B if cached only at L2
  – Section F.4.2 of CUDA Programmer Guide 4.0
Host memory

• Board discussion of PCI express, memory mapping, and direct GPU/host access
• More information available in Section 3.2.4 of the CUDA Programming Guide
**Bandwidths of GeForce 9800 GTX**

- **Frequency**
  - 600 MHz with ALUs running at 1.2 GHz

- **ALU bandwidth (GFLOPs)**
  - \((1.2 \text{ GHz}) \times (16 \text{ SM}) \times ((8 \text{ SP}) \times (2 \text{ MADD}) + (2 \text{ SFU})) = \sim 400 \text{ GFLOPs}\)

- **Register BW**
  - \((1.2 \text{ GHz}) \times (16 \text{ SM}) \times (8 \text{ SP}) \times (4 \text{ words}) = 2.5 \text{ TB/s}\)

- **Shared Memory BW**
  - \((600 \text{ MHz}) \times (16 \text{ SM}) \times (16 \text{ Banks}) \times (1 \text{ word}) = 600 \text{ GB/s}\)

- **Device memory BW**
  - 2 GHz GDDR3 with 256 bit bus: 64 GB/s

- **Host memory BW**
  - PCI-express: 1.5GB/s or 3GB/s with page locking
Bandwidths of GeForce GTX480/Tesla c2050

- **Frequency**
  - 700/650 MHz with ALUs running at 1.4/1.3 GHz

- **ALU bandwidth (GFLOPs)**
  - \((1.4 \text{ GHz}) \times (15 \text{ SM}) \times (32 \text{ SP} \times 2 \text{ MADD})\) = \(\sim 1.3 \text{ TFLOPs}\)
  - \((1.3 \text{ GHz}) \times (14 \text{ SM}) \times (32 \text{ SP} \times 2 \text{ MADD})\) = \(\sim 1.1 \text{ TFLOPs}\)

- **Register BW**
  - \((1.4 \text{ GHz}) \times (15 \text{ SM}) \times (32 \text{ SP} \times 4 \text{ words})\) = \(\sim 2.6 \text{ TFLOPs}\)
  - \((1.3 \text{ GHz}) \times (14 \text{ SM}) \times (32 \text{ SP} \times 4 \text{ words})\) = \(\sim 2.2 \text{ TB/s}\)

- **Shared Memory BW**
  - \((700 \text{ MHz}) \times (15 \text{ SM}) \times (32 \text{ Banks}) \times (1 \text{ word}) = 1.3 \text{ GB/s}\)
  - \((650 \text{ MHz}) \times (14 \text{ SM}) \times (32 \text{ Banks}) \times (1 \text{ word}) = 1.1 \text{ GB/s}\)

- **Device memory BW**
  - GTX480: 3.6 Gb/s GDDR5 with 6 64-bit channels: 177 GB/s
  - C2050: 3 Gb/s GDDR5 with 6 64-bit channels: 144 GB/s (less with ECC on)

- **Host memory BW**
  - PCI-express 2.0: Effective bandwidth is 1.5GB/s or 3GB/s with page locking
Communication

• How do threads communicate?

• Remember the execution model:
  – Data parallel streams that represent independent vertices, triangles, fragments, and pixels in the graphics world
  – These never communicate

• Some communication allowed in compute mode:
  – Shared memory for threads in a thread block
    • No special communication within warp or using registers
  – No communication between thread blocks
    • Except atomics (and derivatives)
  – Kernels communicate through global device memory

• Mechanisms designed to ensure portability
Synchronization

• Do threads need to synchronize?
  – Basically no communication allowed

• Threads in a block share memory – need sync
  – Warps scheduled OoO, can’t rely on warp order
  – Barrier command for all threads in a block
  – __syncthreads()

• Blocks should not synchronize
  – Implicit synchronization at end of kernel
  – Use atomics to form your own primitives
  – Also need to use memory fences
    • Block, device, device+host

• GPUs use streaming rather than parallel concurrency
  – Not safe to assume which, or even how many CTAs are “live” at any given time, although usually works in reality
  – Communication could deadlock, but could be OK
Atomic Operations

• Exception to communication between blocks
• Atomic read-modify-write
  – Shared memory
  – Global memory
    • Executed in each L2 bank
• Simple ALU operations
  – Add, subtract, AND, OR, min, max, inc, dec
• Exchange operations
  – Compare-and-swap, exchange