EE382N (20): Computer Architecture - Parallelism and Locality
Fall 2011
Lecture 19 – GPUs (IV)

Mattan Erez

The University of Texas at Austin
Make the Compute Core The Focus of the Architecture

- The future of GPUs is programmable processing
- So build the architecture around the processor

Manages thread blocks
Used to be only one kernel at a time
SM Memory Architecture

- Registers in SP
  - 1K total per SP
    - shared between thread
    - same per thread in a block)

- Shared memory in SM
  - 16KB total per SM
    - shared between blocks

- Global memory
  - Managed by Texture Units
    - Cache – read only
  - Managed by LD/ST ROP units
    - Uncached – read/Write

Courtesy: John Nicols, NVIDIA
Matrix Multiplication Example

• If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  – Each Block requires 10*256 = 2560 registers
  – 8192 = 3 * 2560 + change
  – So, three blocks can run on an SM as far as registers are concerned

• How about if each thread increases the use of registers by 1?
  – Each Block now requires 11*256 = 2816 registers
  – 8192 < 2816 *3
  – Only two Blocks can run on an SM, 1/3 reduction of parallelism!!!
More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, global loads have 200 cycles
  - 3 Blocks can run on each SM

- If a Compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load
  - Only two can run on each SM
  - However, one only needs $200/(8*4) = 7$ Warps to tolerate the memory latency
  - Two Blocks have 16 Warps. The performance can actually be higher!
SM Memory Architecture

- **Registers in SP**
  - 1K total per SP
    - shared between thread
    - same per thread in a block

- **Shared memory in SM**
  - 16KB total per SM
    - shared between blocks

- **Global memory**
  - Managed by Texture Units
    - Cache – read only
  - Managed by LD/ST ROP units
    - Uncached – read/Write
Constants

• Immediate address constants
• Indexed address constants
• Constants stored in DRAM, and cached on chip
  – L1 per SM
• A constant value can be broadcast to all threads in a Warp
  – Extremely efficient way of accessing a value that is common for all threads in a Block!

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Textures

- Textures are 2D arrays of values stored in global DRAM
- Textures are cached in L1 and L2
- Read-only access
- Caches optimized for 2D access:
  - Threads in a warp that follow 2D locality will achieve better memory performance
SM Memory Architecture

- Registers in SP
  - 1K total per SP
    - shared between thread
    - same per thread in a block

- Shared memory in SM
  - 16KB total per SM
    - shared between blocks

- Global memory
  - Managed by Texture Units
    - Cache – read only
  - Managed by LD/ST ROP units
    - Uncached – read/Write

Courtesy: John Nicols, NVIDIA

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Shared Memory

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32bit words

- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - read and write access

- Not used explicitly for pixel shader programs
  - we dislike pixels talking to each other 😞
Multiply Using Several Blocks

- One block computes one square sub-matrix $P_{\text{sub}}$ of size $\text{BLOCK}_\text{SIZE}$
- One thread computes one element of $P_{\text{sub}}$
- Assume that the dimensions of $M$ and $N$ are multiples of $\text{BLOCK}_\text{SIZE}$ and square shape
Matrix Multiplication
Shared Memory Usage

• Each Block requires $2 \times \text{WIDTH}^2 \times 4$ bytes of shared memory storage
  - For \text{WIDTH} = 16, each BLOCK requires 2KB, up to 8 Blocks can fit into the Shared Memory of an SM
  - Since each SM can only take 768 threads, each SM can only take 3 Blocks of 256 threads each
  - Shared memory size is not a limitation for Matrix Multiplication of

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Parallel Memory Architecture

• In a parallel machine, many threads access memory
  – Therefore, memory is divided into banks
  – Essential to achieve high bandwidth

• Each bank can service one address per cycle
  – A memory can service as many simultaneous accesses as it has banks

• Multiple simultaneous accesses to a bank result in a bank conflict
  – Conflicting accesses are serialized
Bank Addressing Examples

- **No Bank Conflicts**
  - Linear addressing
    - stride == 1

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
<th>Thread 5</th>
<th>Thread 6</th>
<th>Thread 7</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
<th>Thread 5</th>
<th>Thread 6</th>
<th>Thread 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bank 0</td>
<td>Bank 1</td>
<td>Bank 2</td>
<td>Bank 3</td>
<td>Bank 4</td>
<td>Bank 5</td>
<td>Bank 6</td>
<td>Bank 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bank 15</td>
<td>Bank 0</td>
<td>Bank 1</td>
<td>Bank 2</td>
<td>Bank 3</td>
<td>Bank 4</td>
<td>Bank 5</td>
<td>Bank 6</td>
</tr>
</tbody>
</table>

- **No Bank Conflicts**
  - Random 1:1 Permutation

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

Computer Architecture, Fall 2011 -- Lecture 19 (c) Mattan Erez
Bank Addressing Examples

- **2-way Bank Conflicts**
  - Linear addressing
  - stride == 2

- **8-way Bank Conflicts**
  - Linear addressing
  - stride == 8
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
Linear Addressing

- Given:

```c
__shared__ float shared[256];
float foo =
    shared[baseIndex + s * threadIdx.x];
```

- This is only bank-conflict-free if `s` shares no common factors with the number of banks
  - 16 on G80, so `s` must be odd
Data types and bank conflicts

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

```c
foo = shared[baseIndex + threadIdx.x]
```

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

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

```c
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)

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

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

```c
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:

```c
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.

\[
\text{shared}[\text{tid}] = \text{global}[\text{tid}]; \\
\text{shared}[\text{tid} + \text{blockDim.x}] = \text{global}[\text{tid} + \text{blockDim.x}];
\]
Vector Reduction with Bank Conflicts

Array elements

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
No Bank Conflicts
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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
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