Programming Massively Parallel Processors

Lecture Slides for Chapter 6: Performance Considerations
Threading Hardware
Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks

CPU Serial Code

GPU Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);

CPU Serial Code

GPU Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);
Grids and Blocks

• A kernel is executed as a grid of thread blocks
  – All threads share global memory space
• A thread block is a batch of threads that can cooperate with each other by:
  – Synchronizing their execution using barrier
  – Efficiently sharing data through a low latency shared memory
  – Two threads from two different blocks cannot cooperate
CUDA Thread Block: Review

• Programmer declares (Thread) Block:
  – Block size 1 to 512 concurrent threads
  – Block shape 1D, 2D, or 3D
  – Block dimensions in threads

• All threads in a Block execute the same thread program
• Threads share data and synchronize while doing their share of the work
• Threads have thread id numbers within Block
• Thread program uses thread id to select work and address shared data

Courtesy: John Nickolls, NVIDIA
GeForce-8 Series HW Overview

Streaming Processor Array

Texture Processor Cluster

Streaming Multiprocessor
- Instruction L1
- Data L1
- Instruction Fetch/Dispatch
- Shared Memory
  - SP
  - SFU

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
CUDA Processor Terminology

- **SPA**
  - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)

- **TPC**
  - Texture Processor Cluster (2 SM + TEX)

- **SM**
  - Streaming Multiprocessor (8 SP)
  - Multi-threaded processor core
  - Fundamental processing unit for CUDA thread block

- **SP**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- texture and global memory access
G80 Thread Computing Pipeline

• Processors execute computing threads
• Alternative operating mode specifically for computing

Generates Thread grids based on kernel calls

Host
Input Assembler

Vtx Thread Issue
Geom Thread Issue
Pixel Thread Issue

Setup / Rstr / ZCull

408, University of Illinois, Urbana-Champaign
Thread Life Cycle in HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SM’s
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
SM Executes Blocks

- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution
Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
- If one global memory access is needed for every 4 instructions
- A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency
SM Instruction Buffer – Warp Scheduling

• Fetch one warp instruction/cycle
  – from instruction L1 cache
  – into any instruction buffer slot

• Issue one “ready-to-go” warp instruction/cycle
  – from any warp - instruction buffer slot
  – operand scoreboarding used to prevent hazards

• Issue selection based on round-robin/age of warp

• SM broadcasts the same instruction to 32 Threads of a Warp
Scoreboarding

• All register operands of all instructions in the Instruction Buffer are scoreboarded
  – Instruction becomes ready after the needed values are deposited
  – prevents hazards
  – cleared instructions are eligible for issue

• Decoupled Memory/Processor pipelines
  – any thread can continue to issue instructions until scoreboarding prevents issue
  – allows Memory/Processor ops to proceed in shadow of other waiting Memory/Processor ops
Granularity Considerations

• For Matrix Multiplication, should I use 4X4, 8X8, 16X16 or 32X32 tiles?
  – For 4X4, we have 16 threads per block. Since each SM can take up to 768 threads, the thread capacity allows 48 blocks. However, each SM can only take up to 8 blocks, thus there will be only 128 threads in each SM!
    • There are 8 warps but each warp is only half full.

  – For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, it could take up to 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
    • There are 16 warps available for scheduling in each SM
    • Each warp spans four slices in the y dimension

  – For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
    • There are 24 warps available for scheduling in each SM
    • Each warp spans two slices in the y dimension

  – For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!
Memory Hardware
CUDA Device Memory Space: Review

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories
Parallel Memory Sharing

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication
SM Memory Architecture

- Threads in a block share data & results
  - In Memory and Shared Memory
  - Synchronize at barrier instruction
- Per-Block Shared Memory Allocation
  - Keeps data close to processor
  - Minimize trips to global Memory
  - Shared Memory is dynamically allocated to blocks, one of the limiting resources

Credits:
- John Nicols, NVIDIA
- David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
SM Register File

- Register File (RF)
  - 32 KB (8K entries) for each SM in G80
- TEX pipe can also read/write RF
  - 2 SMs share 1 TEX
- Load/Store pipe can also read/write RF
Programmer View of Register File

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all blocks assigned to the SM
  - Once assigned to a block, the register is NOT accessible by threads in other blocks
  - Each thread in the same block only access registers assigned to itself
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
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 $200/(8\times4) = 7$ Warps to tolerate the memory latency
  – Two blocks have 16 Warps. The performance can be actually higher!
Memory Layout of a Matrix in C

\[ \begin{bmatrix}
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3}
\end{bmatrix} \]
Memory Coalescing

- When accessing global memory, peak performance utilization occurs when all threads in a half warp access continuous memory locations.
Memory Layout of a Matrix in C

Access direction in Kernel code

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
Memory Layout of a Matrix in C

Access direction in Kernel code

Time Period 1

Time Period 2

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
Use Shared Memory to Improve Coalescing

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values
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!
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 😊
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

  Thread 0
  Thread 1
  Thread 2
  Thread 3
  Thread 4
  Thread 5
  Thread 6
  Thread 7
  ...
  Thread 15

  Bank 0
  Bank 1
  Bank 2
  Bank 3
  Bank 4
  Bank 5
  Bank 6
  Bank 7
  ...
  Bank 15

• No Bank Conflicts
  – Random 1:1 Permutation

  Thread 0
  Thread 1
  Thread 2
  Thread 3
  Thread 4
  Thread 5
  Thread 6
  Thread 7
  ...
  Thread 15

  Bank 0
  Bank 1
  Bank 2
  Bank 3
  Bank 4
  Bank 5
  Bank 6
  Bank 7
  ...
  Bank 15
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
Control Flow
Control Flow Instructions

• Main performance concern with branching is divergence
  – Threads within a single warp take different paths
  – Different execution paths are serialized in G80
    • The control paths taken by the threads in a warp are traversed one at a time until there is no more.

• A common case: avoid divergence when branch condition is a function of thread ID
  – Example with divergence:
    • If (threadIdx.x > 2) { }
    • This creates two different control paths for threads in a block
    • Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  – Example without divergence:
    • If (threadIdx.x / WARP_SIZE > 2) { }
    • Also creates two different control paths for threads in a block
    • Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
Parallel Reduction

• Given an array of values, “reduce” them to a single value in parallel

• Examples
  – sum reduction: sum of all values in the array
  – Max reduction: maximum of all values in the array

• Typically parallel implementation:
  – Recursively halve # threads, add two values per thread
  – Takes log(n) steps for n elements, requires n/2 threads
A Vector Reduction Example

• Assume an in-place reduction using shared memory
  – The original vector is in device global memory
  – The shared memory used to hold a partial sum vector
  – Each iteration brings the partial sum vector closer to the final sum
  – The final solution will be in element 0
A simple implementation

- Assume we have already loaded array into

\[
\text{__shared__ float partialSum[]}\\
\]

\[
\text{unsigned int t = threadIdx.x; }\\
\text{for (unsigned int stride = 1; }\\
\text{ stride < blockDim.x; stride *= 2) }\\
\{\\
\text{__syncthreads(); }\\
\text{if (t \% (2*stride) == 0) }\\
\text{ partialSum[t] += partialSum[t+stride]; }\\
\}\\
\]
Vector Reduction with Branch Divergence

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0  1  2  3  4  5  6  7  8  9  10  11

0+1  2+3  4+5  6+7  8+9  10+11

0...3  4..7  8..11

0..7  8..15

Array elements
Some Observations

• In each iterations, two control flow paths will be sequentially traversed for each warp
  – Threads that perform addition and threads that do not
  – Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

• No more than half of threads will be executing at any time
  – All odd index threads are disabled right from the beginning!
  – On average, less than $\frac{1}{4}$ of the threads will be activated for all warps over time.
  – After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    • This can go on for a while, up to 4 more iterations ($512/32=16=2^4$), where each iteration only has one thread activated until all warps retire
Shortcomings of the implementation

• Assume we have already loaded array into
  
  ```
  __shared__ float partialSum[]
  ```

  ```
  unsigned int t = threadIdx.x;
  for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
  {
    __syncthreads();
    if (t % (2*stride) == 0)
      partialSum[t] += partialSum[t+stride];
  }
  ```

BAD: Divergence due to interleaved branch decisions
A better implementation

• Assume we have already loaded array into
  __shared__ float partialSum[]

  unsigned int t = threadIdx.x;
  for (unsigned int stride = blockDim.x;
       stride > 1; stride >>= 1)
  {
    __syncthreads();
    if (t < stride)
      partialSum[t] += partialSum[t+stride];
  }
No Divergence until < 16 sub-sums
Registers, ILP and Instruction Mix
Programmer View of Register File

• There are 8192 registers in each SM in G80
  – This is an implementation decision, not part of CUDA
  – Registers are dynamically partitioned across all Blocks assigned to the SM
  – Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  – Each thread in the same Block only access registers assigned to itself
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 $10 \times 256 = 2560$ registers
  - $8192 = 3 \times 2560 + \text{change}$
  - 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 $11 \times 256 = 2816$ registers
  - $8192 < 2816 \times 3$
  - Only two Blocks can run on an SM, 1/3 reduction of thread-level parallelism (TLP)!!!
More on Dynamic Partitioning

• Dynamic partitioning of SM resources 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
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 $200/(8\times4) = 7$ Warps to tolerate the memory latency
  – Two Blocks have 16 Warps. The performance can be actually higher!
Resource Allocation Example

(a) Pre-“optimization”

(b) Post-“optimization”

Increase in per-thread performance, but fewer threads:
Lower overall performance in this case
Prefetching

• One could double buffer the computation, getting better instruction mix within each thread
  – This is classic software pipelining in ILP compilers

```
Loop {
  Load current tile to shared memory
  syncthreads()
  Compute current tile
  syncthreads()
}

Load next tile from global memory

Loop {
  Load next tile from global memory
  Deposit current tile to shared memory
  syncthreads()
  Compute current tile
  syncthreads()
}
```
Prefetch

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
- ....
Instruction Mix Considerations

for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];

There are very few mul/add between branches and address calculation.

Loop unrolling can help.

Pvalue += Ms[ty][k] * Ns[k][tx] + ...
    Ms[ty][k+15] * Ns[k+15][tx];
Unrolling

Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads();

    // compute results for tile
    for (i = 0; i < 16; i++)
        { Ctemp += As[ty][i] * Bs[i][tx]; }
    __syncthreads();
}
C[indexC] = Ctemp;

(b) Tiled Version

Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads();

    // compute results for tile
    Ctemp +=
        As[ty][0] * Bs[0][tx];
    ...
    Ctemp +=
        As[ty][15] * Bs[15][tx];
    __syncthreads();
}
C[indexC] = Ctemp;

(c) Unrolled Version

Removal of branch instructions and address calculations

Does this use more registers?
Major G80 Performance Detractors

• Long-latency operations
  – Avoid stalls by executing other threads

• Stalls and bubbles in the pipeline
  – Barrier synchronization
  – Branch divergence

• Shared resource saturation
  – Global memory bandwidth
  – Local memory capacity