Thread Optimization

Slides adapted from the course at UIUC
by Wen-Mei Hwu and David Kirk
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

Courtesy: NDVIA
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
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)
  - 1 Double Precision ALU
- 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
GeForce-8 Series HW Overview

Streaming Processor Array

Texture Processor Cluster

Streaming Multiprocessor

Instruction Fetch/Dispatch

Shared Memory

Instruction L1

Data L1

TEX

SM

SM

SM

SM

SM

SM

SM

SM

SM

SM

SM

SM

SM

SFU

SFU

SFU

SFU
G80 Thread Computing Pipeline

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

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

TB = Thread Block, W = Warp
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!
Arithmetic Instruction Throughput

- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply

- Integer divide and modulo are expensive
  - Compiler will convert literal power-of-2 divides to shifts
  - Be explicit in cases where compiler can’t tell that divisor is a power of 2!
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2
Arithmetic Instruction Throughput

- Reciprocal, reciprocal square root, sin/cos, log, exp:
  16 cycles per warp
  - These are the versions prefixed with “__”
  - Examples: __rcp(), __sin(), __exp()

- Other functions are combinations of the above
  - $y / x = rcp(x) \times y$ == 20 cycles per warp
  - $\sqrt{x} = rcp(rsqrt(x))$ == 32 cycles per warp
Runtime Math Library

• There are two types of runtime math operations
  – __func(): direct mapping to hardware ISA
    • Fast but low accuracy (see prog. guide for details)
    • Examples: __sin(x), __exp(x), __pow(x,y)
  – func(): compile to multiple instructions
    • Slower but higher accuracy (5 ulp, units in the least place, or less)
    • Examples: sin(x), exp(x), pow(x,y)

• The -use_fast_math compiler option forces every func() to compile to __func()
Make your program float-safe!

- Future hardware will have double precision support
  - G80 is single-precision only
  - Double precision will have additional performance cost
  - Careless use of double or undeclared types may run more slowly on G80+

- Important to be float-safe (be explicit whenever you want single precision) to avoid using double precision where it is not needed
  - Add ‘f’ specifier on float literals:
    - foo = bar * 0.123; // double assumed
    - foo = bar * 0.123f; // float explicit

  - Use float version of standard library functions
    - foo = sin(bar); // double assumed
    - foo = sinf(bar); // single precision explicit
Deviations from IEEE-754

• Addition and Multiplication are IEEE 754 compliant
  – Maximum 0.5 ulp (units in the least place) error
• However, often combined into multiply-add (FMAD)
  – Intermediate result is truncated

• Division is non-compliant (2 ulp)
• Not all rounding modes are supported
• Denormalized numbers are not supported
• No mechanism to detect floating-point exceptions
# GPU Floating Point Features

<table>
<thead>
<tr>
<th></th>
<th>G80</th>
<th>SSE</th>
<th>IBM Altivec</th>
<th>Cell SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precision</strong></td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
</tr>
<tr>
<td><strong>Rounding modes for</strong></td>
<td>Round to nearest and round to zero</td>
<td>All 4 IEEE, round to nearest, zero, inf, -inf</td>
<td>Round to nearest only</td>
<td>Round to zero/truncate only</td>
</tr>
<tr>
<td><strong>Denormal handling</strong></td>
<td>Flush to zero</td>
<td>Supported, 1000’s of cycles</td>
<td>Supported, 1000’s of cycles</td>
<td>Flush to zero</td>
</tr>
<tr>
<td><strong>NaN support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Overflow and Infinity support</strong></td>
<td>Yes, only clamps to max norm</td>
<td>Yes</td>
<td>Yes</td>
<td>No, infinity</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Some</td>
</tr>
<tr>
<td><strong>Square root</strong></td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td>Software only</td>
<td>Hardware</td>
<td>Software only</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Reciprocal estimate accuracy</strong></td>
<td>24 bit</td>
<td>12 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>Reciprocal sqrt estimate accuracy</strong></td>
<td>23 bit</td>
<td>12 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>log2(x) and 2^x estimates accuracy</strong></td>
<td>23 bit</td>
<td>No</td>
<td>12 bit</td>
<td>No</td>
</tr>
</tbody>
</table>