Lecture 9

Thread Divergence
Scheduling
Instruction Level Parallelism
Announcements

• Tuesday’s lecture on 2/11 will be moved to room 4140 from 6.30 PM to 7.50 PM

• Office hours cancelled on Thursday; make an appointment to see me later this week

• CUDA profiling with nvprof

  See slides 13-16

Recapping optimized matrix multiply

- Volkov and Demmel, SC08
- Improve performance using fewer threads
  - Reducing concurrency frees up registers to trade locality against parallelism
  - ILP to increase processor utilization

Vector length: 64 //stripmined into two warps by GPU
Registers: a, c[1:16] //each is 64-element vector
Shared memory: b[16][16] //may include padding

Compute pointers in A, B and C using thread ID
c[1:16] = 0
do
  b[1:16][1:16] = next 16×16 block in B or B^T
  local barrier //wait until b[]] is written by all warps
  unroll for i = 1 to 16 do
    a = next 64×1 column of A
    c[1] += a*b[i][1] // rank-1 update of C’s block
    c[2] += a*b[i][2] // data parallelism = 1024
    c[3] += a*b[i][3] // stripmined in software
    ... // into 16 operations
    c[16] += a*b[i][16] // access to b[]] is stride-1
  endfor
  local barrier //wait until done using b[]]
  update pointers in A and B
repeat until pointer in B is out of range
Merge c[1:16] with 64×16 block of C in memory
Hiding memory latency

• **Parallelism = latency × throughput**

  Arithmetic: 576 ops/SM = 18CP x 32/SM/CP
  Memory: 150KB = ~500CP (1100 nsec) x 150 GB/sec

• **How can we keep 150KB in flight?**
  - Multiple threads: ~35,000 threads @ 4B/thread
  - ILP (increase fetches per thread)
  - Larger fetches (64 or 128 bit/thread)
  - Higher occupancy

Copy 1 float /thread, need 100% occupancy
```c
int indx = threadIdx.x + block * blockDim.x;
float a0 = src[indx];
dest[indx] = a0;
```

Copy 2 floats /thread, need 50% occ
```c
float a0 = src[indx];
float a1 = src[indx+blockDim.x];
dest[indx] = a0;
dst[indx+blockDim.x] = a1;
```

Copy 4 floats /thread, need 25% occ
```c
int indx = threadIdx.x + 4 * block * blockDim.x;
float a[4]; // in registers
for(i=0;i<4;i++) a[i]=src[indx+i*blockDim.x];
for(i=0;i<4;i++) dst[indx+i*blockDim.x]=a[i];
```
Today’s lecture

• Thread divergence
• Scheduling
• Instruction Level Parallelism
Recapping: warp scheduling on Fermi

- Threads assigned to an SM in units of a thread block, multiple blocks
- Each block divided into warps of 32 (SIMD) threads, a schedulable unit
  - A warp becomes eligible for execution when all its operands are available
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
  - All threads in a warp execute the same instruction, branches serialize execution
- Multiple warps simultaneously active, hiding data transfer delays
- All registers in all the warps are available, 0 overhead scheduling
- Hardware is free to assign blocks to any SM

Scott B. Baden /CSE 260/ Winter 2014
Instruction Issue on Fermi

• Each vector unit
  • 32 CUDA cores for integer and floating-point arithmetic
  • 4 special function units for Single Precision transcendental

• 2 Warp schedulers: each scheduler issues: 1 (2) instructions for capability 2.0 (2.1)

• One scheduler in charge of *odd* warp IDs, the other *even* warp IDs

• Only 1 scheduler can issue a double-precision floating-point instruction at a time

• Warp scheduler can issue an instruction to ½ the CUDA cores in an SM

• Scheduler must issue the instruction over 2 clock cycles for an integer or floating-point arithmetic instructions
Dynamic behavior – resource utilization

- Each vector core (SM): 1024 thread slots and 8 block slots
- Hardware partitions slots into blocks at run time, accommodates different processing capacities
- Registers are split dynamically across all blocks assigned to the vector core
- A register is private to a single thread within a single block
Thread Divergence

• All the threads in a warp execute the same instruction

• Different control paths are serialized

• *Divergence when a predicate* is a function of the thread Id

  \[
  \text{if (threadId < 2) \{ \}}
  \]

• No divergence if all follow the same path

  \[
  \text{if (threadId / WARP\_SIZE < 2) \{ \}}
  \]

• Consider reduction, e.g. summation $\sum_i x_i$
Thread divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
Divergence example

if (threadIdx >= 2)
    a = 100;
else
    a = -100;

Mary Hall
Divergence example

if (threadIdx >= 2)
    a=100;
else
    a=-100;

Mary Hall
Divergence example

if (threadIdx >= 2)
    a=100;
else
    a=-100;

Mary Hall

Scott B. Baden /CSE 260/ Winter 2014
Thread Divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
- *Divergence* when a predicate is a function of the threadId
  
  ```
  if (threadId < 2) {} 
  ```

- No divergence if all follow the same path within a warp
  
  ```
  if (threadId / WARP_SIZE < 2) {} 
  ```

- We can have different control paths within the thread block
Example – reduction – thread 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

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Scott B. Baden /CSE 260/ Winter 2014
The naïve code

```c
__global__ void reduce(int *input, unsigned int N, int *total) {
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ int x[BSIZE];
    x[tid] = (i < N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }
    if (tid == 0) atomicAdd(total, x[tid]);
}
```
Reducing divergence and avoiding bank conflicts

Thread 0
The improved code

- All threads in a warp execute the same instruction
  reduceSum \( \lll N/512,512 \rrr \) \((x,N)\)
- No divergence until \( \text{stride} < 32 \)
- All warps active when \( \text{stride} \geq 32 \)

```
__shared__ int x[ ];
  unsigned int tid = threadIdx.x;
  unsigned int s;

for (s = blockDim.x/2;
   s > 1;
   s /= 2) {
  __syncthreads();
  if (tid < s )
     x[tid] += x[tid + s ];
}
```

```
for (stride = 1;
    stride < blockDim.x;
    stride *= 2) {
  __syncthreads();
  if (tid % (2*stride) == 0)
    x[tid] += x[tid + stride];
}
```

- \( s = 256 \): threads 0:255
- \( s = 128 \): threads 0:127
- \( s = 64 \): threads 0:63
- \( s = 32 \): threads 0:31
- \( s = 16 \): threads 0:15
...
Predication on Fermi

• All instructions support predication in 2.x
• Condition code or *predicate* per thread:
  set to true or false
• Execute only if the predicate is true

\[
\text{if } (x > 1) \\
\quad y = 7;
\]

\[
\text{test } = (x > 1) \\
\text{test: } y = 7
\]

• Compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by branch condition is not too large
• If the compiler predicts too many divergent warps…. threshold = 7, else 4
Concurrency – Host & Device

• Nonbocking operations
  • Asynchronous Device ↔ \{Host, Device\}
  • Kernel launch

• Interleaved CPU and device calls: kernel calls run asynchronous with respect to host

• But a kernel call sequence runs sequentially on device

• Multiple kernel invocations running in separate CUDA streams: interleaved, independent computations

```c
cudaStream_t st1, st2, st3, st4; cudaStreamCreate(&st1);
cudaMemcpyAsync(d1, h1, N, H2D, st1);
kernell2<<<Grid, Block, 0, st2 >>> (..., d2,..);
kernell3<<<Grid, Block, 0, st3 >>> (..., d3,..);
cudaMemcpyAsync(h4, d4, N, D2H, st4);
CPU_function(…);
```

on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf
Today’s lecture

• Bank conflicts (previous lecture)
• Further improvements to Matrix Multiplication
• Thread divergence
• Scheduling
Dual warp scheduling on Fermi

- 2 schedulers/SM find an eligible warp
  - Each issues 1 instruction per cycle
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
  - Issue selection: round-robin/age of warp
  - Odd/even warps
  - Warp scheduler can issue instruction to $\frac{1}{2}$ the cores, each scheduler issues:
    1 (2) instructions for capability 2.0 (2.1)
  - 16 load/store units
  - Scheduler must issue the instruction over 2 clock cycles for an integer or floating-point arithmetic instruction
  - Most instructions dual issued 2int/2 FP, int/FP/ ld/st
  - Only 1 double prec instruction at a time
- All registers in all the warps are available, 0 overhead scheduling
- Overhead may be different when switching blocks

Scott B. Baden /CSE 260/ Winter 2014
What makes a processor run faster?

• Registers and cache
• Pipelining
• Instruction level parallelism
• Vectorization, SSE
Pipelining

• Assembly line processing - an auto plant
  Dave Patterson’s Laundry example: 4 people doing laundry

  wash (30 min) + dry (40 min) + fold (20 min) = 90 min

  • Sequential execution takes
    4 * 90min = 6 hours

  • Pipelined execution takes
    30+4*40+20 = 3.5 hours

  • Bandwidth = loads/hour

  • Pipelining helps bandwidth but not latency (90 min)

  • Bandwidth limited by slowest pipeline stage

  • Potential speedup = Number pipe stages
Instruction level parallelism

- Execute more than one instruction at a time
  \[ x = \frac{y}{z}; \]
  \[ a = b + c; \]

- Dynamic techniques
  - Allow stalled instructions to proceed
  - Reorder instructions

\[ x = \frac{y}{z}; \quad x = \frac{y}{z}; \]
\[ a = x + c; \quad t = c - q; \]
\[ t = c - q; \quad a = x + c; \]
Multi-execution pipeline

• MIPS R4000

David Culler

Scott B. Baden /CSE 260/ Winter 2014
Scoreboarding, Tomasulo’s algorithm

• Hardware data structures keep track of
  ◆ When instructions complete
  ◆ Which instructions depend on the results
  ◆ When it’s safe to write a reg.

• Deals with data hazards
  ◆ WAR (Write after read)
  ◆ RAW (Read after write)
  ◆ WAW (Write after write)

\[
\begin{align*}
  a &= b \cdot c \\
  x &= y - b \\
  q &= a / y \\
  y &= x + b
\end{align*}
\]
Scoreboarding

- Keep track of all register operands of all instructions in the Instruction Buffer
  - Instruction becomes ready after the needed values are written
  - Eliminates hazards
  - Ready instructions are eligible for issue

- Decouples the Memory/Processor pipelines
  - Threads can issue instructions until the scoreboard prevents issue
  - Allows Memory/Processor ops to proceed in parallel with other waiting Memory/Processor ops

TB = Thread Block, W = Warp

---

Scott B. Baden /CSE 260/ Winter 2014
Static scheduling limits performance

• The ADDD instruction is stalled on the DIVide..
• Stalling further instruction issue, e.g. the SUBD
  \[
  \text{DIV} \quad F0, \quad F2, \quad F4 \\
  \text{ADDD} \quad F10, \quad F0, \quad F8 \\
  \text{SUBD} \quad F12, \quad F8, \quad F14
  \]
• But SUBD doesn’t depend on ADDD or DIV
• If we have two adder/subtraction units, one will sit idle uselessly until the DIV finishes
Dynamic scheduling

• Idea: modify the pipeline to permit instructions to execute as soon as their operands become available
• This is known as *out-of-order execution (classic dataflow)*
• The SUBD can now proceed normally
• Complications: dynamically scheduled instructions also complete out of order
Dynamic scheduling splits the ID stage

- Issue sub-stage
  - Decode the instructions
  - Check for structural hazards
- Read operands sub-stage
  - Wait until there are no data hazards
  - Read operands
- We need additional registers to store pending instructions that aren’t ready to execute
Consequences of a split ID stage

• We distinguish between the time when an instruction begins execution, and when it completes
• Previously, an instruction stalled in the ID stage, and this held up the entire pipeline
• Instructions can now be in a suspended state, neither stalling the pipeline, nor executing
• They are waiting on operands
Two schemes for dynamic scheduling

• **Scoreboard**
  - CDC 66000

• **Tomasulo’s algorithm**
  - IBM 360/91

• We’ll vary the number of functional units, their latency, and functional unit pipelining
What is a scoreboarding?

• A technique that allows instructions to execute out of order…
  ♦ So long as there are sufficient resources and
  ♦ No data dependencies

• The goal of scoreboarding
  ♦ Maintain an execution rate of one instruction per clock cycle
Multiple execution pipelines in DLX with scoreboard

<table>
<thead>
<tr>
<th>Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Memory</td>
<td>1</td>
</tr>
<tr>
<td>FP Add</td>
<td>2</td>
</tr>
<tr>
<td>FP Multiply</td>
<td>10</td>
</tr>
<tr>
<td>FP Div</td>
<td>40</td>
</tr>
</tbody>
</table>
Scoreboard controls the pipeline

Scoreboard / Control Unit

Instruction Fetch → Instruction Issue → Instruction Decode

Pre-issue buffer

Pre-execution buffers

WAW

Read operands → Execution unit 1

RAW

Read operands → Execution unit 2

WAR

Post-execution buffers

Write results

Mike Frank
What are the requirements?

• Responsibility for instruction issue and execution, including hazard detection
• Multiple instructions must be in the EX stage simultaneously
• Either through pipelining or multiple functional units
• DLX has: 2 multipliers, 1 divider, 1 integer unit (memory, branch, integer arithmetic)
How does it work?

• As each instruction passes through the scoreboard, construct a description of the data dependencies (Issue)
• Scoreboard determines when the instruction can read operands and begin execution
• If the instruction can’t begin execution, the scoreboard keeps a record, and it listens for one the instruction can execute
• Also controls when an instruction may write its result
• All hazard detection is centralized
How is a scoreboard implemented?

• A centralized bookkeeping table
• Tracks instructions, along with register operand(s) they depend on and which register they modify
• Status of result registers (who is going to write to a given register)
• Status of the functional units