CMPSCI 691AD – General Purpose Computation on the GPU

Spring 2009

Lecture 10: Performance Guidelines I

Rui Wang

Overview

- Last two lectures:
  CUDA memory models and the performance impact of memory instructions

  General pattern:

  - Load data from GMEM to SMEM (coalesced)
  - Synchronize
  - Process data in SMEM (padding to avoid SMEM bank confclits)
  - Synchronize
  - Write results back to GMEM (coalesced)

Overview

- Last two lectures:
  CUDA memory models and the performance impact of memory instructions
- This time:
  additional topics related to performance tuning:
  - Arithmetic instructions
  - Control flow instructions
  - Hiding memory latency
  - Tools (occupancy graph, profiler, bank conflict checker)

Arithmetic Instructions

- A multiprocessor takes:
  - 4 clock cycles
    - single-precision add, mul, mad
    - Integer add, __mul24, __umul24
    - Bitwise op, compare, min, max, type convert
  - 16 cycles for
    - Reciprocal, reciprocal sqrt, __logf
    - Integer mult (32-bit)
  - 32 cycles for sqrt
  - 36 cycles for floating point division (20 for __fdividef)

Arithmetic Instructions

- Integer division and modulo very expensive (really, very!)
  - Convert to bitwise op if possible (e.g. i/n and i%n when n is a power of 2)

Look how CUDA implements integer division!

```
__global__ void testKernel1(int a, int b, int *res) {
    res[threadIdx.x] = a / b;
}
```

```
000000: 10000009 0403c780 mov.b32 $r2, $r0
000008: 1100ea04          mov.half.b32 $r1, s[0x0014]
00000c: 1100e800          mov.half.b32 $r0, s[0x0010]
000010: a000081d 04000780 cvt.u32.u16 $r7, $r2.lo
000018: 20008003 00000780 call.label label0
000020: 30020e05 c4100780 shl.u32 $r1, $r7, 0x00000002
000028: 2000cc05 04204780 add.u32 $r1, s[0x0018], $r1
000030: d00e0201 a0c00780 mov.u32 g[$r1], $r0
000038: 30000003 00000780 return
000040: a0000209 04114780 label0: cvt.u32.s32 $r2, $r1 (Unknown subsubop 45)
000048: a000040d 44004780 cvt.rn.f32.u32 $r3, $r2
000050: a0000011 04114780 cvt.u32.s32 $r4, $r0 (Unknown subsubop 45)
000058: 90000615 00000780 rcp.f32 $r5, $r3
64780 cvt.rz.f32.u32 $r3, $r4
fffff add.b32 $r5, $r5, 0xfffffffe
0c7c0 mul.rz.f32 $p0|$r3, $r3, $r5
64780 cvt.rzi.u32.f32 $r3, $r3
00780 mul24.lo.u32.u16.u16 $r6, $r2.lo, $r3.hi
000088: 60060a19 00018780 mad24.lo.u32.u16.u16.u32 $r6, $r2.hi, $r3.lo, $r6
000090: 30100c19 c4100780 shl.u32 $r6, $r6, 0x00000010
000098: 60060819 00018780 mad24.lo.u32.u16.u16.u32 $r6, $r2.lo, $r3.lo, $r6
0000a0: 20400819 04018780 sub.u32 $r6, $r4, $r6
0000a8: a0000c19 44064780 cvt.rz.f32.u32 $r6, $r6
0000b0: c0050c15 0000c7c0 mul.rz.f32 $p0|$r5, $r6, $r5
0000b8: a0000a15 84064780 cvt.rzi.u32.f32 $r5, $r5
0000c0: 2000060d 04014780 add.u32 $r3, $r3, $r5
0000c8: 40040e15 00000780 mul24.lo.u32.u16.u16 $r5, $r3.hi, $r2.lo
0000d0: 60050c15 00014780 mad24.lo.u32.u16.u16.u32 $r5, $r3.lo, $r2.hi, $r5
0000d8: 30100a15 c4100780 shl.u32 $r5, $r5, 0x00000010
0000e0: 60040c15 00014780 mad24.lo.u32.u16.u16.u32 $r5, $r3.lo, $r2.lo, $r5
0000e8: 30000a11 04010780 subr.u32 $r4, $r5, $r4
0000f0: 30040409 6400c780 set.le.u32 $r2, $r2, $r4
0000f8: d0000201 04008780 xor.b32 $r0, $r1, $r0
000100: 301f0001 e4100780 shr.u32 $r0, $r0, 0x0000001f
000108: 3000040d 0400c780 subr.u32 $r3, $r2, $r3
000110: a0000009 2c014780 cvt.neg.s32 $r2, $r0
000118: d0030409 04008780 xor.b32 $r2, $r2, $r3
000120: 307c03fd 6c0147c8 set.ne.s32 $p0|$o127, $r1, $r60 (unk0 00400000)
000128: 20000001 04008780 add.u32 $r0, $r0, $r2
000130: d0010001 0402c500 @$p0.equ not.b32 $r0, $r1
000138: 30000003 00000780 return
000140: f0000001 e0000001 nop.end
```

http://dl.getdropbox.com/u/484203/Lectures/Guests/KiptonBarros__mit_cuda_09.pdf

Single Precision FP Instructions

- A multiprocessor takes:
  - 32 clock cycles
    - __sinf, __cosf, __expf
  - Much more expensive for the regular sinf, cosf, tanf
    - You can examine the binary code to find out
- Convertion from single-precision to double precision
  - 1.0 is interpreted as double-precision
  - Be explicit! (use 1.0f as opposed to 1.0)

Control Flow Instructions

- if, switch, do, for, while
- If threads in the same warp diverge, execution path must be serialized (effectively increasing the total # of inst)
- Managing divergent threads also consumes resources for book-keeping

Control Flow Instructions

- Try to minize divgergent warps
  - Optimization is possible because we know how the device organizes threads (according to id) into warps
    - Trivial: when branching depends only on tid / WSIZE
  - Sometimes this involves simply rearranging tasks: group similar tasks together
    - Recall the parallel reduction example

Compiler Optimizations

- Branch predication
  - Each instruction is associated with a per-thread condition code, which is set to true or false based on the result of the controlling condition.
  - Instruction gets scheduled for all threads, but only the ones with a true predicate are executed.

Compiler Optimizations

- Branch predication
- Example:

```
:
:
if (x == 10)
   c = c + 1;
:
:
```

```
      :
      :
      LDR r5, X
      p1 <- r5 eq 10
<p1> LDR  r1 <- C
<p1> ADD r1, r1, 1
<p1> STR  r1 -> C
      :
      :
```

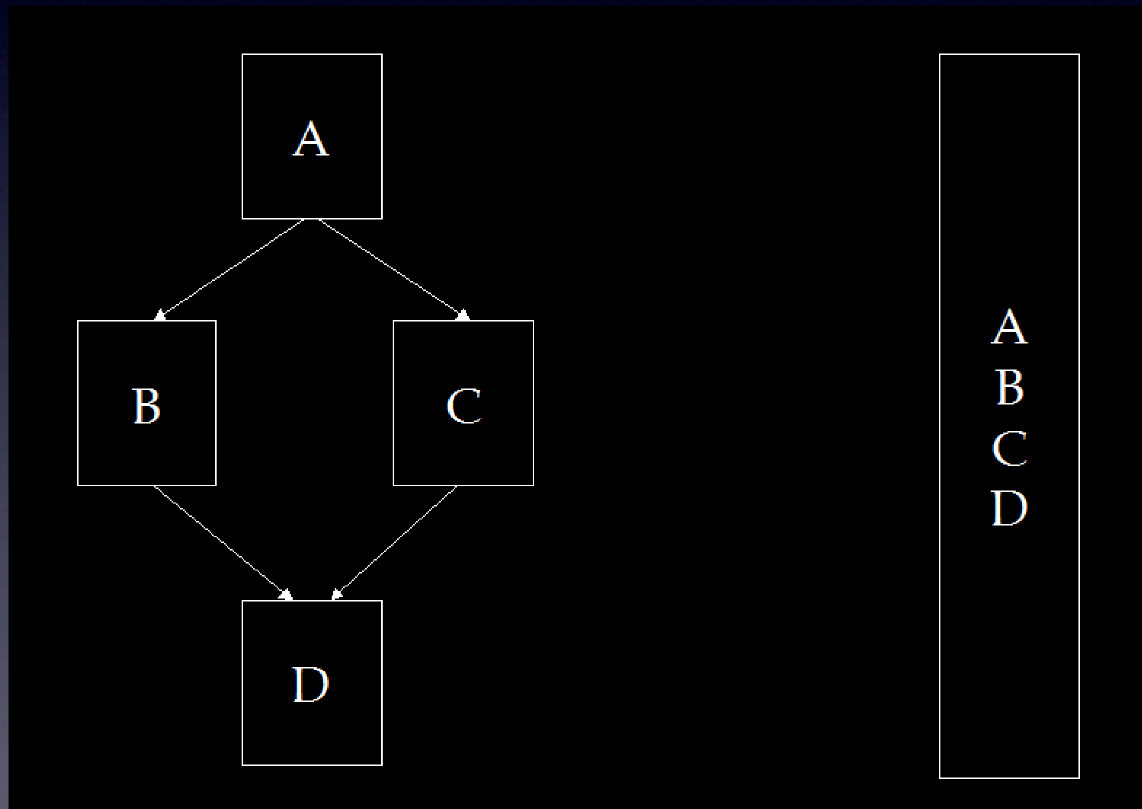
http://sbel.wisc.edu/Courses/ME964/2008/index.htm

Compiler Optimizations

- Branch predication
  - Advantage: eliminates divergence
  - Disadvantage: extra instructions – each predicated instruction will be scheduled for all threads; however,
    - Not as expensive as serializing code
    - Allows instructions with different predicates to mix, improving processor scheduling.

Compiler Optimizations

- Branch predication
- Example:




```
      :
      :
      p1,p2 <- r5 eq 10
<p1>  inst 1 from B
<p2>  inst 1 from C

<p1>  inst 2 from B
<p2>  inst 2 from C

<p1>  :
      :
```

http://sbel.wisc.edu/Courses/ME964/2008/index.htm

Compiler Optimizations

- Branch predication
  - The CUDA compiler will try to replace a branch with predicated instructions, but it only happens if the number of branch instructions is small – less than
    - 7 for highly divergent branches (predicted by compiler)
    - 4 otherwise
  - If the number of branch instructions is large, do not convert to predication.

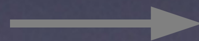
Compiler Optimizations

- Loop Unrolling

  Unwind instructions across different loops in order to reduce (or eliminate) end of the loop condition test; also enables better instruction scheduling.

- Example:

```
for (i = 0; i < 5; i ++)
{
    a[i] += b[i];
}
```



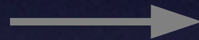
```
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
a[3] += b[3];
a[4] += b[4];
```

Compiler Optimizations

- Loop Unrolling

  Example:

```
for (i = 0; i < 1000; i ++)
{
      a[i] += b[i];
}
```



```
for (i = 0; i < 1000; i += 5)
{
      a[i+0] += b[i+0];
      a[i+1] += b[i+1];
      a[i+2] += b[i+2];
      a[i+3] += b[i+3];
      a[i+4] += b[i+4];
}
```

Compiler Optimizations

- Loop Unrolling
  - Use #pragma unroll directive to control the unrolling count

```
#pragma unroll 5

for (i = 0; i < n; i ++)
{
	a[i] += b[i];
}
```

  - It's up to the programmer to make sure that the unrolling will not affect the correctness of the program (which it might, say, if $n < 5$)

Hiding Memory Latency

- An important resource limitation: GMEM access.
- Although GMEM access speed is low, the latency can be hidden well if there are many independent instructions.
  - Processor can schedule these instructions while waiting for memory access results.

Hiding Memory Latency

- An important resource limitation: GMEM access.
- Although GMEM access speed is low, the latency can be hidden well if there are many independent instructions.
  - Processor can schedule these instructions while waiting for memory access results.
- Recall the matrix multiplication example:
  - Although SMEM is used to coalesce GMEM access, the computation of each tile is clearly data dependent.

Hiding Memory Latency

- Matrix Multiplication:

```
Loop over all tiles {
    Load current tile to SMEM;
    __syncthreads();
    Compute current tile;
    __syncthreads();
}
```

Hiding Memory Latency

- Matrix Multiplication:

```
Loop over all tiles {
    Load current tile to SMEM;
    __syncthreads();
    Compute current tile;
    __syncthreads();
}
```

Data dependency

Hiding Memory Latency

- Data Prefetching

  Prefetch the next block of data elements while consuming the current block → increased number of independent instructions → GMEM latency hidden.

Hiding Memory Latency

- Data Prefetching

```
Load first file from GMEM to SMEM;
__syncthreads();
Loop over all tiles {
    Load next file from GMEM to SMEM;
    Compute current tile;
    __syncthreads();
}
```

Hiding Memory Latency

- Data Prefetching

```
Load first file from GMEM to SMEM;
__syncthreads();
Loop over all tiles {
    Load next file from GMEM to SMEM;
    Compute current tile;
    __syncthreads();
}
```

- What's the catch (tradeoff) ?

Hiding Memory Latency

- Data Prefetching

```
Load first file from GMEM to SMEM;
__syncthreads();
Loop over all tiles {
    Load next file from GMEM to SMEM;
    Compute current tile;
    __syncthreads();
}
```

- What's the catch (tradeoff) ?
  - Requires twice the amount of SMEM (one serves as a buffer).

Tools

- Occupancy

  The ratio of the number of actual concurrent warps to the maximum number of warps supported.

- Limited by resource usages:
  - Registers
  - Shared memory consumption
- Find out resource consumption by
  - Use -ptroptions=-v
  - Or -cubin flag

Tools

- .cubin

```
architecture {sm_10}
abiversion   {1}
modname      {cubin}
code {
	name = _Z11testKernel1ffPi
	lmem = 0
	smem = 28
	reg  = 3
	bar  = 0
	const {
			segname = const
			segnum  = 1
			offset  = 0
			bytes   = 4
		mem {
			0x00000004
		}
	}
	bincode {
		0xb000c809 0xc0200780 0x1000cc05 0x0423c780
	..................
```

Tools

- Occupancy calculator

Microsoft Excel - CUDA_Occupancy_calculator.xls

CUDA GPU Occupancy Calculator

lick Here for detailed instructions on how to use this occupancy calculato

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select a GPU from the list (click):	G80
2.) Enter your resource usage:	
Threads Per Block	192
Registers Per Thread	20
Shared Memory Per Block (bytes)	68
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	384
Active Warps per Multiprocessor	12
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	50%
Maximum Simultaneous Blocks per GPU	32
(Note: This assumes there are at least this many blocks)	
Physical Limits for GPU:	G80
Multiprocessors per GPU	16
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384
Allocation Per Thread Block	
Warps	6
Registers	3840
Shared Memory	512
These data are used in computing the occupancy data in blue	
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	2
Limited by Shared Memory / Multiprocessor	32
Thread Block Limit Per Multiprocessor is the minimum of these 3	
CUDA Occupancy Calculator	
Version:	1.1

Copyright and License

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Varying Block Size — Multiprocessor Warp Occupancy vs. Threads Per Block (My Block Size 192)

Varying Register Count — Multiprocessor Warp Occupancy vs. Registers Per Thread (My Register Count 20)

Varying Shared Memory Usage — Multiprocessor Warp Occupancy vs. Registers Per Thread (My Shared Memory 68)

Tools

- Occupancy calculator

  Provides an idea of how well you are keeping the device busy

- General rules of thumb:
  - Choose thread number to be a multiple of warp size
  - More threads per block → better memory latency hiding; on the other hand allows less registers and SMEM per thread
  - Heuristics: minimum 64 threads per block (192 or 256 better)
- Also, increasing occupancy does not necessarily mean increased performance! (no analysis on the instructions)