Module 3 Part 2
Application Design for Manycore
Advanced Optimizations and Performance Measurements

Carnegie Mellon University
18-645

Jike Chong
Ian Lane

What we discussed last time:

Fast Platforms

- **Multicore platforms**
- **Manycore platforms**
- **Cloud platforms**

+

Good Techniques

- **Data structures**
- **Algorithms**
- **Software Architecture**

- Highlighted the difference between multicore and manycore platforms
- **Exposing** concurrency in k-means, **Exploiting** parallelism by **exploring** mappings from application to platform
- Multicore platforms: OpenMP, performance metrics using Roofline model
- Manycore platforms: GPUs and CUDA

Answers you should know after this…

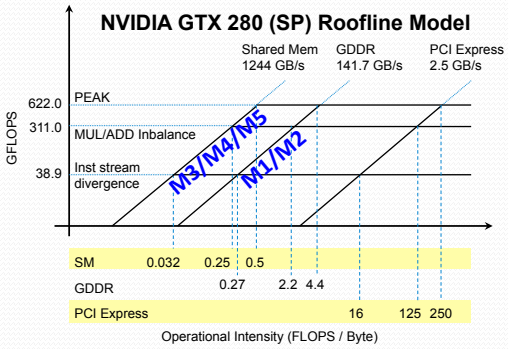
- What are the three ways to improve execution throughput?
- When to use SOA vs AOS?
- What is memory Coalescing? When to use it? Why is it important?
- What is shared memory? How to use it?
- What is memory bank conflict? How to work around it?
- What is branch divergence?
- How to optimize for instruction mix?
- What is occupancy? How to model/measure it?
- How to use the code profiler with CUDA?

Outline

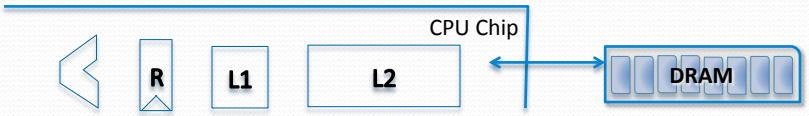
- Maximizing Memory Throughput
- Maximizing Instruction Throughput
- Maximizing Scheduling Throughput
- More Special Optimizations
- Performance Analysis: NVIDIA Visual Profiler

Maximize Memory Throughput

M1. SoA vs AoS
M2. Memory coalescing
M3. Use of shared memory
M4. Memory bank conflict
M5. Padding



"Highly parallel processors turns compute-limited algorithms into memory-limited algorithms"



(M1) SOA vs AOS

Struct of Arrays vs. **Array of Struct**

```
typedef struct
{
    float* x;
    float* y;
    float* z;

} Constraints;
```

x	x	x	y	y	y	z	z	z

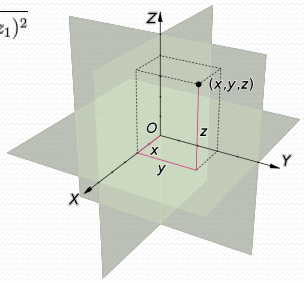
```
typedef struct __align__(16)
{
    float3 position;

} Constraint;
```

x	y	z	x	y	z	x	y	z

- Important distinction based on application memory access patterns
  - Which one is "better"?

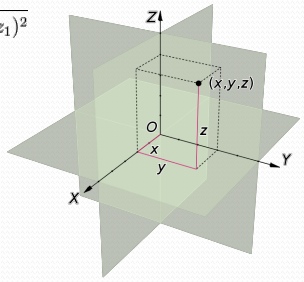
SOA vs AOS? It depends…

- What is the computation?
  - For each point: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$
- What is the optimization goal?
- Example 1:
  - You have a list of coordinates, and you want to find the distance from the origin to all points
  - What should be the data structure?



SOA vs AOS? Example 1

- What is the computation?
  - For each point: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$
- What is the optimization goal?
  - Minimize memory bandwidth to DRAM
- Example 1:
  - You have a list of coordinates, and you want to find the distance from the origin to all points
  - What should be the data structure?




```
typedef struct
{
      float* x;
      float* y;
      float* z;
} pointIn3D;
```

x	x	x
y	y	y
z	z	z

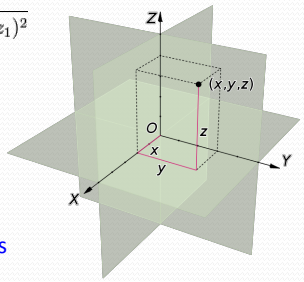
OR

```
typedef struct __align__(16)
{
      float3 position;
} pointIn3D;
```

x	y	z	x	y	z	x	y	z

SOA vs AOS? Example 2

- What is the computation?
  - For a point: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$
- What is the optimization goal?
  - Minimize memory bandwidth to DRAM
- Example 2:
  - Find the distance from the origin to a list of points estimated to be ~1% of all points
  - What should be the data structure?




```
typedef struct
{
      float* x;
      float* y;
      float* z;

} pointIn3D;
```

x	x	x
y	y	y
z	z	z

OR

```
typedef struct __align__(16)
{
      float3 position;

} pointIn3D;
```

x	y	z	x	y	z	x	y	z

(M2) Memory Coalescing

- **Hardware Constraint**: DRAM is accessed in "segments" of 32B/64B/128B
  - Unused data loaded in a "segment" still takes up valuable bandwidth
- **Goal:** combine multiple memory accesses generated from multiple threads into a single physical transaction
  - increases effective throughput to DRAM
- **Rules** for maximizing DRAM memory bandwidth:
  - Possible bus transaction sizes: 32B, 64B, or 128B
  - Memory segment must be aligned: First address = multiple of segment size
  - Hardware coalescing for each half-warp: 16-word wide

Kirk, Hwu, Programming Massively Parallel Processors, Chapter 6.2

When is the Access Coalesced?

- "Threads can access any words in any order, including the same words, and a single memory transaction for each segment addressed by a half-warp"
- Detailed protocol:

  Find the memory segment that contains the address requested by the active thread with the lowest thread ID.
  - 32 bytes for 1-byte words
  - 64 bytes for 2-byte words
  - 128 bytes for 4-, 8- and 16-byte words.

  Find all other active threads whose requested address lies in the same segment.

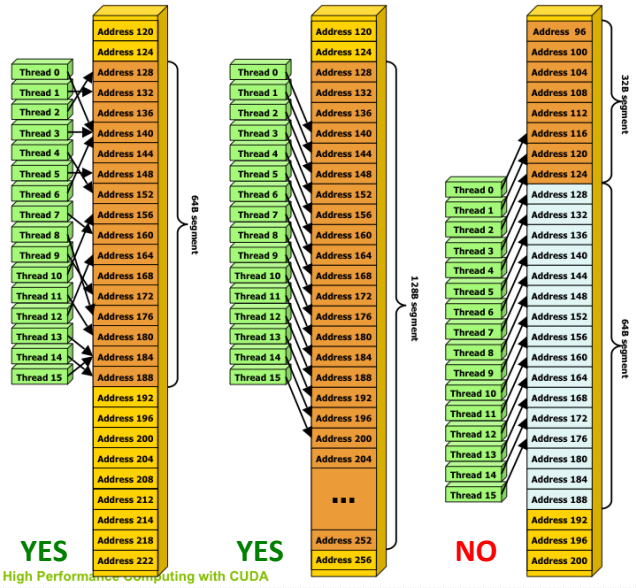
  Reduce the transaction size, if possible:
  - If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
  - If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.

  Carry out the transaction and mark the serviced threads as inactive

  Repeat until all threads in the half-warp are serviced.

Examples:

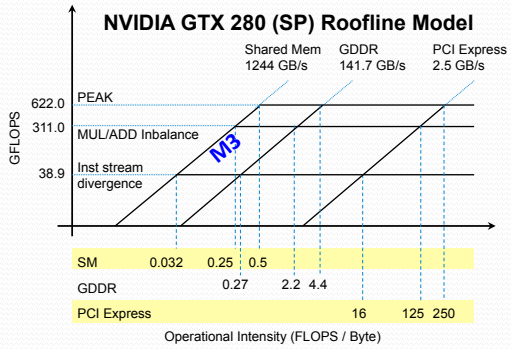
- Are these coalesced?

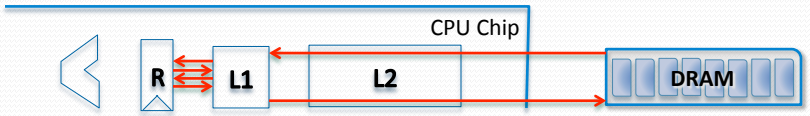


Paulius Micikeviciu, "M02: High Performance Computing with CUDA: Optimizing CUDA", SC08, Austin, TX.

(M3) Use of Shared Memory

- Take advantage of **9x** faster memory bandwidth
- Process:
  - Load from DRAM to shared memory
  - **Synchronize**
  - Perform work on data in shared memory
  - **Synchronize**
  - Write out results to DRAM





Double Buffering

- One could double buffer the computation
  - Getting better instruction mix within each thread
  - Classic software pipelining in ILP compilers

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

  Compute current tile
  syncthreads()
}
```

Original

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

Double Buffered

Kirk, Hwu, Programming Massively Parallel Processors, Figure 6.13

Using Shared Memory

- Two approaches to using shared memory

(1) Declared a fixed sized variable at **compile time**

```
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
```

(2) Define a size to be used at **run time**

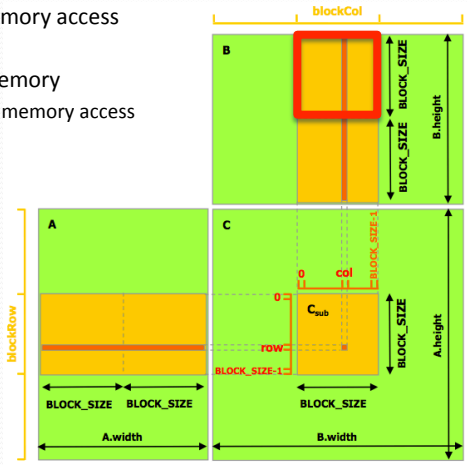
```
__global__ void mykernel(int a, float *objects) {

  extern __shared__ char sharedMemory[];
  unsigned char *membershipChanged = (unsigned char *)sharedMemory;
  float         *clusters          = (float *)(sharedMemory + blockDim.x);
  ...
}

// In host code
mykernel <<< nBlks, nThds, shmemByteSize >>> (a, objects);
```


Remember Matrix Transposition?

- **One can use the shared memory**:
  - Load Matrix B with coalesced memory access into shared memory
  - "Random access" from shared memory
    - No longer limited by uncoalesced memory access

load → use

32	33	34	35	...
R+32	R+33	R+34	R+35	...
2R+32	2R+33	2R+34	2R+35	...
3R+32	3R+33	3R+34	3R+35	...
...	...	...	...	...



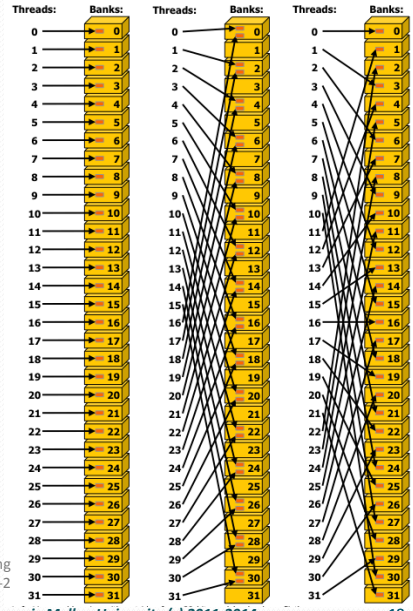
Good conceptually, but does it work?

(M4) Memory Bank Conflicts

- Shared memory has 32 banks
  - Organized such that successive 32-bit words are assigned to successive banks
  - Each bank has a bandwidth of 32 bits per two clock cycles (2 cycle latency)
- A **bank conflict** occurs if two or more threads access any bytes within *different* 32-bit words belonging to the same bank

Examples:

- **Left**: Linear addressing with a stride of one 32-bit word
  - no bank conflict
- **Middle**: Linear addressing with a stride of two 32-bit words
  - 2-way bank conflicts
- **Right**: Linear addressing with a stride of three 32-bit words
  - no bank conflict

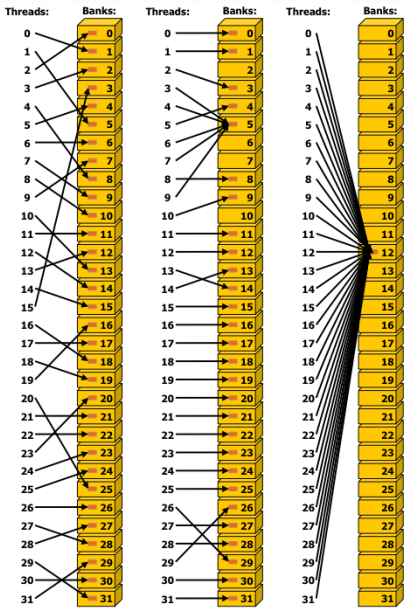


NVIDIA CUDA C Programming Guide Version 4.0, Figure F-2

Examples:

- **Left:** Conflict-free access via random permutation.
- **Middle:** Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
- **Right:** Conflict-free broadcast access
  - all threads access the same word

How do we make the matrix transposition work?



NVIDIA CUDA C Programming Guide Version 4.0, Figure F-3

(M5) Padding Technique

- We have seen padding used to align data structures in project 1
- Padding can also be used to **offset** memory bank conflicts

load → use

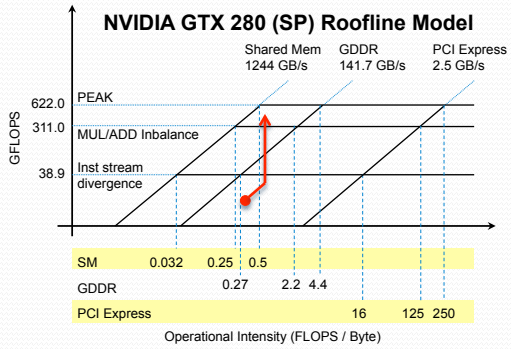
32	33	34	35	…	62	63
R+32	R+33	R+34	R+35	…	R+62	R+63
2R+32	2R+33	2R+34	2R+35	…	2R+62	2R+63
3R+32	3R+33	3R+34	3R+35	…	2R+62	2R+63
…	…	…	…	…	…	…

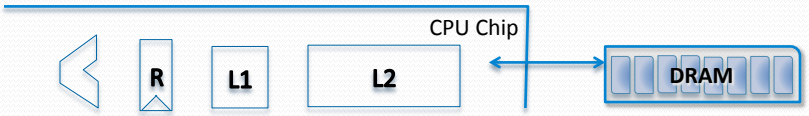
load → use

32	33	34	35	…	62	63	*
R+32	R+33	R+34	R+35	…	R+62	R+63	*
2R+32	2R+33	2R+34	2R+35	…	2R+62	2R+63	*
3R+32	3R+33	3R+34	3R+35	…	2R+62	2R+63	*
…	…	…	…	…	…	…	

Maximize Memory Throughput

M1. SoA vs AoS
M2. Memory coalescing
M3. Use of shared memory
M4. Memory bank conflict
M5. Padding





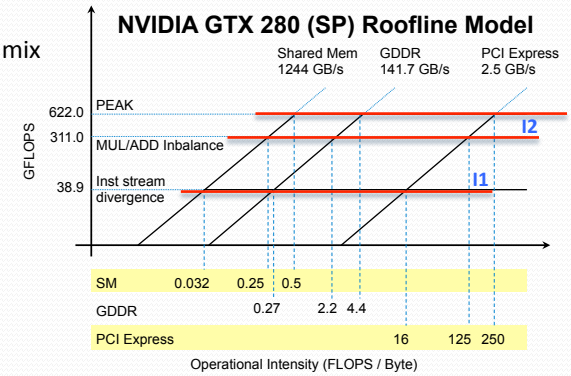
Outline

- Maximizing Memory Throughput
- Maximizing Instruction Throughput
- Maximizing Scheduling Throughput
- More Special Optimizations
- Performance Analysis: NVIDIA Visual Profiler

Maximizing Instruction Throughput

(I1) Branch divergence

(I2) Optimize instruction mix



Examples:

Example 1

```
tid = threadIdx.x;
if (a[tid] > 0) {
    x += 1;
} else {
    if (b[tid] > 0) {
        x += 2;
    } else {
        x += 3;
    }
}
```

Example 2

```
if(c>0){
    x = x*a1 + b1;
    y = y*a1 + b1;
} else {
    x = x*a2 + b2;
    y = y*a2 + b2;
}
```

Original Code

```
if (c > 0) {
    a = a1;
    b = b1;
} else {
    a = a2;
    b = b2;
}
x = x*a + b;
y = y*a + b;
```

Optimized Code

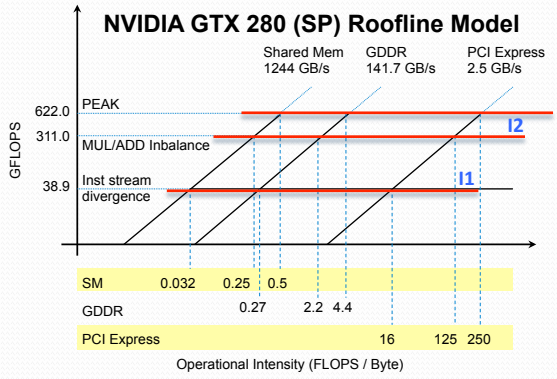
- Optimization:
  - Factor out decision variables to have shorter sequence of divergent code

Tianyi David Han, Tarek S. Abdelrahman, **Reducing Branch Divergence in GPU Programs,** *GPGPU-4* Mar 05-05 2011, Newport Beach, CA USA

(I1) Branch Divergence

- At every instruction issue, SIMT unit selects a warp that is ready to execute
- A warp executes one common instruction at a time
  - **Full efficiency is realized** when **all 32 thread**s of a warp **agree** on their path
- If threads of a warp diverge via a data-dependent conditional branch
  - the warp serially executes each branch path taken
  - disables threads that are not on that path
  - when all paths complete
  - the threads converge back to the same execution path
- **Branch divergence occurs only within a warp**

Maximizing Instruction Throughput



(I2) Optimizing Instruction Mix

- Compiler Assisted Loop Unrolling
  - Provides more instruction level parallelism for the compiler to use
  - Improves the ability for the **compiler to find** the **instruction mix** that increases instructions executed per cycle (IPC)
- By default, the *compiler unrolls small loops* with a *known trip count*
- In CUDA, **#pragma unroll** directive can control unrolling of any given loop
  - Must be placed immediately before the loop and only applies to that loop
  - Optionally followed by a number
    - Specifies how many times the loop must be unrolled

Compiler Assisted Loop Unrolling

- Example 1:
  - Loop to be unrolled 5 times

```
#pragma unroll 5
for (int i = 0; i < n; ++i)
```

- Example 2:
  - Preventing the compiler from unrolling a loop

```
#pragma unroll 1
for (int i = 0; i < n; ++i)
```

- Example 3:

```
#pragma unroll
for (int i = 0; i < n; ++i)
```

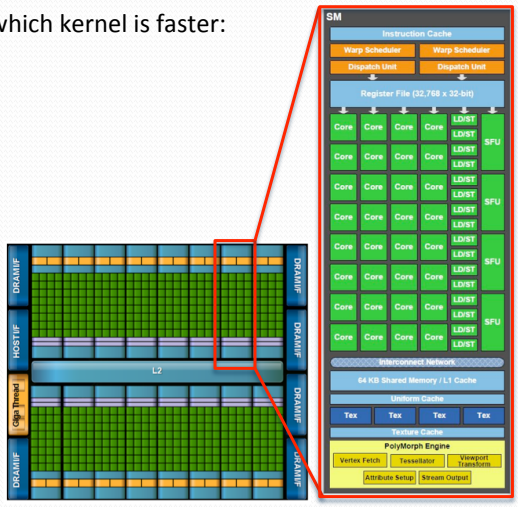
  - If n is a constant, loop is fully unrolled
  - If n is a variable loop is not rolled at all

Outline

- Maximizing Memory Throughput
- Maximizing Instruction Throughput
- Maximizing Scheduling Throughput
- More Special Optimizations
- Performance Analysis: NVIDIA Visual Profiler

Maximizing Scheduling Throughput

- For a particular application, which kernel is faster:
  - Kernel 1:
    - 256 threads/block
    - 17 registers/thread
    - 10KB Shared mem/block
  - Kernel 2:
    - 196 threads/block
    - 28 registers/thread
    - 4KB Shared mem/block
- Must respect the physical limitations of the processor!



NVIDIA Fermi Architecture

- **Occupancy**:
  - Ability of a **CUDA kernel** to **occupy concurrent contexts** in a SM (Streaming Multiprocessor)
  - Specifically, the ratio of active warps to the maximum number of warps supported
  - Helpful in determining how efficient the kernel could be on the GPU
- **CUDA Occupancy Calculator**
  - A programmer tool for computing the multiprocessor "occupancy"

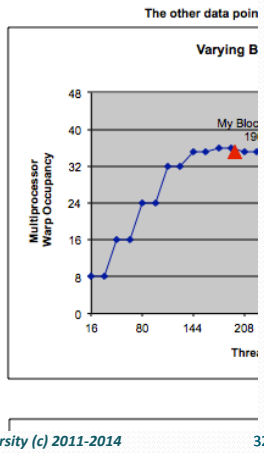
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

CUDA Occupancy Calculator

- 196 threads/block, 28 registers/thread, 4KB Shared mem/block ➔ **73%**

	A	B	C
3			
4	Just follow steps 1, 2, and 3 below! (or click here for help)		
5			
6	1.) Select Compute Capability (click):	2.0	(Help)
7			
8	2.) Enter your resource usage:		
9	Threads Per Block	196	(Help)
10	Registers Per Thread	28	
11	Shared Memory Per Block (bytes)	4,000	
12			
13	(Don't edit anything below this line)		
14			
15	3.) GPU Occupancy Data is displayed here and in the graphs:		
16	Active Threads per Multiprocessor	980	(Help)
17	Active Warps per Multiprocessor	35	
18	Active Thread Blocks per Multiprocessor	5	
19	Occupancy of each Multiprocessor	73%	
20			
21			
22	Physical Limits for GPU Compute Capability:	2.0	
23	Threads per Warp	32	
24	Warps per Multiprocessor	48	
25	Threads per Multiprocessor	1536	
26	Thread Blocks per Multiprocessor	8	
27	Total # of 32-bit registers per Multiprocessor	32768	
28	Register allocation unit size	64	
29	Register allocation granularity	warp	
30	Shared Memory per Multiprocessor (bytes)	49152	
31	Shared Memory Allocation unit size	128	
32	Warp allocation granularity (for register allocation)	0	



How to Get the Parameters

- Threads/block:
  - Programmer specified at __global__ function launch
- Registers/thread:
  - Use compiler option to display at compile time

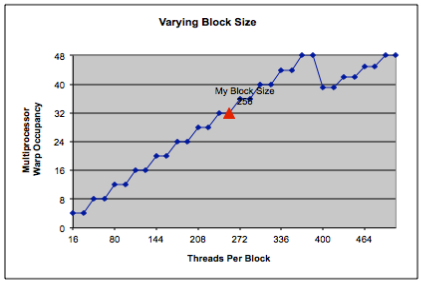
    `--ptxas-options=-v`
  - Expected output:

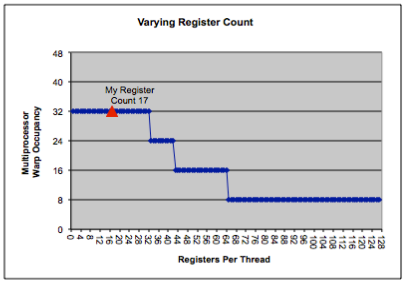
```
ptxas info: Compiling entry function '_XYZ_' for 'sm_20'
ptxas info: Used 25 registers, 3616+0 bytes smem, 53 bytes cmem[0], 4 bytes cmem[16]
```

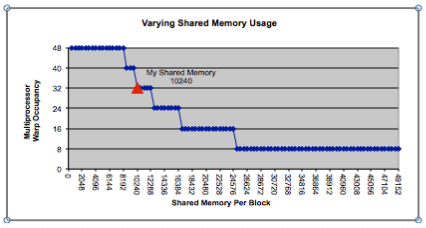
- Shared memory/block:
  - If determined at runtime, user specified variable
  - If determined at compile time, see output from:

    `--ptxas-options=-v`

Maximizing Scheduling Throughput

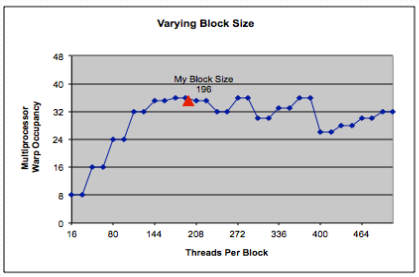


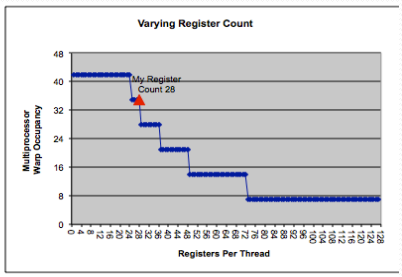


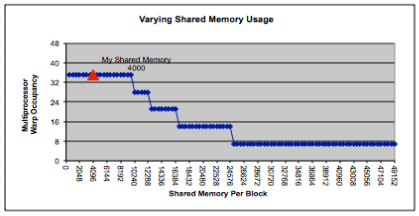


- Kernel 1:
  - 256 threads/block
  - 17 registers/thread
  - 10KB Shared mem/block
- Occupancy: 67%

Maximizing Scheduling Throughput







- Kernel 2:
  - 196 threads/block
  - 28 registers/thread
  - 4KB Shared mem/block
- Occupancy: 73%

Outline

- Maximizing Memory Throughput
- Maximizing Instruction Throughput
- Maximizing Scheduling Throughput
- More Special Optimizations
- Performance Analysis: NVIDIA Visual Profiler

Special Optimizations

- Device-only CUDA intrinsic functions
  - Faster implementation with reduced accuracy
  - Use compiler option (**-use_fast_math**) to force each function on the left to compile to its intrinsic counterpart.
  - Or selectively replace mathematical function calls by calls to intrinsic functions

Operator/Function	Device Function
`x/y`	`__fdividef(x,y)`
`sinf(x)`	`__sinf(x)`
`cosf(x)`	`__cosf(x)`
`tanf(x)`	`__tanf(x)`
`sincosf(x,sptr,cptr)`	`__sincosf(x,sptr,cptr)`
`logf(x)`	`__logf(x)`
`log2f(x)`	`__log2f(x)`
`log10f(x)`	`__log10f(x)`
`expf(x)`	`__expf(x)`
`exp10f(x)`	`__exp10f(x)`
`powf(x,y)`	`__powf(x,y)`

Outline

- Maximizing Memory Throughput
- Maximizing Instruction Throughput
- Maximizing Scheduling Throughput
- More Special Optimizations
- Performance Analysis: NVIDIA Visual Profiler

Tools for Measuring Performance

- CUDA Profiler Tutorial – by Erik Reed

How is this relevant to writing fast code?

Fast Platforms

- **Multicore platforms**
- **Manycore platforms**
- **Cloud platforms**

+

Good Techniques

- **Data structures**
- **Algorithms**
- **Software Architecture**

- Introduced the manycore platform HW and SW mental models
- Introduced the terminologies for you to start FLIRTing with the technology
- Introduced design trade-offs in data structures with some algorithms

- Next lectures:
  - Focus on algorithms and software architecture