CUDA Performance

Patrick Cozzi
University of Pennsylvania
CIS 565 - Fall 2012

Announcements

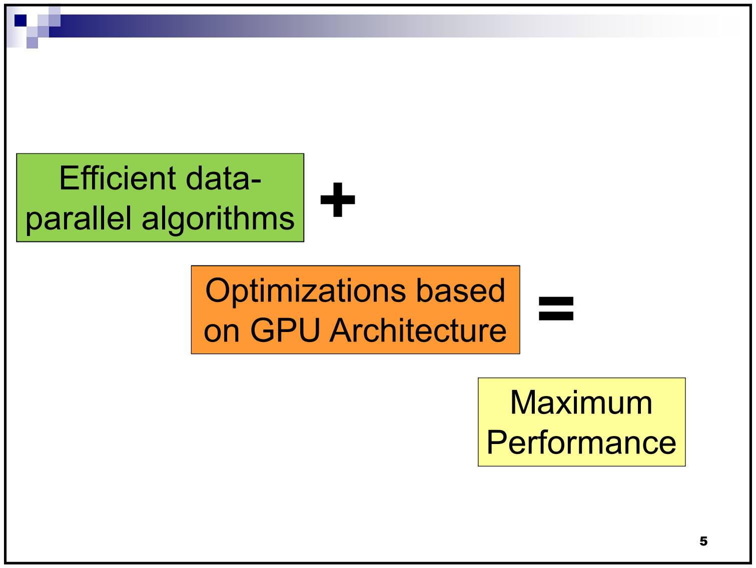
- Project 2 due Friday 10/12
  - Be ready to present on Monday, 10/15
- Move class on Halloween to Tuesday, 10/30?

Acknowledgements

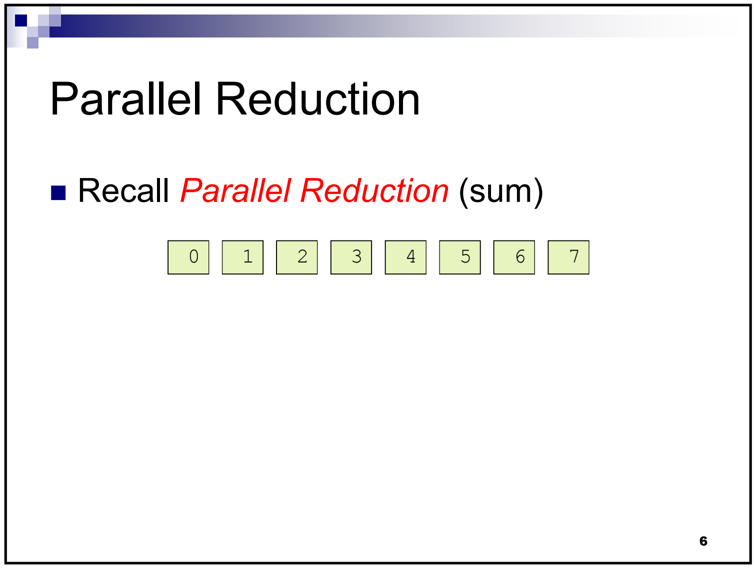
- Some slides from Varun Sampath

Agenda

- Parallel Reduction Revisited
- Warp Partitioning
- Memory Coalescing
- Bank Conflicts
- Dynamic Partitioning of SM Resources
- Data Prefetching
- Instruction Mix
- Loop Unrolling

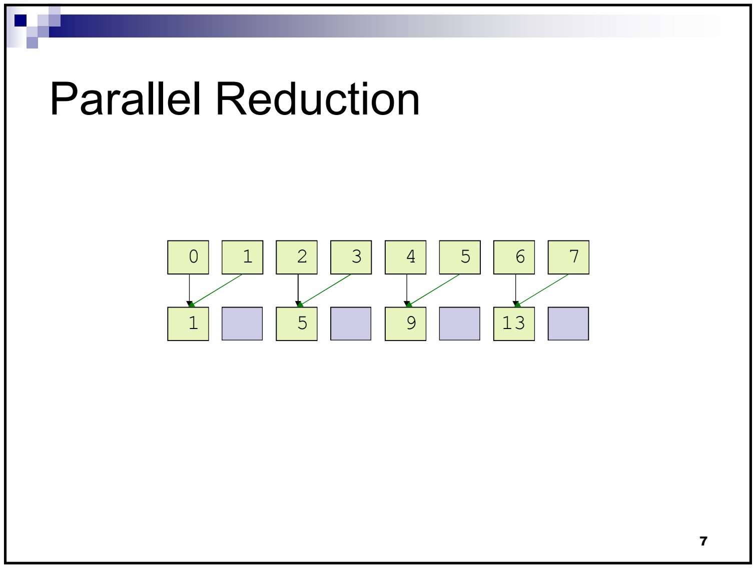


Efficient data-parallel algorithms + Optimizations based on GPU Architecture = Maximum Performance

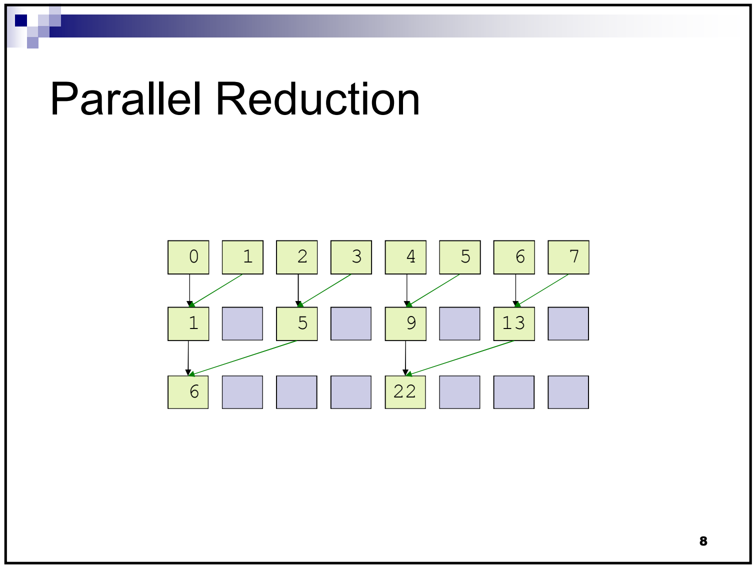


Parallel Reduction

- Recall *Parallel Reduction* (sum)

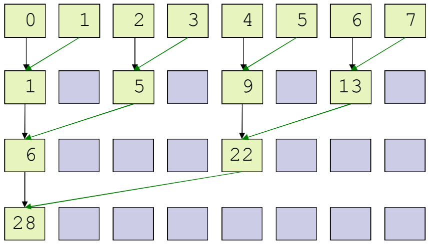


Parallel Reduction



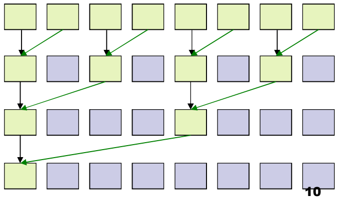
Parallel Reduction

Parallel Reduction



Parallel Reduction

- Similar to brackets for a basketball tournament
- log(`n`) passes for `n` elements
- How would you implement this in CUDA?



```
__shared__ float partialSum[];
// ... load into shared memory
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];
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

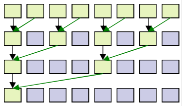
```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
      stride < blockDim.x;
      stride *= 2)
{
  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] +=
      partialSum[t + stride];
}
```

Computing the sum for the elements in shared memory

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__shared__ float partialSum[];
// ... load into shared memory
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];
}
```

Stride:
1, 2, 4, …



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

```
__shared__ float partialSum[];
// ... load into shared memory
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];
}
```

Why?

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

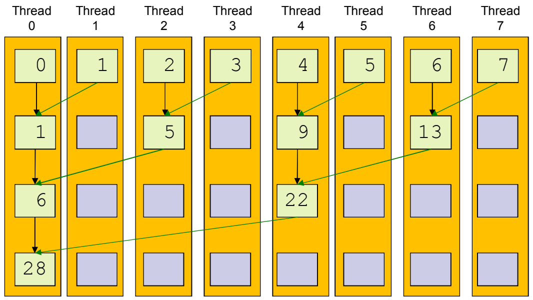
```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
       stride < blockDim.x;
       stride *= 2)
{
  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] +=
      partialSum[t + stride];
}
```

- Compute sum in same shared memory
- As stride increases, what do more threads do?

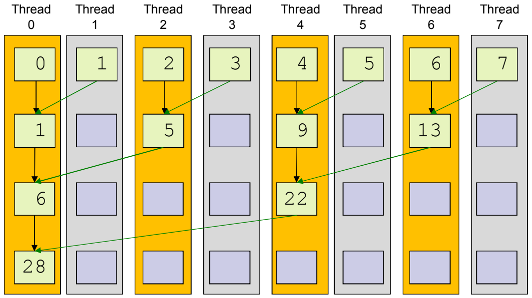
Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Parallel Reduction

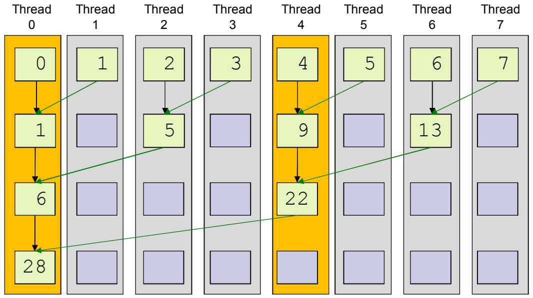


Parallel Reduction



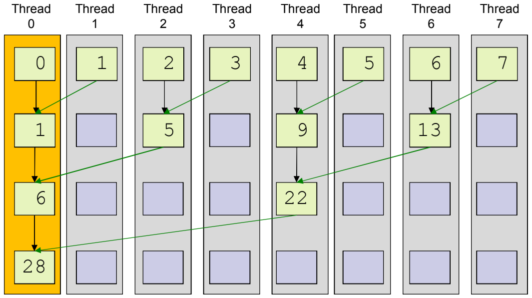
- 1st pass: threads 1, 3, 5, and 7 don't do anything
  - Really only need `n/2` threads for `n` elements

Parallel Reduction



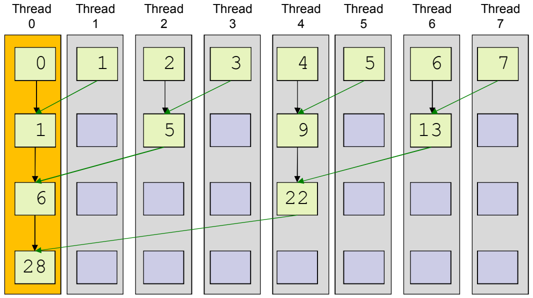
- 2nd pass: threads 2 and 6 also don't do anything

Parallel Reduction



- 3rd pass: thread 4 also doesn't do anything

Parallel Reduction

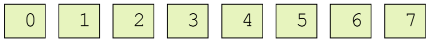


- In general, number of required threads cuts in half after each pass

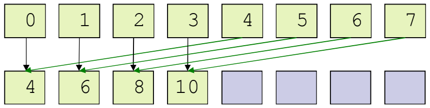
Parallel Reduction

- What if we *tweaked* the implementation?

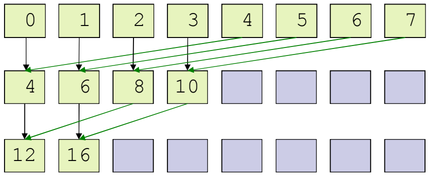
Parallel Reduction

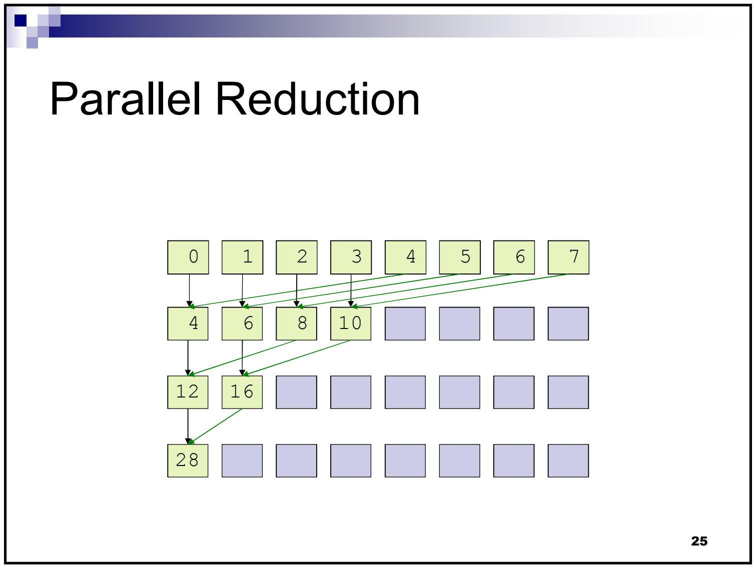


Parallel Reduction

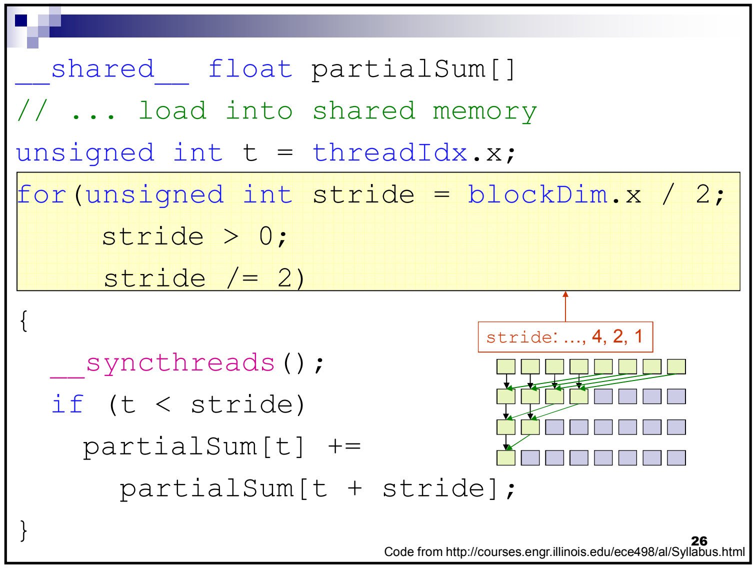


Parallel Reduction





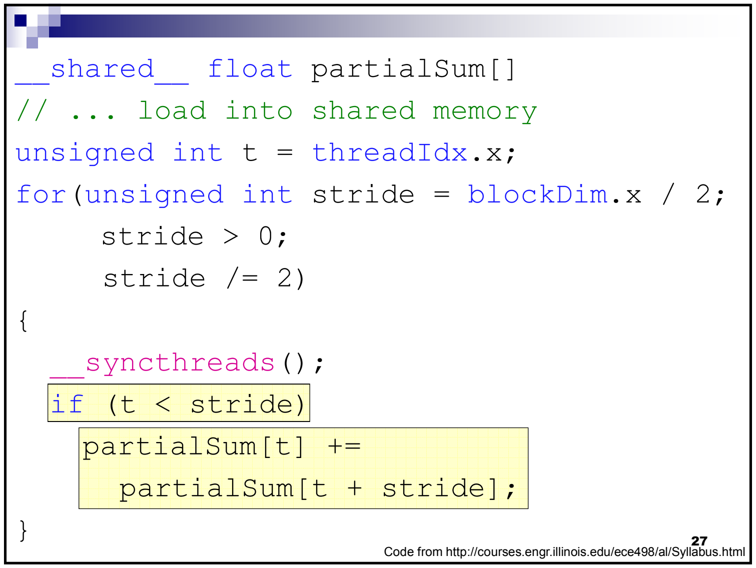
Parallel Reduction



```
__shared__ float partialSum[]
// ... load into shared memory
unsigned int t = threadIdx.x;
for(unsigned int stride = blockDim.x / 2;
      stride > 0;
      stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] +=
      partialSum[t + stride];
}
```

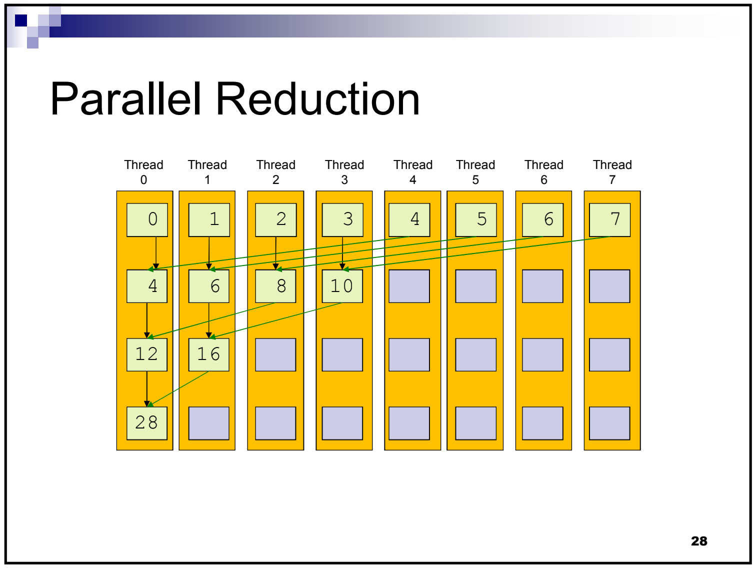
stride: …, 4, 2, 1

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html



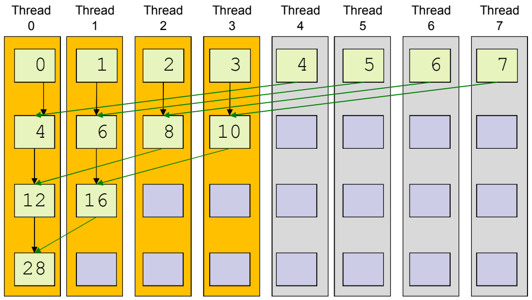
```
__shared__ float partialSum[]
// ... load into shared memory
unsigned int t = threadIdx.x;
for(unsigned int stride = blockDim.x / 2;
      stride > 0;
      stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] +=
      partialSum[t + stride];
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html



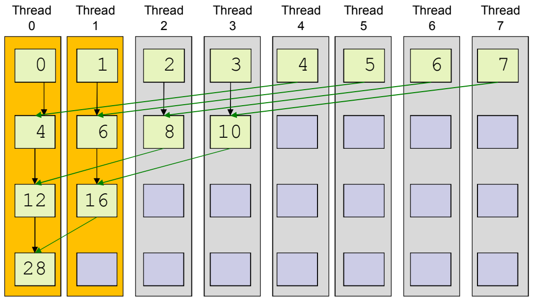
Parallel Reduction

Parallel Reduction



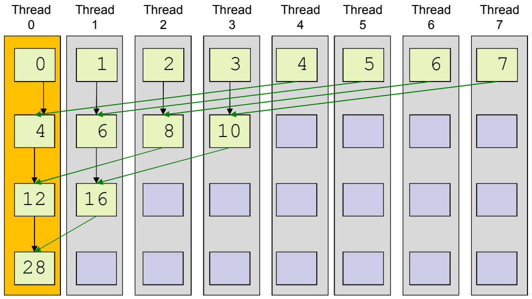
- 1st pass: threads 4, 5, 6, and 7 don't do anything
  - Really only need `n/2` threads for `n` elements

Parallel Reduction



- 2nd pass: threads 2 and 3 also don't do anything

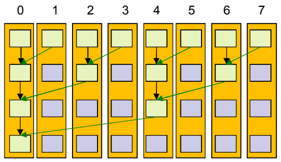
Parallel Reduction



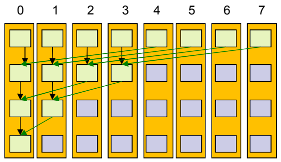
- 3rd pass: thread 1 also doesn't do anything

Parallel Reduction

- What is the difference?



`stride` = 1, 2, 4, …



`stride` = 4, 2, 1, …

Parallel Reduction

- What is the difference?

```
if (t % (2 * stride) == 0)
  partialSum[t] +=
    partialSum[t + stride];
```

`stride` = 1, 2, 4, …

```
if (t < stride)
  partialSum[t] +=
    partialSum[t + stride];
```

`stride` = 4, 2, 1, …

Warp Partitioning

- *Warp Partitioning*: how threads from a block are divided into warps
- Knowledge of warp partitioning can be used to:
  - Minimize divergent branches
  - Retire warps early

Warp Partitioning

- Partition based on *consecutive increasing* `threadIdx`

Warp Partitioning

- 1D Block
  - `threadIdx.x` between 0 and 512 (G80/GT200)
  - Warp `n`
    - Starts with thread `32n`
    - Ends with thread `32(n + 1) – 1`
  - Last warp is padded if block size is not a multiple of 32

Warp 0	Warp 1	Warp 2	Warp 3	
0…31	32...63	64...95	96...127	…

Warp Partitioning

- 2D Block
  - ☐ Increasing `threadIdx` means
    - Increasing `threadIdx.x`
    - Starting with row `threadIdx.y == 0`

Warp Partitioning

- 2D Block

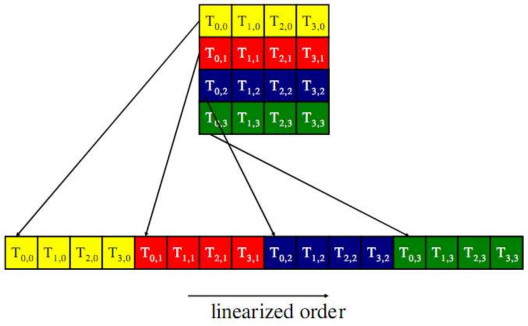


Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

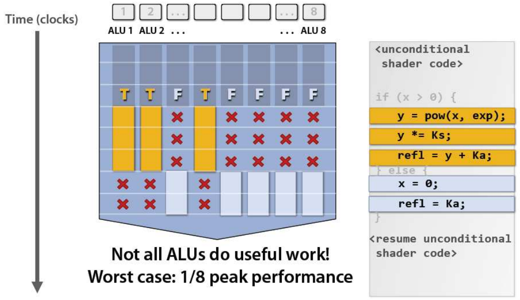
---

Warp Partitioning

- 3D Block
  - ☐ Start with `threadIdx.z == 0`
  - ☐ Partition as a 2D block
  - ☐ Increase `threadIdx.z` and repeat

Warp Partitioning

Divergent branches are within a warp!



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

Warp Partitioning

- For `warpSize == 32`, does any warp have a divergent branch with this code:

```
if (threadIdx.x > 15)
{
  // ...
}
```

Warp Partitioning

- For any `warpSize > 1`, does any warp have a divergent branch with this code:

```
if (threadIdx.x > warpSize - 1)
{
  // ...
}
```

Warp Partitioning

- Given knowledge of warp partitioning, which parallel reduction is better?

```
if (t % (2 * stride) == 0)
  partialSum[t] +=
    partialSum[t + stride];
```

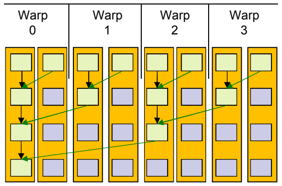
`stride` = 1, 2, 4, …

```
if (t < stride)
  partialSum[t] +=
    partialSum[t + stride];
```

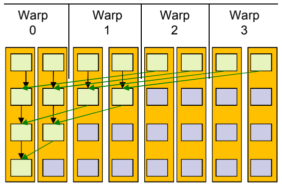
`stride` = 4, 2, 1, …

Warp Partitioning

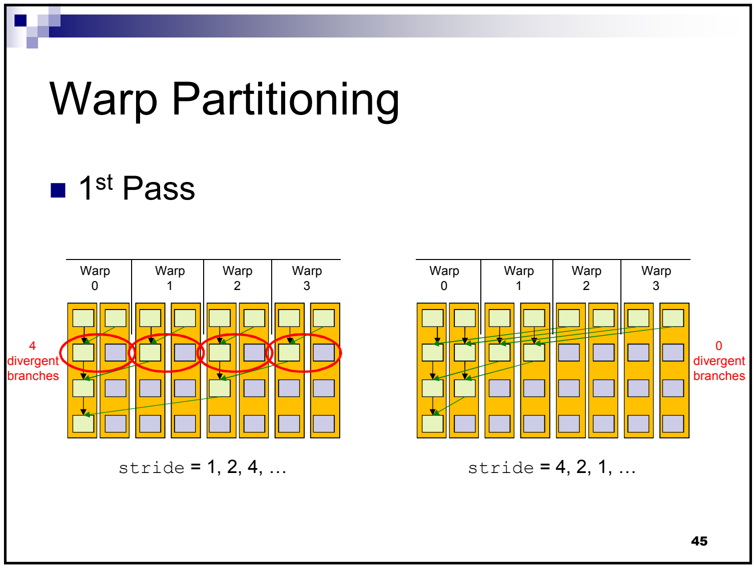
- Pretend `warpSize == 2`



`stride` = 1, 2, 4, …



`stride` = 4, 2, 1, …

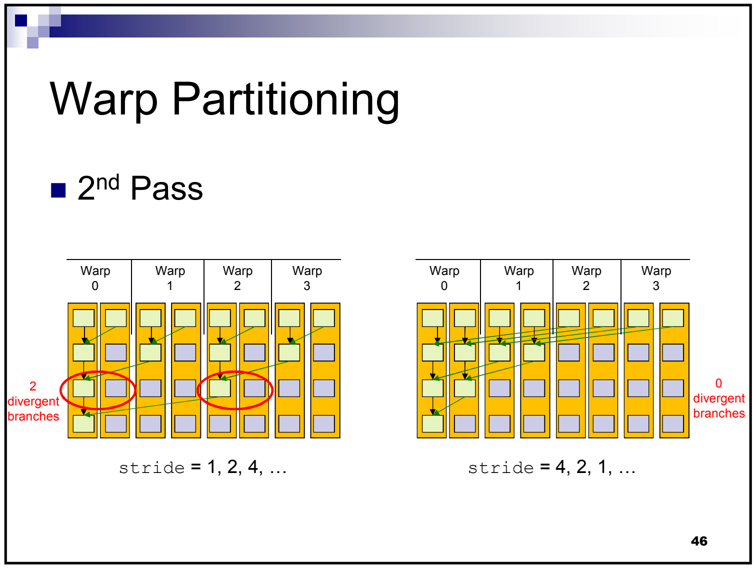


Warp Partitioning

- 1st Pass

`stride` = 1, 2, 4, …

`stride` = 4, 2, 1, …

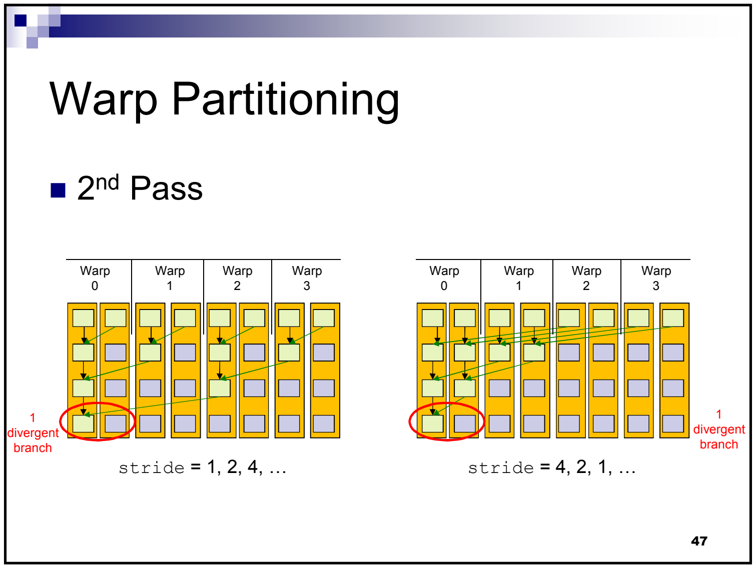


Warp Partitioning

- 2nd Pass

`stride` = 1, 2, 4, …

`stride` = 4, 2, 1, …

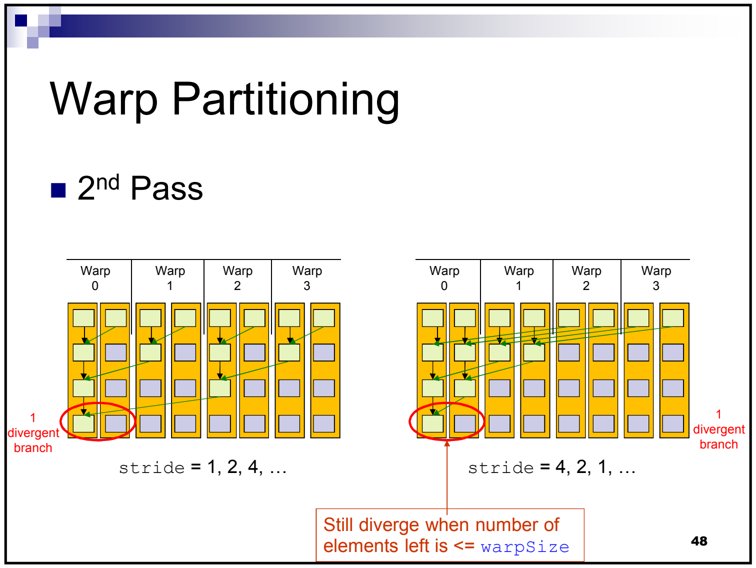


Warp Partitioning

- 2nd Pass

`stride` = 1, 2, 4, …

`stride` = 4, 2, 1, …



Warp Partitioning

- 2nd Pass

`stride` = 1, 2, 4, …

`stride` = 4, 2, 1, …

Still diverge when number of elements left is <= `warpSize`

Warp Partitioning

- Good partitioning also allows warps to be retired early.
  - Better hardware utilization

```
if (t % (2 * stride) == 0)
  partialSum[t] +=
    partialSum[t + stride];
```

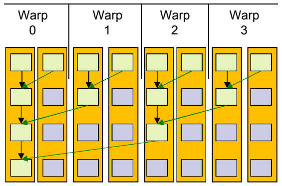
`stride` = 1, 2, 4, …

```
if (t < stride)
  partialSum[t] +=
    partialSum[t + stride];
```

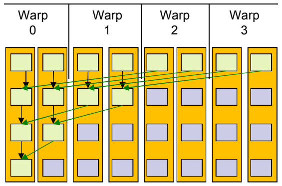
`stride` = 4, 2, 1, …

Warp Partitioning

- Parallel Reduction



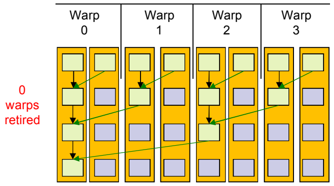
`stride` = 1, 2, 4, …



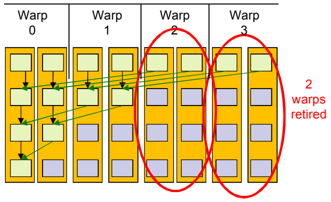
`stride` = 4, 2, 1, …

Warp Partitioning

- 1st Pass



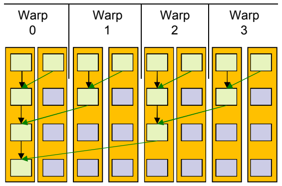
`stride` = 1, 2, 4, …



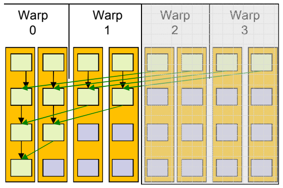
`stride` = 4, 2, 1, …

Warp Partitioning

- 1st Pass



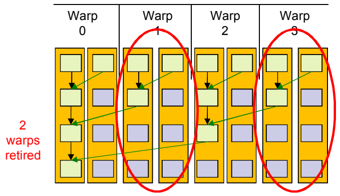
`stride` = 1, 2, 4, …



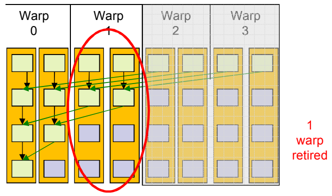
`stride` = 4, 2, 1, …

Warp Partitioning

- 2nd Pass



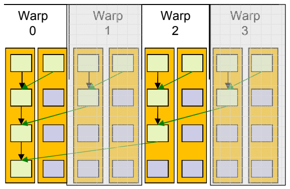
`stride` = 1, 2, 4, …



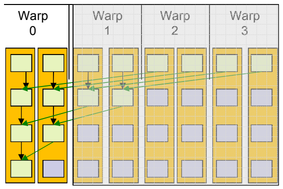
`stride` = 4, 2, 1, …

Warp Partitioning

- 2nd Pass



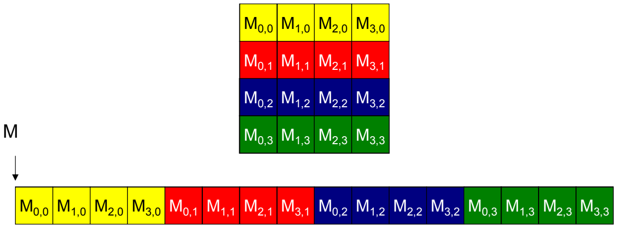
`stride` = 1, 2, 4, …



`stride` = 4, 2, 1, …

Memory Coalescing

- Given a matrix stored *row-major* in *global memory*, what is a *thread*'s desirable access pattern?



$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

M

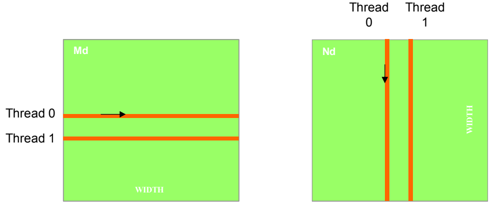
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

---

Memory Coalescing

- Given a matrix stored *row-major* in *global memory*, what is a *thread*'s desirable access pattern?



a) column after column?

b) row after row?

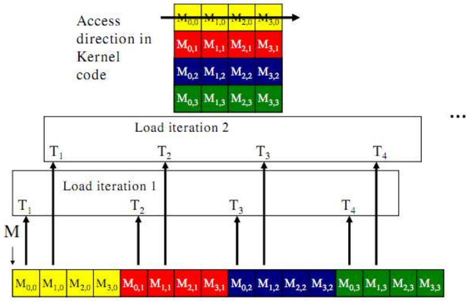
Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Memory Coalescing

- Given a matrix stored *row-major* in *global memory*, what is a *thread*'s desirable access pattern?
  - a) column after column
    - *Individual threads* read increasing, consecutive memory address
  - b) row after row
    - *Adjacent threads* read increasing, consecutive memory addresses

---

Memory Coalescing

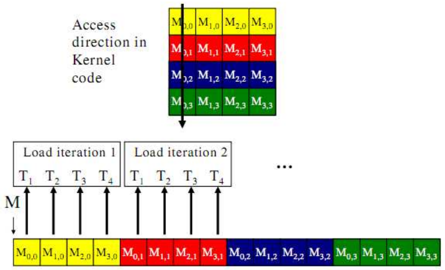


a) column after column

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

---

Memory Coalescing

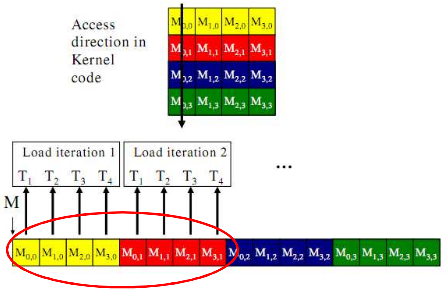


b) row after row

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

---

Memory Coalescing



Recall warp partitioning; if these threads are in the same warp, global memory addresses are increasing and consecutive across warps.

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf

Memory Coalescing

- Global memory bandwidth (DRAM)
  - G80 – 86.4 GB/s
  - GT200 – 150 GB/s
- Achieve peak bandwidth by requesting large, consecutive locations from DRAM
  - Accessing random location results in much lower bandwidth

---

Memory Coalescing

- *Memory coalescing* – rearrange access patterns to improve performance
- Useful today but will be less useful with large on-chip caches

---

Memory Coalescing

- The GPU coalesce consecutive reads in a *half-warp* into a single read
- *Strategy*: read global memory in a coalesce-able fashion into shared memory
  - Then access shared memory randomly at maximum bandwidth
    - Ignoring *bank conflicts*…

See Appendix G in the NVIDIA CUDA C Programming Guide for coalescing alignment requirements

---

Bank Conflicts

- Shared Memory
  - Sometimes called a *parallel data cache*
    - Multiple threads can access shared memory at the same time
  - Memory is divided into *banks*

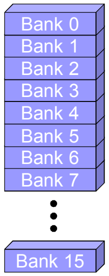


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Banks
  - ☐ Each bank can service one address per two cycles
  - ☐ Per-bank bandwidth: 32-bits per two (shader clock) cycles
  - ☐ Successive 32-bit words are assigned to successive banks

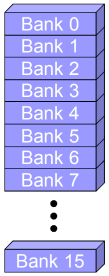


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Bank Conflicts

- *Bank Conflict*: Two simultaneous accesses to the same bank, but not the same address
  - ☐ Serialized
- G80-GT200: 16 banks, with 8 SPs concurrently executing
- Fermi: 32 banks, with 16 SPs concurrently executing
  - ☐ What does this mean for conflicts?

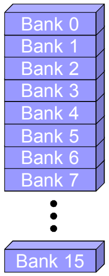


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Bank Conflicts

- Bank Conflicts?
  - ☐ Linear addressing stride == 1
- Bank Conflicts?
  - ☐ Random 1:1 Permutation

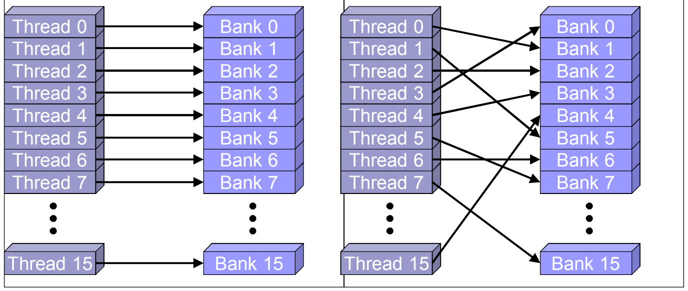


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Bank Conflicts

- Bank Conflicts?
  - ☐ Linear addressing stride == 2
- Bank Conflicts?
  - ☐ Linear addressing stride == 8

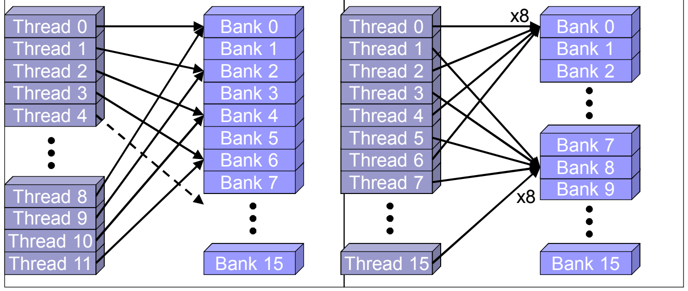


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bank Conflicts

- Fast Path 1 (G80)
  - All threads in a half-warp access different banks

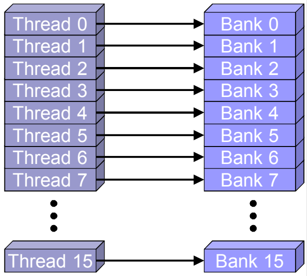


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Bank Conflicts

- Fast Path 2 (G80)
  - All threads in a half-warp access the same address

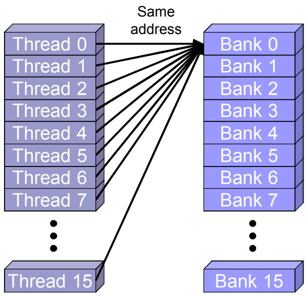


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Bank Conflicts

- Slow Path (G80)
  - Multiple threads in a half-warp access the same bank
  - Access is serialized
  - What is the cost?

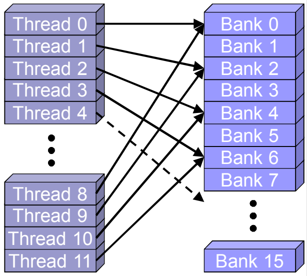


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Bank Conflicts

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

- For what values of `s` is this conflict free?
  - Hint: The G80 has 16 banks

Bank Conflicts

```
__shared__ float shared[256];
// ...
float f = shared[index + s * threadIdx.x];
```

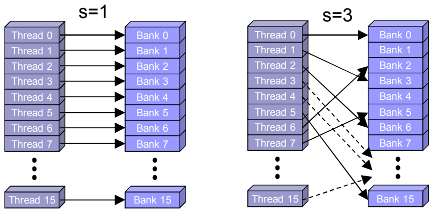


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

---

Bank Conflicts

- Without using a profiler, how can we tell what kind of speedup we can expect by removing bank conflicts?
- What happens if more than one thread in a warp writes to the same shared memory address (non-atomic instruction)?

SM Resource Partitioning

- Recall a SM dynamically partitions resources:

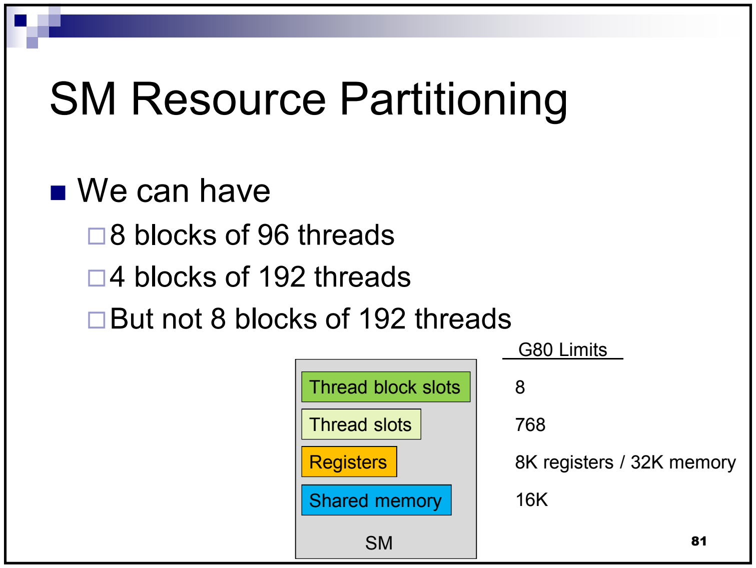
Thread block slots
Thread slots
Registers
Shared memory
SM

SM Resource Partitioning

- Recall a SM dynamically partitions resources:

	G80 Limits
Thread block slots	8
Thread slots	768
Registers	8K registers / 32K memory
Shared memory	16K

SM

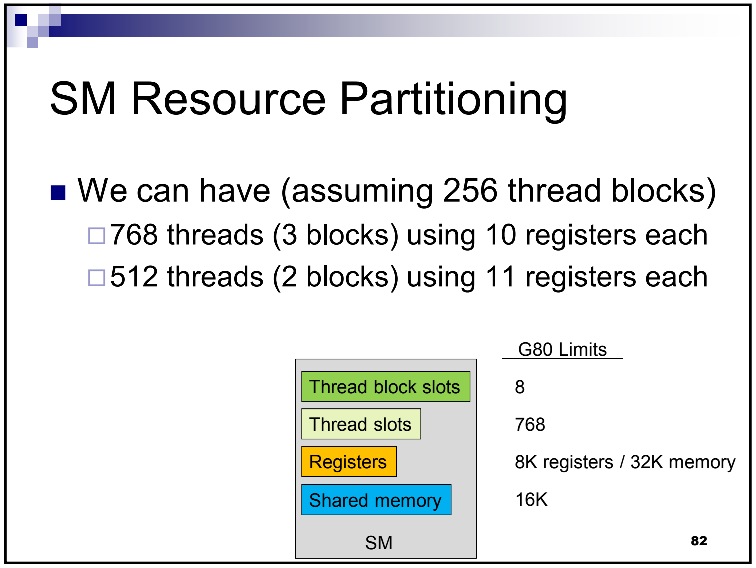


SM Resource Partitioning

- We can have
  - 8 blocks of 96 threads
  - 4 blocks of 192 threads
  - But not 8 blocks of 192 threads

	G80 Limits
Thread block slots	8
Thread slots	768
Registers	8K registers / 32K memory
Shared memory	16K

SM

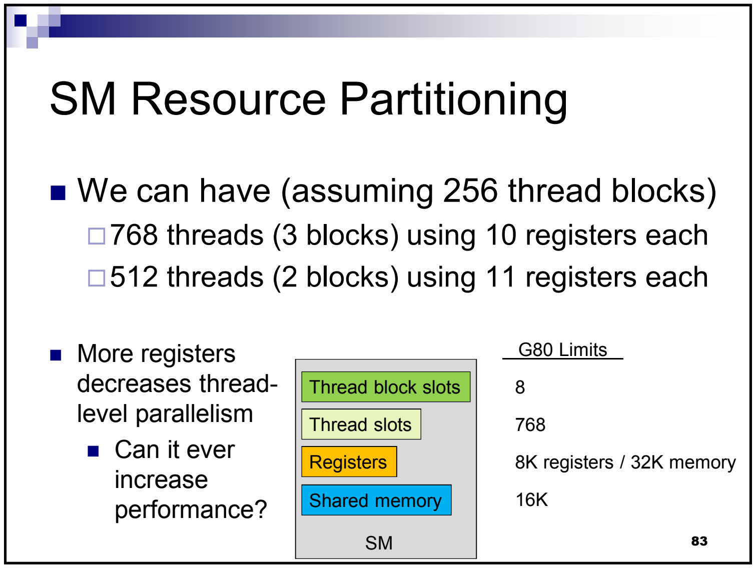


SM Resource Partitioning

- We can have (assuming 256 thread blocks)
  - 768 threads (3 blocks) using 10 registers each
  - 512 threads (2 blocks) using 11 registers each

	G80 Limits
Thread block slots	8
Thread slots	768
Registers	8K registers / 32K memory
Shared memory	16K

SM

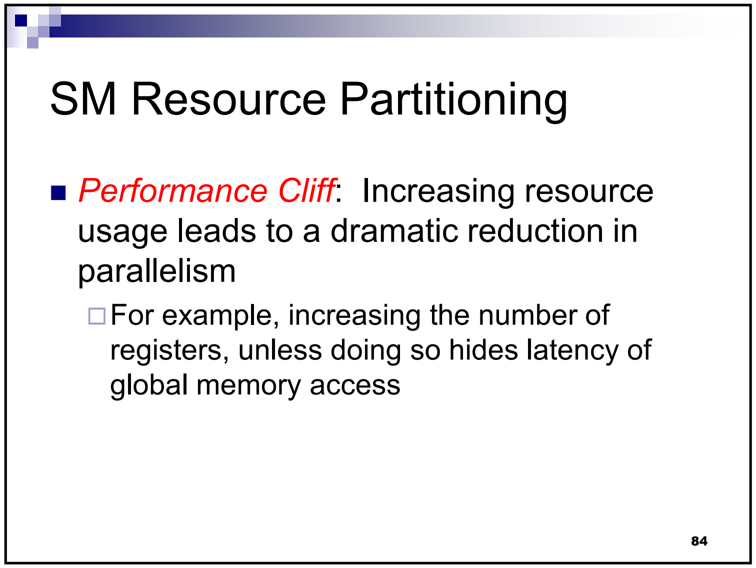


SM Resource Partitioning

- We can have (assuming 256 thread blocks)
  - 768 threads (3 blocks) using 10 registers each
  - 512 threads (2 blocks) using 11 registers each
- More registers decreases thread-level parallelism
  - Can it ever increase performance?

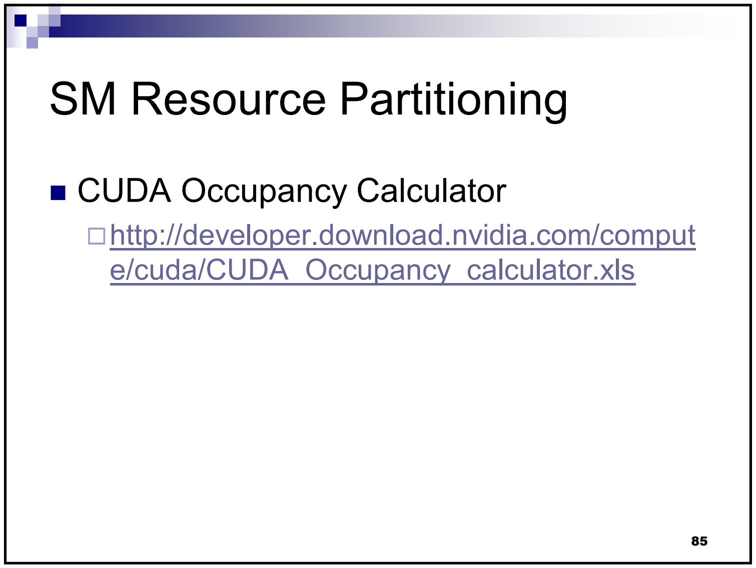
	G80 Limits
Thread block slots	8
Thread slots	768
Registers	8K registers / 32K memory
Shared memory	16K

SM



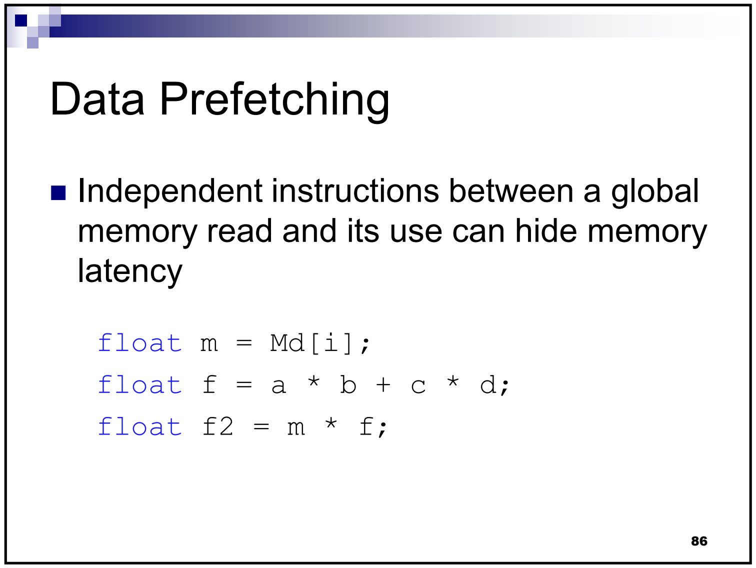
SM Resource Partitioning

- *Performance Cliff*: Increasing resource usage leads to a dramatic reduction in parallelism
  - For example, increasing the number of registers, unless doing so hides latency of global memory access



SM Resource Partitioning

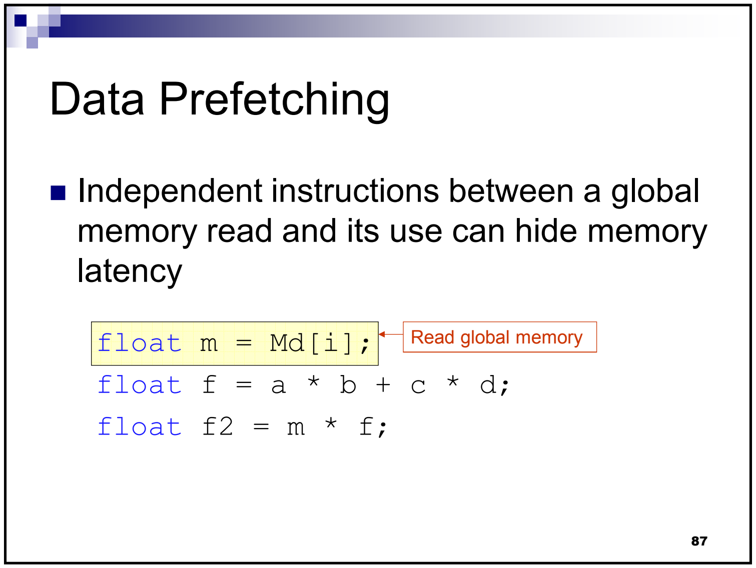
- CUDA Occupancy Calculator
  - http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls



Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

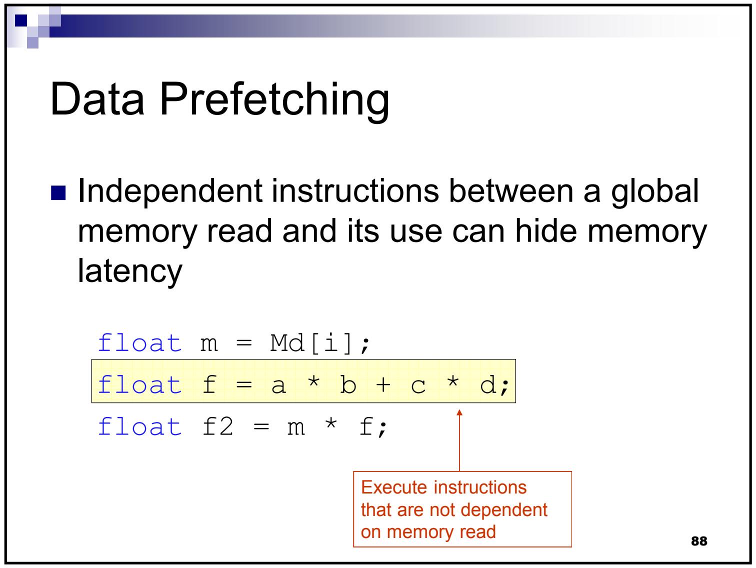


Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Read global memory



Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Execute instructions that are not dependent on memory read

Data Prefetching

- Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Use global memory after the above line from enough warps hide the memory latency

---

Data Prefetching

- *Prefetching* data from global memory can effectively increase the number of independent instructions between global memory read and use

---

Data Prefetching

- Recall tiled matrix multiply:

```
for (/* ... */)
{
  // Load current tile into shared memory
  __syncthreads();
  // Accumulate dot product
  __syncthreads();
}
```

Data Prefetching

- Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

Data Prefetching

- Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

Data Prefetching

- Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

Prefetch for next iteration of the loop

Data Prefetching

- Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
  // Deposit registers into shared memory
  __syncthreads();
  // Load next tile into registers
  // Accumulate dot product
  __syncthreads();
}
```

These instructions executed by enough threads will hide the memory latency of the prefetch

---

Instruction Mix

- Special Function Units (SFUs)
  - Use to compute `__sinf(), __expf()`
  - Only 4, each can execute 1 instruction per clock

Fermi Streaming Multiprocessor (SM)

Image: NVIDIA Fermi Whitepaper

Loop Unrolling

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

- Instructions per iteration
  - One floating-point multiply
  - One floating-point add
  - What else?

Loop Unrolling

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

- Other instructions per iteration
  - Update loop counter

Loop Unrolling

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

- Other instructions per iteration
  - Update loop counter
  - Branch

Loop Unrolling

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

- Other instructions per iteration
  - Update loop counter
  - Branch
  - Address arithmetic

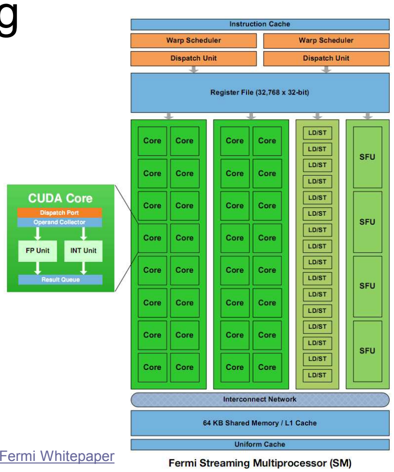
Loop Unrolling

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

- Instruction Mix
  - 2 floating-point arithmetic instructions
  - 1 loop branch instruction
  - 2 address arithmetic instructions
  - 1 loop counter increment instruction

Loop Unrolling

- Only 1/3 are floating-point calculations
  - But I want my full theoretical 1 TFLOP (Fermi)
  - Consider *loop unrolling*



Fermi Streaming Multiprocessor (SM)

Image: NVIDIA Fermi Whitepaper

Loop Unrolling

```
Pvalue +=
  Ms[ty][0] * Ns[0][tx] +
  Ms[ty][1] * Ns[1][tx] +
  ...
  Ms[ty][15] * Ns[15][tx]; // BLOCK_SIZE = 16
```

- No more loop
  - No loop count update
  - No branch
  - Constant indices – no address arithmetic instructions

Loop Unrolling

- Automatically:

```
#pragma unroll BLOCK_SIZE
for (int k = 0; k < BLOCK_SIZE; ++k)
{
  Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

- Disadvantages to unrolling?