Basic CUDA Optimization

Bin ZHOU @ NVIDIA & USTC

Jan. 2015

Agenda

- Parallel Reduction
- 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

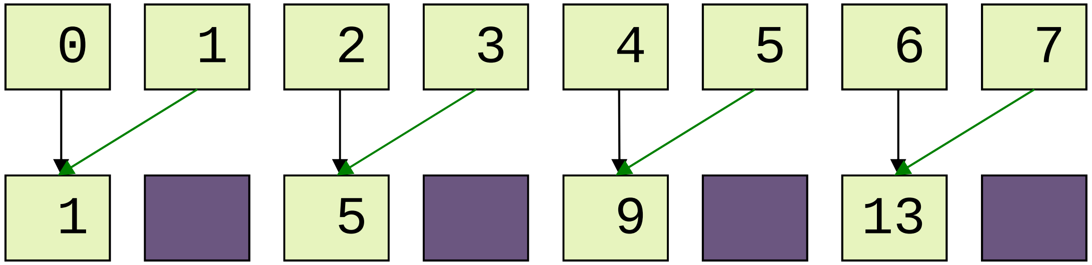
Parallel Reduction

- Recall *Parallel Reduction* (sum)

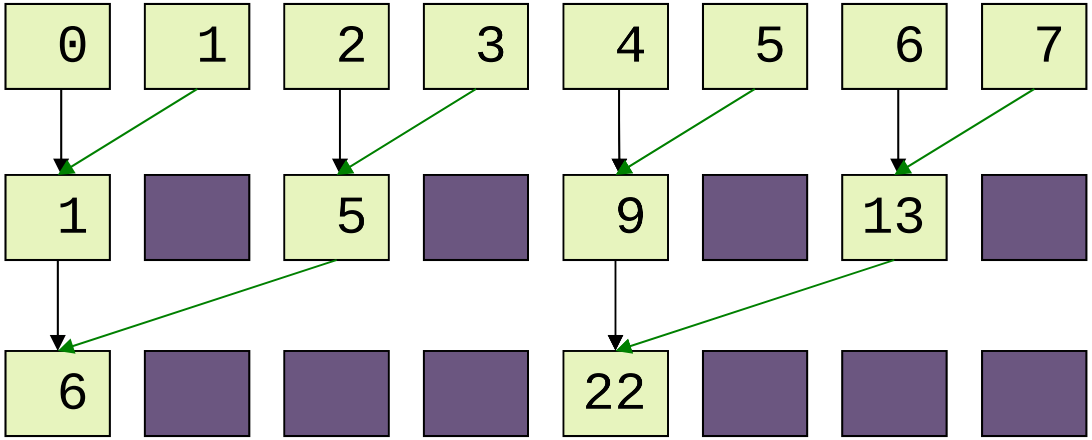
0	1	2	3	4	5	6	7

- S= $\sum_{i=0}^{N} a_i$

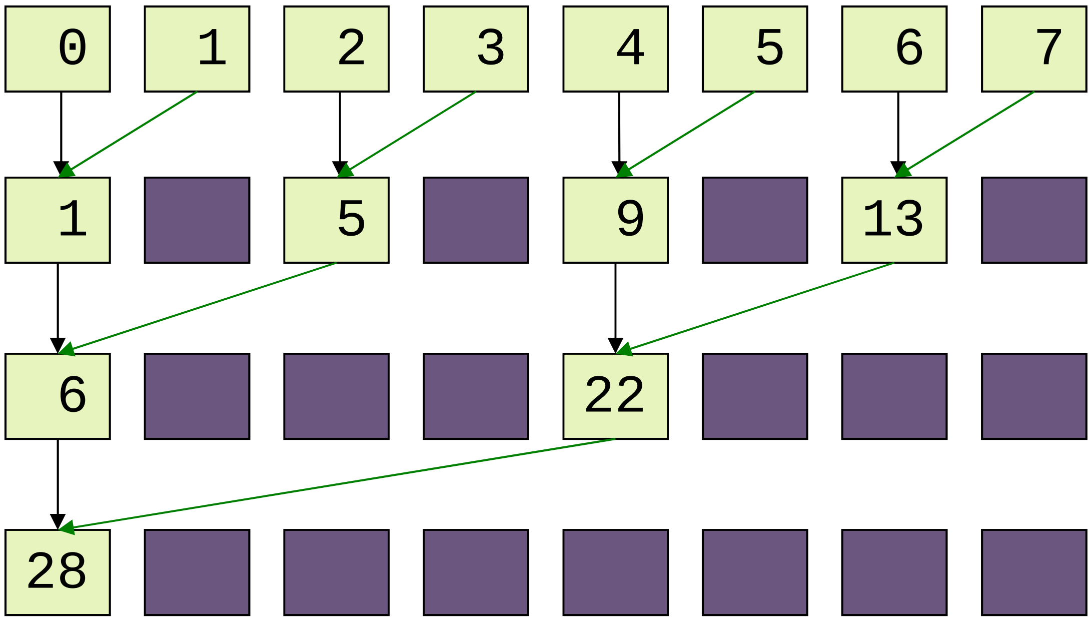
Parallel Reduction



Parallel Reduction



Parallel Reduction

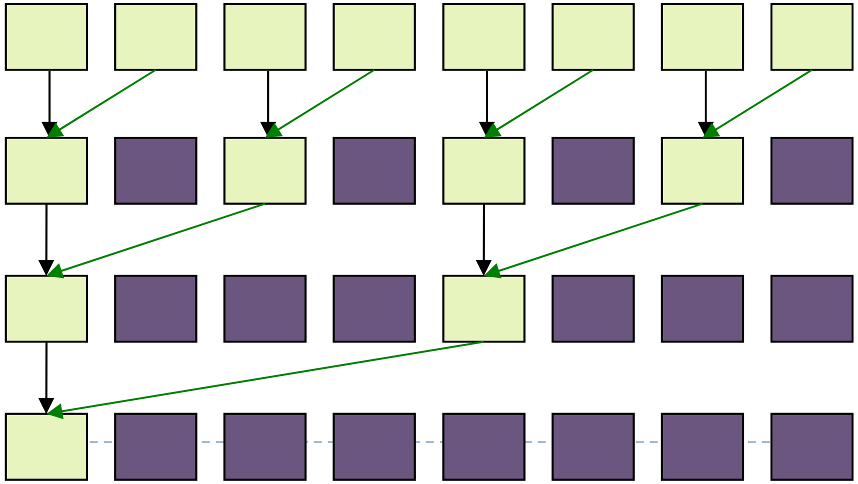


Parallel Reduction

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

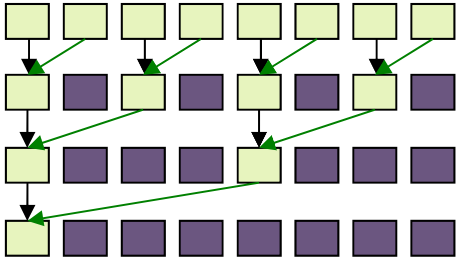
```
__shared__ float partialSum[];
// ... load into shared memory
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;
     stride *= 2)
{
  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] +=
      partialSum[t + stride];
}
```

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?

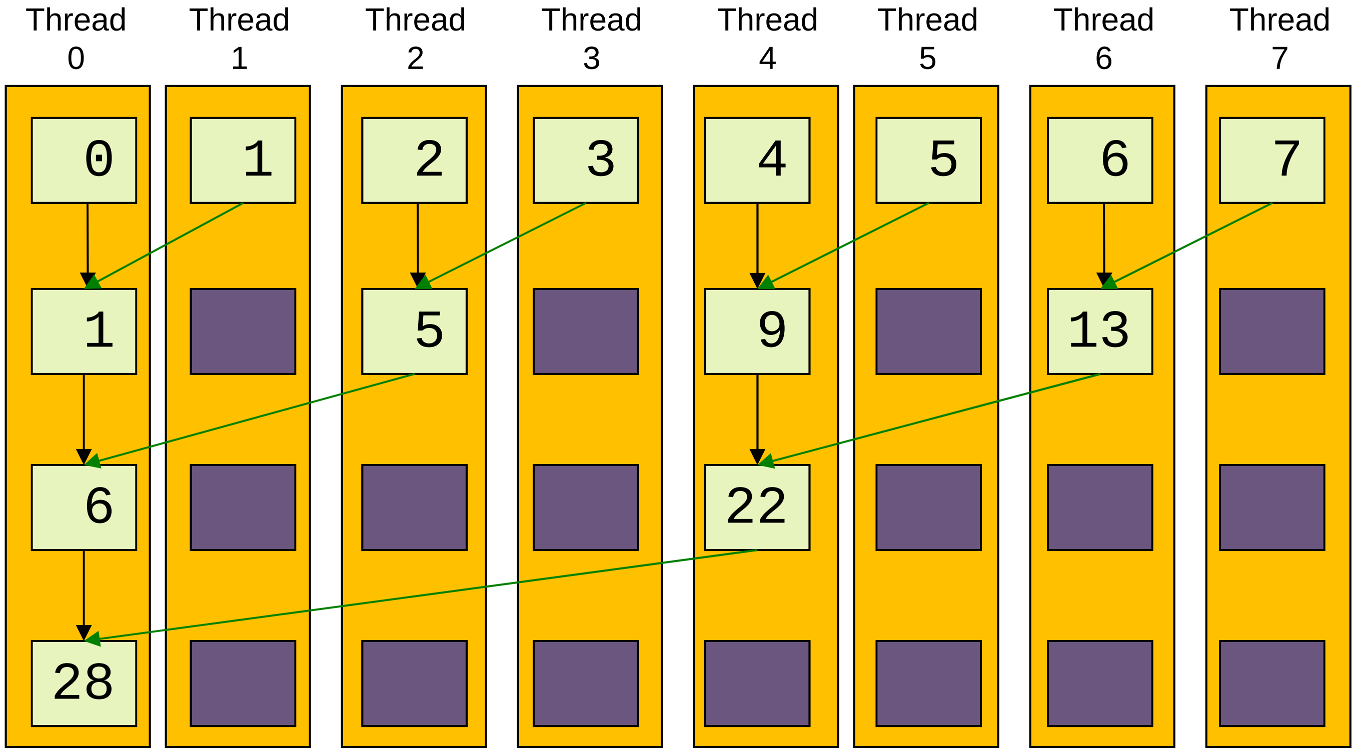
Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__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];
}
```

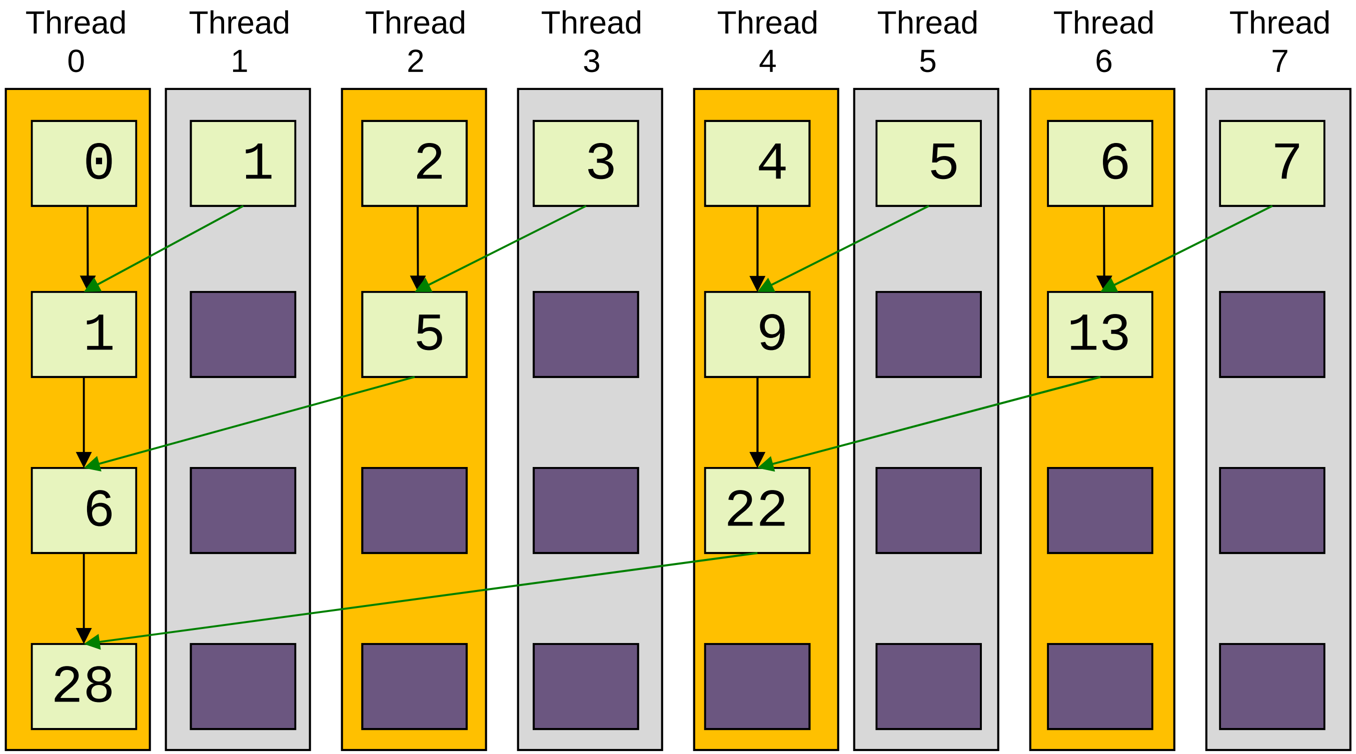
- Compute sum in same shared memory
- As stride increases, what do more threads do?

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

Parallel Reduction



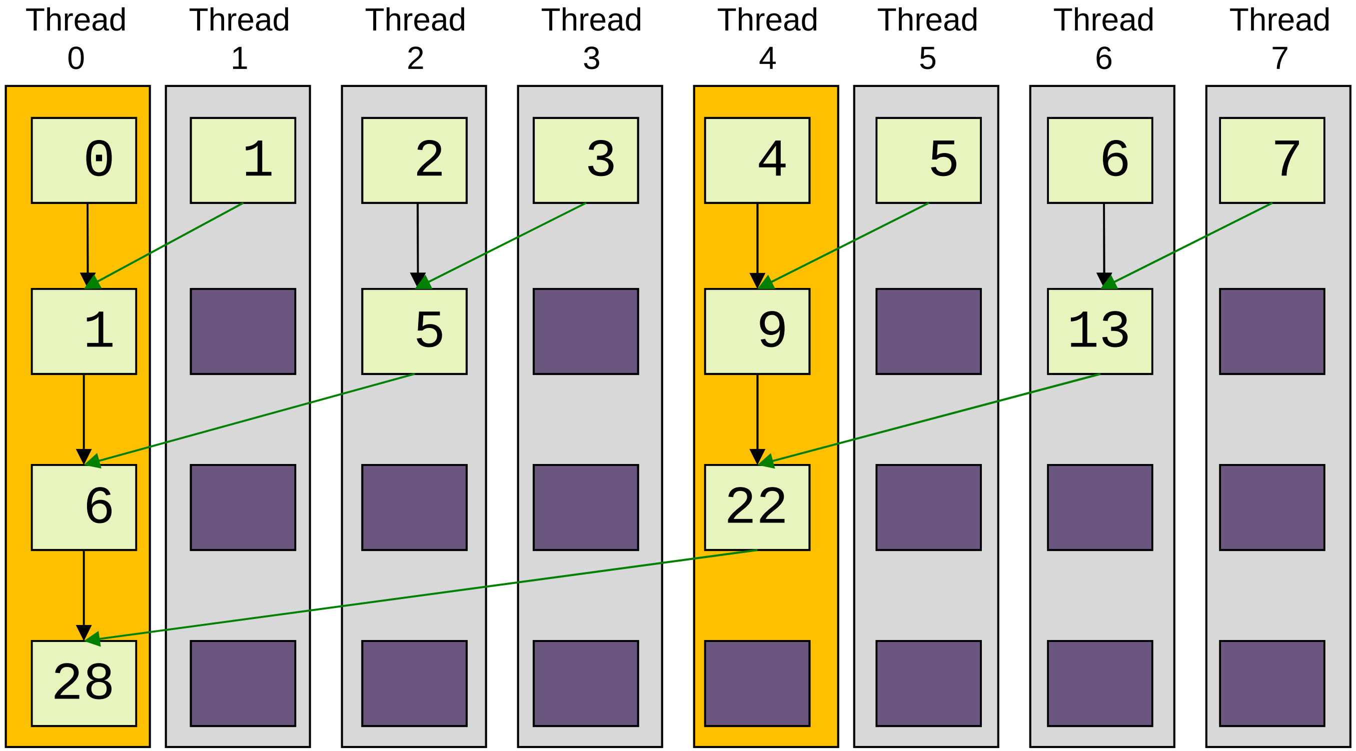
Parallel Reduction



1st pass: threads 1, 3, 5, and 7 don' t do anything

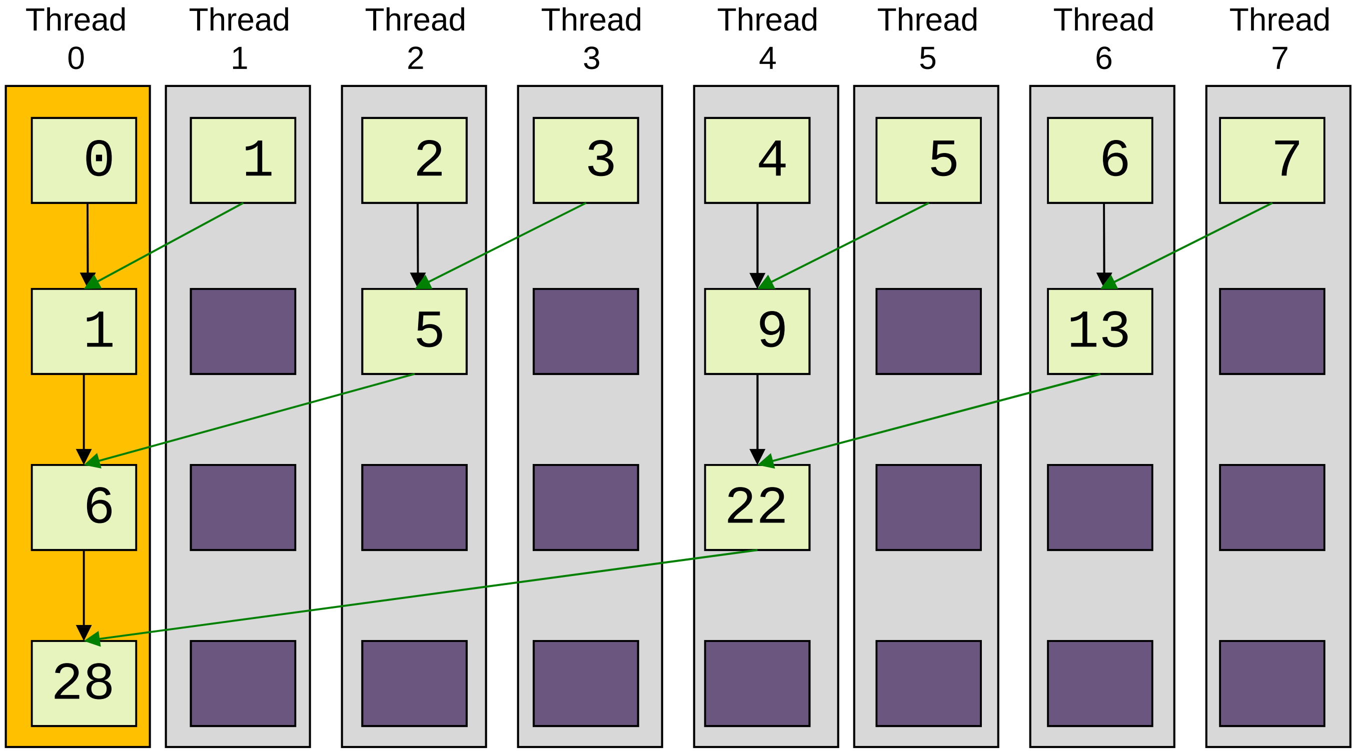
Really only need n/2 threads for n elements

Parallel Reduction



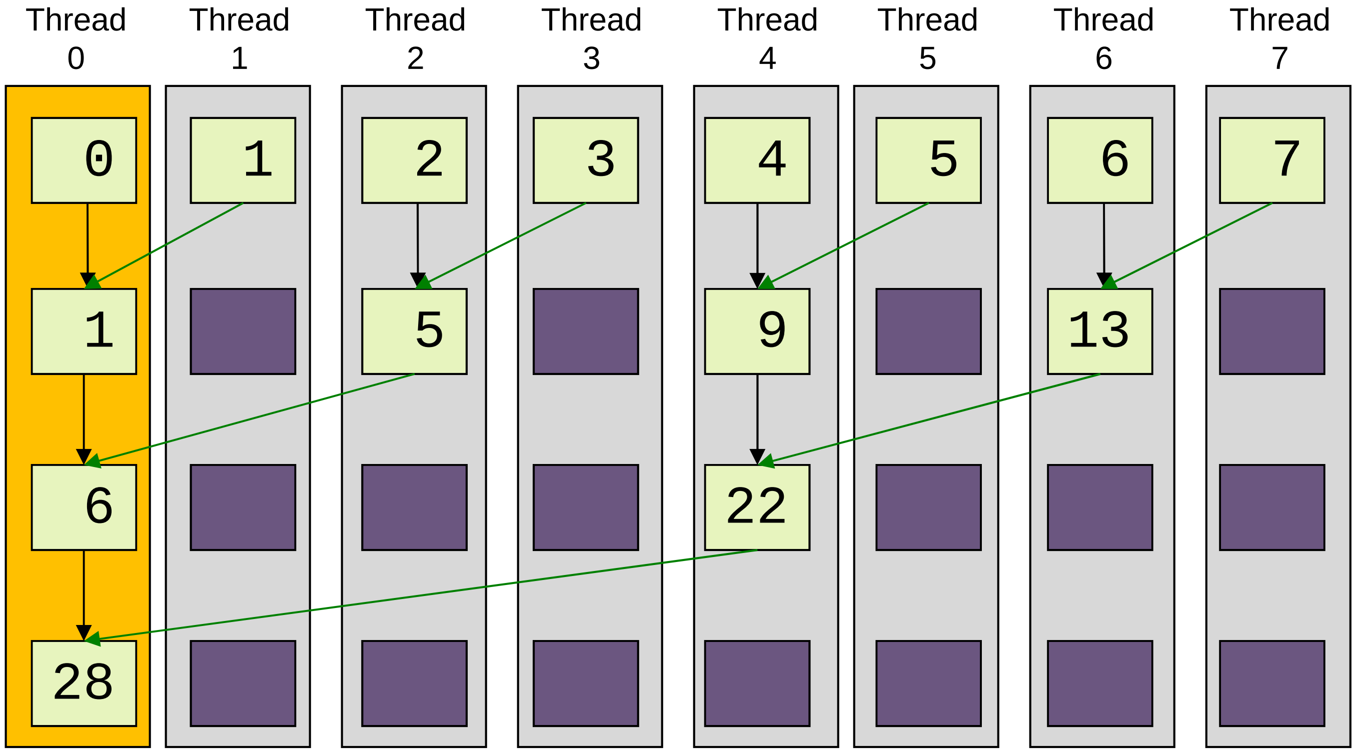
2nd pass: threads 2 and 6 also don’ t do anything

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

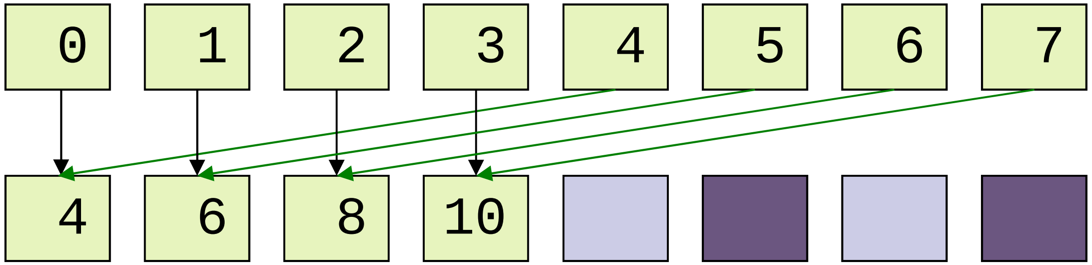
Parallel Reduction

What if we *tweaked* the implementation?

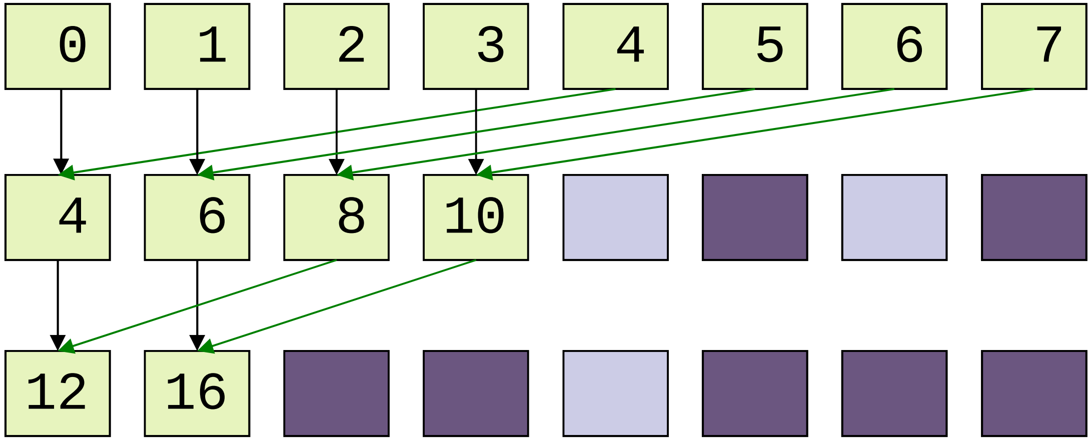
Parallel Reduction

0	1	2	3	4	5	6	7

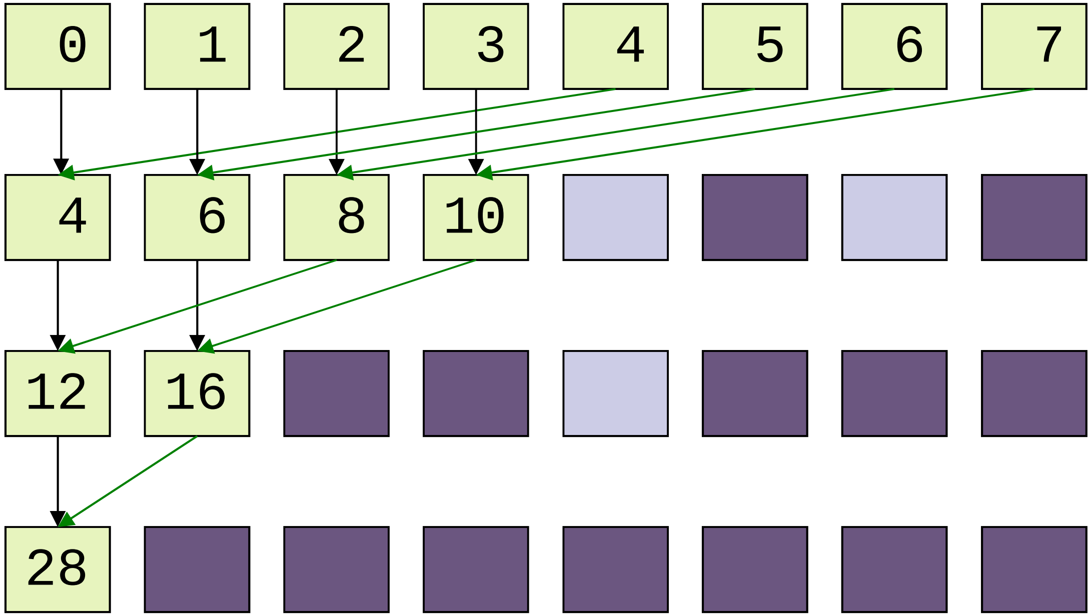
Parallel Reduction



Parallel Reduction

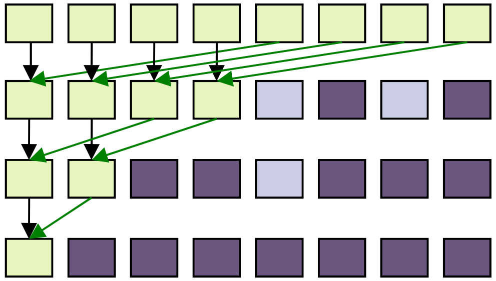


Parallel Reduction



```
__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

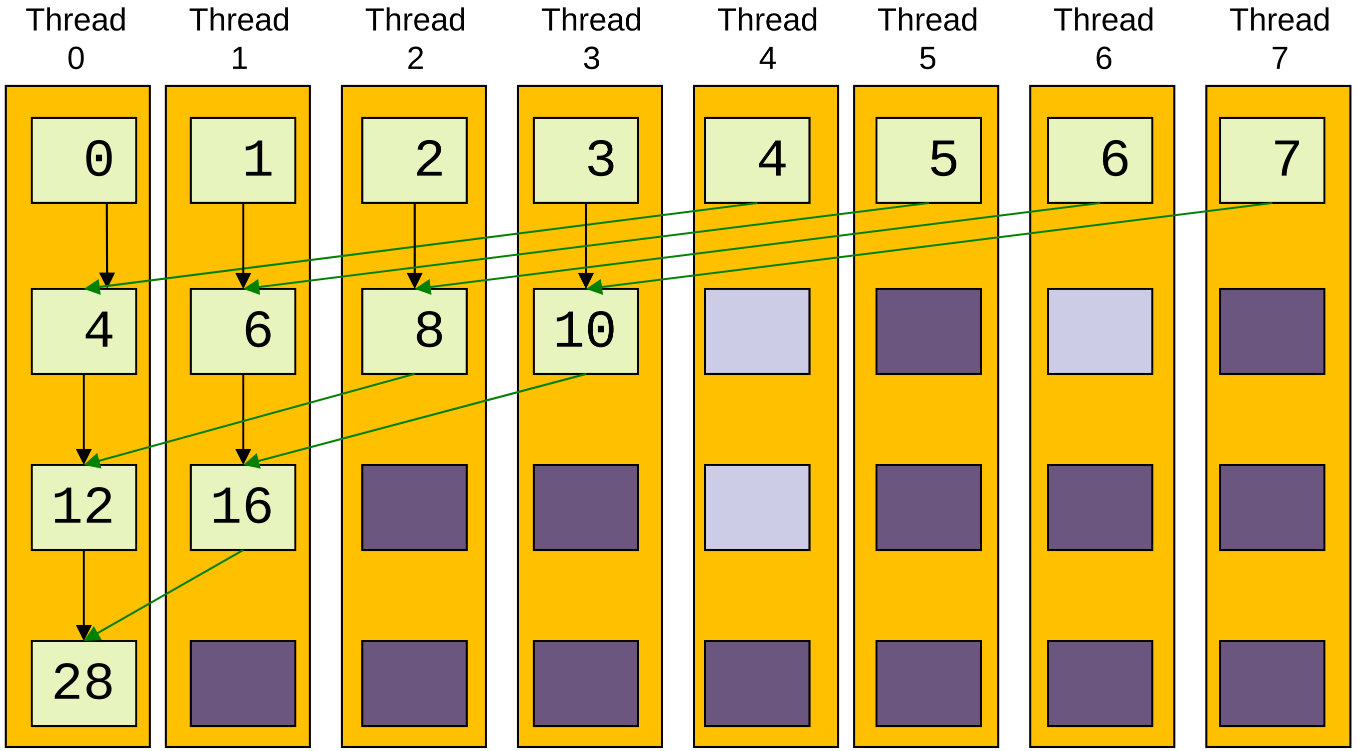


Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

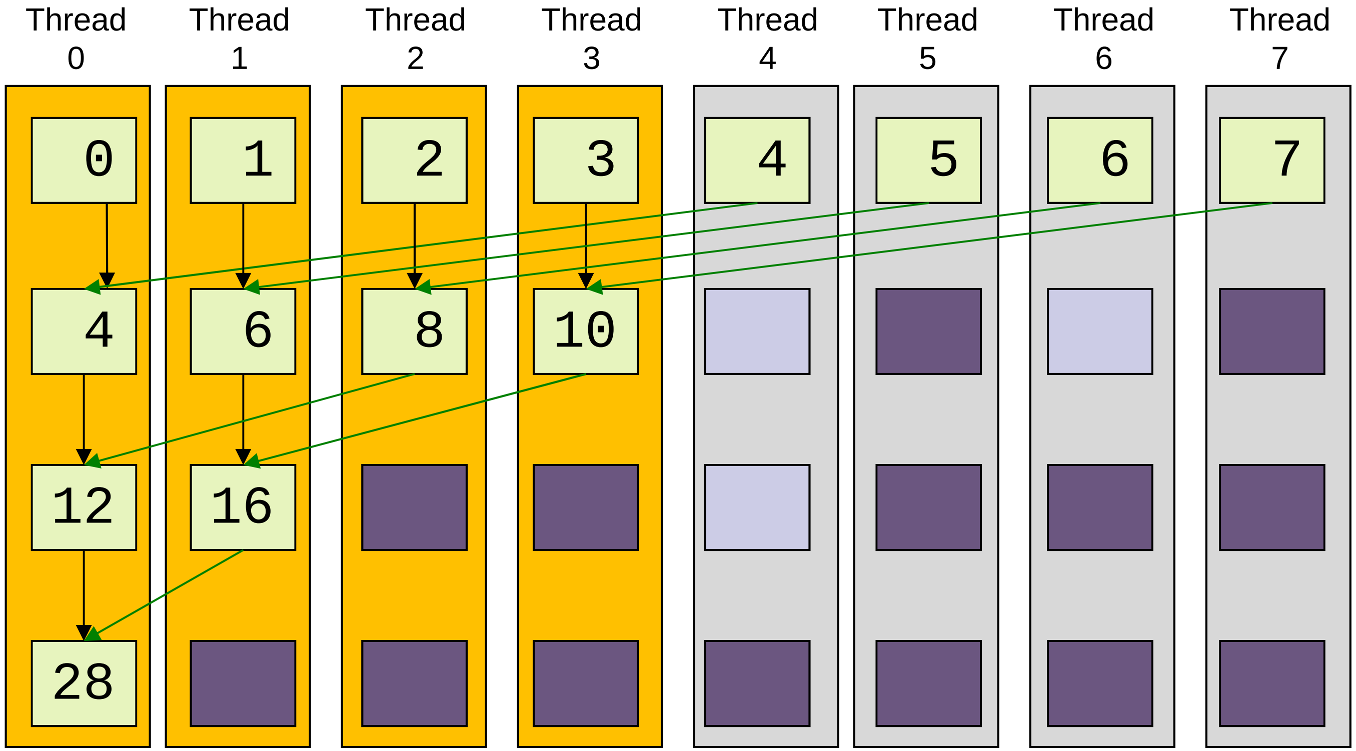
```
__shared__ float partialSum[]
// ... load into shared memory
unsigned int t = threadIdx.x;
for(unsigned int stride = blockDim.x / 2;
     stride > 0;
     stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] +=
      partialSum[t + stride];
}
```

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

Parallel Reduction



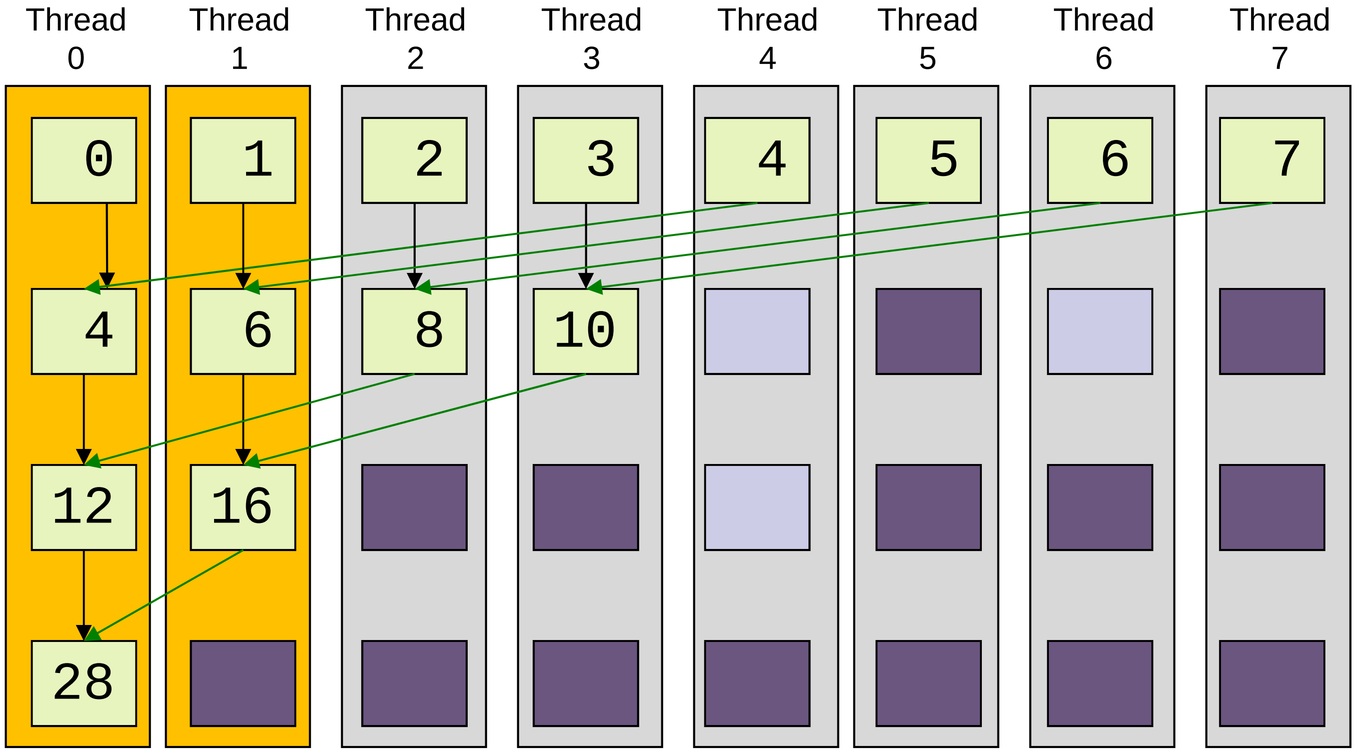
Parallel Reduction



1st pass: threads 4, 5, 6, and 7 don' t do anything

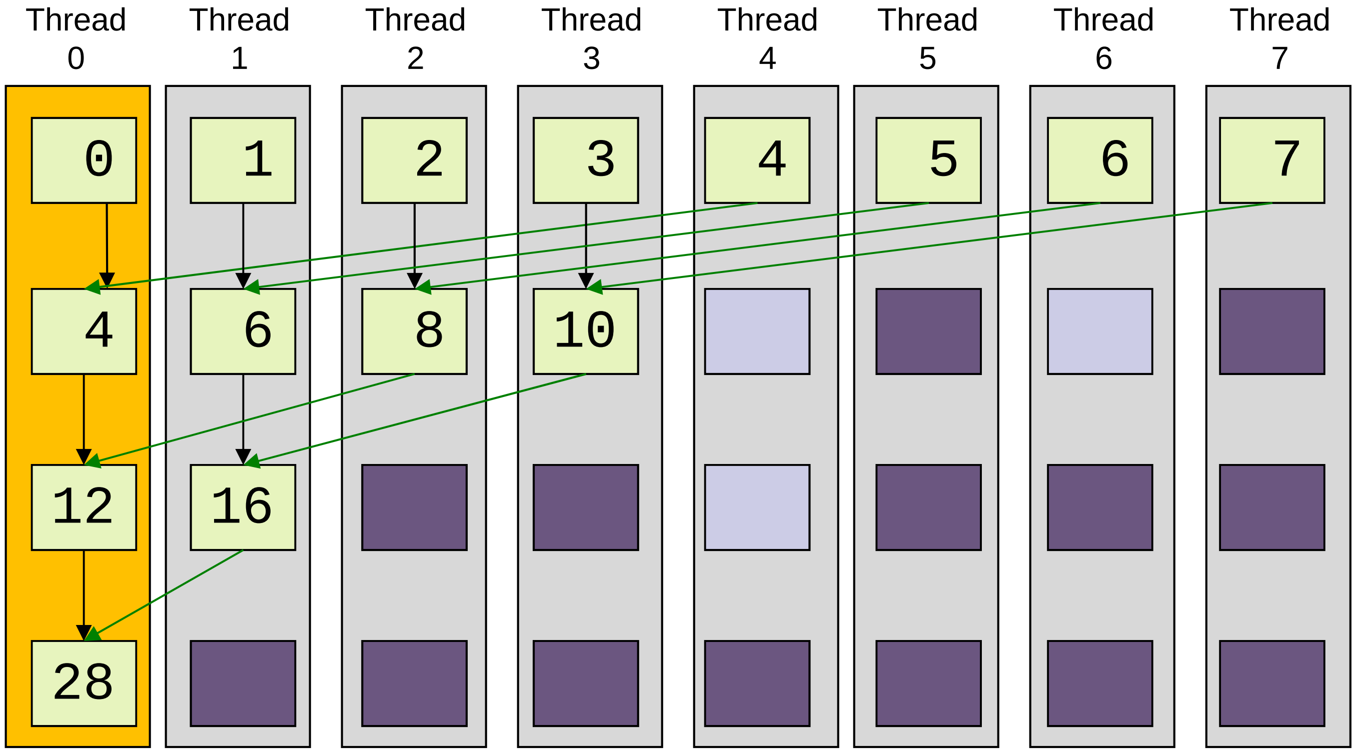
Really only need n/2 threads for n elements

Parallel Reduction



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

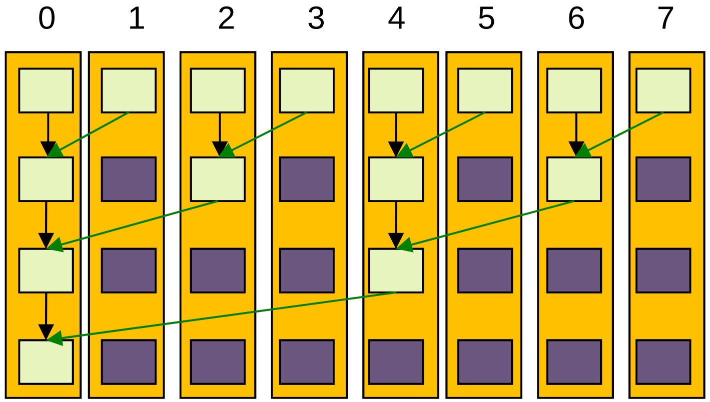
Parallel Reduction



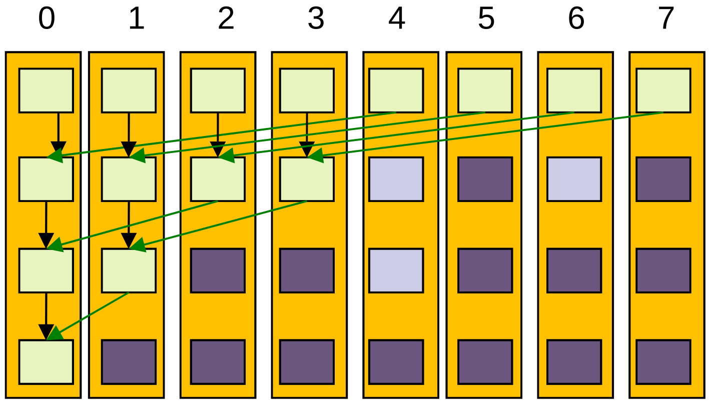
3rd pass: thread 1 also doesn' t do anything

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, …

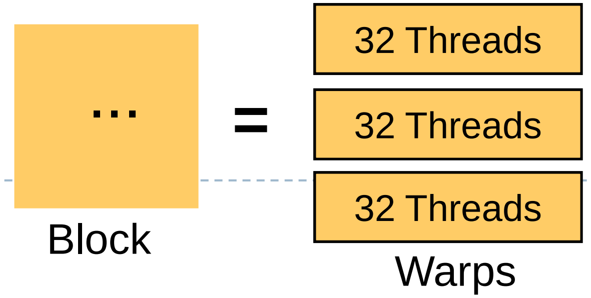
Warp Partitioning

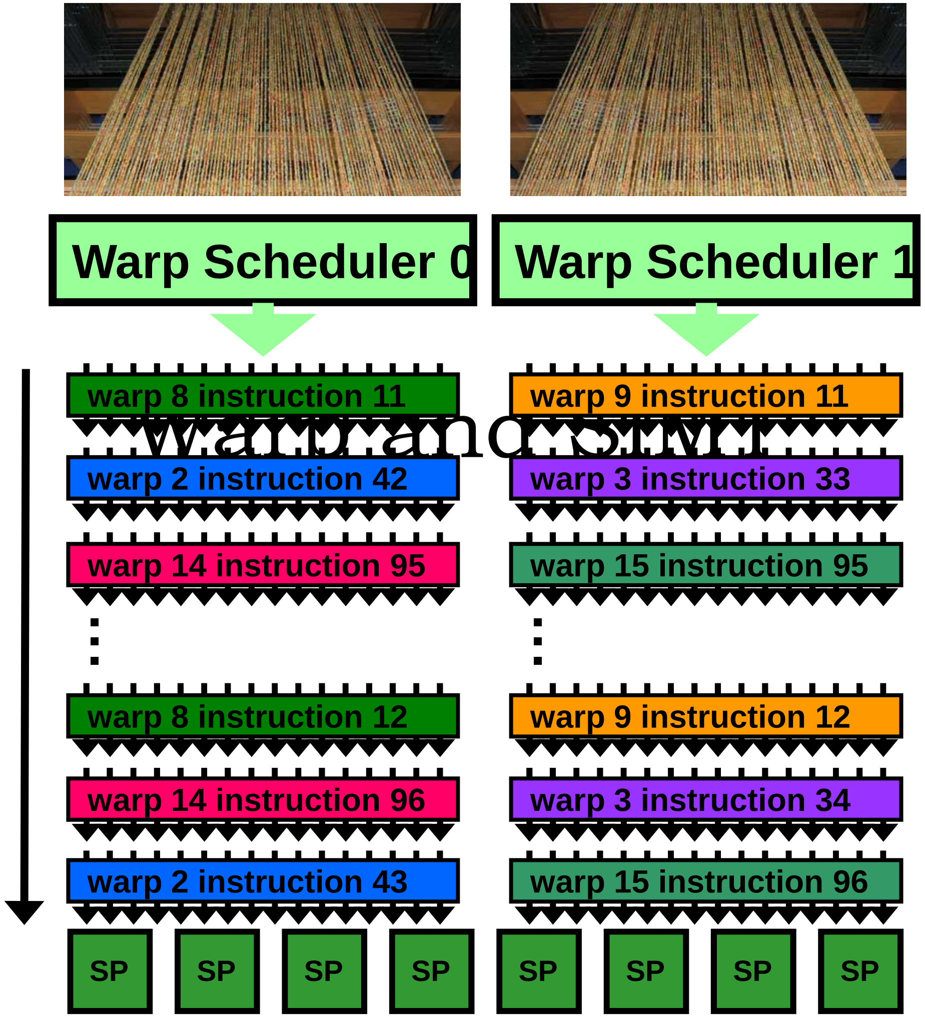
Warp Partitioning: how threads from a block are divided into warps

Knowledge of warp partitioning can be used to:

- Minimize divergent branches
- Retire warps early

Warp and SIMT





- Blocks divide into groups of 32 threads called warps.
- Warps are basic scheduling units
- Warps always perform same instruction (SIMT)
- Each thread **CAN** execute its own code path
- Fermi SM has 2 warp schedulers (Tesla has 1).
- Context switching is free
- A lot of warps can hide memory latency

Warp Partitioning

Partition based on *consecutive increasing* `threadIdx`

Warp Partitioning

1D Block

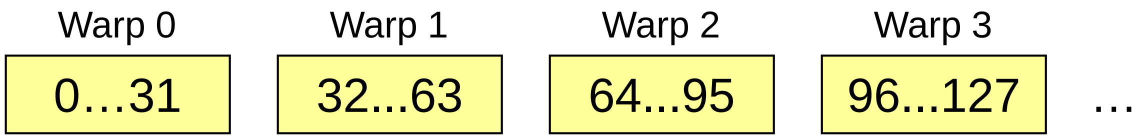
threadIdx.x between 0 and 512 (G80/GT200)

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

0…31

Warp 1

32...63

Warp 2

64...95

Warp 3

96...127

…

Warp Partitioning

2D Block

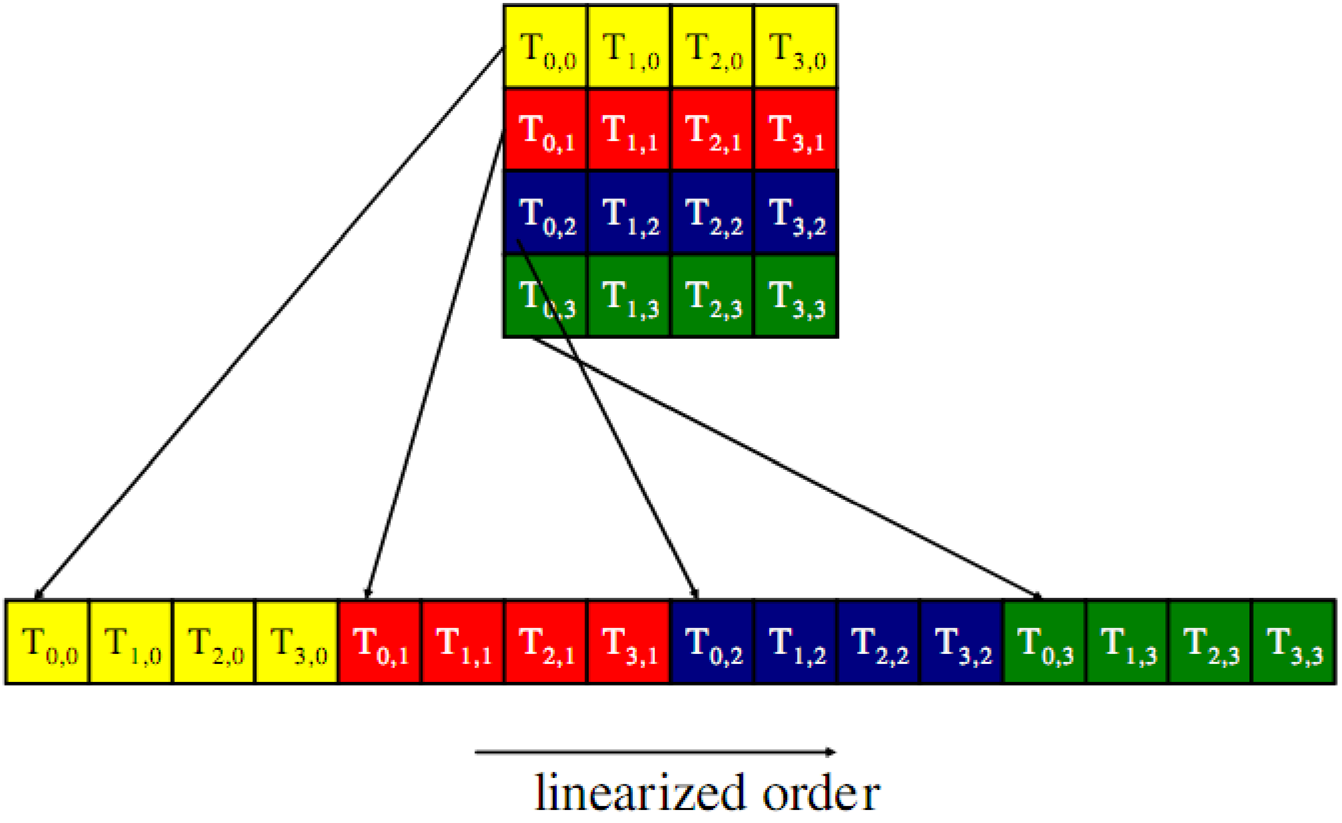
Increasing `threadIdx` means

Increasing `threadIdx.x`

Starting with row `threadIdx.y == 0`

Warp Partitioning

2D Block



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

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

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

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

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

linearized order

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

Warp Partitioning

3D Block

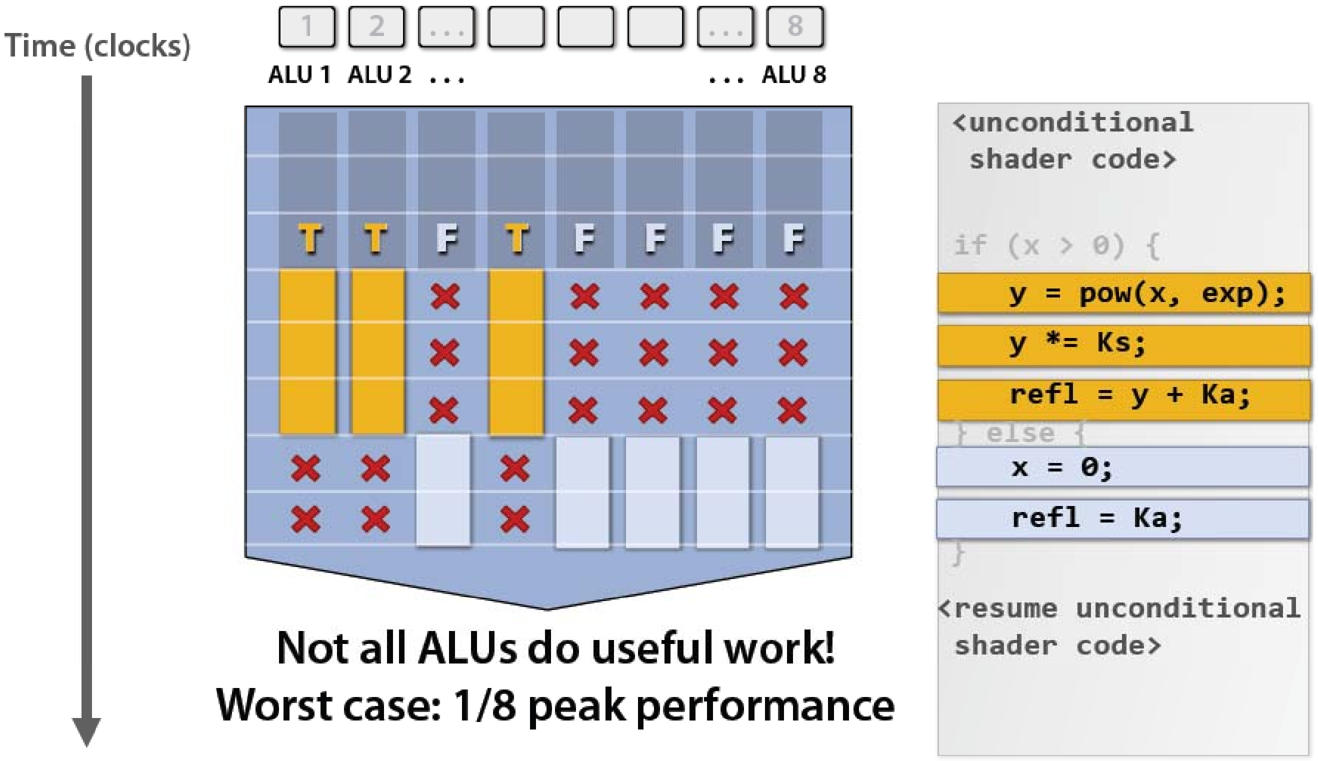
Start with `threadIdx.z == 0`

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>
```

Not all ALUs do useful work!
Worst case: 1/8 peak performance

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];
```

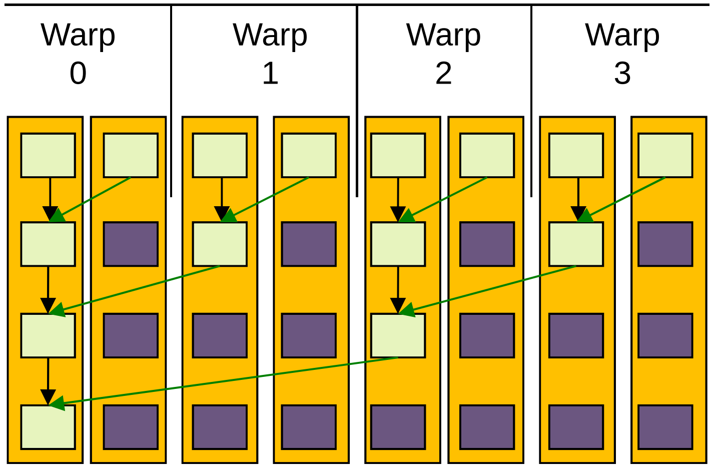
`stride` = 1, 2, 4, …

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

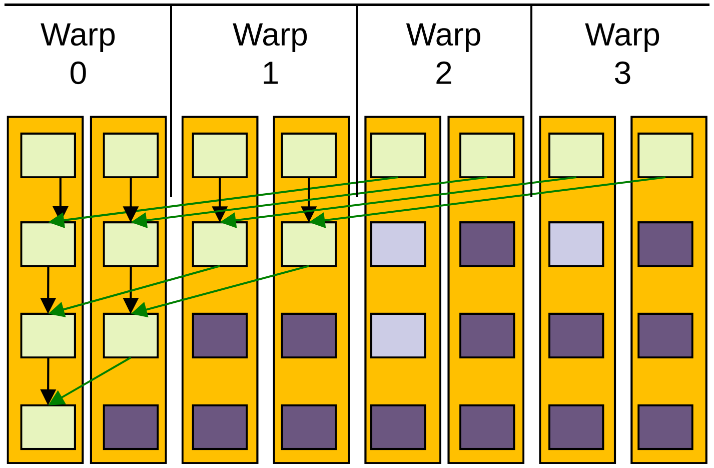
`stride` = 4, 2, 1, …

Warp Partitioning

Pretend `warpSize == 2`



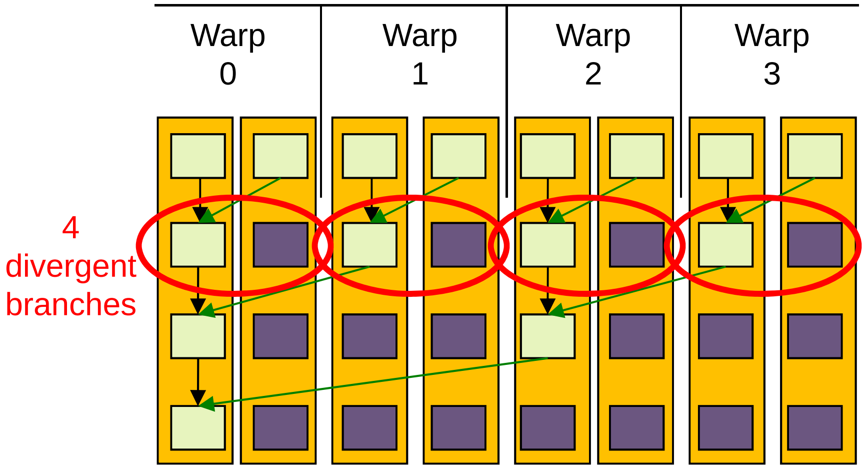
`stride` = 1, 2, 4, …



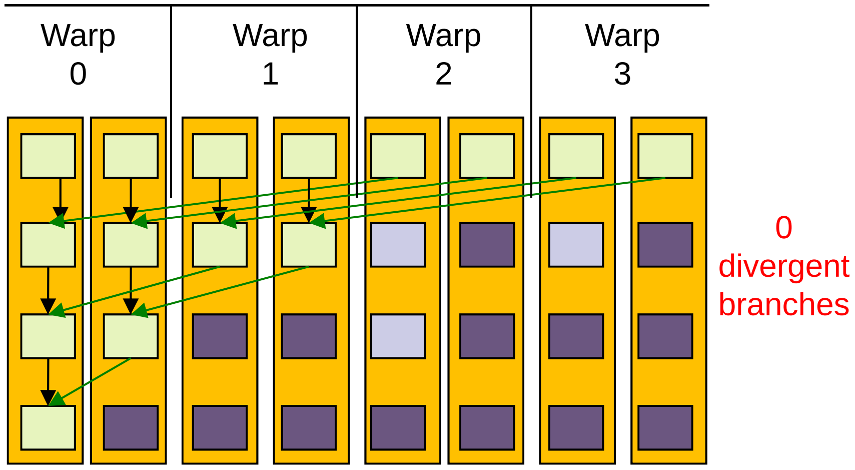
`stride` = 4, 2, 1, …

Warp Partitioning

1st Pass



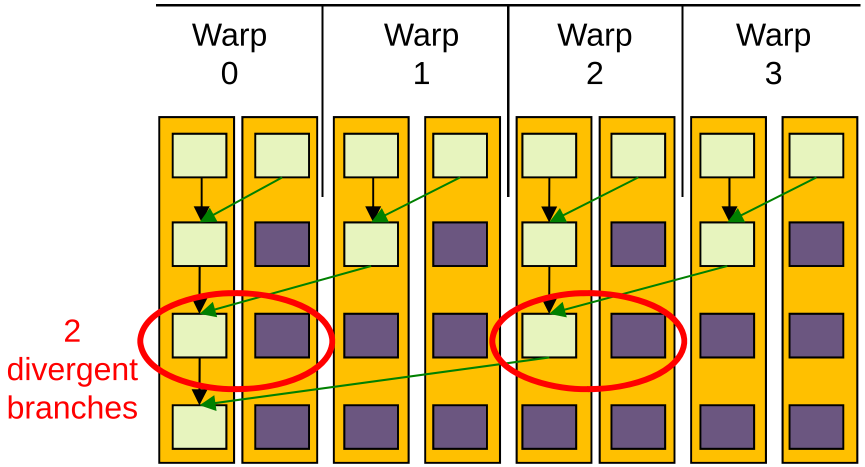
`stride` = 1, 2, 4, …



`stride` = 4, 2, 1, …

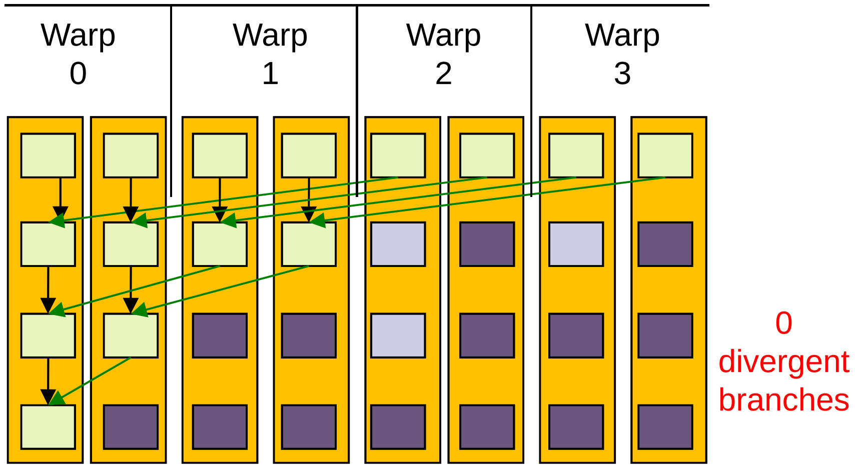
Warp Partitioning

2nd Pass



2
divergent
branches

`stride` = 1, 2, 4, …

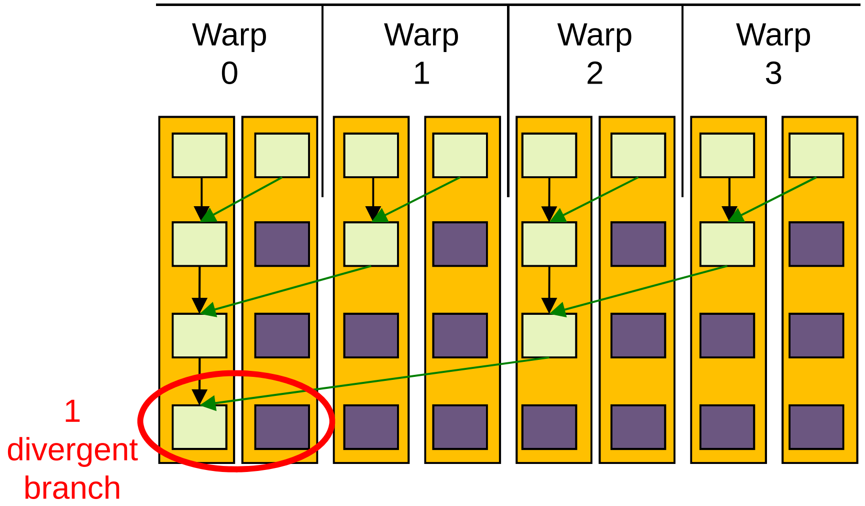


0
divergent
branches

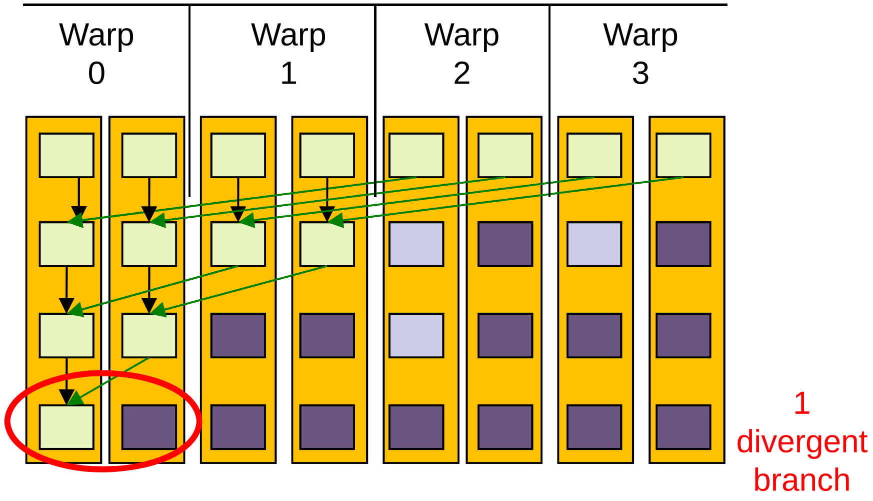
`stride` = 4, 2, 1, …

Warp Partitioning

2^{nd} Pass



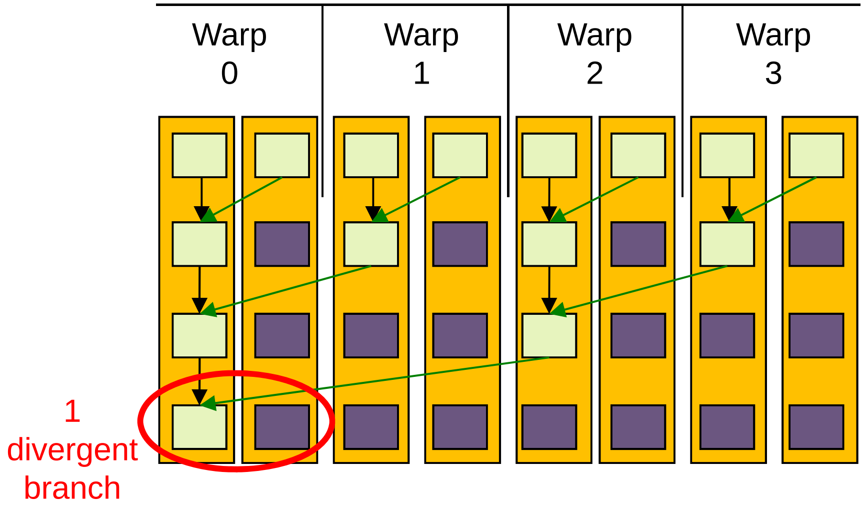
`stride` = 1, 2, 4, …



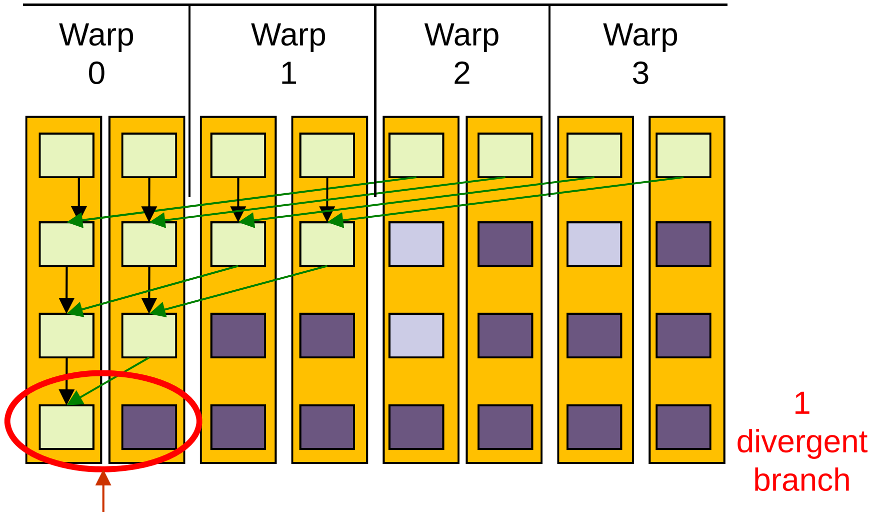
`stride` = 4, 2, 1, …

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];
```

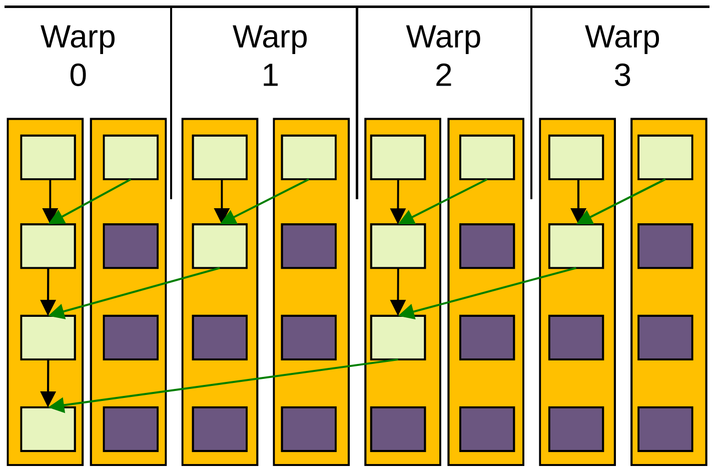
`stride` = 1, 2, 4, …

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

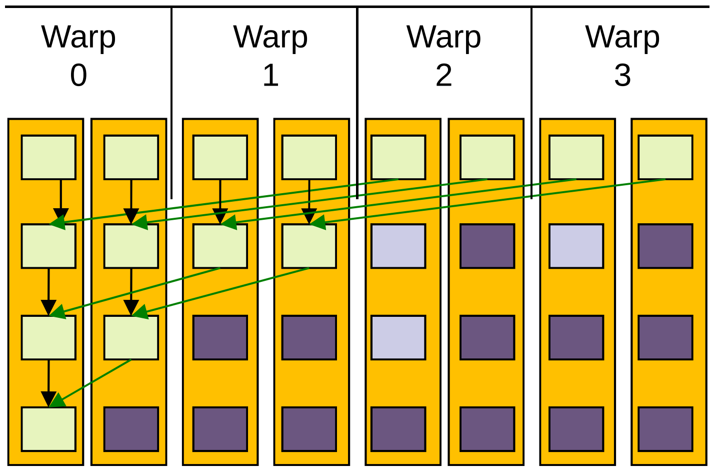
`stride` = 4, 2, 1, …

Warp Partitioning

Parallel Reduction



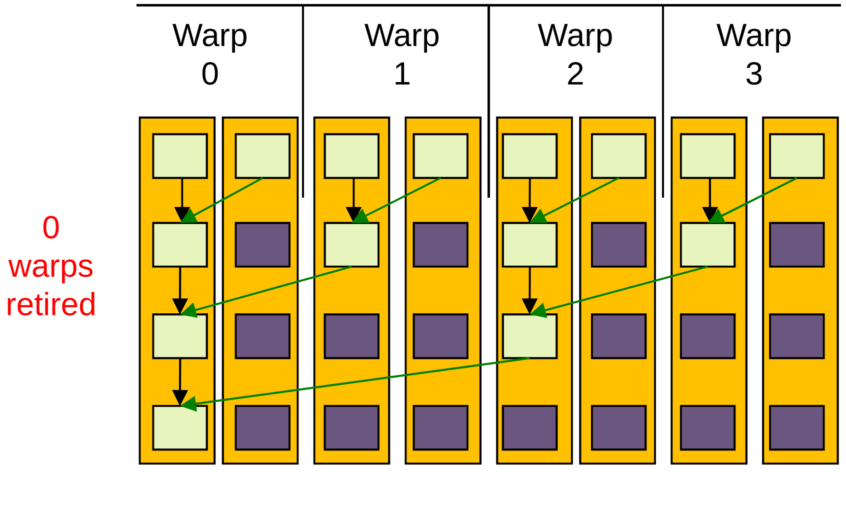
`stride` = 1, 2, 4, …



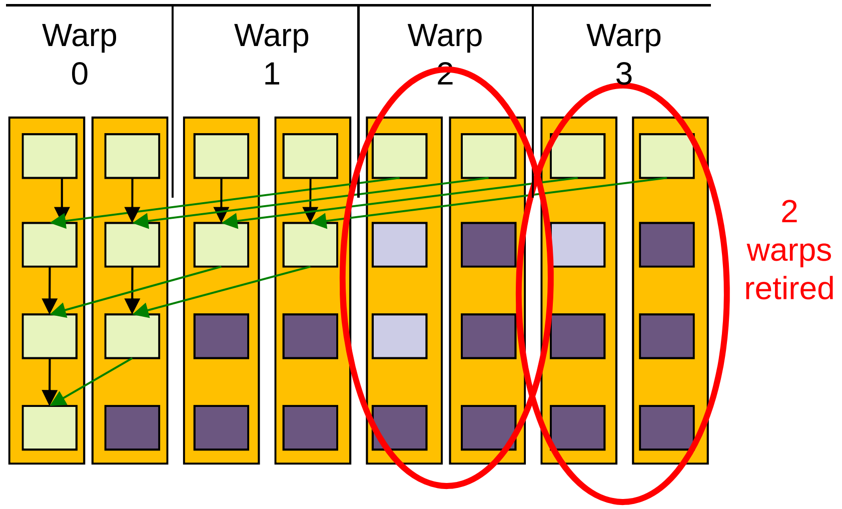
`stride` = 4, 2, 1, …

Warp Partitioning

1st Pass



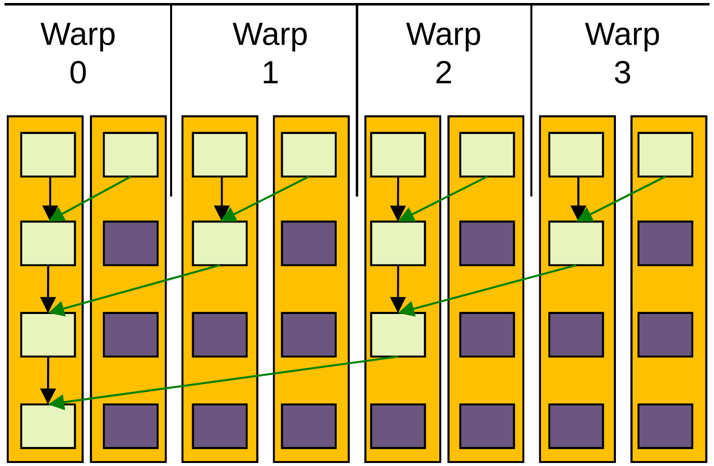
`stride` = 1, 2, 4, …



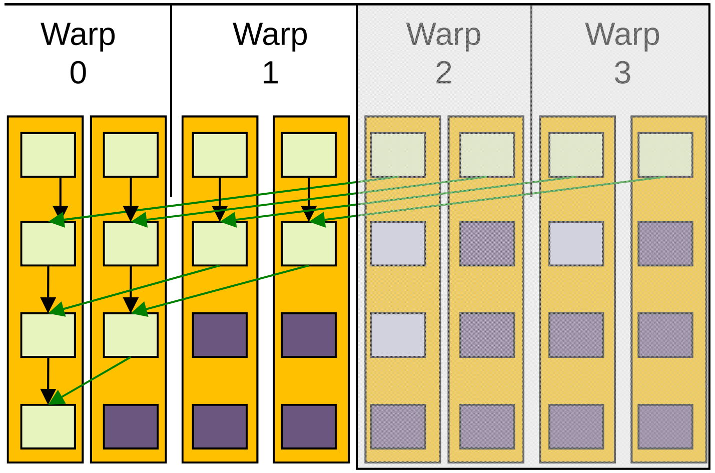
`stride` = 4, 2, 1, …

Warp Partitioning

1st Pass



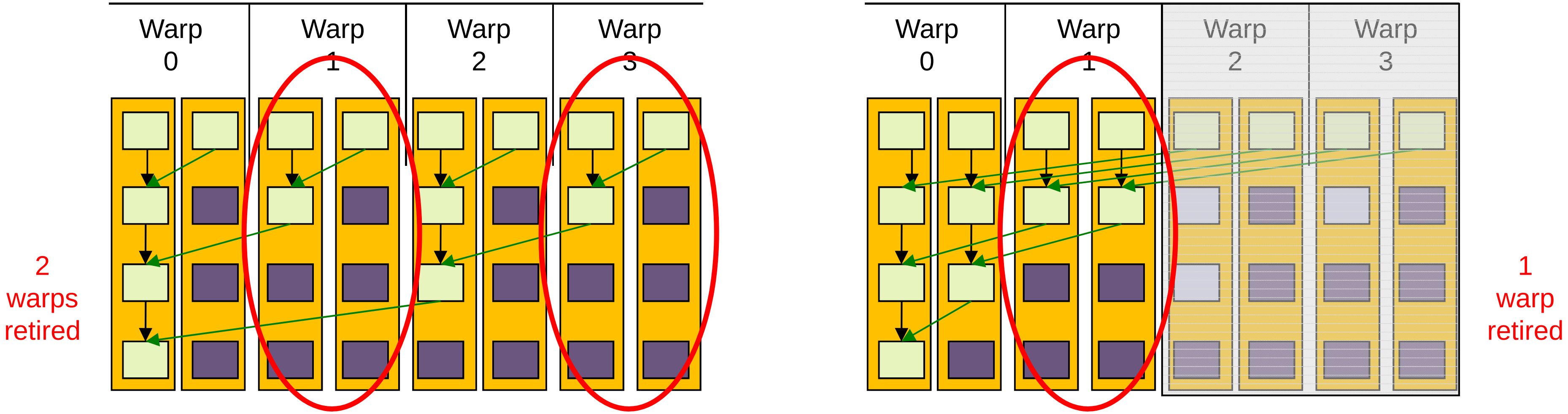
`stride` = 1, 2, 4, …



`stride` = 4, 2, 1, …

Warp Partitioning

2nd Pass

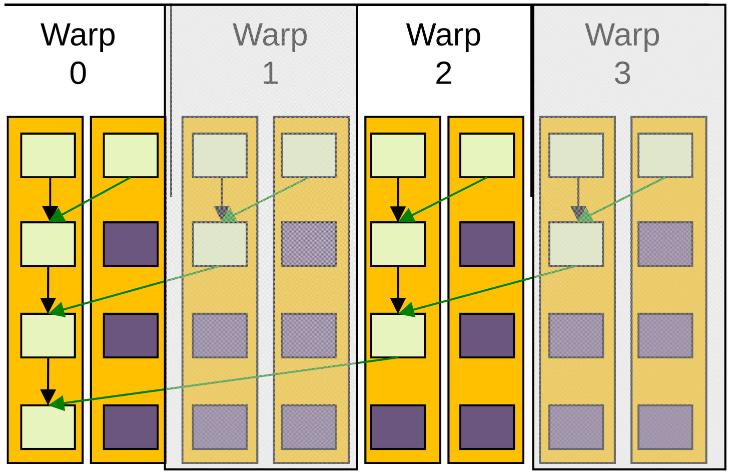


`stride` = 1, 2, 4, …

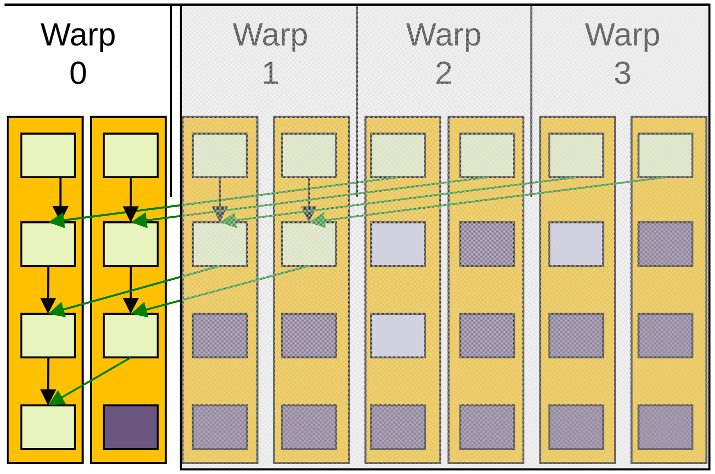
`stride` = 4, 2, 1, …

Warp Partitioning

2nd Pass



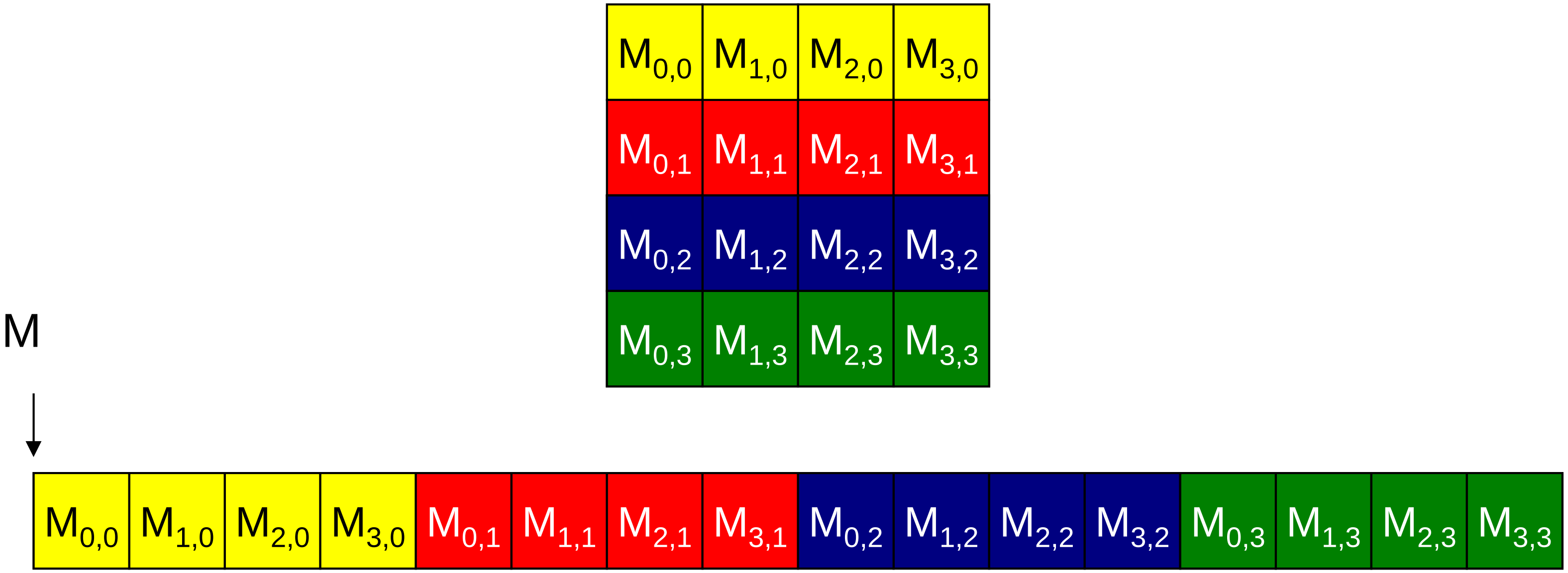
`stride` = 1, 2, 4, …



`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 Optimization

Minimizing CPU-GPU data transfer

- Host<->device data transfer has much lower bandwidth than global memory access.
  - 8 GB/s (PCIe x16 Gen2) vs 156 GB/s & 515 Ginst/s (C2050)
- Minimize transfer
  - Intermediate data can be allocated, operated, de-allocated directly on GPU
  - Sometimes it' s even better to recompute on GPU
  - Move CPU codes to GPU that do not have performance gains if it can reduce data transfer
- Group transfer
  - One large transfer much better than many small ones: 10 microsec latency, 8 GB/s => latency dominated if data size < 80 KB

Overlap memory transfer with computation

Double buffering

Coalescing

- Global memory latency: 400-800 cycles.

The single most important performance consideration!

On Fermi, by default all global memory access are cached in L1.

L1 can be by-passed by passing "-Xptxas –dlcm=cg" to nvcc: cache only in L2

If cached: requests from a warp falling in a L1 cache line, one transaction

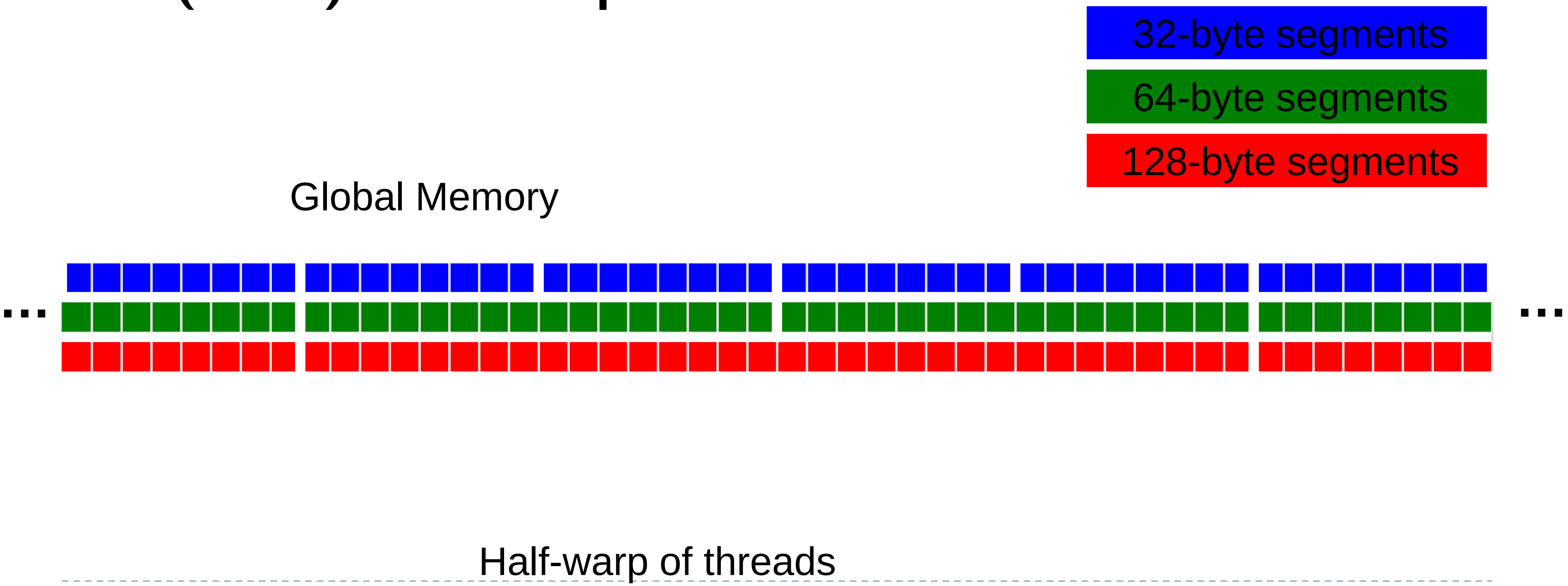
# transaction = # L1 line accessed

If non-cached: same coalescing criterion

But transaction size can be reduced to 32B segment

Coalescing

- **Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met**
- **Depends on compute capability**
  - **1.0 and 1.1 have stricter access requirements**
- **Float (32-bit) data example:**



32-byte segments

64-byte segments

128-byte segments

Global Memory

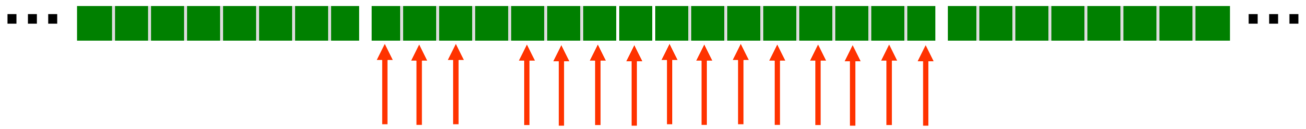
Half-warp of threads

Coalescing

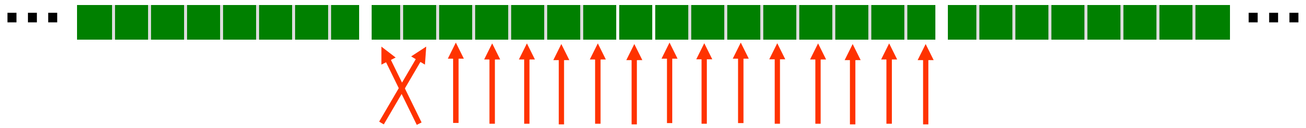
Compute capability 1.0 and 1.1

- **K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate**

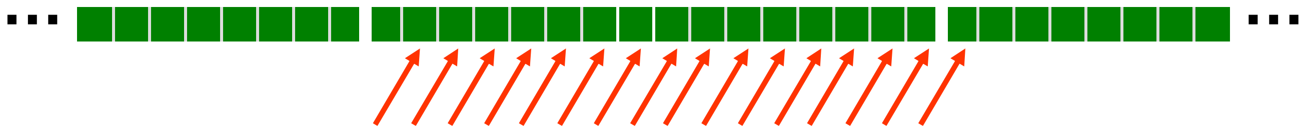
Coalesces – 1 transaction



Out of sequence – 16 transactions



Misaligned – 16 transactions

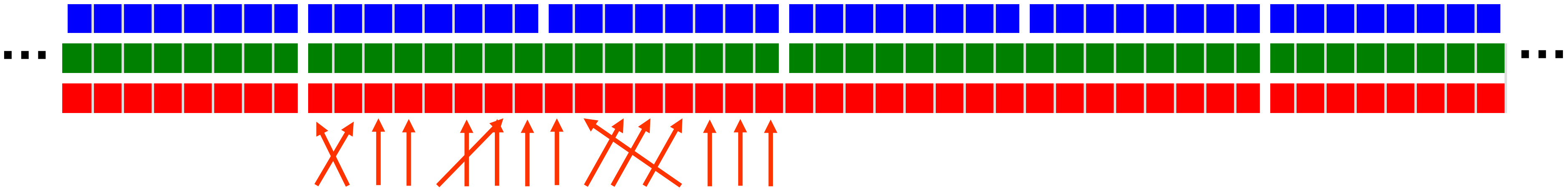


Coalescing

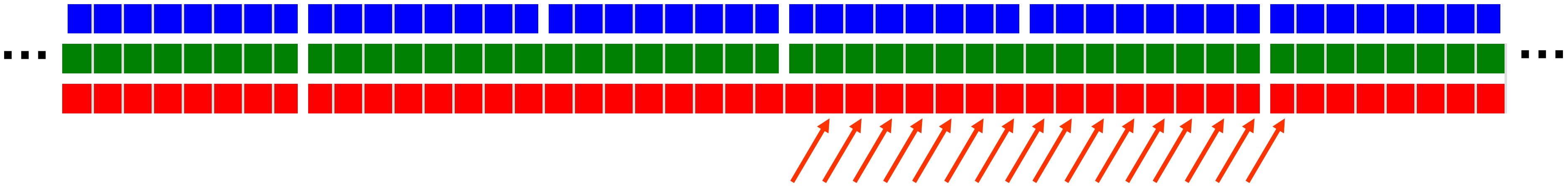
Compute capability 1.2 and higher

- **Issues transactions for segments of 32B, 64B, and 128B**
- **Smaller transactions used to avoid wasted bandwidth**

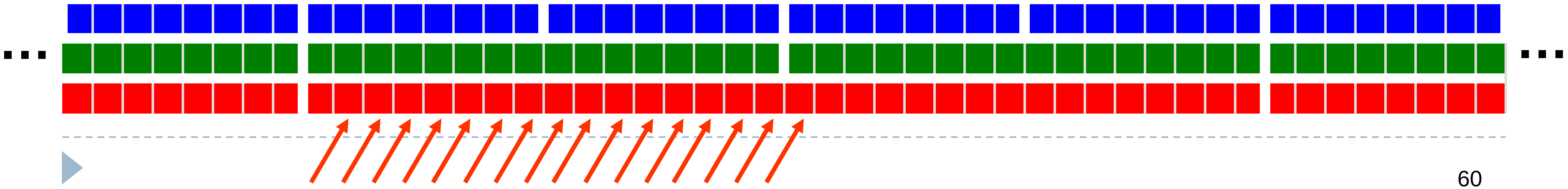
1 transaction - 64B segment



2 transactions - 64B and 32B segments



1 transaction - 128B segment



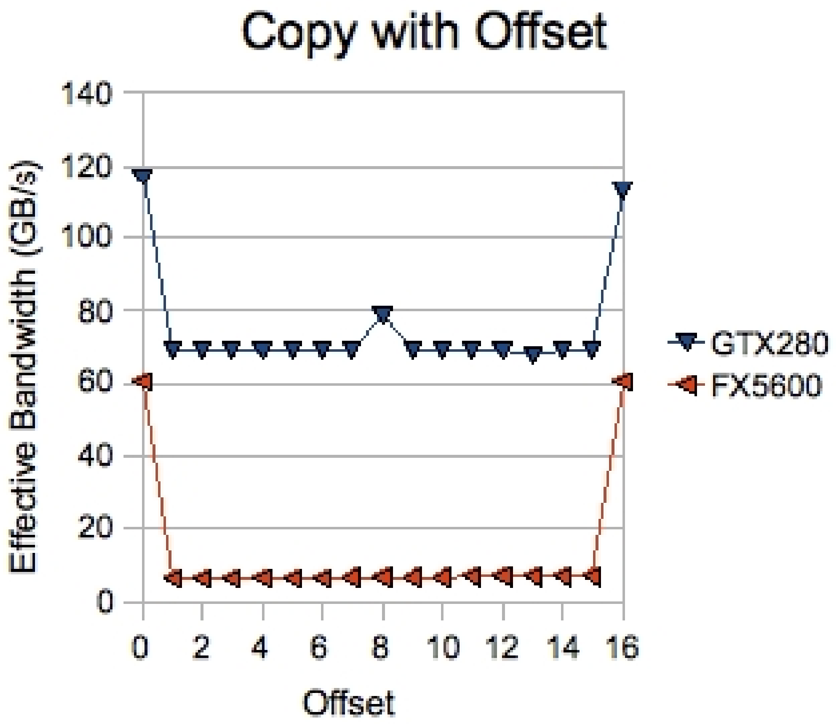
Coalescing Examples

- Effective bandwidth of small kernels that copy data
  - Effects of offset and stride on performance

- Two GPUs
  - GTX 280
    - Compute capability 1.3
    - Peak bandwidth of 141 GB/s
  - FX 5600
    - Compute capability 1.0
    - Peak bandwidth of 77 GB/s

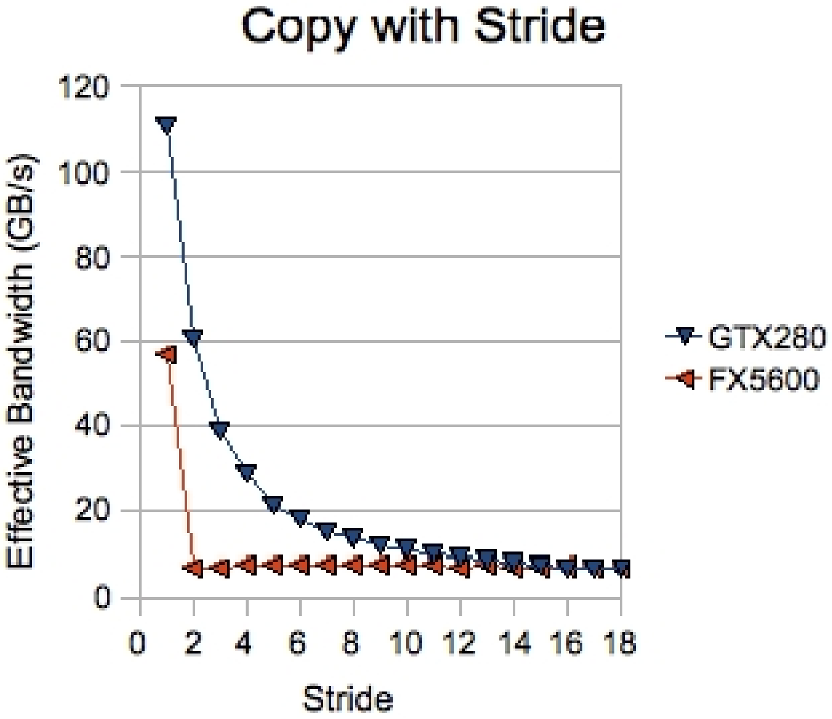
Coalescing Examples

```
__global__ void offsetCopy(float *odata, float *idata,
                           int offset)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
  odata[xid] = idata[xid];
}
```



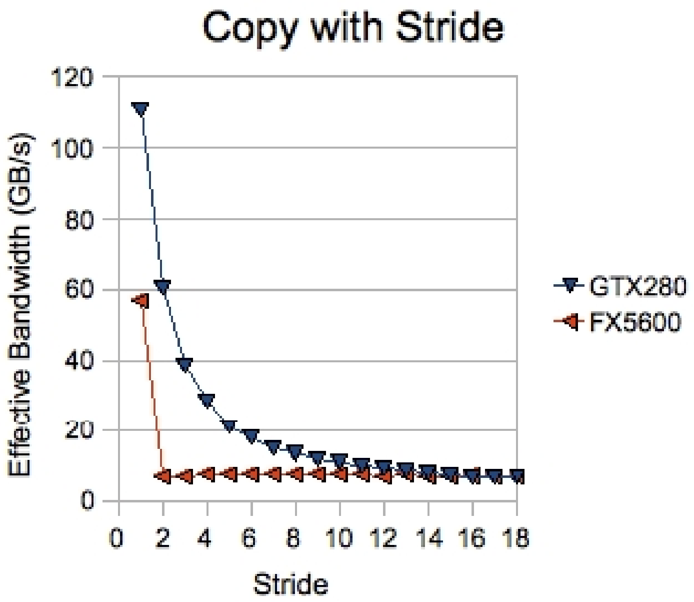
Coalescing Examples

```
__global__ void strideCopy(float *odata, float *idata,
                           int stride)
{
  int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
  odata[xid] = idata[xid];
}
```



Coalescing Examples

- Strided memory access is inherent in many multidimensional problems
  - Stride is generally large (>> 18)



However …

- Strided access to *global memory* can be avoided using ***shared memory***

Shared Memory

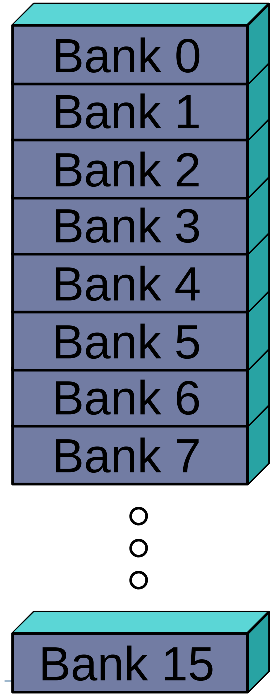
- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - **Stage loads and stores in shared memory to re-order non-coalesceable addressing**

Shared Memory Architecture

- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks

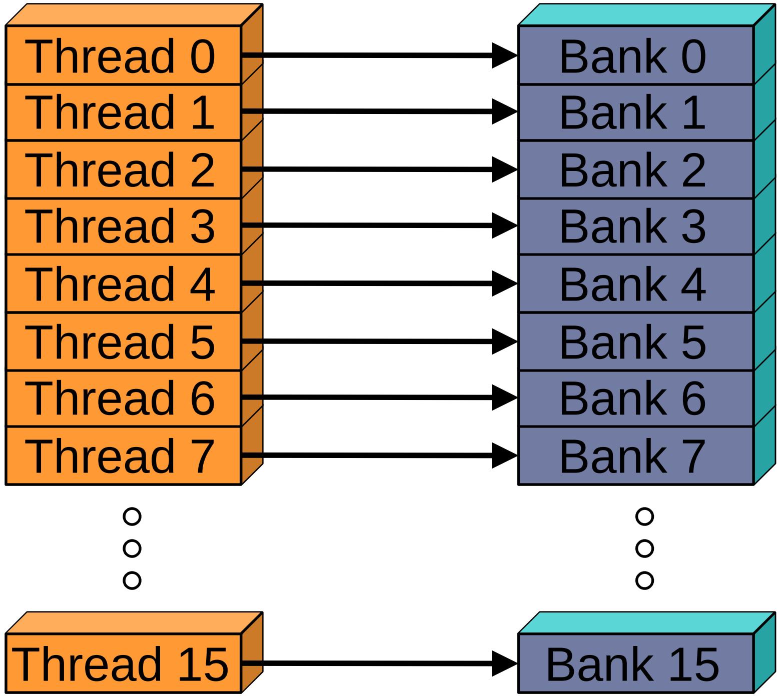
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

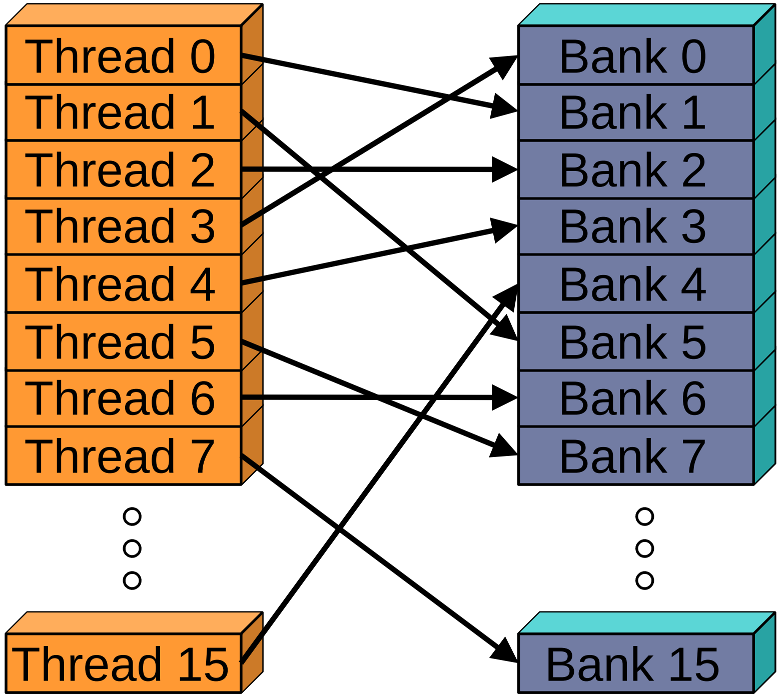


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

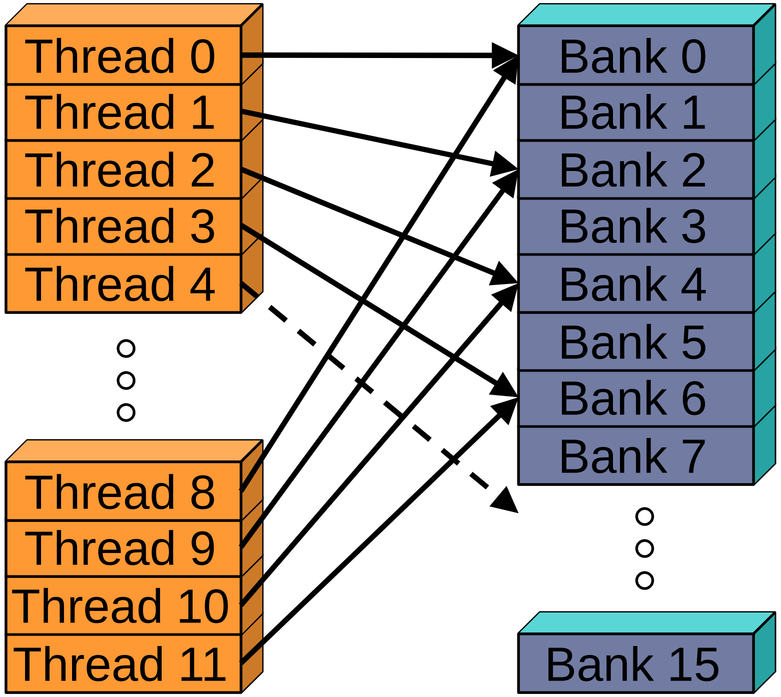


- No Bank Conflicts
  - Random 1:1 Permutation

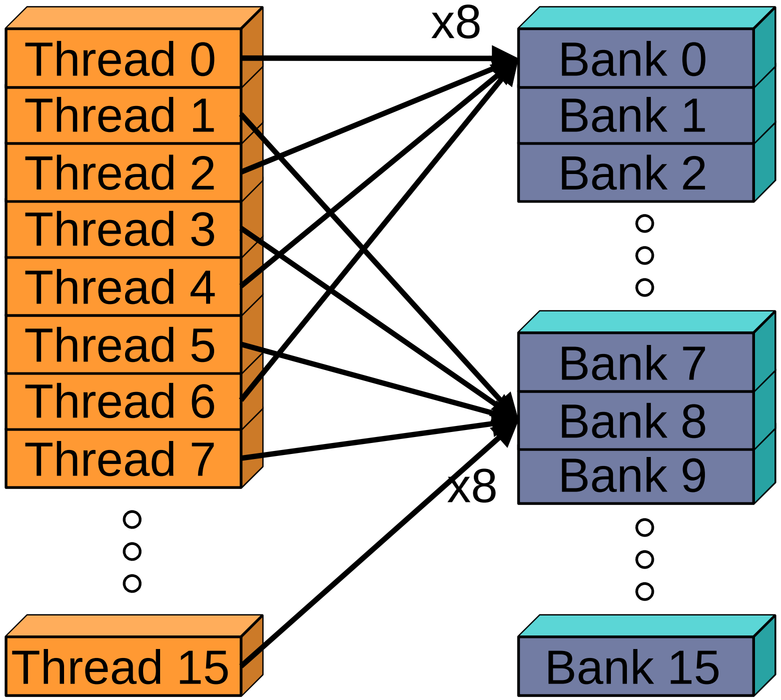


Bank Addressing Examples

- **2-way Bank Conflicts**
  - **Linear addressing stride == 2**



- **8-way Bank Conflicts**
  - **Linear addressing stride == 8**

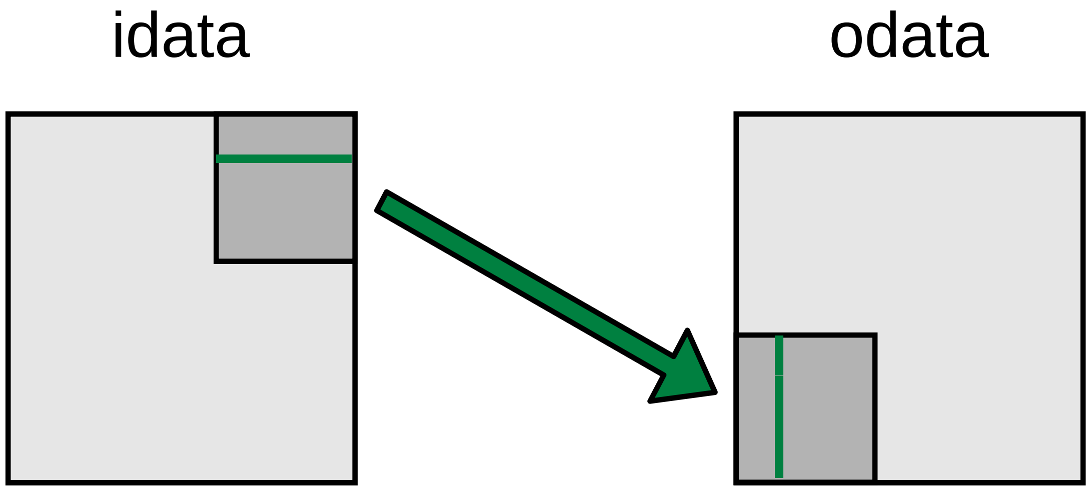


Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts
- warp_serialize profiler signal reflects conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

Shared Memory Example: Transpose

- **Each thread block works on a tile of the matrix**
- **Naïve implementation exhibits strided access to global memory**

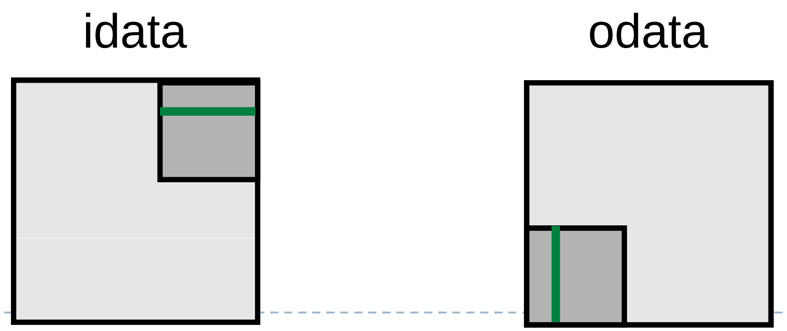


Elements transposed by a half-warp of threads

Naïve Transpose

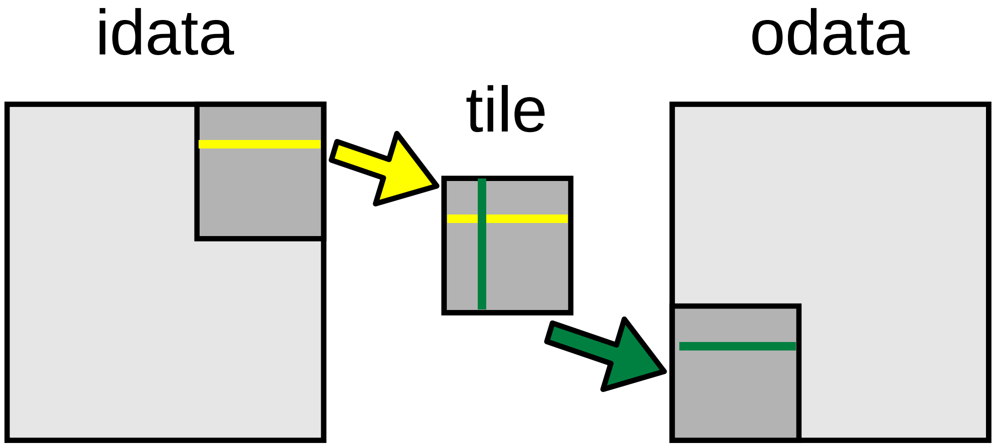
- **Loads are coalesced, stores are not (strided by height)**

```
__global__ void transposeNaive(float *odata, float *idata,
                               int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;

  odata[index_out] = idata[index_in];
}
```



Coalescing through shared memory

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored by other threads



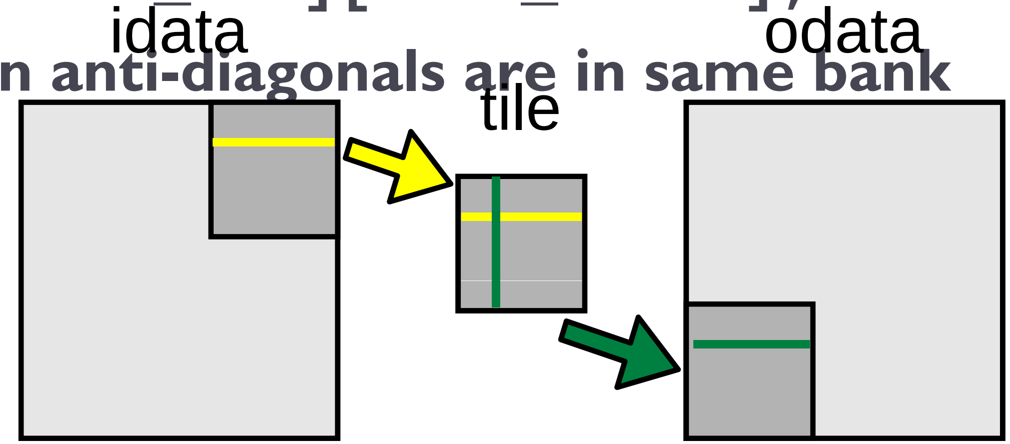
Elements transposed by a half-warp of threads

Coalescing through shared memory

```
__global__ void transposeCoalesced(float *odata, float *idata,
                                   int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  tile[threadIdx.y][threadIdx.x] = idata[index_in];

  __syncthreads();

  odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

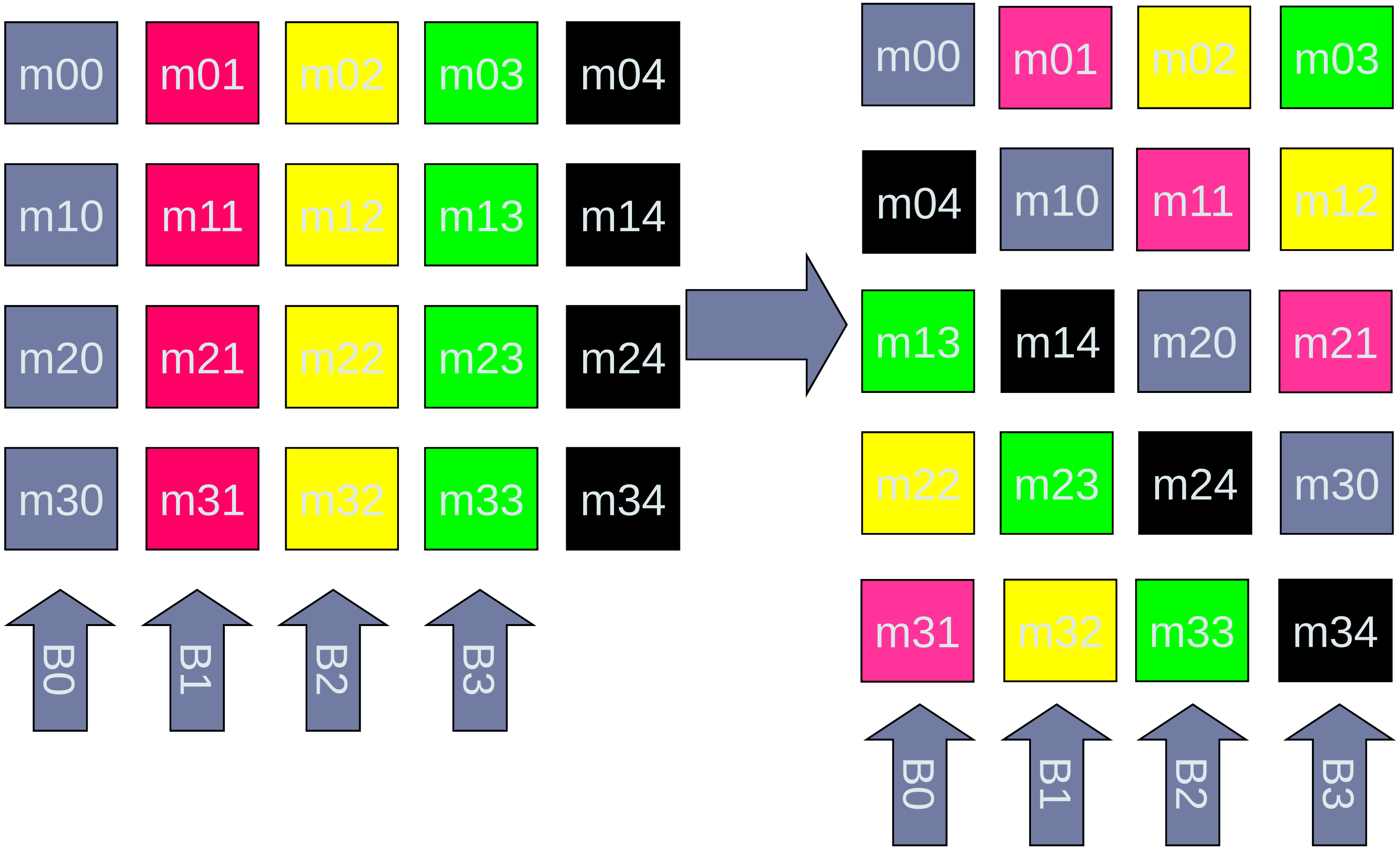
Bank Conflicts in Transpose

- 16x16 shared memory tile of floats
  - Data in columns are in the same bank
  - 16-way bank conflict reading columns in tile
- Solution - pad shared memory array
  - `__shared__ float tile[TILE_DIM][TILE_DIM+1];`
  - Data in anti-diagonals are in same bank



Elements transposed by a half-warp of threads

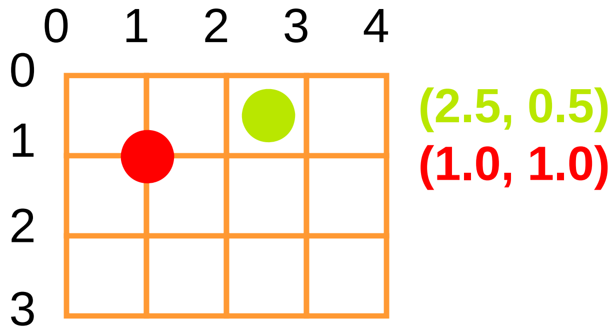
Padding Shared Memory



Textures in CUDA

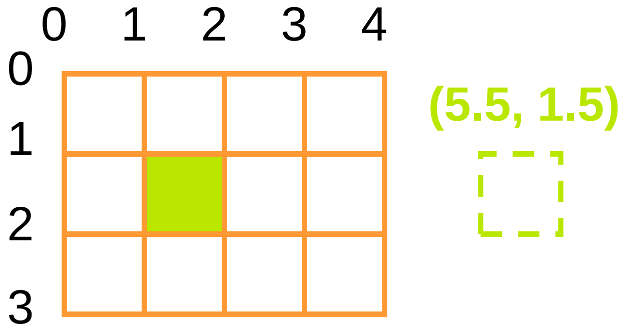
- Texture is an object for *reading* data
- Benefits:
  - Data is cached
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear interpolation
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable in 1D, 2D, or 3D
    - Using integer or normalized coordinates

Texture Addressing



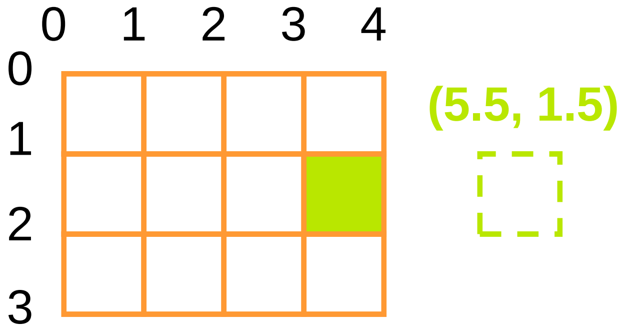
Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



Clamp

- Out-of-bounds coordinate is replaced with the closest boundary



CUDA Texture Types

- Bound to linear memory
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes
- Bound to CUDA arrays
  - Block linear CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- Bound to pitch linear (CUDA 2.2)
  - Global memory address is bound to a texture
  - 2D
  - Float/integer addressing, filtering, and clamp/repeat addressing modes similar to CUDA arrays

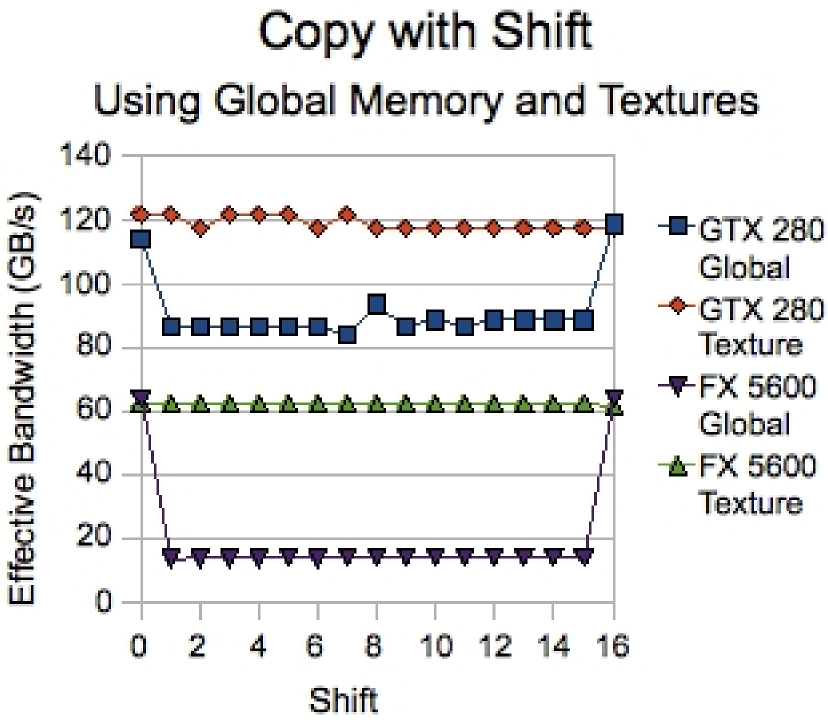
CUDA Texturing Steps

- **Host (CPU) code:**
  - **Allocate/obtain memory (global linear/pitch linear, or CUDA array)**
  - **Create a texture reference object**
    - **Currently must be at file-scope**
  - **Bind the texture reference to memory/array**
  - **When done:**
    - **Unbind the texture reference, free resources**

- **Device (kernel) code:**
  - **Fetch using texture reference**
  - **Linear memory textures: tex1Dfetch()**
  - **Array textures: tex1D() or tex2D() or tex3D()**
  - **Pitch linear textures: tex2D()**

Texture Example

```
__global__ void
shiftCopy(float *odata,
          float *idata,
          int shift)
{
  int xid = blockIdx.x * blockDim.x
          + threadIdx.x;
  odata[xid] = idata[xid+shift];
}


texture <float> texRef;

__global__ void
textureShiftCopy(float *odata,
                 float *idata,
                 int shift)
{
  int xid = blockIdx.x * blockDim.x
          + threadIdx.x;
  odata[xid] = tex1Dfetch(texRef, xid+shift);
}
```

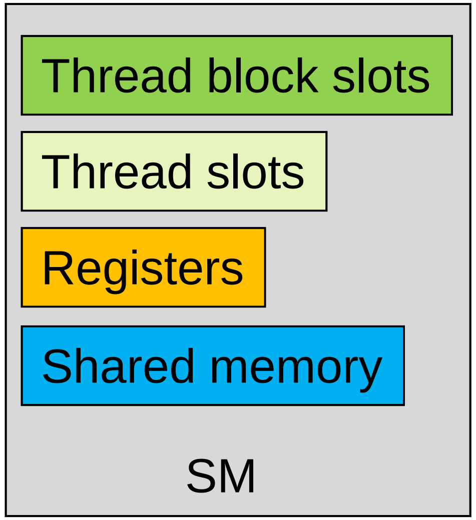


Summary

- GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
  - Use parallelism efficiently
  - Coalesce memory accesses if possible
  - Take advantage of shared memory
  - Explore other memory spaces
    - Texture
    - Constant
  - Reduce bank conflicts

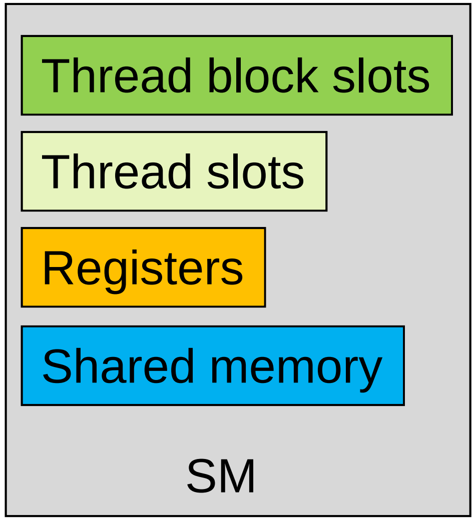
SM Resource Partitioning

Recall a SM dynamically partitions resources:



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

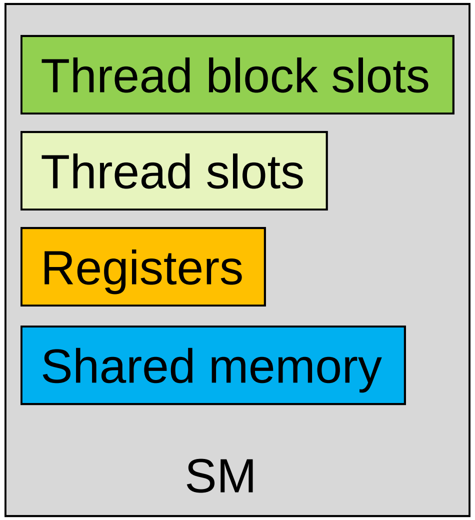
SM Resource Partitioning

We can have

8 blocks of 96 threads

4 blocks of 192 threads

But not 8 blocks of 192 threads



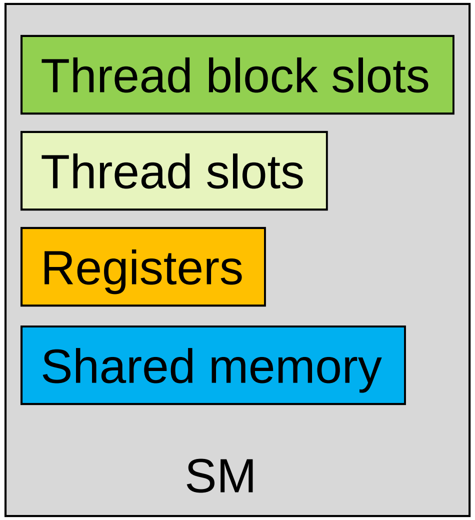
	G80 Limits
Thread block slots	8
Thread slots	768
Registers	8K registers / 32K memory
Shared memory	16K

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

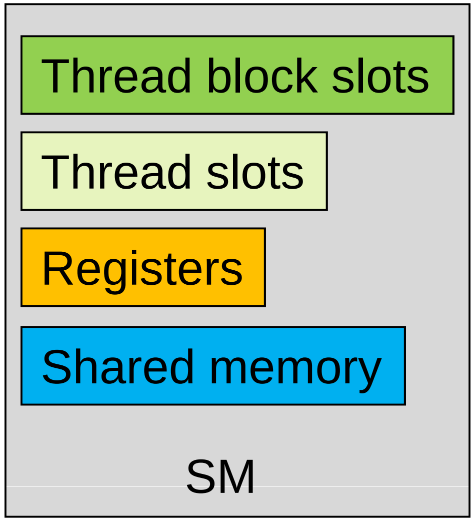
SM Resource Partitioning

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?



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

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

Kernel Launch Configuration

Grid Size Heuristics

- # of blocks > # of SM
  - Each SM has at least one work-group to execute
- # of blocks / # of SM > 2
  - Multi blocks can run concurrently on a SM
  - Work on another block if one block is waiting on barrier
- # of blocks / # of SM > 100 to scale well to future device

Block Size Heuristics

- Block size should be a multiple of 32 (warp size)
- Want as many warps running as possible to hide latencies
- Minimum: 64. I generally use 128 or 256. But use whatever is best for your app.
- Depends on the problem, do experiments!

Latency Hiding

Key to understanding:

- Instructions are issued in order
- A thread blocks when one of the operands isn' t ready:
- Latency is hidden by switching threads

Conclusion:

- Need enough threads to hide latency

Occupancy

- Occupancy: ratio of active warps per SM to the maximum number of allowed warps

Maximum number: 32 in Tesla, 48 in Fermi

Dynamical Partitioning of SM Resources

Shared memory is partitioned among blocks

Registers are partitioned among threads: <= 63

Thread block slots: <= 8

Thread slots: <= 1536

Any of those can be the limiting factor on how many threads can be launched at the same time on a SM

Latency Hiding Occupancy Calculation

- Assume global memory takes 400 cycles, we need 400/2 = 200 arithmetic instructions to hide the latency.
- For example, assume the code has 8 independent arithmetic instructions for every one global memory access. Thus 200/8~26 warps would be enough (54% occupancy).
- Note beyond 54%, in this example higher occupancy won't lead to performance increase.

Register Dependency Latency Hiding

- If an instruction uses a result stored in a register written by an instruction before it, this is ~ 24 cycles latency

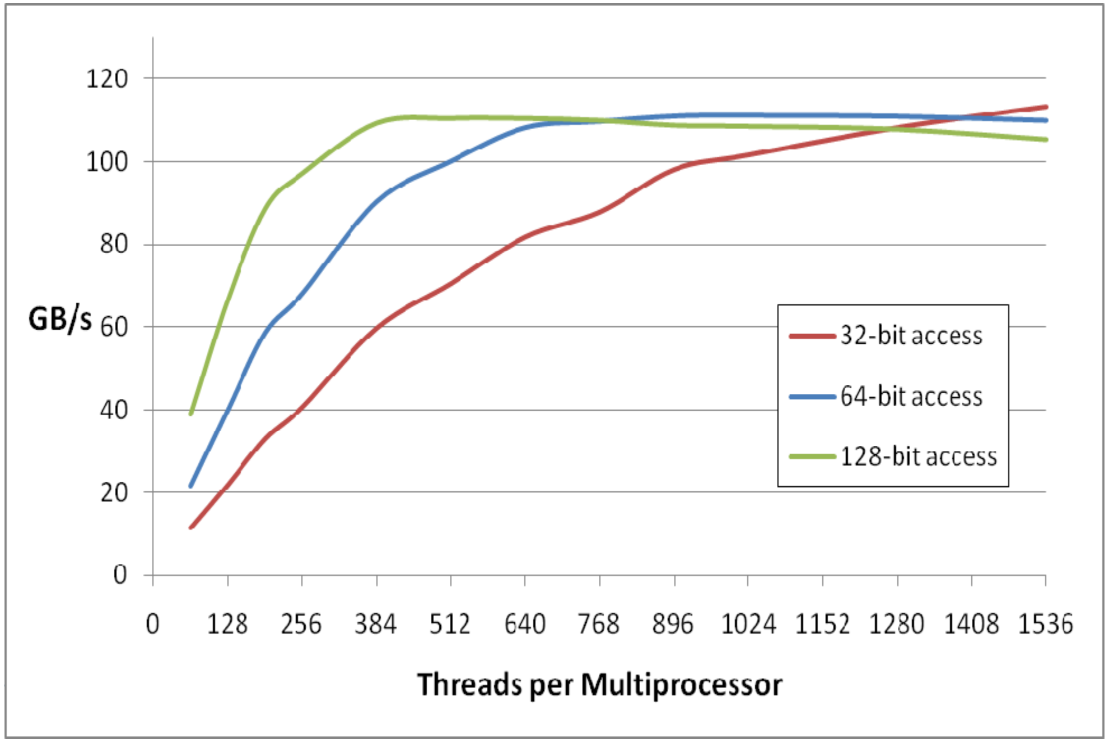
- So, we need 24/2=13 warps to hide register dependency latency. This corresponds to 27% occupancy

Concurrent Accesses and Performance

Increment a 64M element array

- Two accesses per thread (load then store, but they are dependent)
  - Thus, each warp (32 threads) has one outstanding transaction at a time

Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s



Several independent smaller accesses have the same effect as one larger one.

For example:

Four 32-bit ~= one 128-bit

Occupancy Optimizations

- Increase occupancy to achieve latency hiding

If adding a single instruction leads to significant perf drop, occupancy is the primary suspect

--ptxas-options=-v: output resource usage info

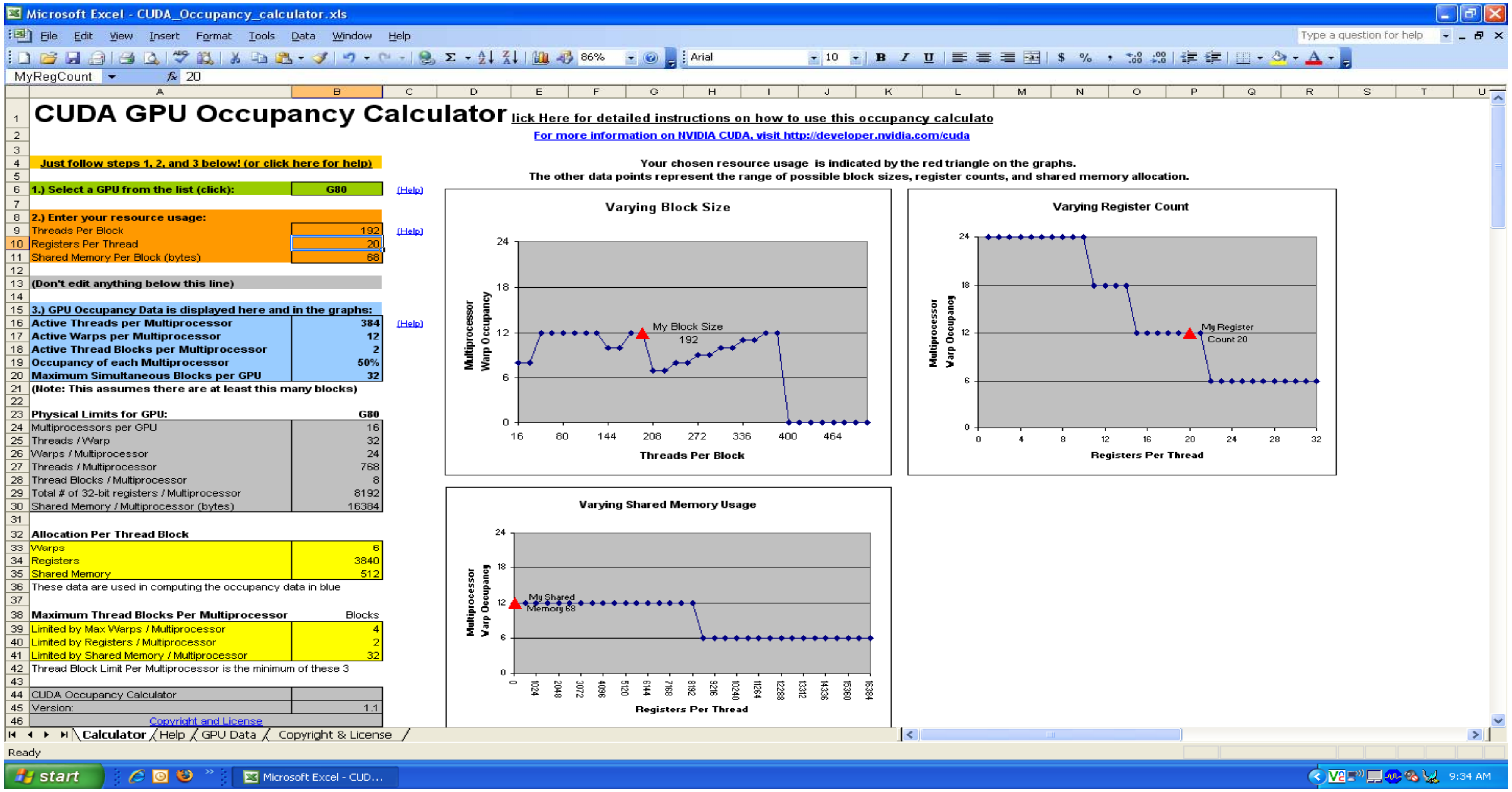
Compiler option –maxrregcount=n: per file

__launch_bounds__: per kernel

Use of template to reduce register usage

Dynamical allocating shared memory

After some point (generally 50%), further increase in occupancy won’t lead to performance increase: got enough warps for latency hiding



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

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.

Just follow steps 1, 2, and 3 below! (or click here for help)		
1.) Select a GPU from the list (click):	G80	(Help)
2.) Enter your resource usage:		
Threads Per Block	192	(Help)
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	(Help)
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		

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 Throughput Optimizations

Instruction Optimization

If you find out the code is instruction bound

- Compute-intensive algorithm can easily become memory-bound if not careful enough
- Typically, worry about instruction optimization after memory and execution configuration optimizations

Fermi Arithmetic Instruction Throughputs

Int & fp32: 2 cycles

fp64: 2 cycles

Fp32 transendental: 8 cycles

Int divide and modulo are expensive

- Divide by 2^n, use "\>> n"
- Modulo 2^n, use "& (2^n – 1)"

Avoid automatic conversion of double to float

- Adding "f" to floating literals (e.g. 1.0f) because the default is double

Fermi default: -ftz=false, -prec-div=true, -prec-sqrt=true for IEEE compliance

Runtime Math Library and Intrinsics

Two types of runtime math library functions

- func():
  - Slower but higher accuracy (5 ulp or less)
  - Examples: sin(x), exp(x), pow(x, y)
- __func():
  - Fast but lower accuracy (see prog. guide for full details)
  - Examples: __sin(x), __exp(x), __pow(x, y)

A number of additional intrinsics:

- __sincos(), __rcp(), ...
- Full list in Appendix C.2 of the CUDA Programming Guide

-use-fast-math: forces every func() to __func ()

Control Flow

Instructions are issued per 32 threads (warp)

Divergent branches:

- Threads within a single warp take different paths
    - if-else, ...
- Different execution paths within a warp are serialized

Different warps can execute different code with no impact on performance

Avoid diverging within a warp

- Example with divergence:

  ```
  if (threadIdx.x > 2) {...} else {...}
  ```

    - Branch granularity < warp size
- Example without divergence:

  ```
  if (threadIdx.x / WARP_SIZE > 2) {...} else {...}
  ```

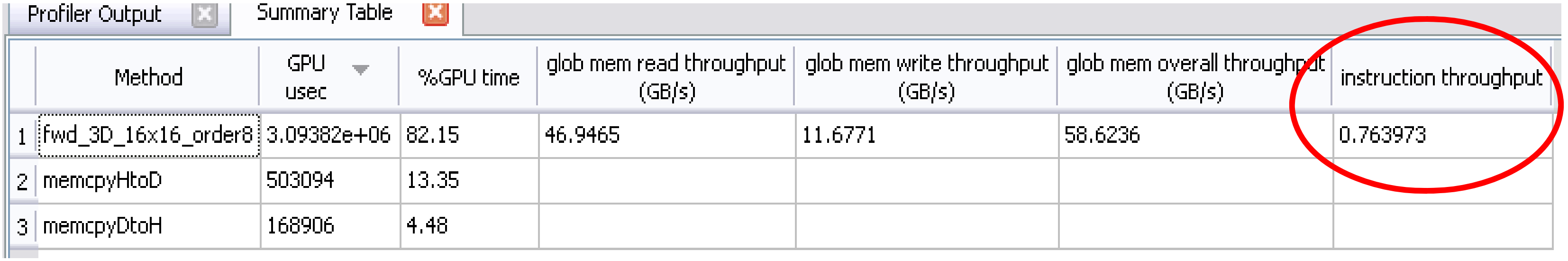
    - Branch granularity is a whole multiple of warp size

Profiler and Instruction Throughput

Visual Profiler derives:

- Instruction throughput
    - Fraction of SP arithmetic instructions that could have been issued in the same amount of time
        - So, not a good metric for code with DP arithmetic or transcendentals
- Extrapolated from one multiprocessor to GPU

Change the conditional statement and see how that affect the instruction throughput



	Method	GPU usec	%GPU time	glob mem read throughput (GB/s)	glob mem write throughput (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	fwd_3D_16x16_order8	3.09382e+06	82.15	46.9465	11.6771	58.6236	0.763973
2	memcpyHtoD	503094	13.35				
3	memcpyDtoH	168906	4.48				

Optimizing CPU/GPU interaction

Pinned (non-pageable) memory

Pinned memory enables:

- faster PCIe copies (~2x throughput on FSB systems)
- memcopies asynchronous with CPU
- memcopies asynchronous with GPU

Usage

- cudaHostAlloc / cudaFreeHost
  - instead of malloc / free
- Additional flags if pinned region is to be shared between lightweight CPU threads

Implication:

- pinned memory is essentially removed from virtual memory
- cudaHostAlloc is typically very expensive

Streams and Async API

Default API:

- Kernel launches are asynchronous with CPU
- Memcopies (D2H, H2D) block CPU thread
- CUDA calls are serialized by the driver

Streams and async functions provide:

- Memcopies (D2H, H2D) asynchronous with CPU
- Ability to concurrently execute a kernel and a memcopy
- Concurrent kernel in Fermi

Stream = sequence of operations that execute in issue-order on GPU

- Operations from different streams can be interleaved
- A kernel and memcopy from different streams can be overlapped

Overlap kernel and memory copy

Requirements:

D2H or H2D memcopy from pinned memory

Device with compute capability ≥ 1.1 (G84 and later)

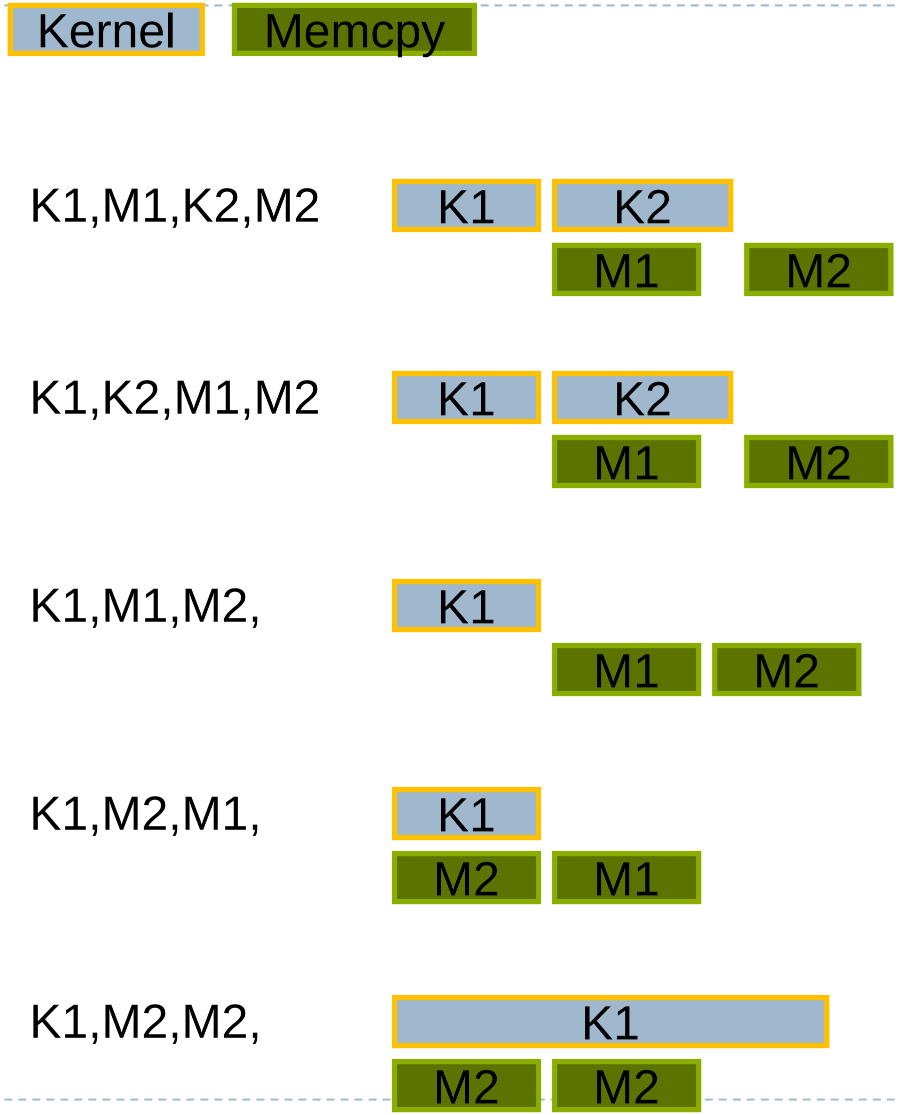
Kernel and memcopy in different, non-0 streams

Code:

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

potentially overlapped

Stream Examples



Kernel Memcpy

K1,M1,K2,M2

K1 K2

M1 M2

K1,K2,M1,M2

K1 K2

M1 M2

K1,M1,M2,

K1

M1 M2

K1,M2,M1,

K1

M2 M1

K1,M2,M2,

K1

M2 M2

Summary

Optimization needs an understanding of GPU architecture

Memory optimization: coalescing, shared memory

Execution configuration: latency hiding

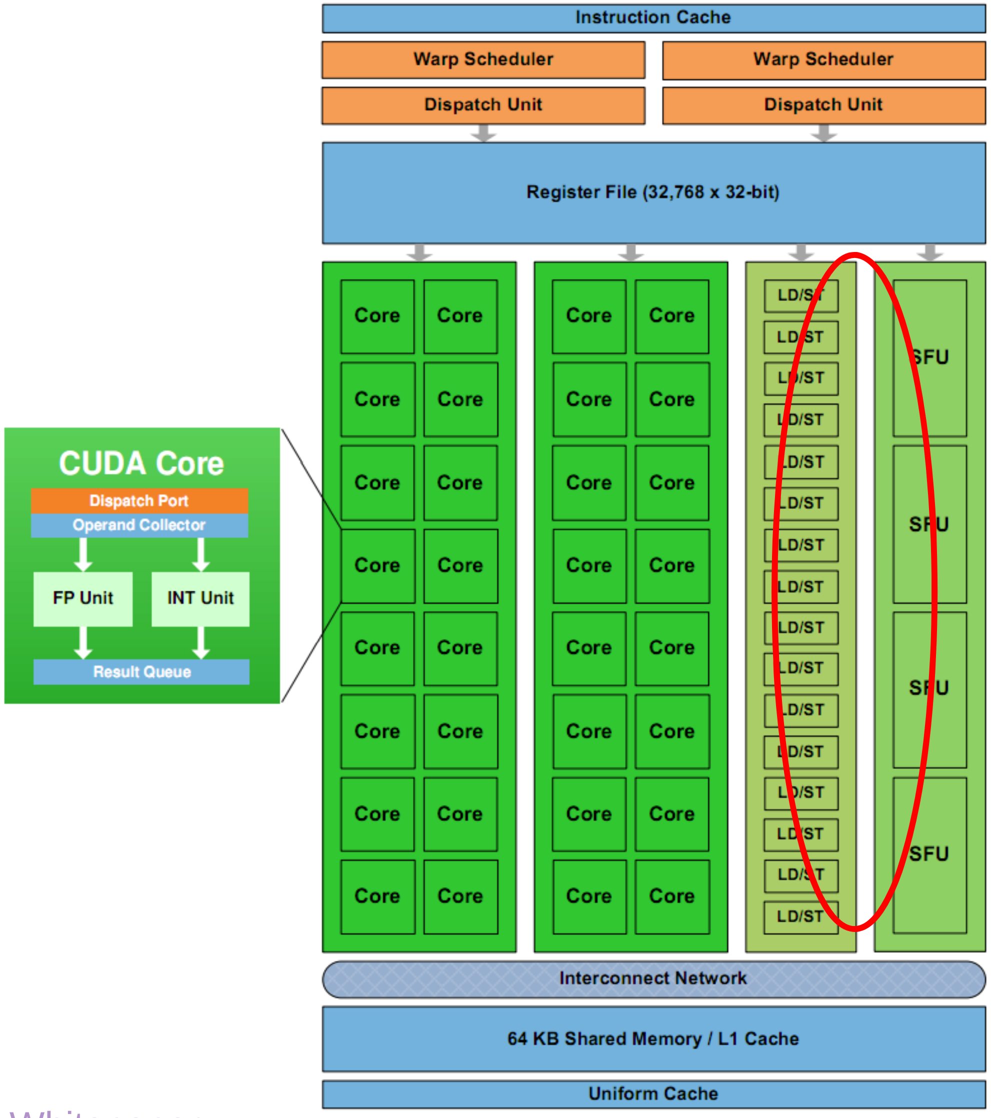
Instruction throughput: use high throughput inst

Do measurements!

- Use the Profiler, simple code modifications
- Compare to theoretical peaks

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

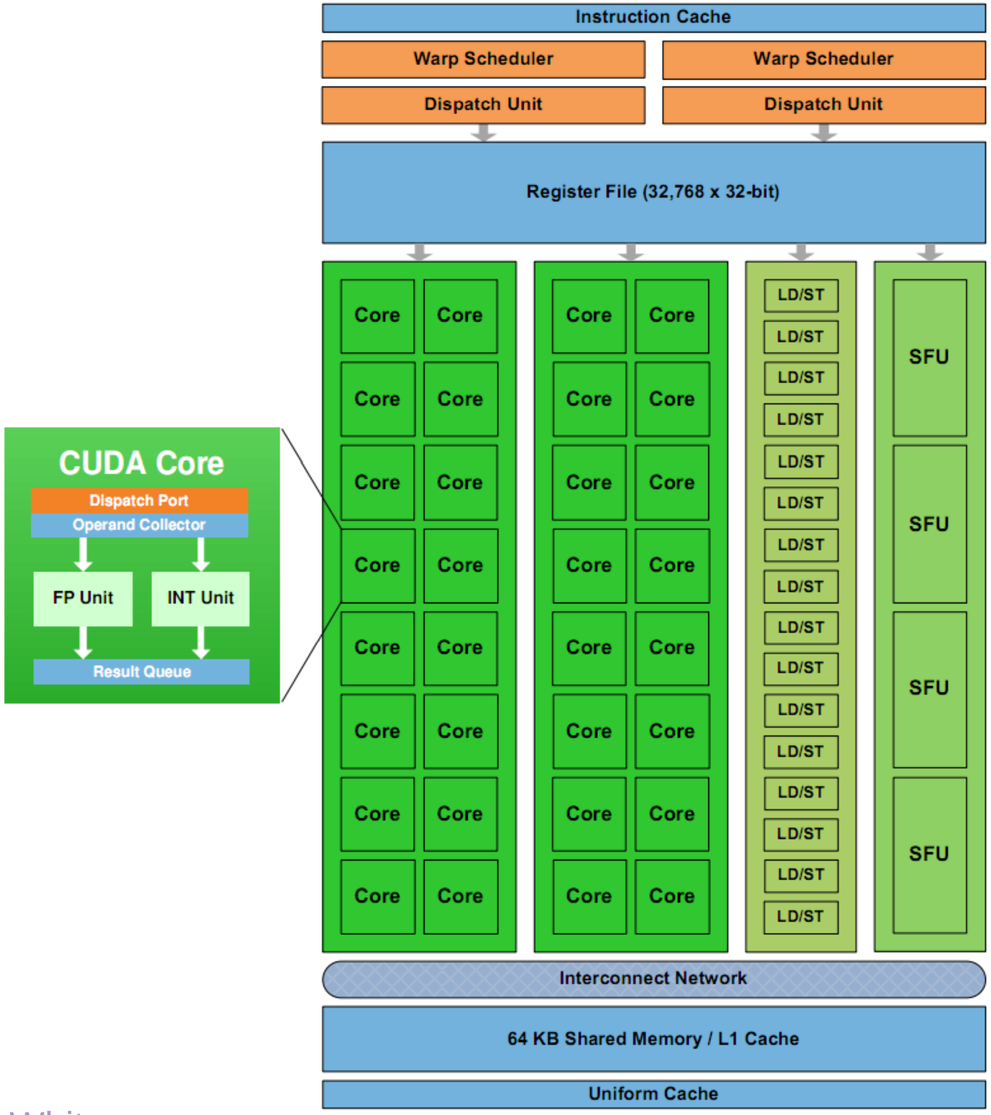
1 loop branch instruction

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?