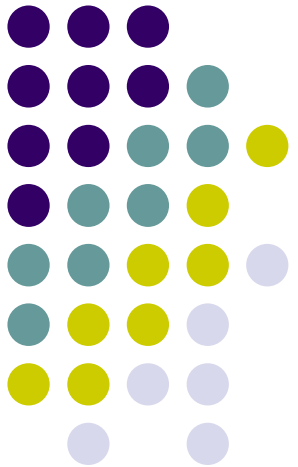
ME964
High Performance Computing for Engineering Applications

CUDA Optimization

Tiling as a Design Pattern in CUDA

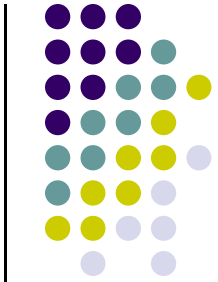
Vector Reduction Example

March 13, 2012



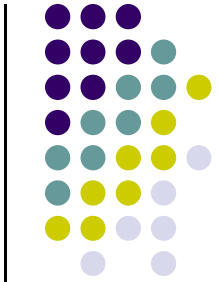
"The inside of a computer is as dumb as hell but it goes like mad!"
Richard Feynman

Before We Get Started…



- Last time
  - Shared memory & bank conflicts
  - Synchronization issues
  - Atomic operations
- Today
  - Tiling – a programming design pattern in CUDA [quick discussion]
  - CUDA Optimization/Best Practices issues
    - Example: Vector Reduction
    - Execution Configuration Heuristics
- Other issues
  - HW7 posted online, due on Th at 11:59 PM
  - The Midterm Project race is underway
    - Default project: solving dense banded linear system
    - For default project: I will provide guidelines in terms of profiling, comparison to existing solutions, etc.
  - The syllabus has been updated online, check it out…

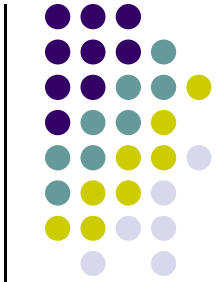
Putting Things in Perspective…



- Here's what we've covered so far:
  - CUDA execution configuration (grids, blocks, threads)
  - CUDA scheduling issues (warps, thread divergence, synchronization, etc.)
  - CUDA Memory ecosystem (registers, shared mem, device mem, L1/L2 cache, etc.)
  - Practical things: building, debugging, profiling CUDA code

- Next: CUDA GPU Programming - Examples & Code Optimization Issues
  - Tiling: a CUDA programming pattern
  - Example: CUDA optimization exercise in relation to a vector reduction operation
  - CUDA Execution Configuration Optimization Heuristics: Occupancy issues
  - CUDA Optimization Rules of Thumb

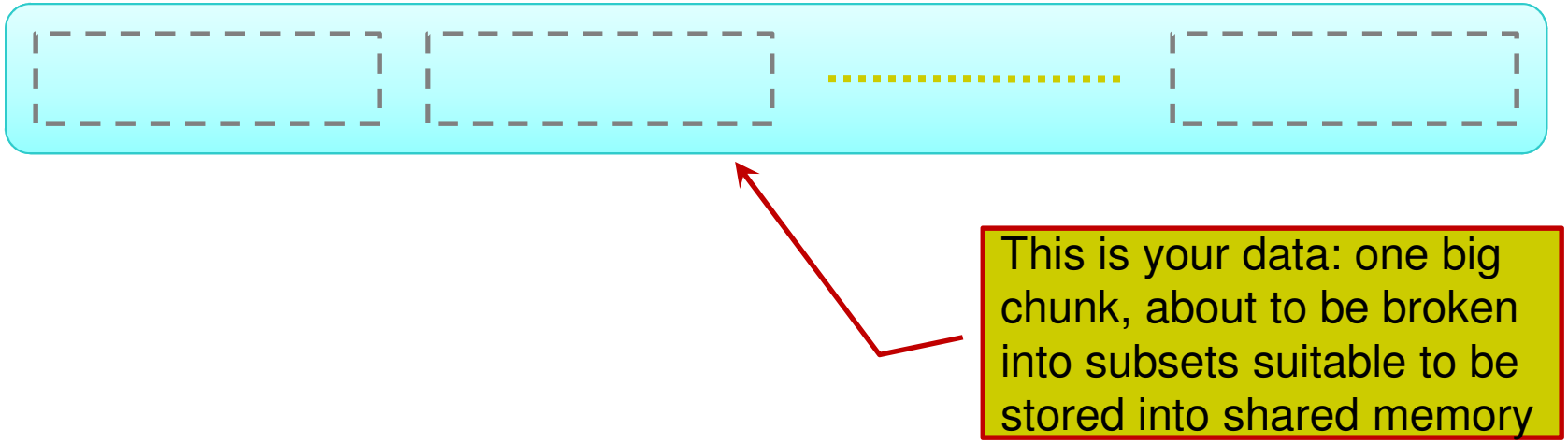
Tiling [Blocking]:
A Fundamental CUDA Programming Pattern



- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a block of threads
  - No different from cache blocking!
    - Except you now have full control over it

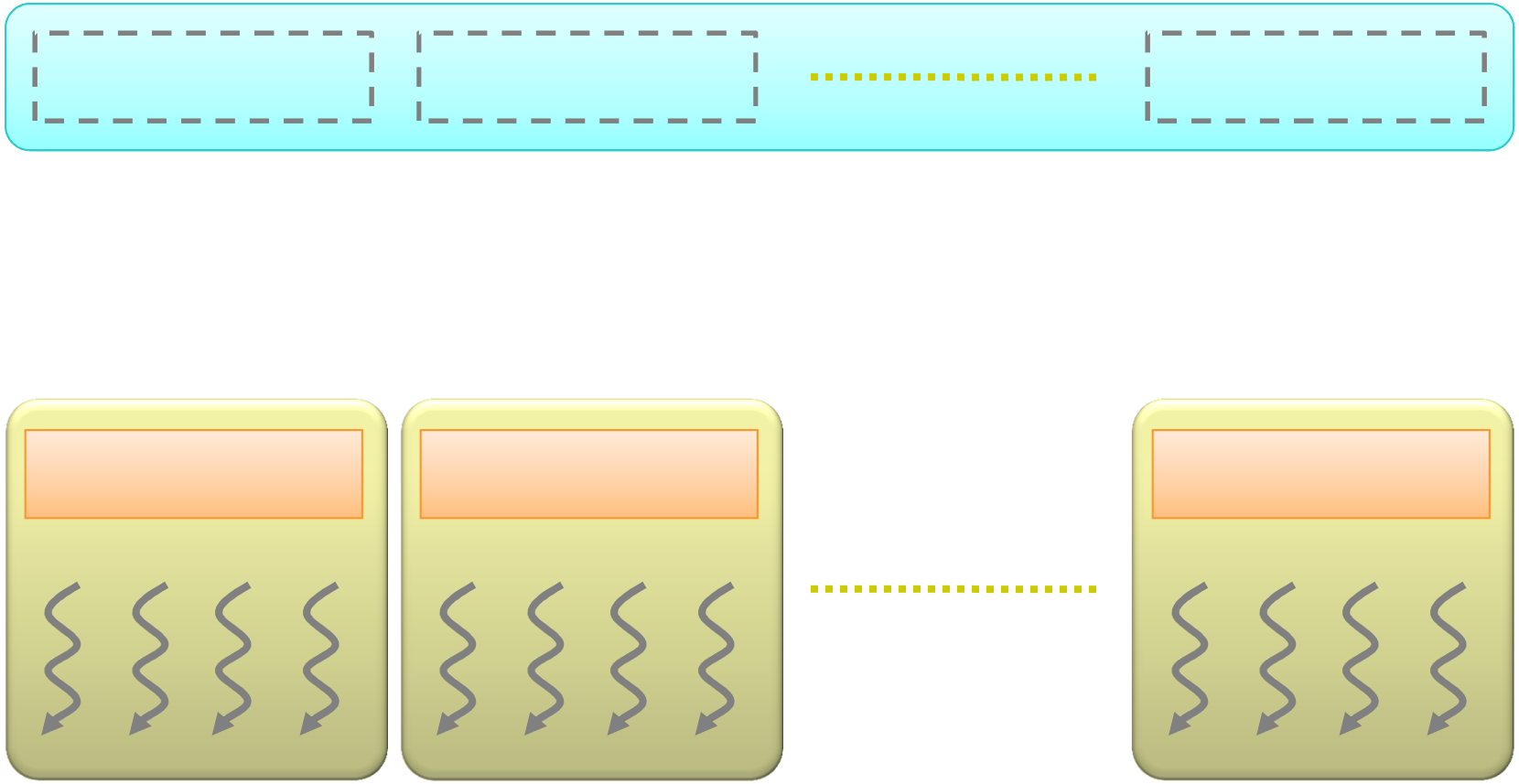
- Provides several significant performance benefits
  - Working in shared memory reduces memory latency dramatically
  - More likely to have address access patterns that coalesce well on load/store to shared memory

Fundamental CUDA Pattern: Tiling



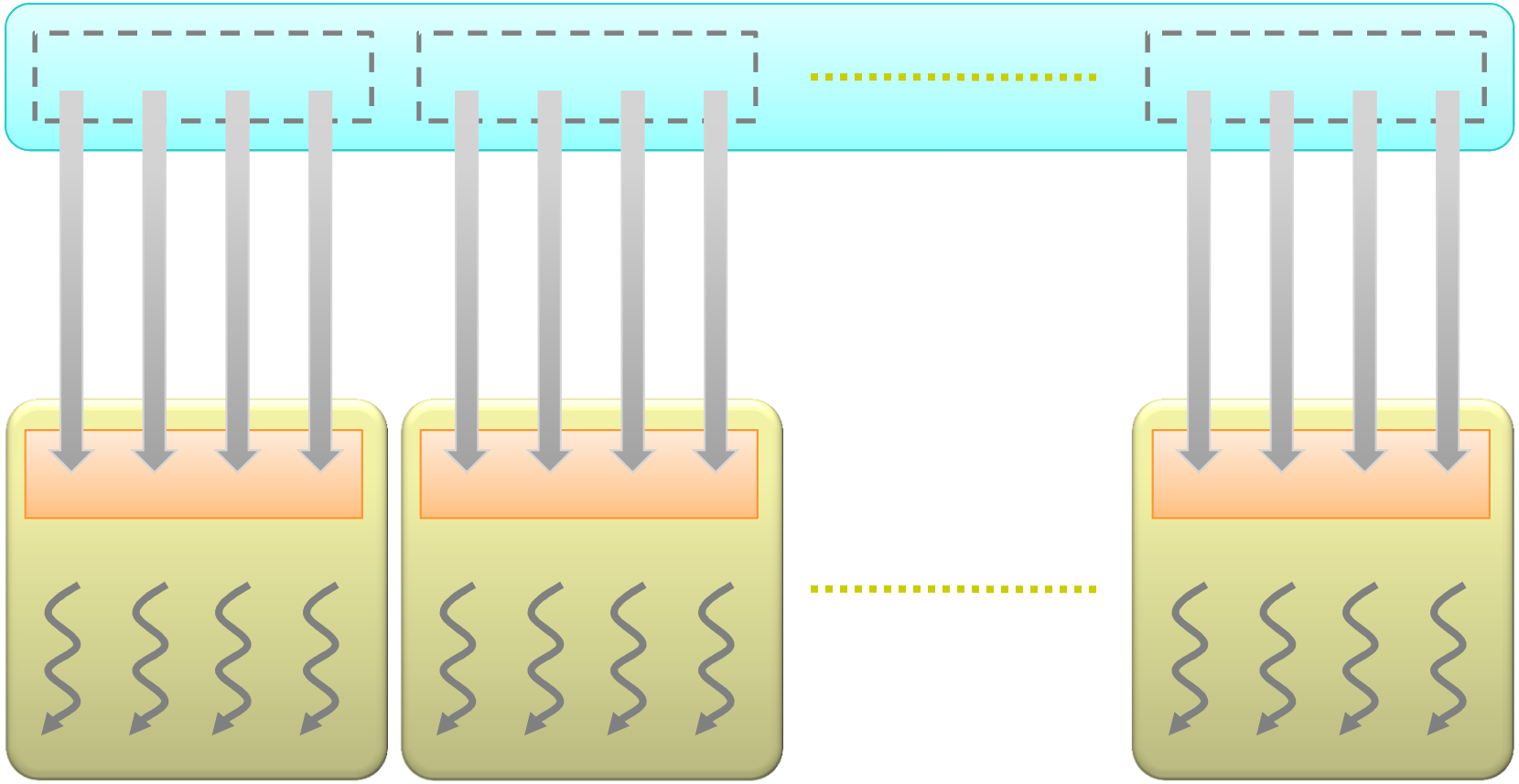
- Partition data into subsets that fit into **__shared__** memory

Fundamental CUDA Pattern: Tiling



- Process each data subset with one thread block

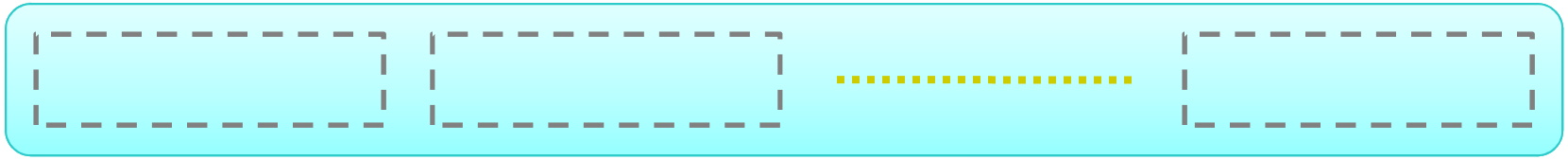
Fundamental CUDA Pattern: Tiling

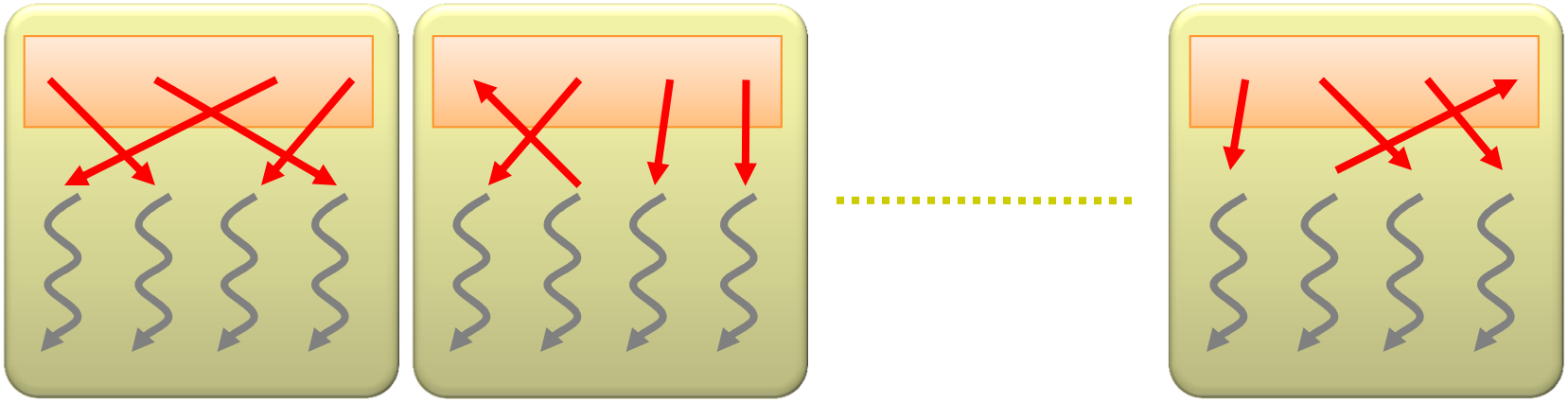


- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

NVIDIA [J. Balfour]→

Fundamental CUDA Pattern: Tiling

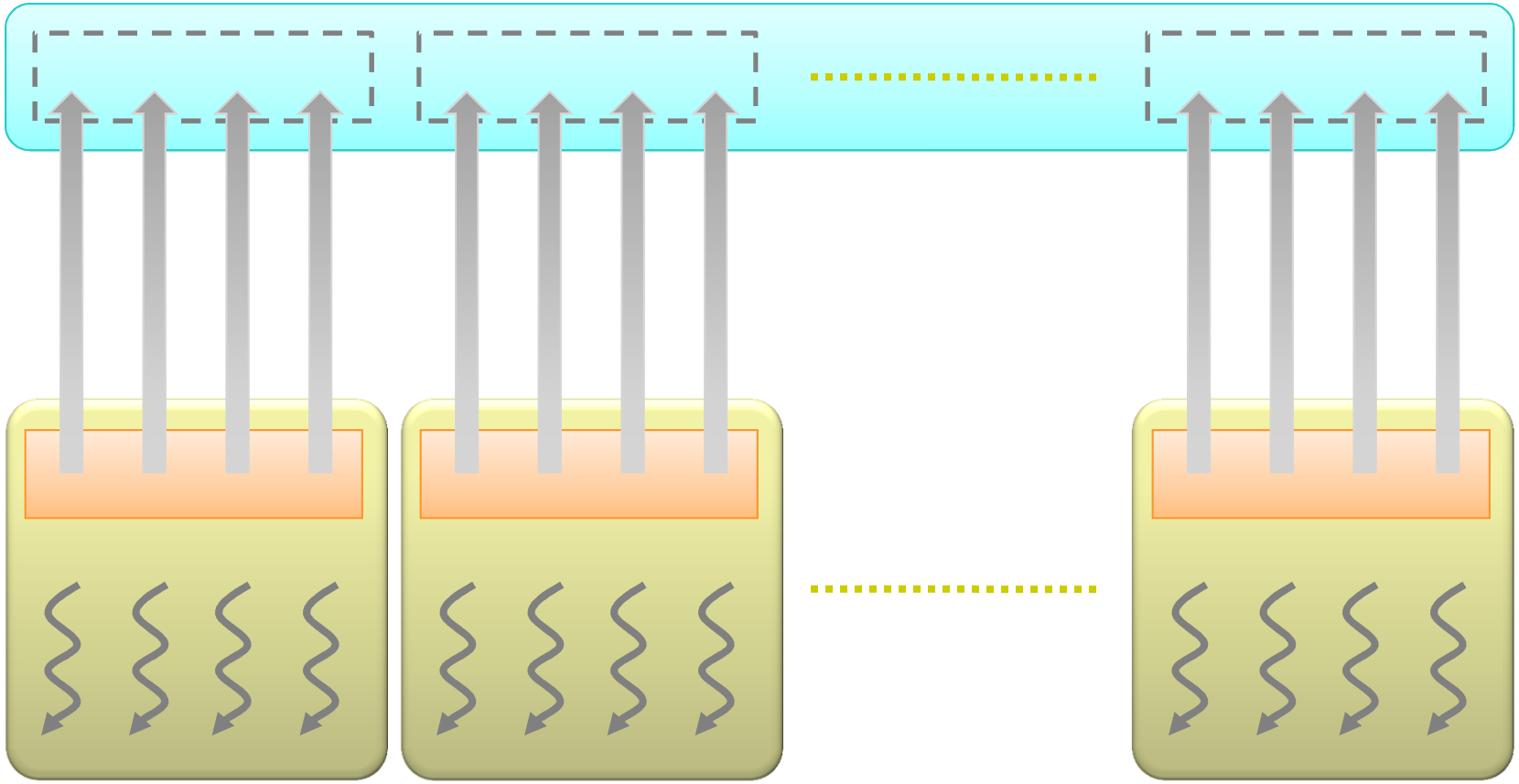




- Perform the computation on the subset from shared memory

NVIDIA [J. Balfour]→

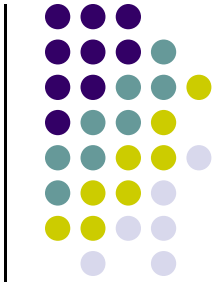
Fundamental CUDA Pattern: Tiling



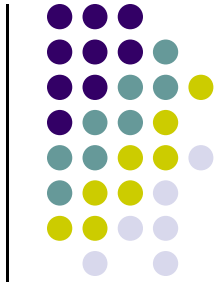
- Copy the result from __shared__ memory back to global memory

NVIDIA [J. Balfour]→

Tiling: Closing Remarks



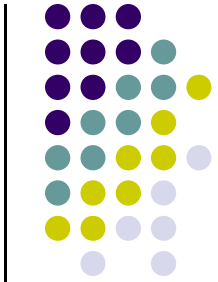
- A large number of CUDA kernels are built this way
- However, tiling [blocking] may not be the only approach to solving a problem, sometimes it might not apply…
- Two questions that can guide you in deciding if tiling is it:
  - Does a thread require several loads from global memory to serve its purpose?
  - Could data used by a thread be used by some other thread in the same block?
  - If answer to both questions is “yes”, consider tiling as a design pattern
- The answer to these two questions above is not always obvious
  - Sometime it’s useful to craft an altogether new approach (algorithm) that is capable of using tiling: you force the answers to be “yes”



A CUDA Optimization Exercise

[A Demonstration Using the Parallel Reduction Application]

Parallel Reduction in CUDA



- Exercise draws on material made available by Mark Harris of NVIDIA
  [acknowledgement at bottom of slides]

- Parallel Reduction: Common and very important data parallel primitive
  - Example: Used to compute the norm of a large vector

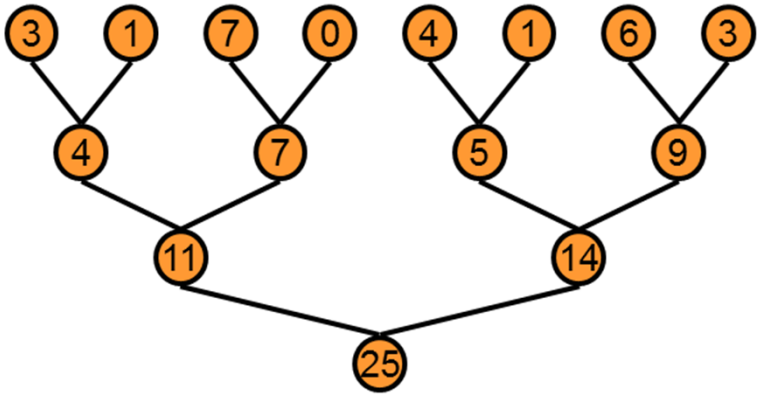
- Easy to implement in CUDA
  - Challenging to get it right though

- Serves as a great optimization example
  - Walk step by step through several different versions
  - Demonstrates several important optimization strategies

NVIDIA [M. Harris]→

Parallel Reduction

- Basic Idea: tree-based approach used within each thread block

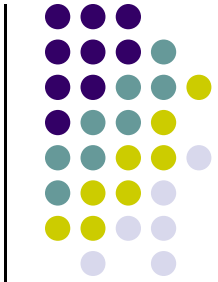


- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array to one single value

- Q: How do we communicate partial results between thread blocks?

NVIDIA [M. Harris]→

Problem: Global Synchronization



- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively

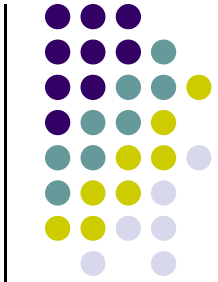
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than number of SMs times the number of resident blocks / SM) → this may reduce overall efficiency

- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

NVIDIA [M. Harris]→

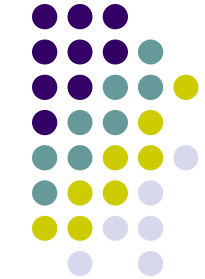
Multiple Kernel Calls

[An Example, and how it all works out…]

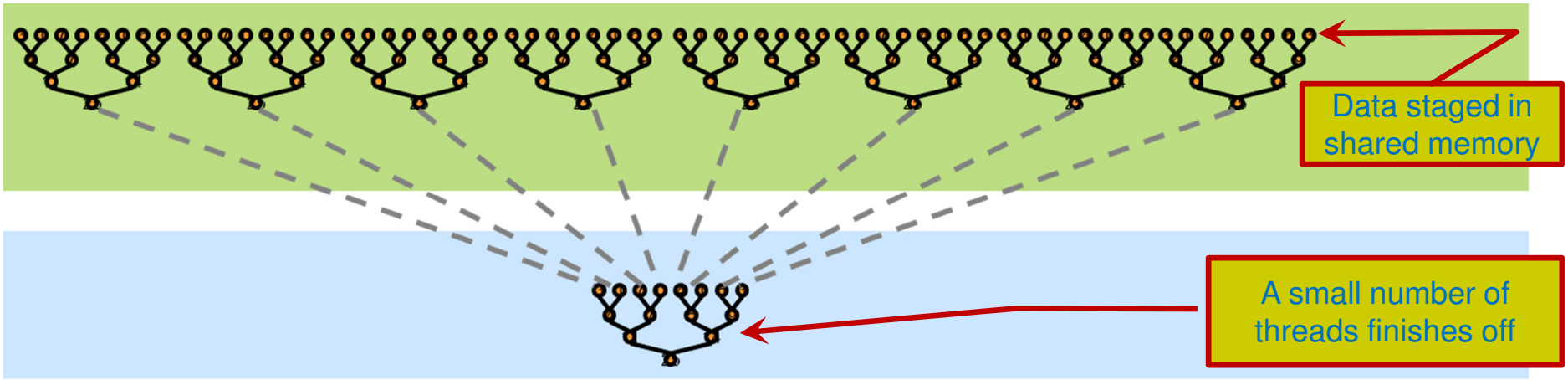


- Imagine you launch a grid in which each block has 256 threads
- Assume that the number or elements in the array is N=100,000
  - Note that 100,000= 390*256 + 160, therefore **`ceil[N/256.0]=391`**
- For the first stage, you launch 391 blocks of 256 threads
  - At the end of this stage you still have to operate on 391 elements
- For the second stage, you launch two blocks of 256 threads
  - At the end of this stage you only have to operate on two elements
- For the third and last stage, you launch one block of 256 threads
  - Almost all threads idle…
- NOTE: after the first stage, each subsequent stage operates on a number of entries equal to the number of blocks in the previous stage

Vector Reduction: 30,000 Feet Perspective

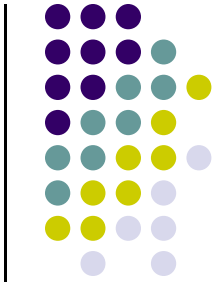


- At the block level: Bring data in shared memory, then start adding in parallel
- Fewer and fewer threads of a block participate
- The process is memory bound, low arithmetic intensity…



NVIDIA [M. Harris]→

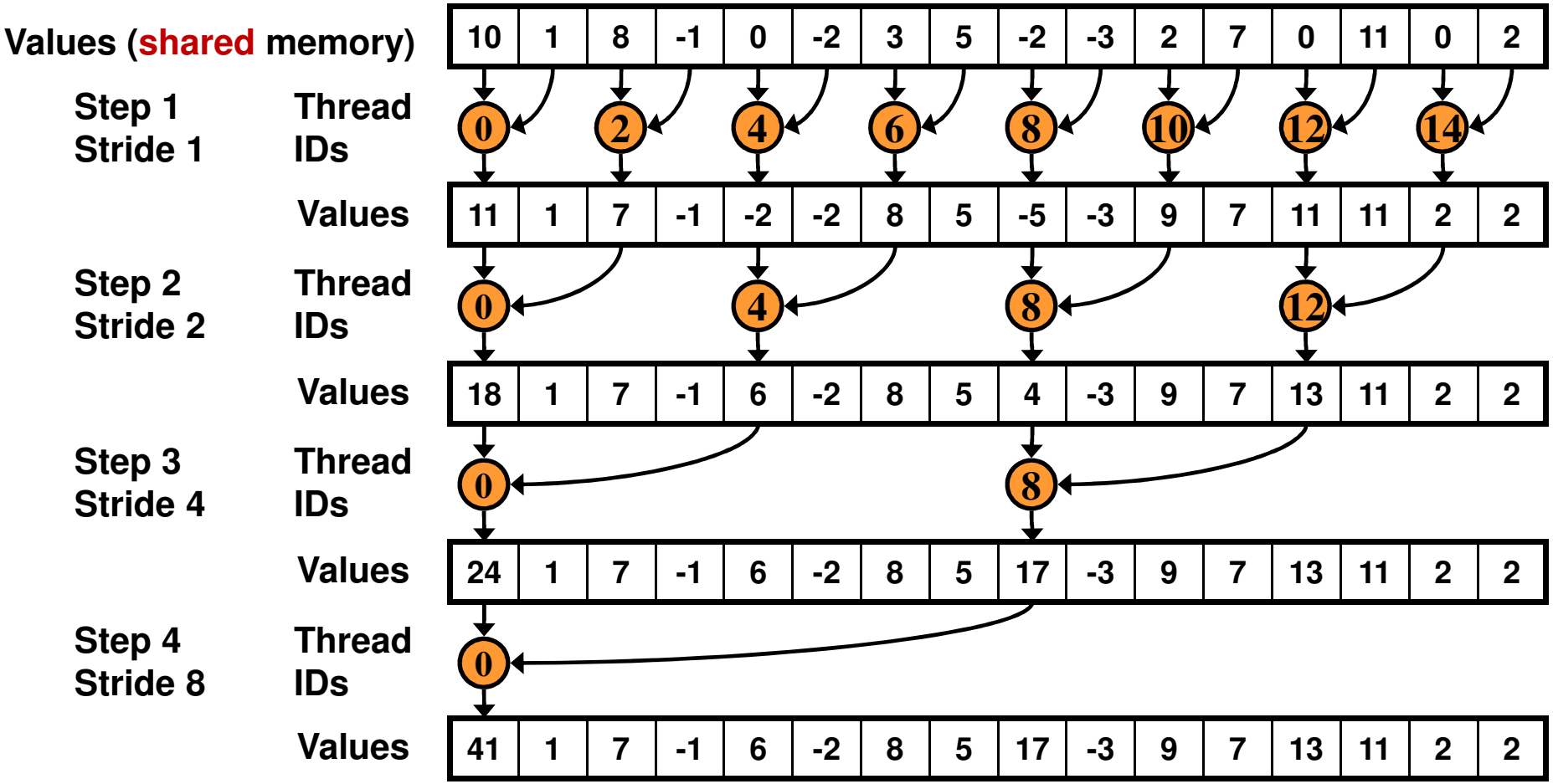
What is Our Optimization Goal?



- We should strive to reach GPU peak performance
- Choose the right metric:
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth
- This example uses results generated using a G80 GPU
  - Compute capability (CC) 1.0
  - 384-bit memory interface, 900 MHz DDR
  - 384 * 900 *2 / 8 = 86.4 GB/s
  - Example carries over to other CCs, this algorithm will be memory bound

NVIDIA [M. Harris]→

Parallel Reduction: Interleaved Addressing



NVIDIA [M. Harris]→

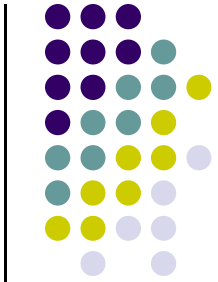
Reduction #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global memory
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

NVIDIA [M. Harris]→

Performance for 4M element reduction

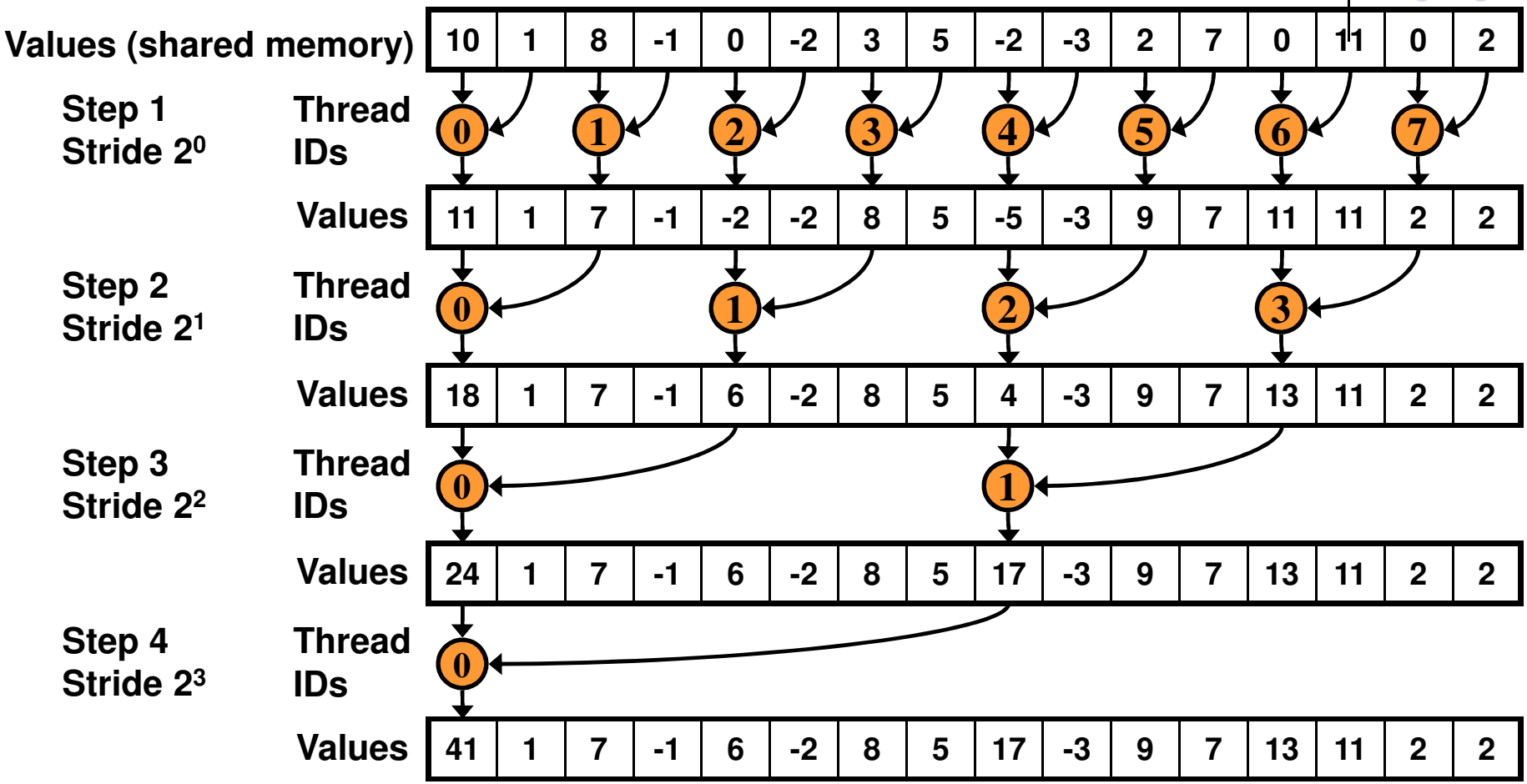


	Time (2^{22} ints)	Bandwidth
Kernel 1: **interleaved addressing with divergent branching**	**8.054 ms**	**2.083 GB/s**

Note: Block Size = 128 threads for all tests

NVIDIA [M. Harris]→

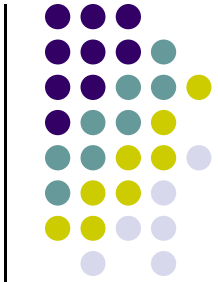
Parallel Reduction: Interleaved Addressing



New Problem: Shared Memory Bank Conflicts

NVIDIA [M. Harris]→

Reduction #2: Interleaved Addressing



Just replace divergent branch in inner loop…

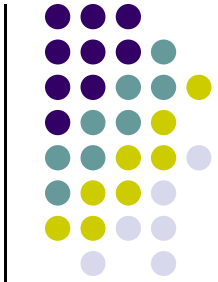
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

…with strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

NVIDIA [M. Harris]→

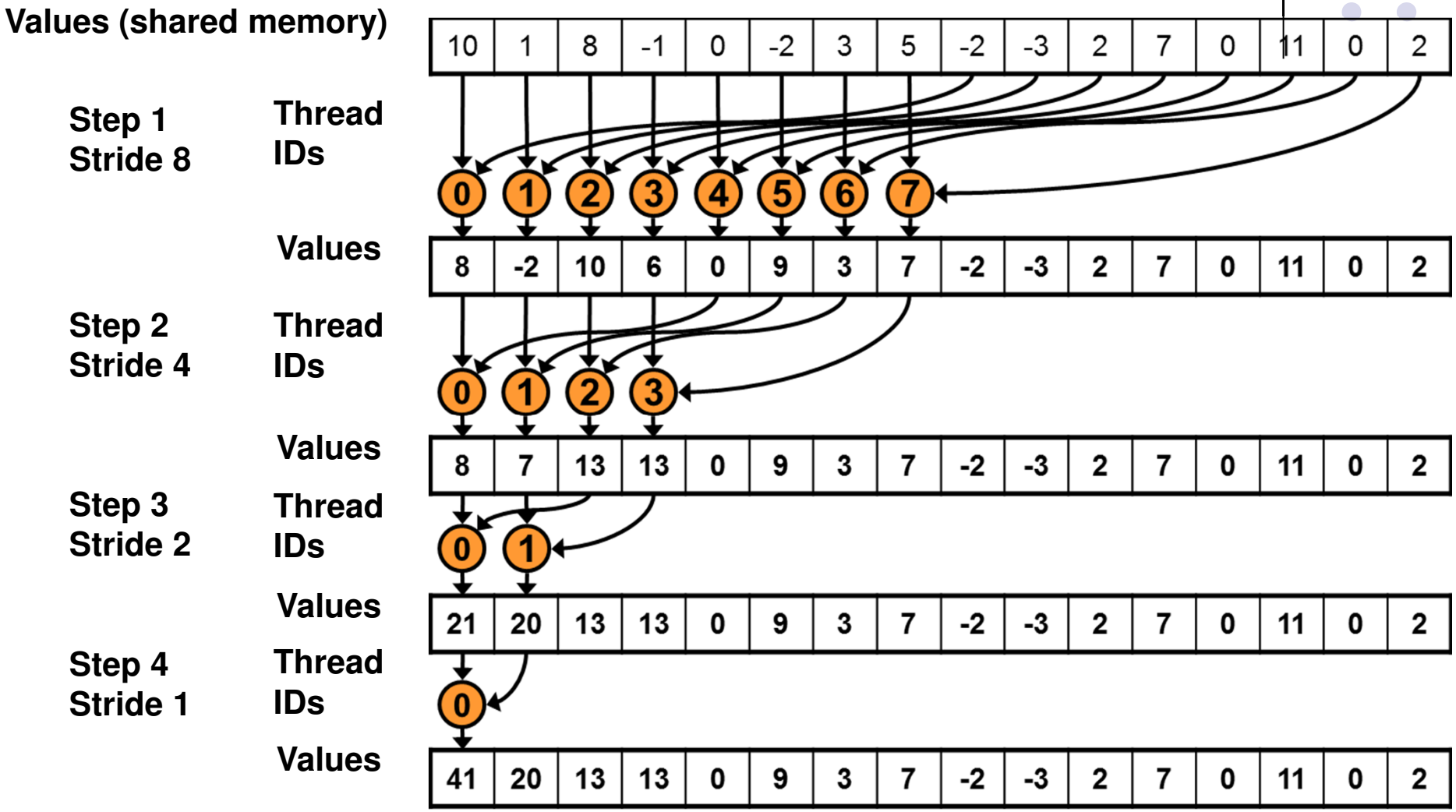
Performance for 4M element reduction



	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

NVIDIA [M. Harris]→

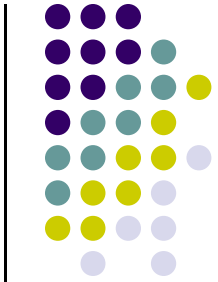
Parallel Reduction: Sequential Addressing



Sequential addressing is conflict free

NVIDIA [M. Harris]→

Reduction #3: Sequential Addressing



Just replace strided indexing in inner loop…

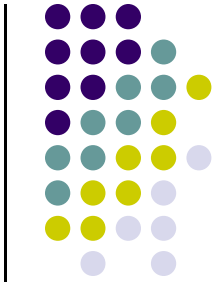
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

…with reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```


NVIDIA [M. Harris]→

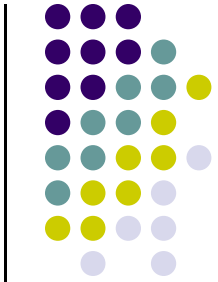
Performance for 4M element reduction



	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

NVIDIA [M. Harris]→

Idle Threads…



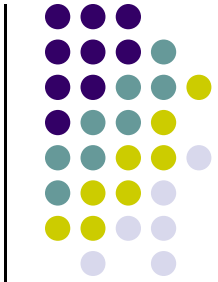
Current solution:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Note that half of the threads are idle on first loop iteration!
This is wasteful…

NVIDIA [M. Harris]→

Reduction #4: First Add During Load



Replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

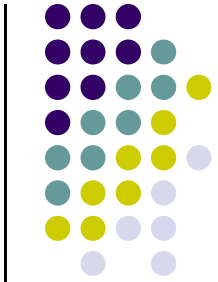
…With two loads and first add of the reduction:

```
// perform first level of reduction upon reading from
// global memory and writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

One side effect: the number of blocks you need now is half of what it used to be…

NVIDIA [M. Harris]→

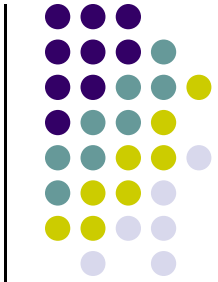
Performance for 4M element reduction



	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

NVIDIA [M. Harris]→

Instruction Bottleneck



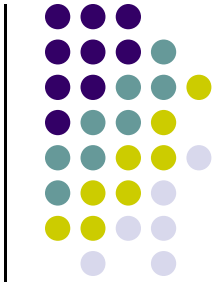
- At 17 GB/s, we're far from bandwidth bound
  - And we know reduction has low arithmetic intensity

- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead

- Strategy: unroll loops

NVIDIA [M. Harris]→

Unrolling the Last Warp



- As reduction proceeds, the number of “active” threads decreases
  - When `s <= 32`, we have only one warp left

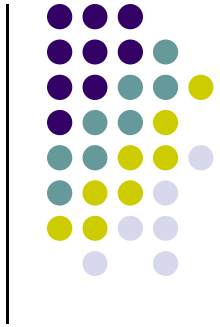
- Instructions are SIMD synchronous within a warp
  - All threads in a warp proceed in lockstep fashion

- That means when `s <= 32`:
  - We don’t need to `__syncthreads()`
  - We don’t need “`if (tid < s)`” because it doesn’t save any work

- Let’s unroll the last 6 iterations of the inner loop

NVIDIA [M. Harris]→

Reduction #5: Unroll the Last Warp



```
// and use later like this…
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32) warpReduce(sdata, tid);
```

This used to be: `s>0`

```
__device__ void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid +  8];
    sdata[tid] += sdata[tid +  4];
    sdata[tid] += sdata[tid +  2];
    sdata[tid] += sdata[tid +  1];
}
```

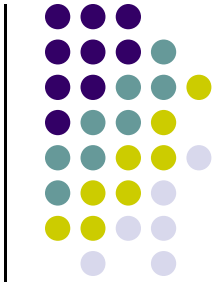
IMPORTANT: For this to be correct, we must use the "`volatile`" keyword!

Note: This saves useless work in *all* warps, not just the last one!

Without unrolling, all warps execute every iteration of the for loop and if statement

NVIDIA [M. Harris]→

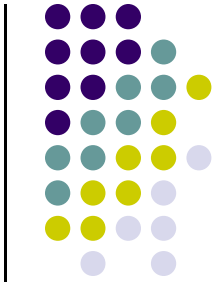
Performance for 4M element reduction



	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

NVIDIA [M. Harris]→

Complete Unrolling



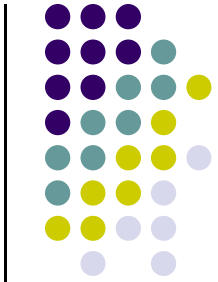
- If we knew the number of iterations (or equivalently, of threads in a block) at compile time, we could completely unroll the reduction
  - Luckily, the block size on G80 is limited by the GPU to 512 threads
    - 1024 on newer Fermi GPUs
  - Also, we are sticking to power-of-2 block sizes

- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?

- Use of templates can solve this issue…
  - CUDA supports C++ template parameters on device and host functions

NVIDIA [M. Harris]→

Unrolling with Templates



- Specify block size as a function template parameter
- The kernel is parameterized:

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata)
```

NVIDIA [M. Harris]→

Reduction #6: Completely Unrolled

```
if (blockSize >= 512) {
    if(tid < 256){ sdata[tid] += sdata[tid + 256]; } __syncthreads();}
if (blockSize >= 256) {
    if(tid < 128){ sdata[tid] += sdata[tid + 128]; } __syncthreads();}
if (blockSize >= 128) {
    if(tid <  64){ sdata[tid] += sdata[tid +  64]; } __syncthreads();}

if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
}
```

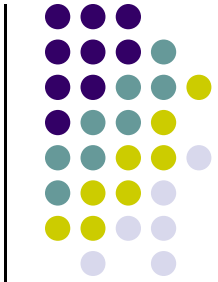
- All code in RED will be evaluated at compile time. Results in a very efficient inner loop.
- For Fermi, you'd have one more `if` statement that covers the case when `blockSize>=1024`

Invoking Template Kernels

```
switch (threads) {
case 512:
    reduce6<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case 256:
    reduce6<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case 128:
    reduce6<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case 64:
    reduce6< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case 32:
    reduce6< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case 16:
    reduce6< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case  8:
    reduce6<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case  4:
    reduce6<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case  2:
    reduce6<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
case  1:
    reduce6<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```

NVIDIA [M. Harris]→

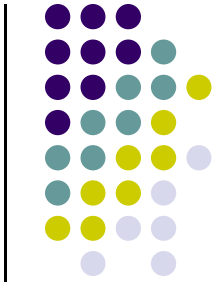
Performance for 4M element reduction



	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

NVIDIA [M. Harris]→

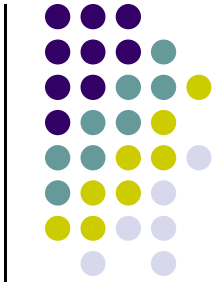
Parallel Reduction Complexity



- Assume that the number of elements in array is of the form $N=2^D$
- Log(N) parallel stages, each stage S requires $N/2^S$ independent ops
  - Stage Complexity is O(log N)
- For $N=2^D$, approach requires a total of $\sum_{S\in[1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is O(N) – It is work-efficient
  - That is, it does not perform more operations than a sequential algorithm
- Time complexity, for P threads physically in parallel (P processors): O(N/P + log N)
  - Compare to O(N) for sequential reduction
  - In a thread block, N=P, so O(log N)

NVIDIA [M. Harris]→

What About *Cost?*



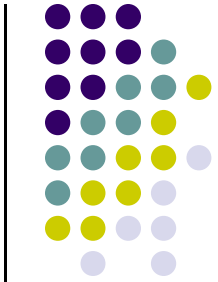
- *Cost* of a parallel algorithm is processors × time complexity
  - Allocate threads instead of processors: O(*N*) threads
  - Time complexity is O(log *N*), so *cost* is O(*N* log *N*) : not cost efficient!

- Brent's theorem suggests O(*N*/log *N*) threads
  - Each thread does O(log *N*) sequential work
  - Then all O(*N*/log *N*) threads cooperate for O(log *N*) stages
  - Cost = O((*N*/log *N*) * log *N*) = O(*N*) → cost efficient

- Sometimes called *algorithm cascading*
  - Can lead to significant speedups in practice

NVIDIA [M. Harris]→

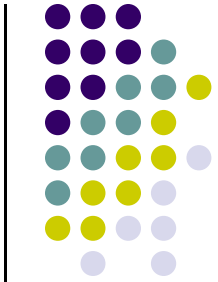
Algorithm Cascading



- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Brent's theorem says each thread should sum O(log n) elements
  - i.e. 1024 or 2048 elements per block vs. 256
- Probably beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks
- On G80, best performance with 64-256 blocks of 128 threads
  - 1024-4096 elements per *thread*

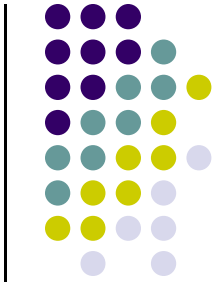
NVIDIA [M. Harris]→

Kernel 7, Comments



- For the first six kernels a large number of blocks was used to “tile” the array
- Kernel 7: reduce the number of blocks and have a thread do more work than just fetch something to shared memory
- Example [cooked up, not related to actual CUDA warp size, typical CUDA block dim, etc.]:
  - Say you have 1024 elements stored in an array; you need to reduce that array
  - You start with 32 blocks, each with 4 threads
  - Then, 128 threads total. It means that a thread, say in block 11, would have to add two numbers, then two numbers, then two numbers, then two more numbers.
  - At this point, everything is in the union of the shared memory associated with the 32 blocks. At this point proceed like before with kernel 6.

Reduction #7: Multiple Adds / Thread



Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

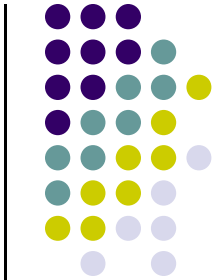
With a while loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride to maintain coalescing!

NVIDIA [M. Harris]→

Performance for 4M element reduction

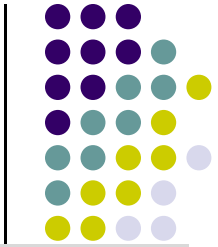


Kernel 7 on 32M elements: 73 GB/s!

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

NVIDIA [M. Harris]→

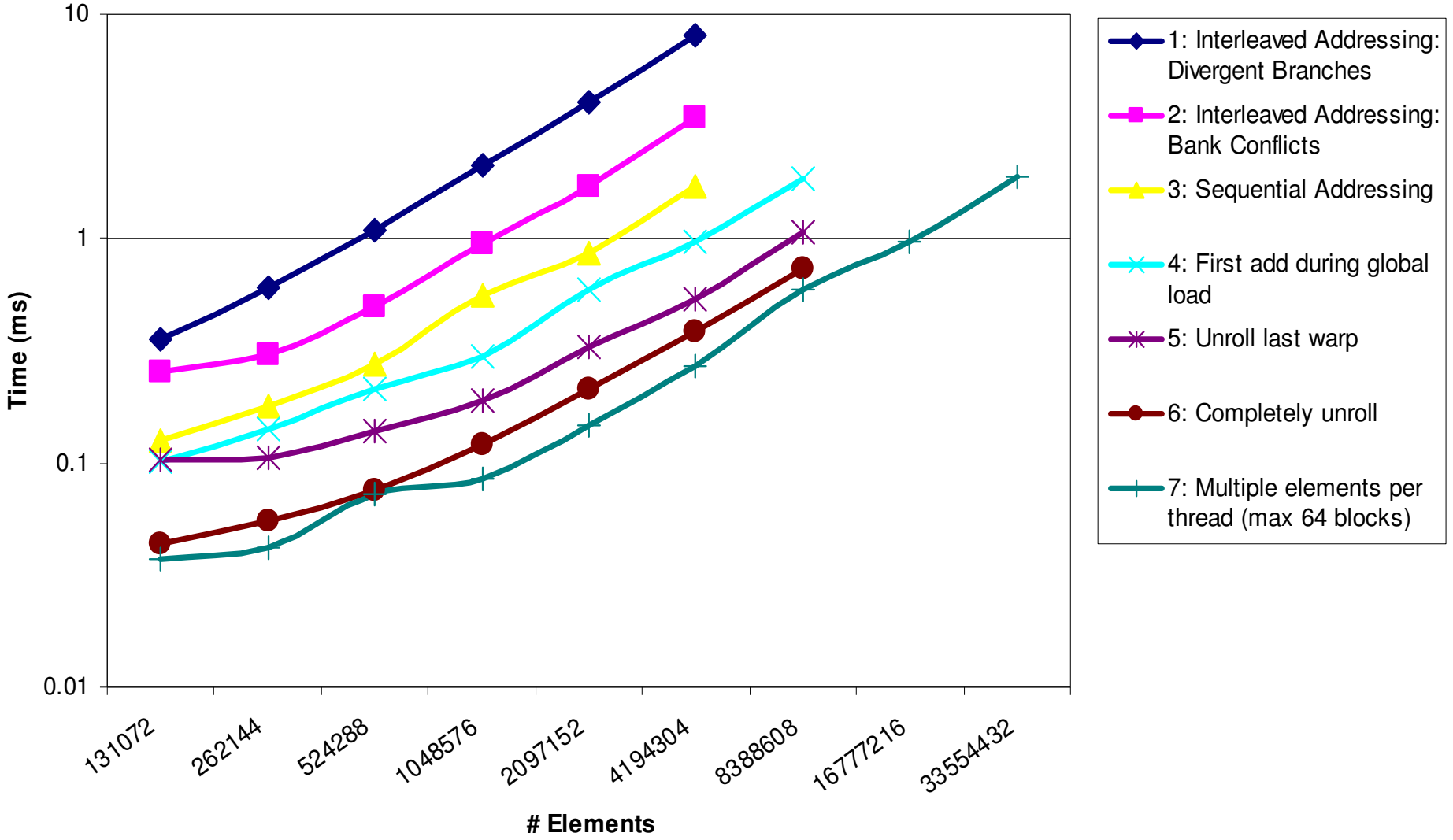
Final Kernel…



```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}

template <unsigned int blockSize>
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

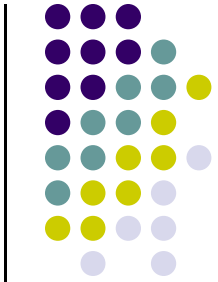
NVIDIA [M. Harris]→

Performance Comparison



NVIDIA [M. Harris]→

Sources of Efficiency Improvement

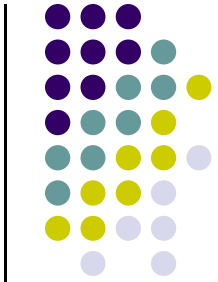


- Algorithmic optimizations
  - Changes to addressing, algorithm cascading
  - 11.84x speedup, combined!

- Code optimizations
  - Loop unrolling
  - 2.54x speedup, combined

NVIDIA [M. Harris]→

Lessons Learned, Vector Reduction



- Understand CUDA performance characteristics
  - Memory coalescing
  - Warp divergence
  - Bank conflicts
  - Latency hiding
- Use peak performance metrics to guide optimization
- Know how to identify type of bottleneck
  - E.g. memory, core computation, or instruction overhead
- Optimize your algorithm, *then* unroll loops
- Use template parameters to generate optimal code
- Understand parallel algorithm complexity theory

NVIDIA [M. Harris]→