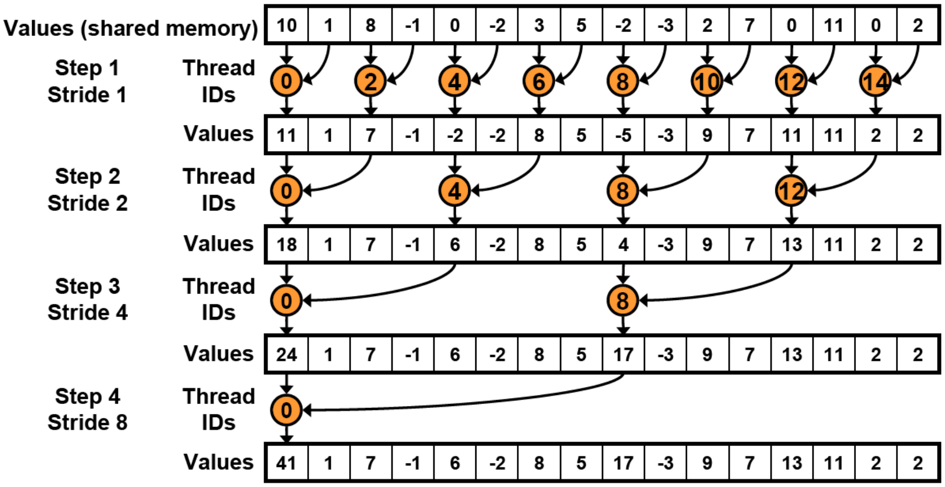
Implementing Reductions II

Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green, Mark Harris, and many others
Optimizations studied on G80 hardware

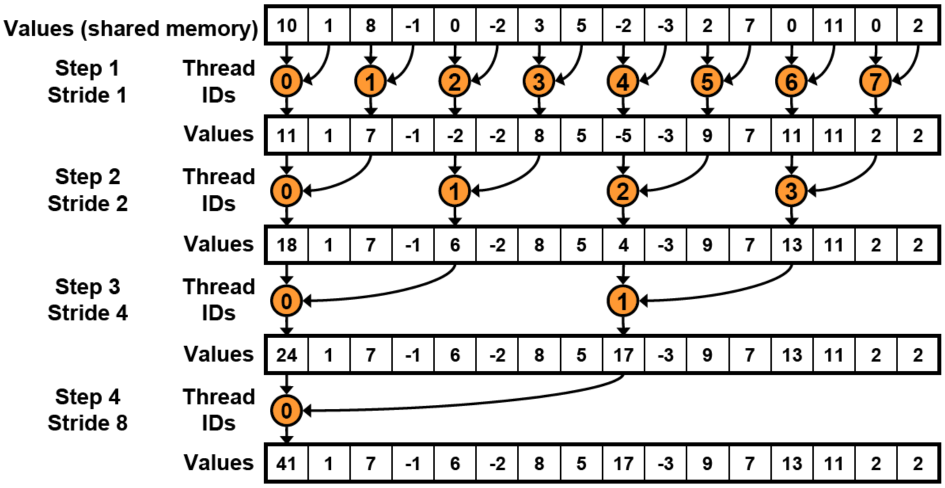
Reduction #1: interleaved addressing, interleaved thread idx



Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

Problem: active threads → divergent branching

Reduction #2: interleaved addressing, sequential thread idx

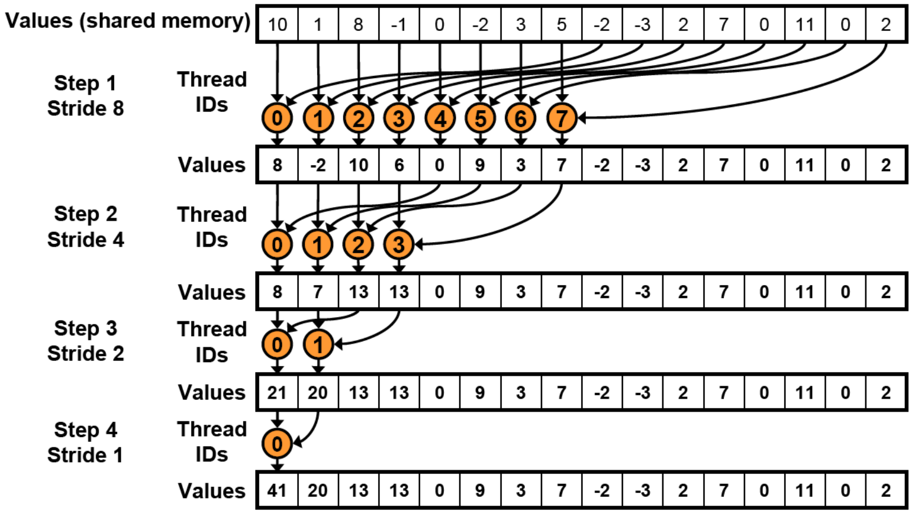


Values (shared memory)		10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1	Thread IDs	0		1		2		3		4		5		6		7	
	Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2	Thread IDs	0				1				2				3			
	Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4	Thread IDs	0								1							
	Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Step 4 Stride 8	Thread IDs	0															
	Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

Problem: interleaved addressing → bank conflicts

3

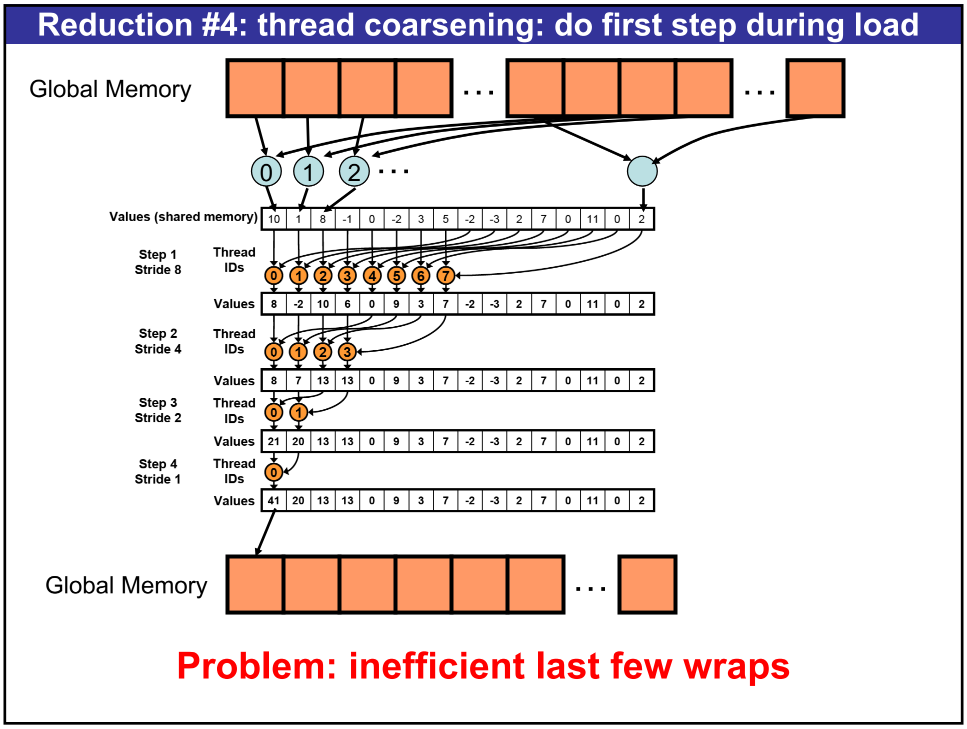
Reduction #3: sequential addressing, sequential thread idx



Values (shared memory)		10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 8	Thread IDs	0	1	2	3	4	5	6	7								
	Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 2 Stride 4	Thread IDs	0	1	2	3												
	Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 3 Stride 2	Thread IDs	0	1														
	Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Step 4 Stride 1	Thread IDs	0															
	Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Problem: wasting thread resources after GMEM load

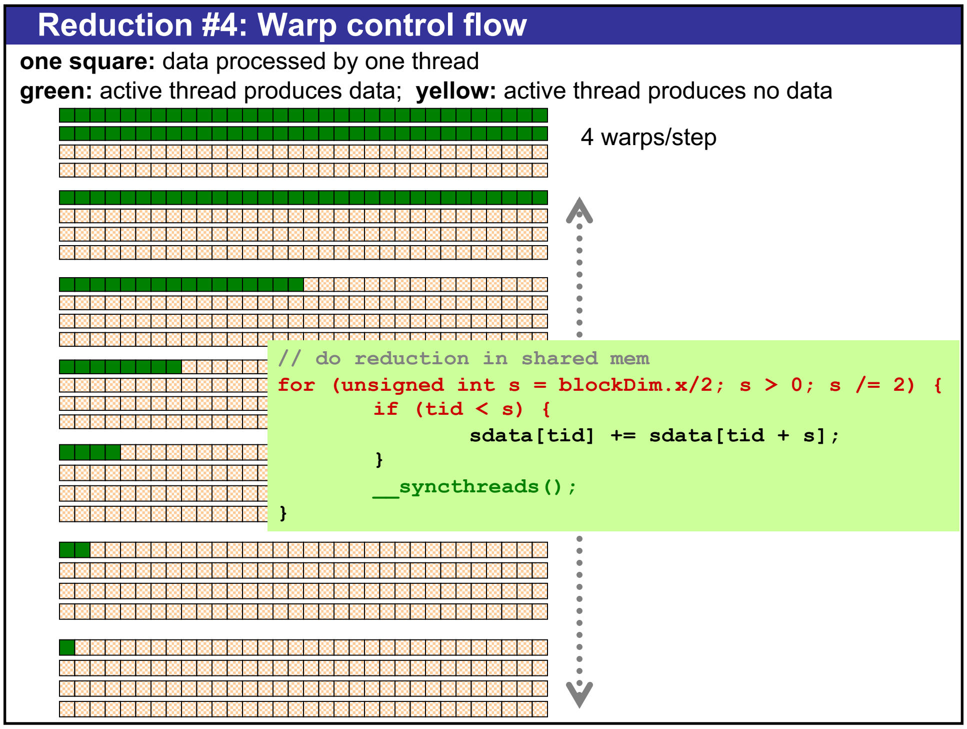
4



Reduction #4: thread coarsening: do first step during load

Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Problem: inefficient last few wraps



Reduction #4: Warp control flow

one square: data processed by one thread

green: active thread produces data; **yellow:** active thread produces no data

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

Reduction #5: Unrolling the last 6 iterations

```
    // do reduction in shared mem
 for (unsigned int s = blockDim.x/2; s > 32; s /= 2) {

     if (tid < s) {
           sdata[tid] += sdata[tid + s];
     }
     __syncthreads();
 }
```

```
 if (tid <32)
   {
     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];
 }
```

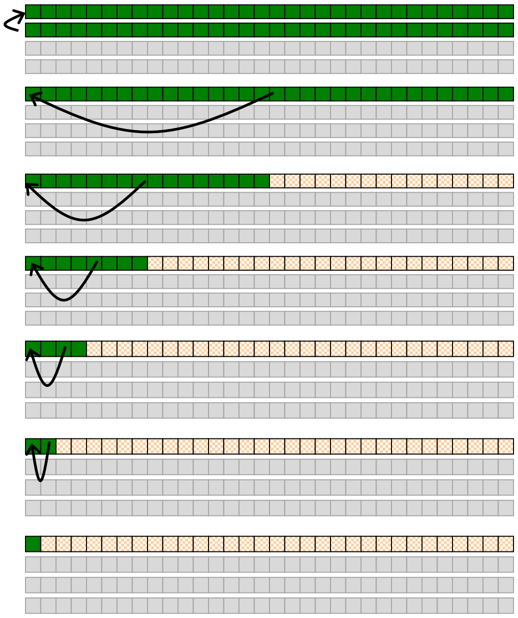
- This saves work in all warps not just the last one
  - Without unrolling all warps execute every iteration of the for loop

Reduction #5: unroll last wrap

one square: data processed by one thread

green: active thread produces data; **yellow:** active thread produces no data

grey: inactive thread



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];

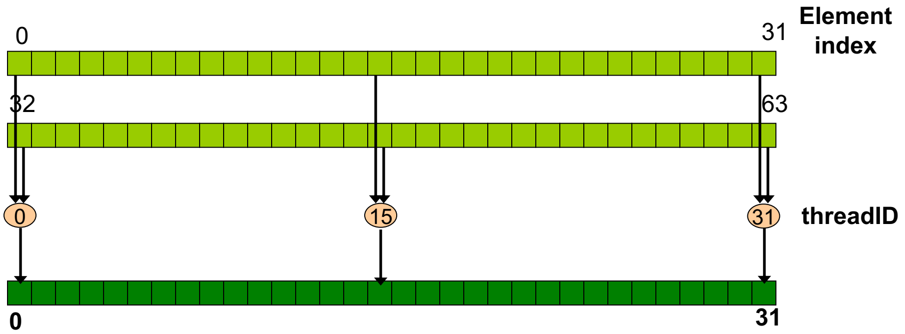
Unrolling the Last Warp: A closer look

Keep in mind:

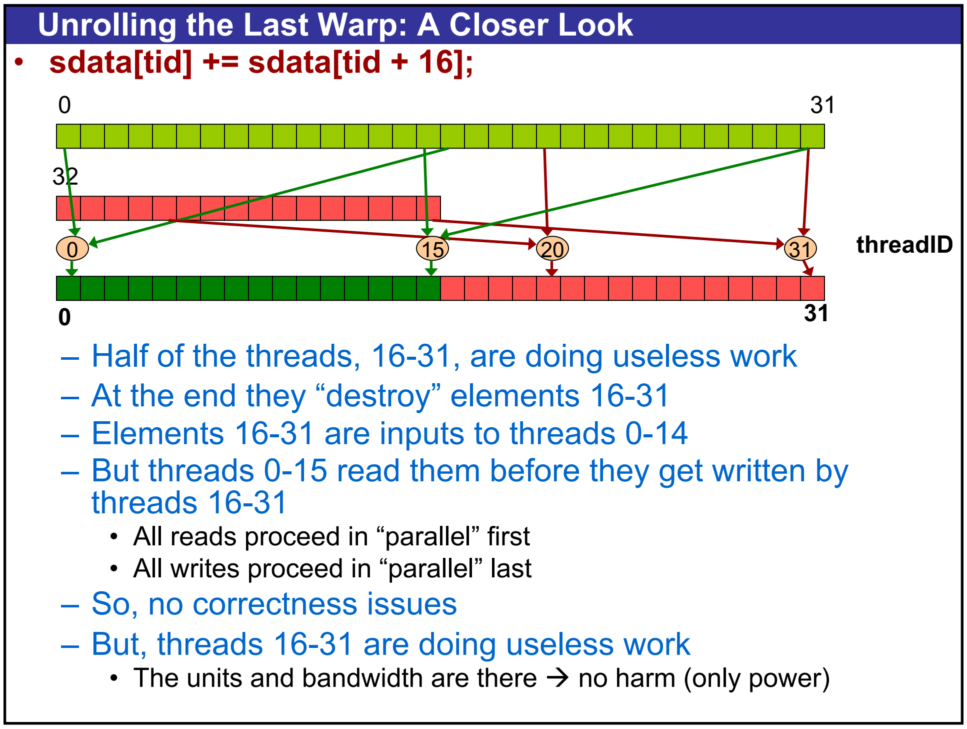
1. Warp execution proceeds in lock step for all threads
   - All threads execute the same instruction
   - So:
     - **sdata[tid] += sdata[tid + 32];**
   - Becomes:
     - Read into a register: sdata[tid]
     - Read into a register: sdate[tid+32]
     - Add the two
     - Write: sdata[tid]
2. Shared memory can provide up to 32 4-byte words per cycle
   - If we don't use this capability it just gets wasted

Unrolling the Last Warp: A Closer Look

- **sdata[tid] += sdata[tid + 32];**



- **All threads doing useful work**

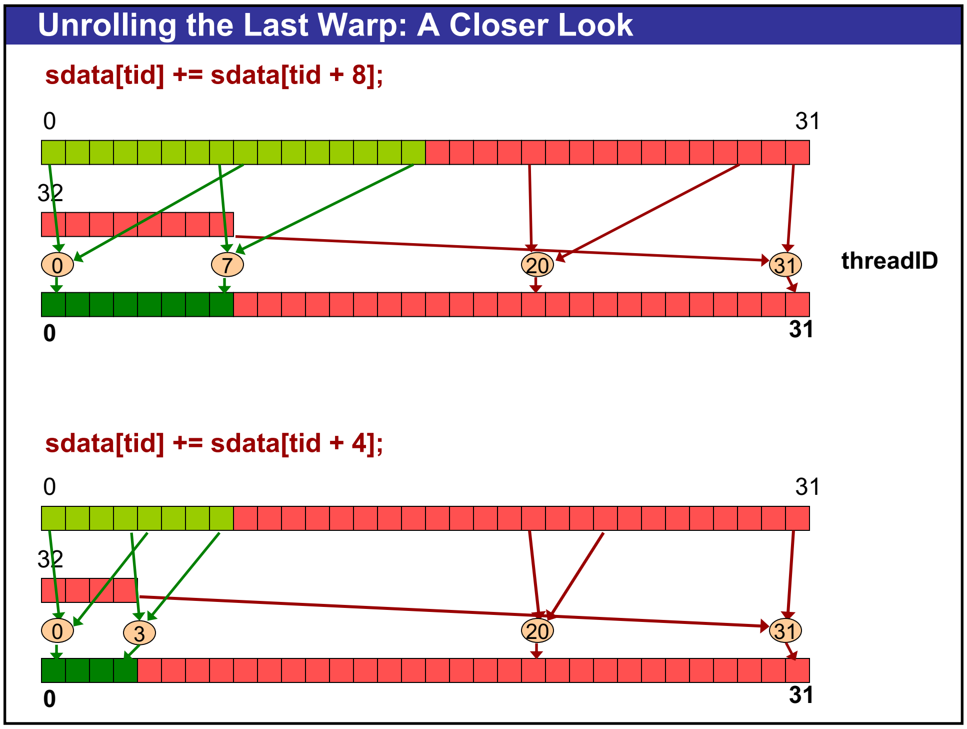


Unrolling the Last Warp: A Closer Look

- **sdata[tid] += sdata[tid + 16];**

- Half of the threads, 16-31, are doing useless work
- At the end they “destroy” elements 16-31
- Elements 16-31 are inputs to threads 0-14
- But threads 0-15 read them before they get written by threads 16-31
  - All reads proceed in “parallel” first
  - All writes proceed in “parallel” last
- So, no correctness issues
- But, threads 16-31 are doing useless work
  - The units and bandwidth are there → no harm (only power)

11

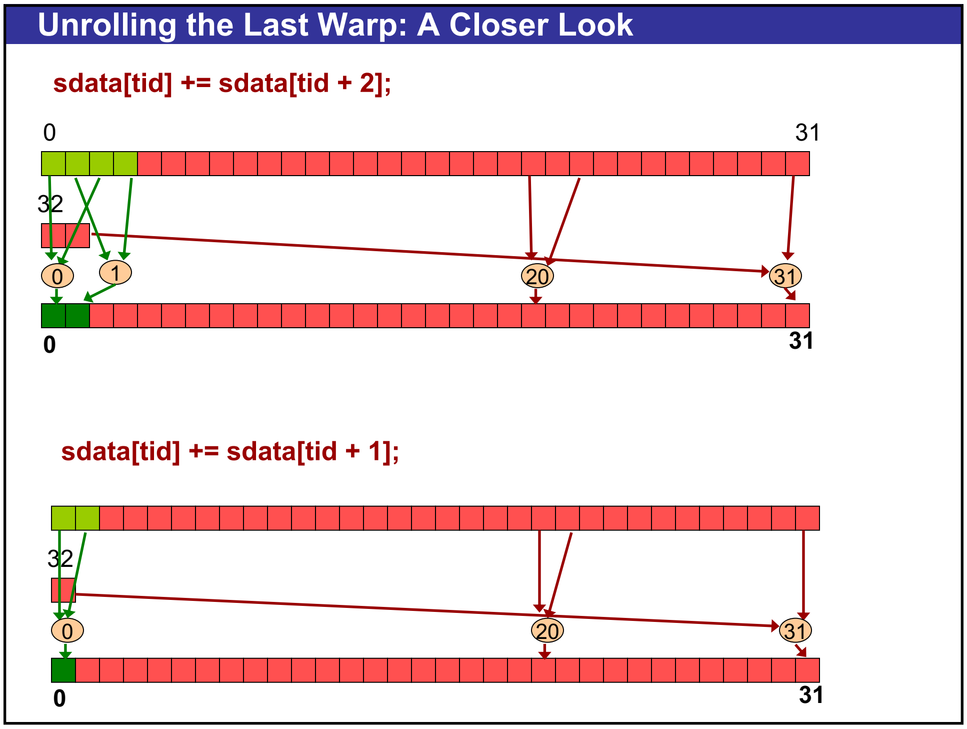


Unrolling the Last Warp: A Closer Look

sdata[tid] += sdata[tid + 8];

sdata[tid] += sdata[tid + 4];

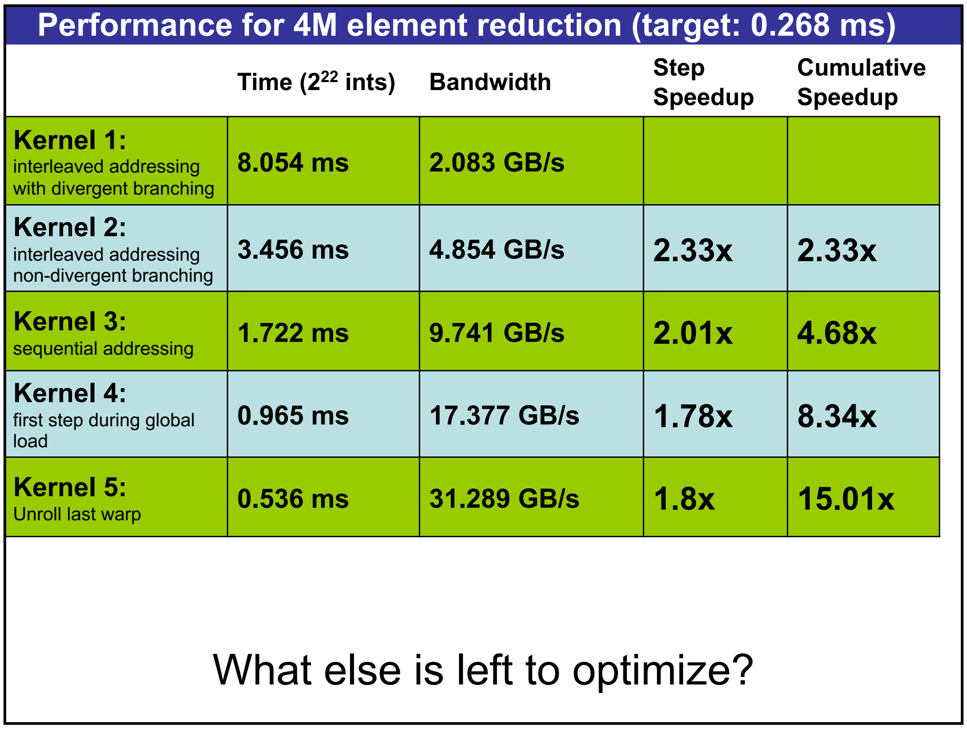
12



Unrolling the Last Warp: A Closer Look

sdata[tid] += sdata[tid + 2];

sdata[tid] += sdata[tid + 1];



Performance for 4M element reduction (target: 0.268 ms)

	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 non-divergent branching	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 step 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

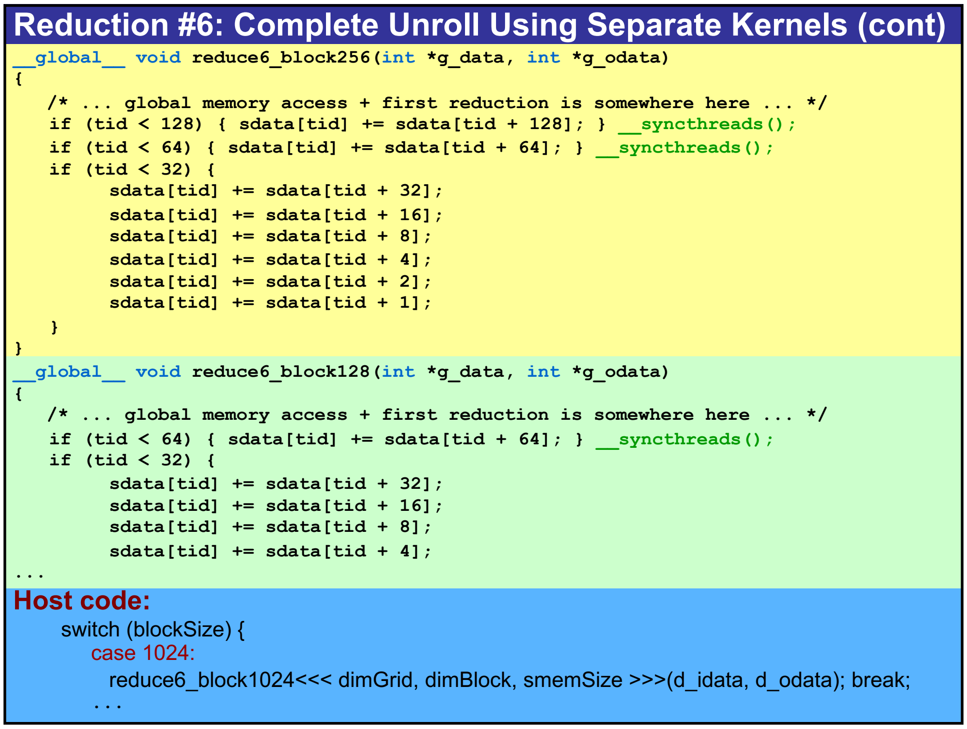
What else is left to optimize?

Reduction #6: Complete Unroll

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Block size is limited to 1024
  - We can restrict our attention to powers-of-2 block sizes
- We can easily unroll for a fixed block size
  - But we need to be generic
  - How can we unroll for block sizes we don’t know at compile time?
- Possible strategies
  - Define a separate kernel for each block size of interest
    - Cumbersome, error prone, verbose/duplicated code
  - Fully unroll kernel, use branching
    - Still high overhead from useless branch instructions
  - Use C++ templates
    - Elegant, compile time evaluation results in very efficient code

Reduction #6: Complete Unroll Using Separate Kernels

```
__global__ void reduce6_block1024(int *g_data, int *g_odata)
{  /* ... global memory access + first reduction is somewhere here ... */
   if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __syncthreads();
   if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
   if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
   if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
   if (tid < 32) {
        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];
   }
}
__global__ void reduce6_block512(int *g_data, int *g_odata)
{
   /* ... global memory access + first reduction is somewhere here ... */
   if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
   if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
   if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
   if (tid < 32) {
        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];
   }
}
```



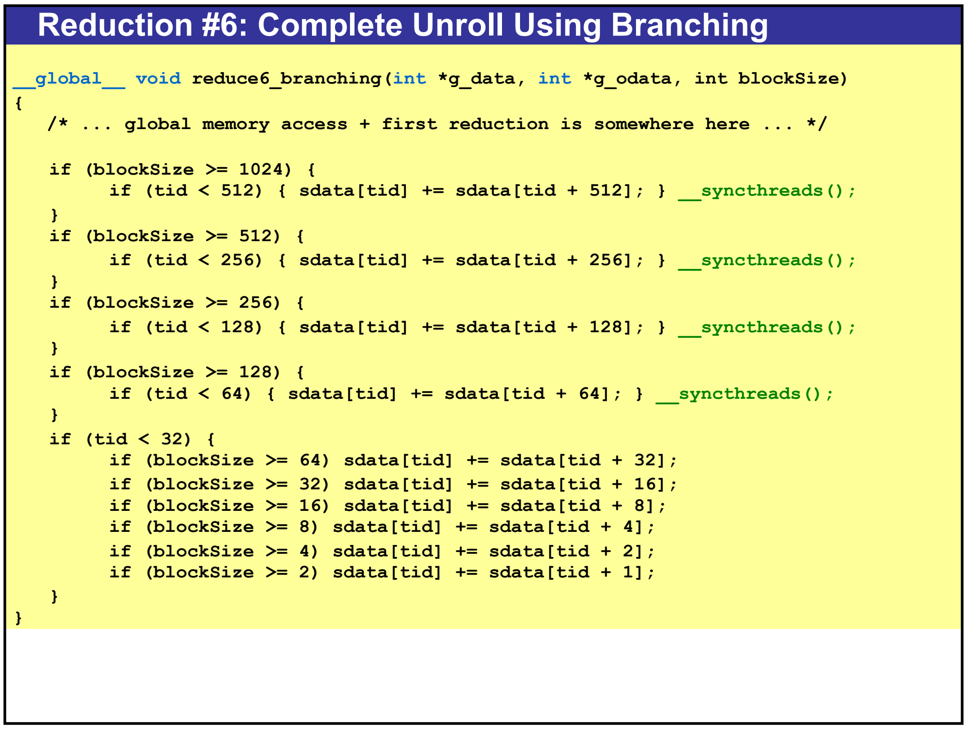
Reduction #6: Complete Unroll Using Separate Kernels (cont)

```
__global__ void reduce6_block256(int *g_data, int *g_odata)
{
   /* ... global memory access + first reduction is somewhere here ... */
   if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
   if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
   if (tid < 32) {
        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];
   }
}
__global__ void reduce6_block128(int *g_data, int *g_odata)
{
   /* ... global memory access + first reduction is somewhere here ... */
   if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
   if (tid < 32) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid + 8];
        sdata[tid] += sdata[tid + 4];
...
```

Host code:

```
    switch (blockSize) {
       case 1024:
         reduce6_block1024<<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
       . . .
```

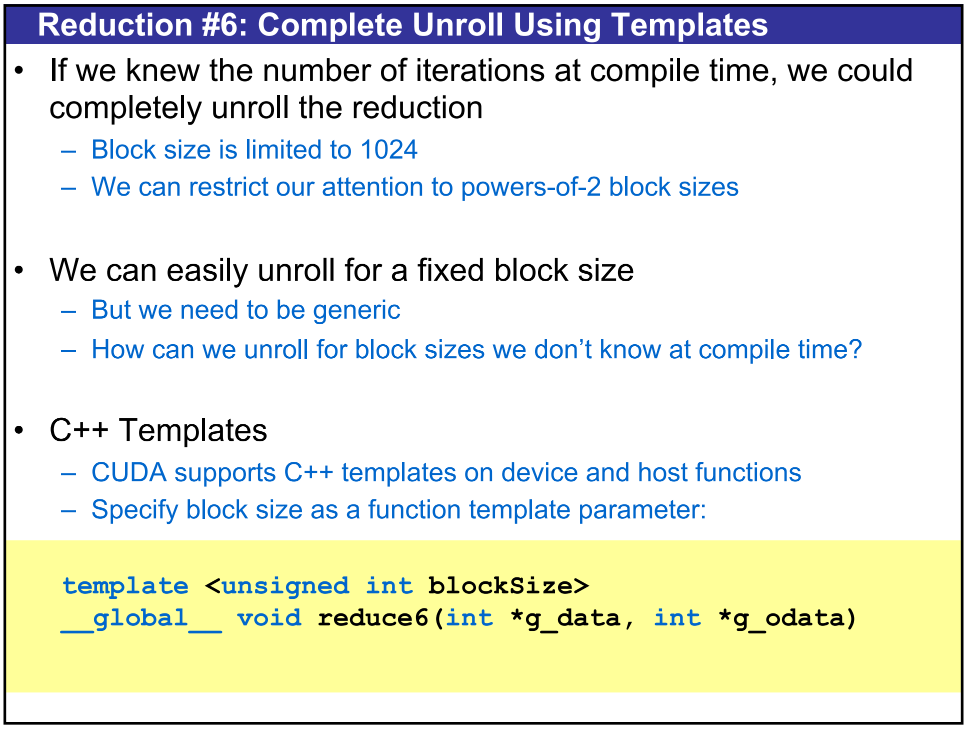
17



Reduction #6: Complete Unroll Using Branching

```
__global__ void reduce6_branching(int *g_data, int *g_odata, int blockSize)
{
   /* ... global memory access + first reduction is somewhere here ... */

   if (blockSize >= 1024) {
        if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __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) {
        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];
   }
}
```

18

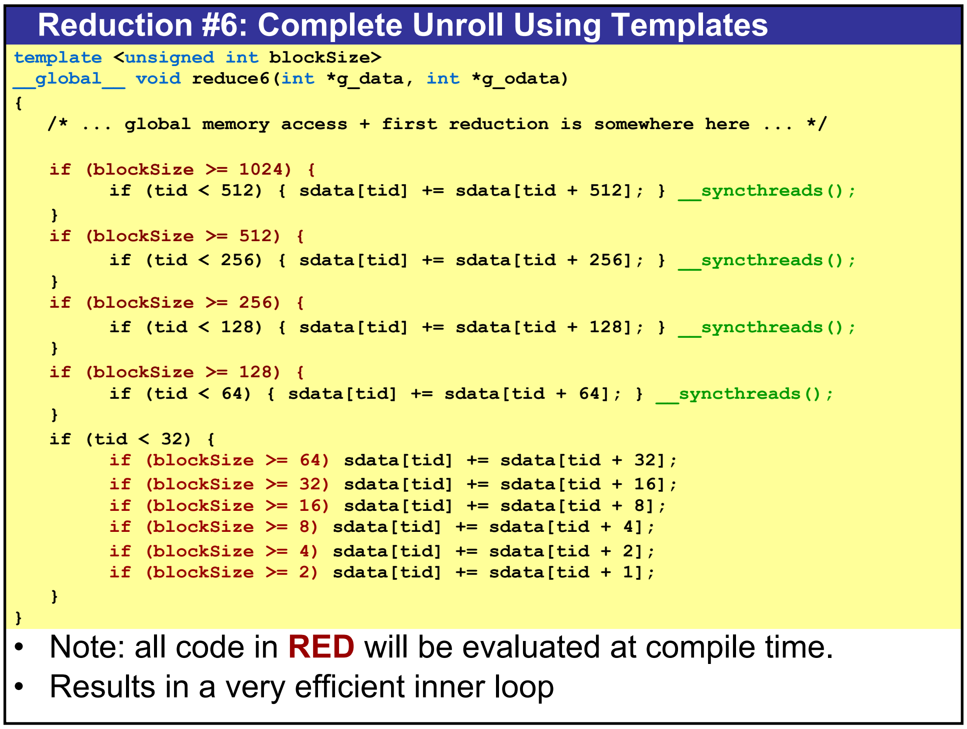


Reduction #6: Complete Unroll Using Templates

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Block size is limited to 1024
  - We can restrict our attention to powers-of-2 block sizes
- We can easily unroll for a fixed block size
  - But we need to be generic
  - How can we unroll for block sizes we don't know at compile time?
- C++ Templates
  - CUDA supports C++ templates on device and host functions
  - Specify block size as a function template parameter:

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

19

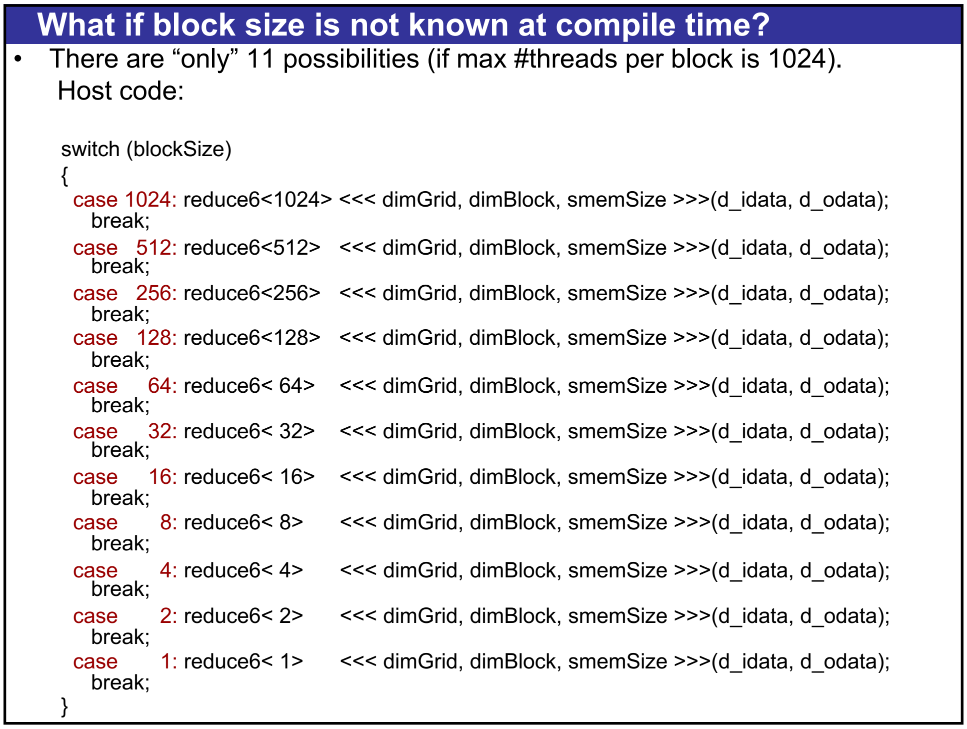


Reduction #6: Complete Unroll Using Templates

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_data, int *g_odata)
{
   /* ... global memory access + first reduction is somewhere here ... */

   if (blockSize >= 1024) {
        if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __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) {
        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];
   }
}
```

- Note: all code in **RED** will be evaluated at compile time.
- Results in a very efficient inner loop

20

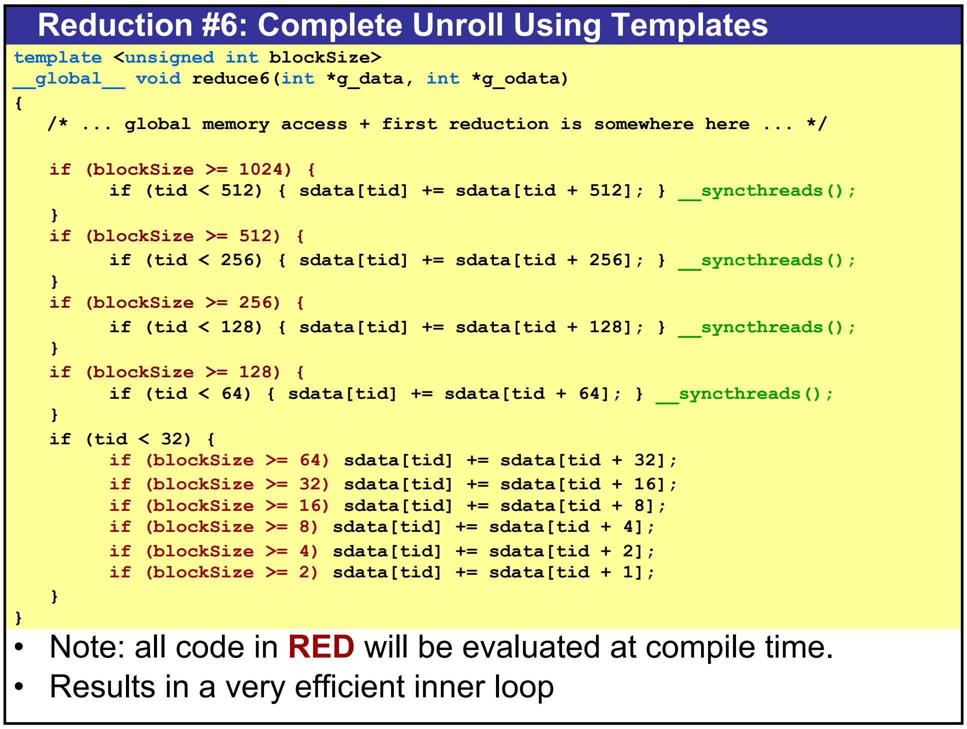


What if block size is not known at compile time?

- There are “only” 11 possibilities (if max #threads per block is 1024).
  Host code:

```
 switch (blockSize)
 {
   case 1024: reduce6<1024> <<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
      break;
   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;
 }
```

21

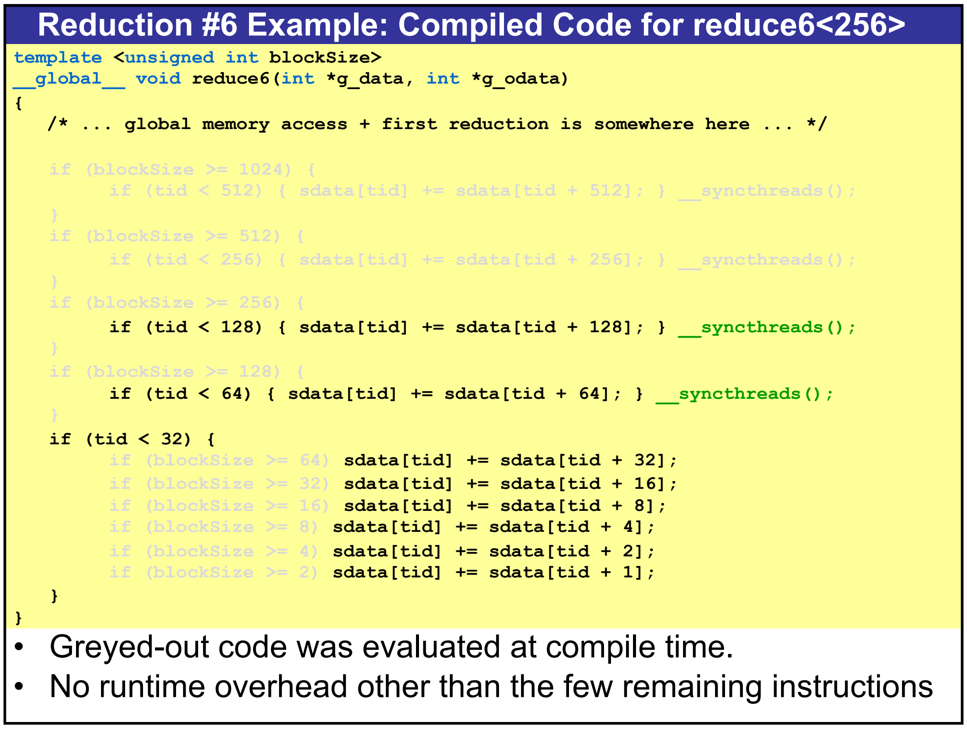


Reduction #6: Complete Unroll Using Templates

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_data, int *g_odata)
{
   /* ... global memory access + first reduction is somewhere here ... */

   if (blockSize >= 1024) {
        if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __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) {
        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];
   }
}
```

- Note: all code in **RED** will be evaluated at compile time.
- Results in a very efficient inner loop

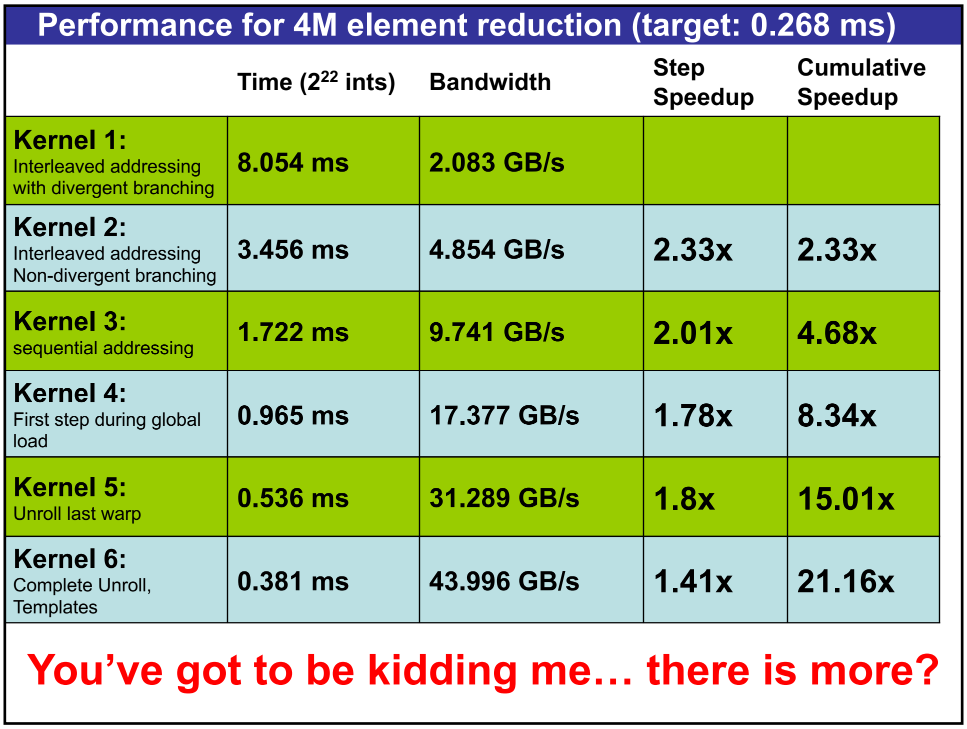
22



Reduction #6 Example: Compiled Code for reduce6<256>

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_data, int *g_odata)
{
   /* ... global memory access + first reduction is somewhere here ... */

   if (blockSize >= 1024) {
        if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __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) {
        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];
   }
}
```

- Greyed-out code was evaluated at compile time.
- No runtime overhead other than the few remaining instructions

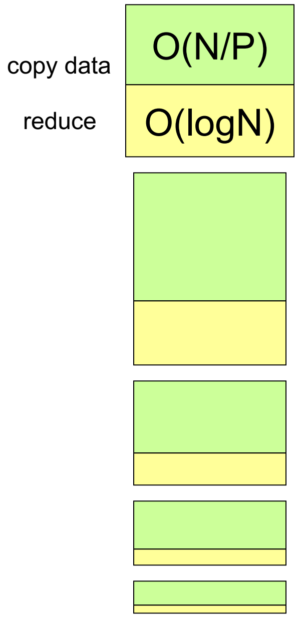


Performance for 4M element reduction (target: 0.268 ms)

	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 Non-divergent branching	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 step 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: Complete Unroll, Templates	0.381 ms	43.996 GB/s	1.41x	21.16x

You’ve got to be kidding me… there is more?

What on earth is left for us to improve?



N: number of data elements
P: number of processors

If we can choose P → can we make this O(logN)

I’ll tell you what is left for us to improve…

- Two parts:
  - Loading elements + pair-wise reduction
    - **O(*N*/*P*)**
  - Tree-like reduction
    - **O(log*N*)**

- Overall: O(N/P + logN)

- Can we make the O(N/P) part become O(logN)?
- What should be P?
  - N/logN

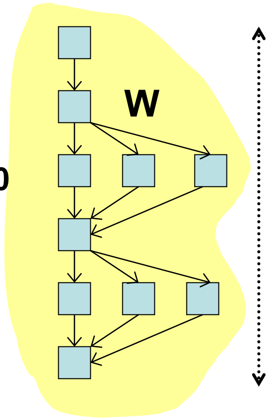
- So, first step should be:
  - load N/logN elements
  - do the first N/logN reductions

Brent's theorem

$$t_p \leq t_{inf} + (W - t_{inf}) / p$$

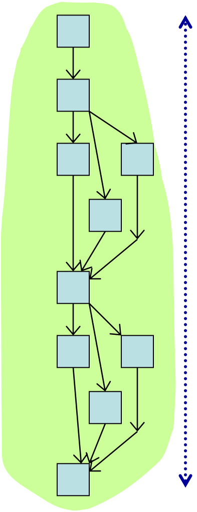
$t_p \leq 6 + (10 - 6) / 2 = 8$

W = 10
p >= 3



$t_{inf} = 6$

W = 10
p = 2



$t_p = 8$

If **given infinite processors** an algorithm takes t_{inf} **steps** then **given p processors** it should take at most t_p **steps**

27

Can we improve the algorithm?

- **Work** or "**element complexity**": `W`
  - total number of operations performed collectively by all processors
  - Time to execute on a single processor
- **Time Step:** All operations with no unresolved dependencies are performed in parallel
- **Depth** or "**step complexity**": t_{inf}
  - time to complete all time steps
  - time to execute if we had infinite processors
- **Computation time on P Processors:** t_p
  - time to execute when there are P processors

28

Brent’s Theorem

$$t_p \le t_{inf} + (W - t_{inf}) / p$$

- p =1, t_1 = W (seq. algorithm)
- P → inf, t → t_{inf} (limit)
- if sequential, t = W

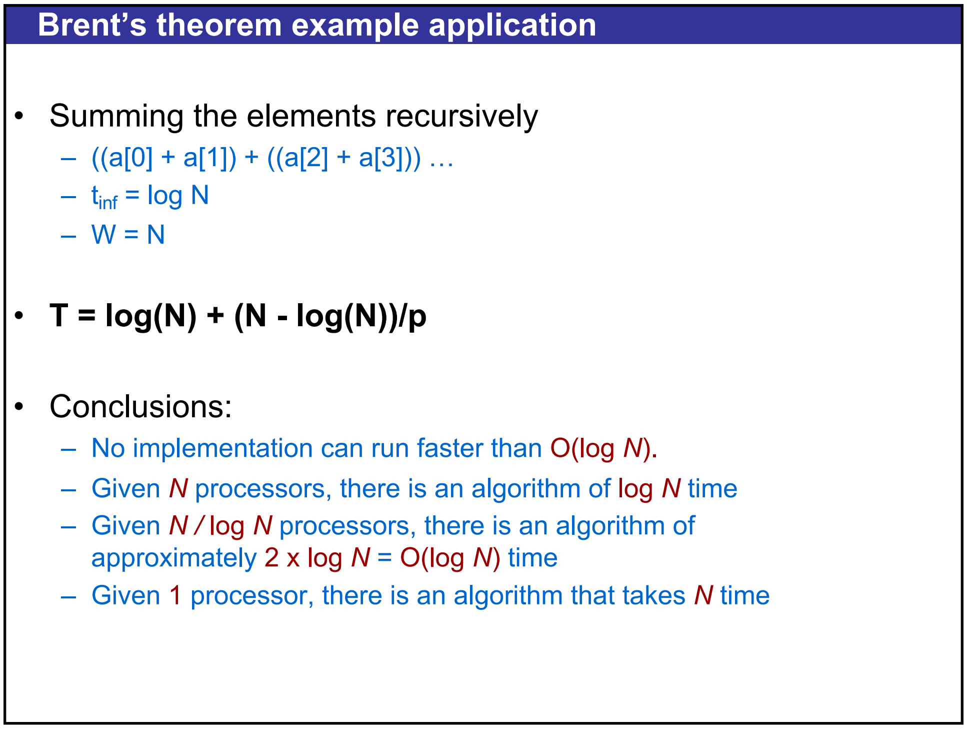
- **Work** or **element complexity**: **W**
- **Depth** or **step complexity**: t_{inf}
- **Computation time on P Processors:** t_p

What is Brent’s theorem useful for?

- Tells us whether an algorithm can be improved for sure
- Example:

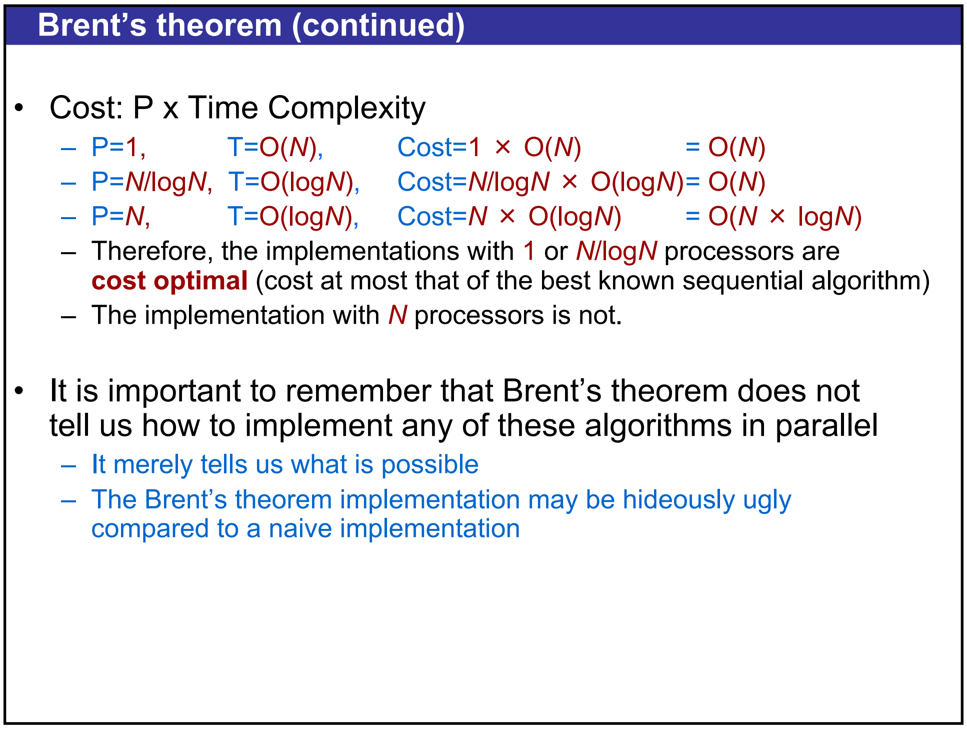
$$t_p \le t_{inf} + (W - t_{inf}) / p$$

  – summing the elements of an array sequentially
    - for (i=0; i < N; i++)
      sum = sum + a[i];
    - W = N, t_{inf} = N
    - t_p = N + (N – N) / p = N
    - No matter what, this will take N time



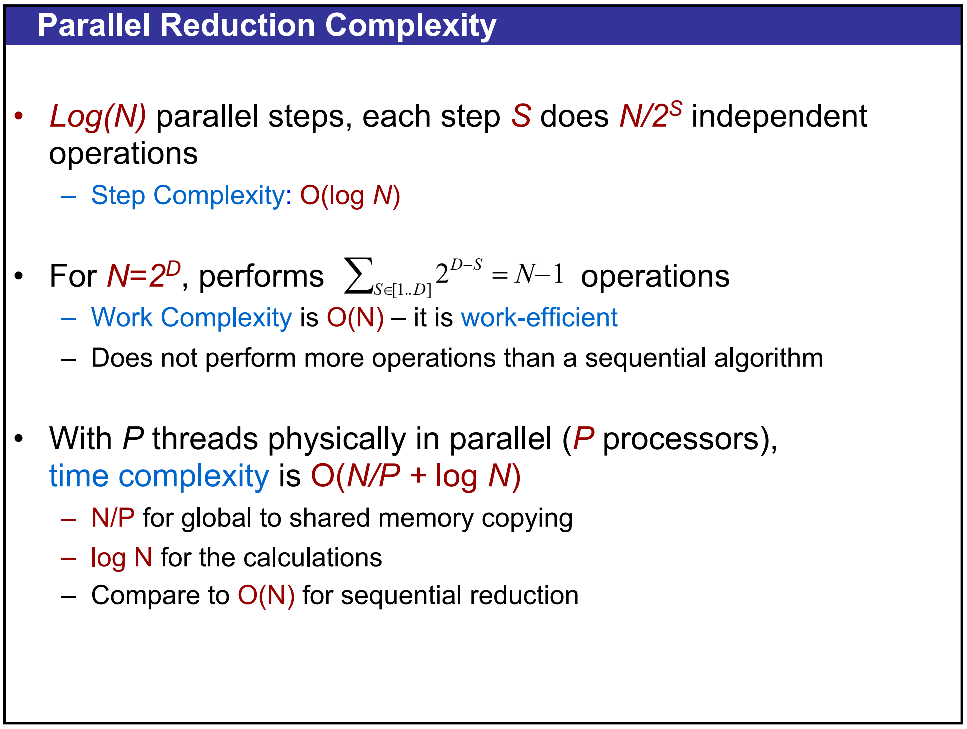
Brent’s theorem example application

- Summing the elements recursively
  - ((a[0] + a[1]) + ((a[2] + a[3])) …
  - t_{inf} = log N
  - W = N
- **T = log(N) + (N - log(N))/p**
- Conclusions:
  - No implementation can run faster than O(log *N*).
  - Given *N* processors, there is an algorithm of log *N* time
  - Given *N* / log *N* processors, there is an algorithm of approximately 2 x log *N* = O(log *N*) time
  - Given 1 processor, there is an algorithm that takes *N* time



Brent’s theorem (continued)

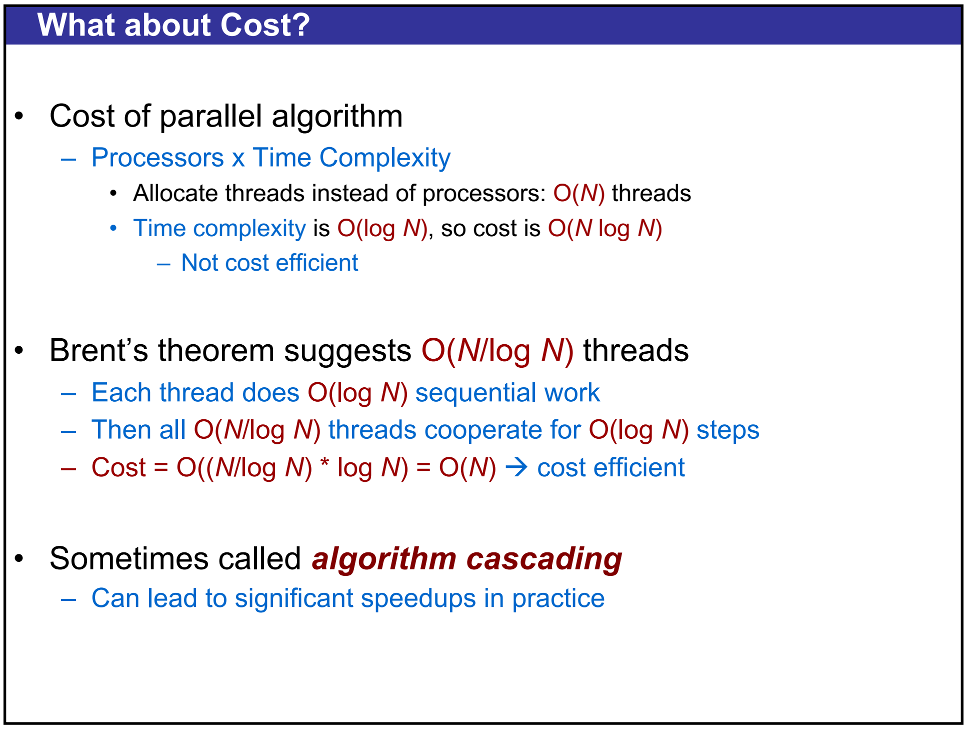
- Cost: P x Time Complexity
  - P=1, T=O(*N*), Cost=1 × O(*N*) = O(*N*)
  - P=*N*/log*N*, T=O(log*N*), Cost=*N*/log*N* × O(log*N*)= O(*N*)
  - P=*N*, T=O(log*N*), Cost=*N* × O(log*N*) = O(*N* × log*N*)
  - Therefore, the implementations with 1 or *N*/log*N* processors are **cost optimal** (cost at most that of the best known sequential algorithm)
  - The implementation with *N* processors is not.
- It is important to remember that Brent’s theorem does not tell us how to implement any of these algorithms in parallel
  - It merely tells us what is possible
  - The Brent’s theorem implementation may be hideously ugly compared to a naive implementation



Parallel Reduction Complexity

- *Log(N)* parallel steps, each step *S* does *N/2^S* independent operations
  - Step Complexity: O(log *N*)
- For *N=2^D*, performs $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is O(N) – it is work-efficient
  - Does not perform more operations than a sequential algorithm
- With *P* threads physically in parallel (*P* processors), time complexity is O(*N/P* + log *N*)
  - N/P for global to shared memory copying
  - log N for the calculations
  - Compare to O(N) for sequential reduction

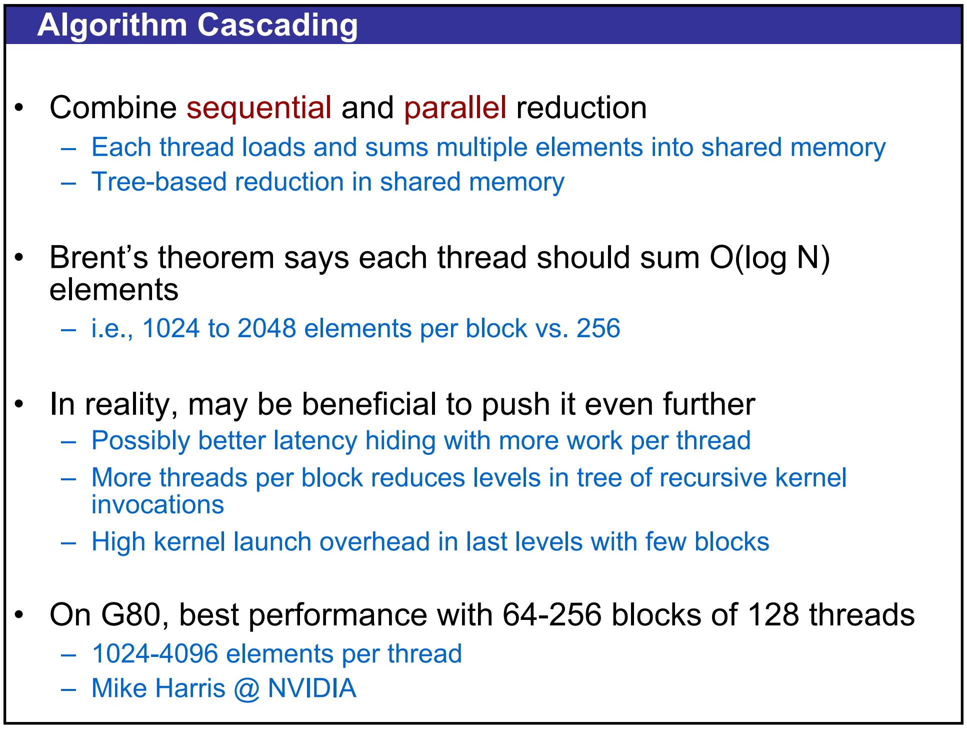
33



What about Cost?

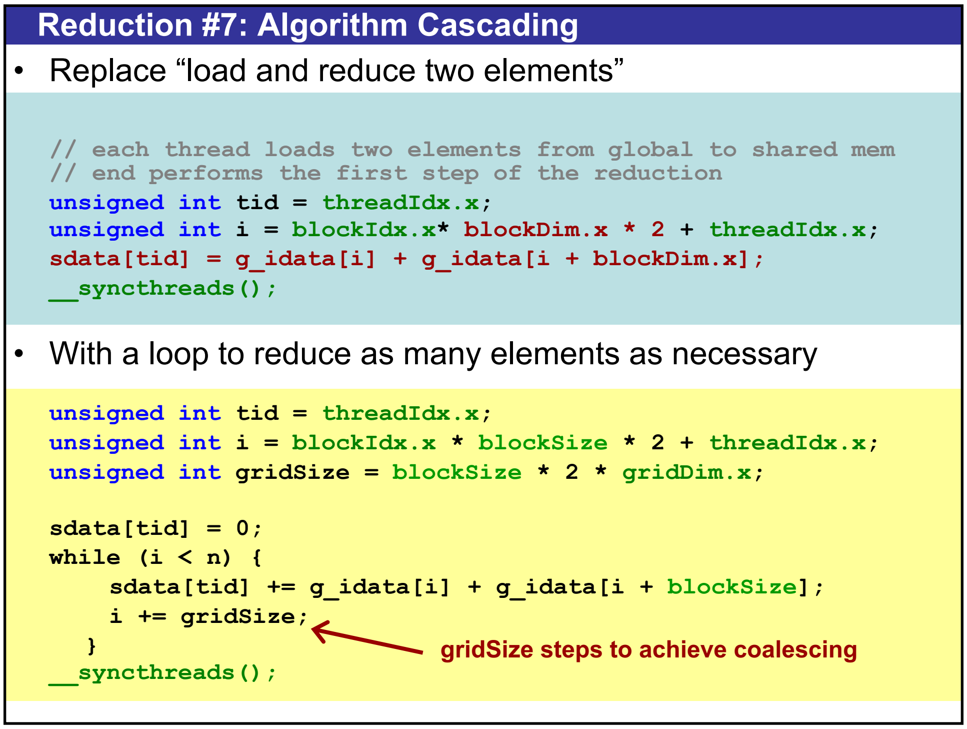
- Cost of parallel algorithm
  - Processors x Time Complexity
    - Allocate threads instead of processors: O(*N*) threads
    - Time complexity is O(log *N*), so cost is O(*N* log *N*)
      - Not cost efficient
- Brent's theorem suggests O(*N*/log *N*) threads
  - Each thread does O(log *N*) sequential work
  - Then all O(*N*/log *N*) threads cooperate for O(log *N*) steps
  - Cost = O((*N*/log *N*) * log *N*) = O(*N*) → cost efficient
- Sometimes called ***algorithm cascading***
  - Can lead to significant speedups in practice

34



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 to 2048 elements per block vs. 256
- In reality, may be 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
  - Mike Harris @ NVIDIA



Reduction #7: Algorithm Cascading

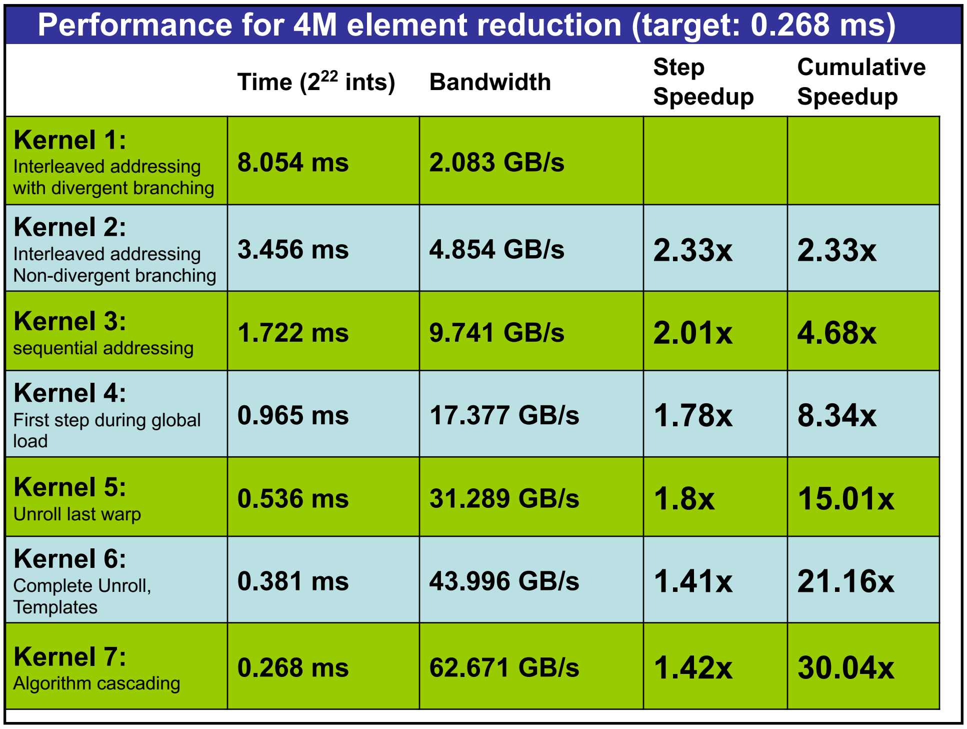
- Replace “load and reduce two elements”

```
// each thread loads two elements from global to shared mem
// end performs the first step of the reduction
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 loop to reduce as many elements 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();
```

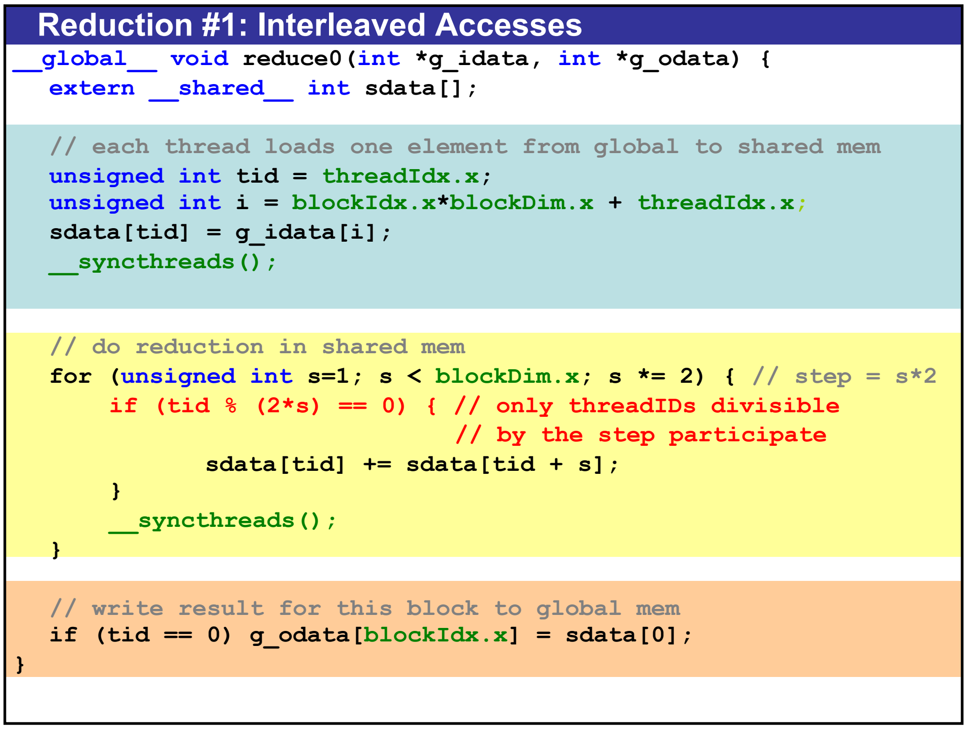
gridSize steps to achieve coalescing



Performance for 4M element reduction (target: 0.268 ms)

	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 Non-divergent branching	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 step 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: Complete Unroll, Templates	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: Algorithm cascading	0.268 ms	62.671 GB/s	1.42x	30.04x

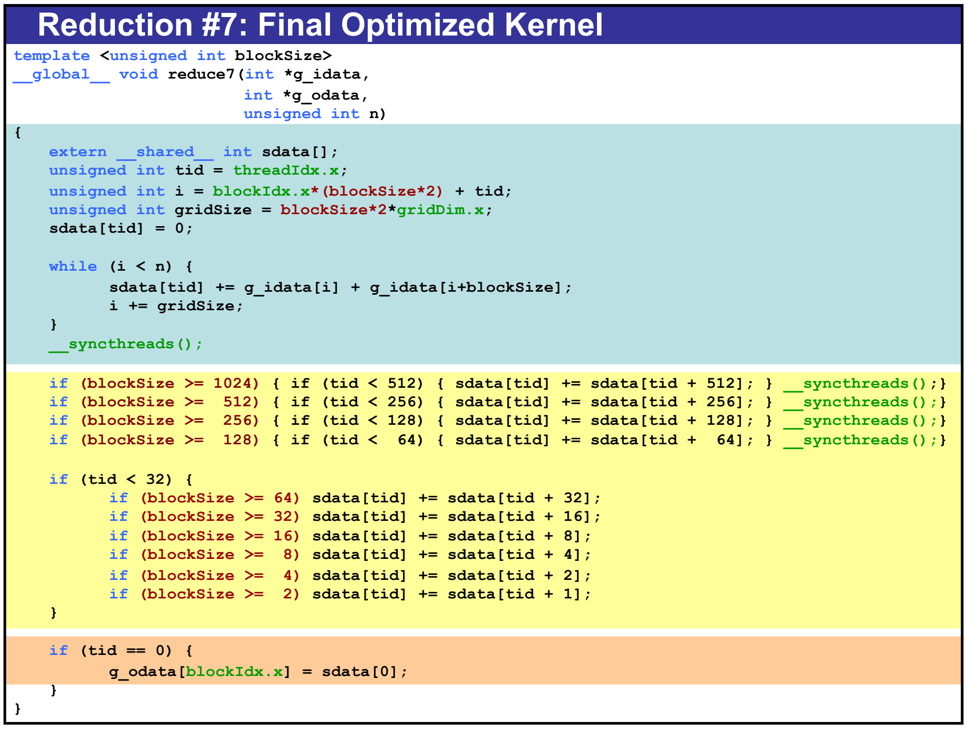
37



Reduction #1: Interleaved Accesses

```
__global__ void reduce0(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) { // step = s*2
        if (tid % (2*s) == 0) { // only threadIDs divisible
                                // by the step participate
              sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
   }

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

38

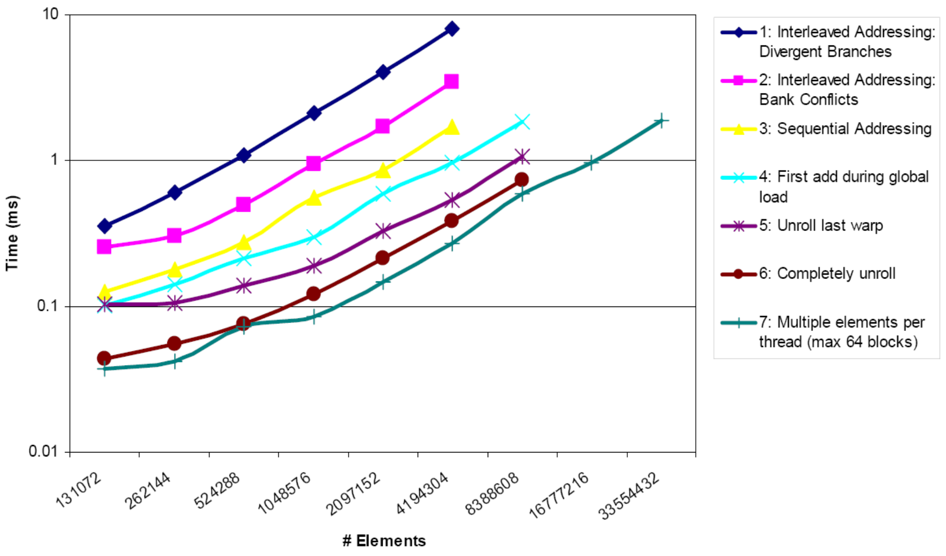


Reduction #7: Final Optimized Kernel

```
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 >= 1024) { if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __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) {
          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];
    }

    if (tid == 0) {
          g_odata[blockIdx.x] = sdata[0];
    }
}
```

39

Performance comparison on G80



40

Optimization types

- Algorithmic optimizations
  - Changes to addressing, algorithm cascading
  - 11.84x speedup
- Code optimizations:
  - Loop unrolling
  - 2.54x speedup over algorithmic optimizations

41

Summary

- Understand CUDA performance characteristics
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding
- Use peak performance metrics to guide optimization
- Understand parallel algorithm complexity theory
- 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

42