CUDA Optimization with Parallel Reduction

Jeong-Gun Lee

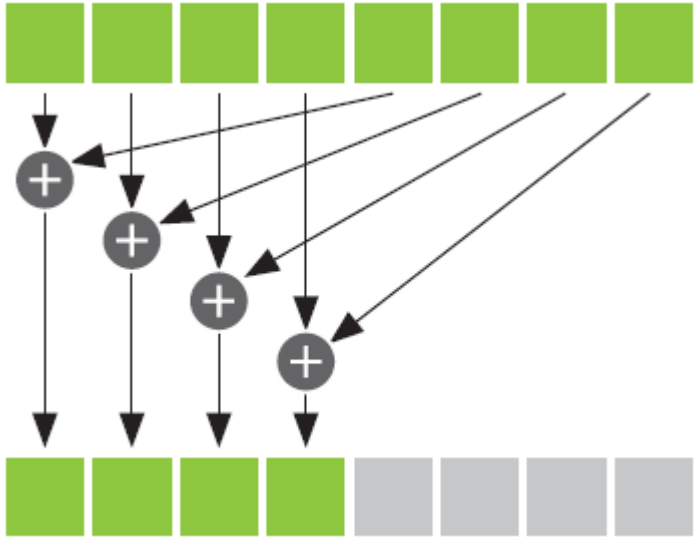
Dept. of Computer Engineering, Hallym Univ

Email: Jeonggun.Lee@gmail.com

KIRIA 한국로봇산업진흥원
KOREA INSTITUTE FOR ROBOT INDUSTRY ADVANCEMENT

CUDA Optimization

- I will use "**Optimizing Parallel Reduction in CUDA**" written by Mark Harris for demonstrating how to optimize your CUDA applications

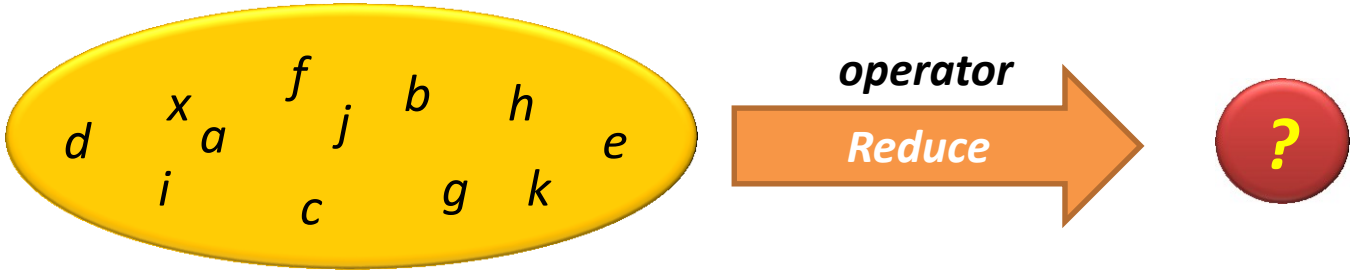


Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - We'll walk ***step by step*** through 7 different versions
  - Demonstrates several important optimization strategies

Parallel Reduction ?

- ***Reduction***: An operation that computes a single result from a set of data

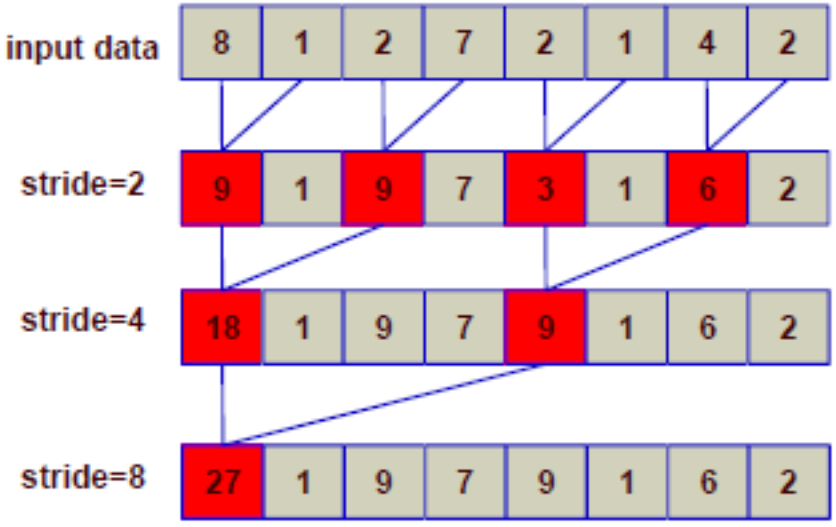


- Examples:
  - Minimum/maximum value
  - Average, sum, product, etc.

- Parallel Reduction: Do it in ***parallel*** obviously

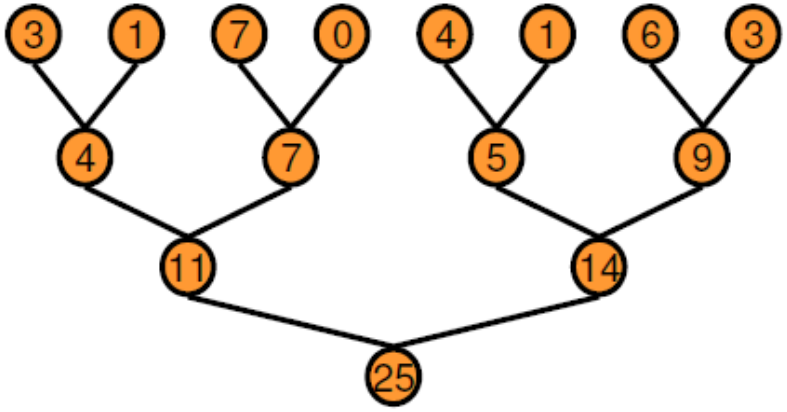
Parallel Reduction ??

- Calculating "**x1 + x2 + x3 + … + xn**"
- What is the ***best parallel implementation*** for this computation
  - What is the time complexity for the best solution
  - With how many parallel cores ?
- Reduction with any binary operators such as sum, min, max.



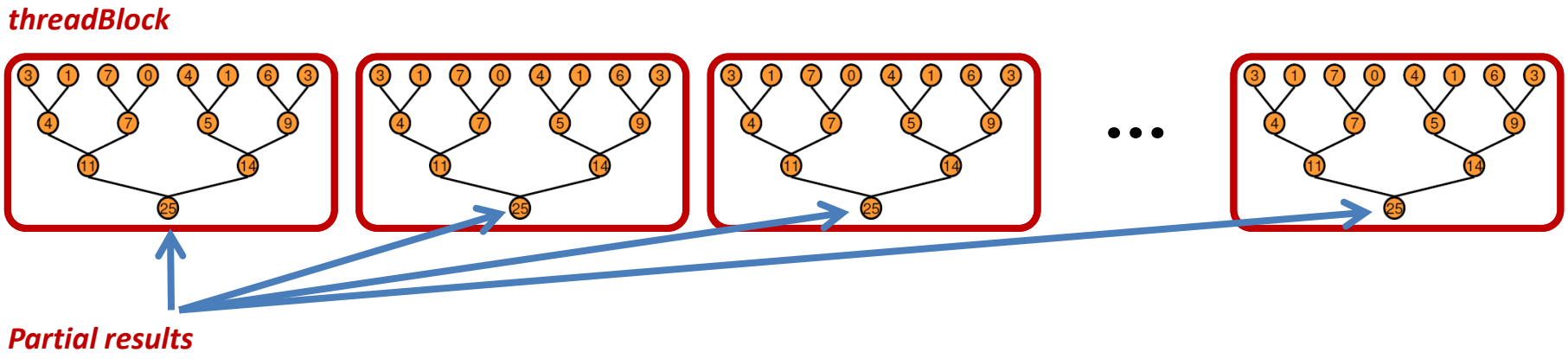
Parallel Reduction

- 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
- But how do we ***communicate partial results*** between thread blocks?

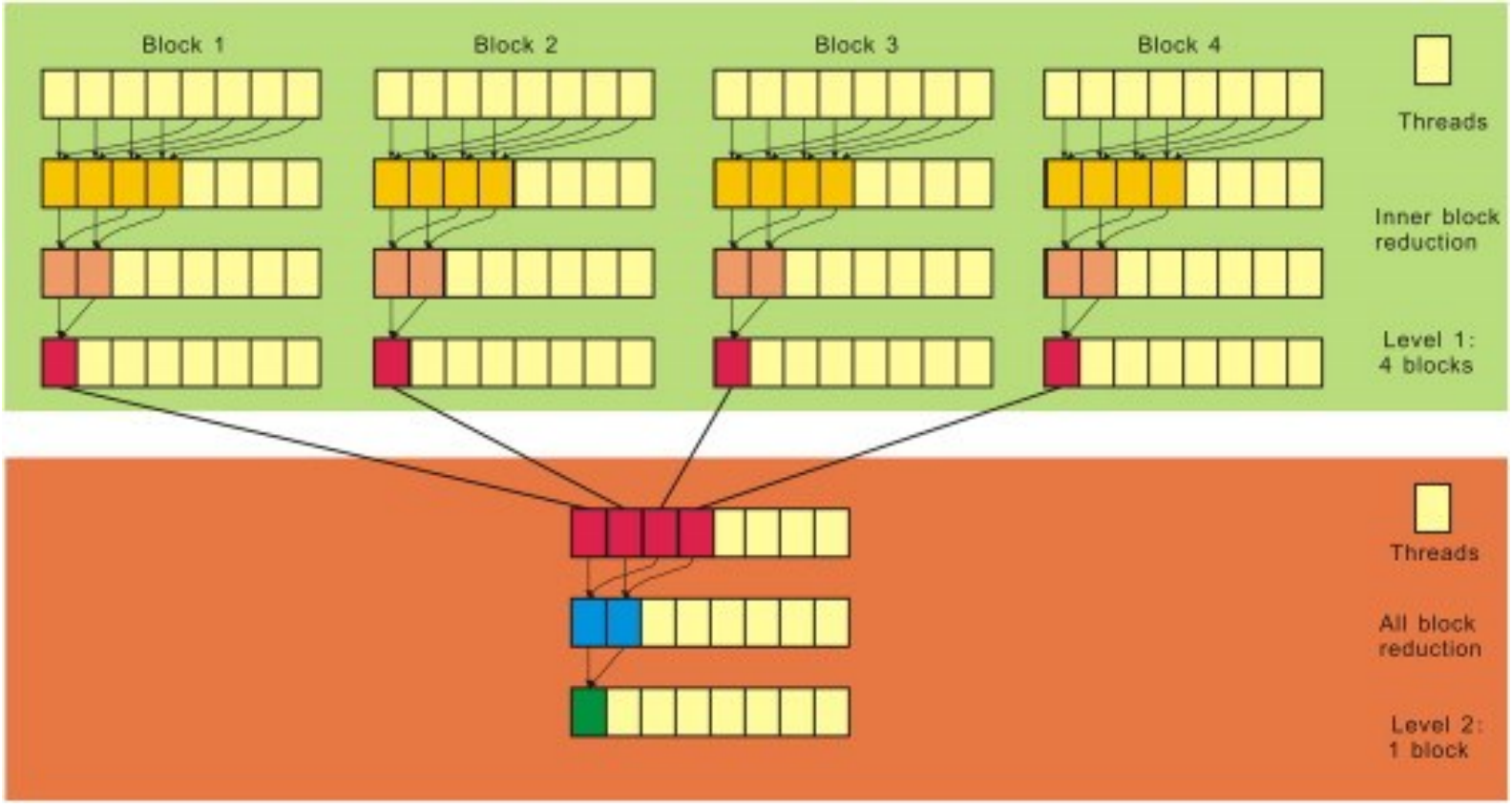
Parallel Reduction



No intra block level synchronization support !

What is inter block synchronization support ?

Parallel Reduction

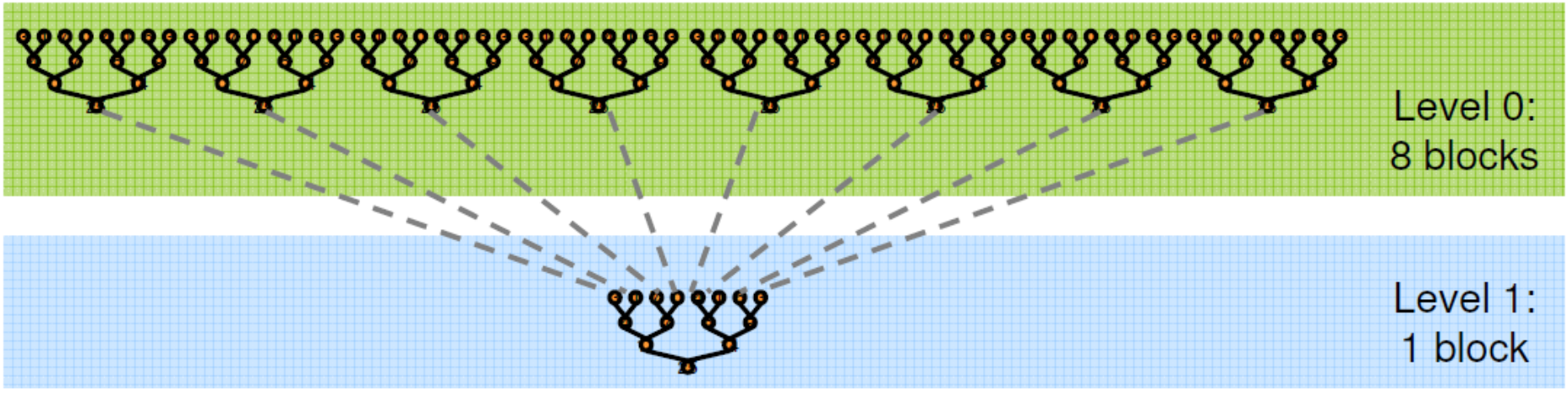


Problem: Global Synchronization

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

Solution: Kernel Decomposition

- Avoid global sync by ***decomposing computation into multiple kernel invocations***



- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

Synchronization between kernel calls -- *cudaDeviceSynchronize()*

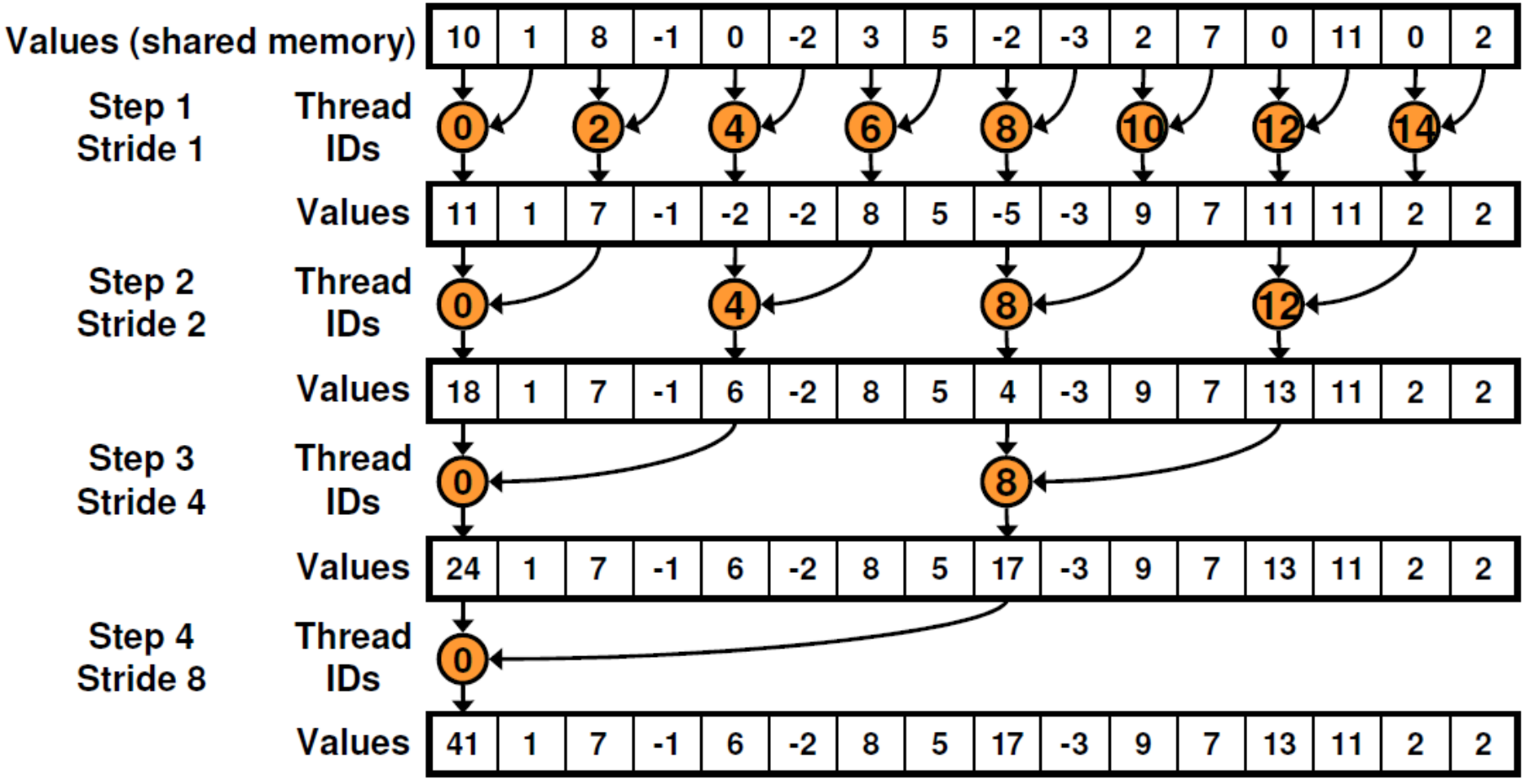
What is Our Optimization Goal?

- 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 floating-point op per element loaded (bandwidth-optimal)
- Therefore we ***should strive for peak bandwidth***
- Will use G80 GPU for this example
  - 384-bit memory interface, 900 MHz DDR (Double Data Rate)
  - 384 * (1800) / 8 = **86.4 GB/s**

Reduction #1: Interleaved Addressiing

```
__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) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Parallel Reduction: Interleaved Addressing

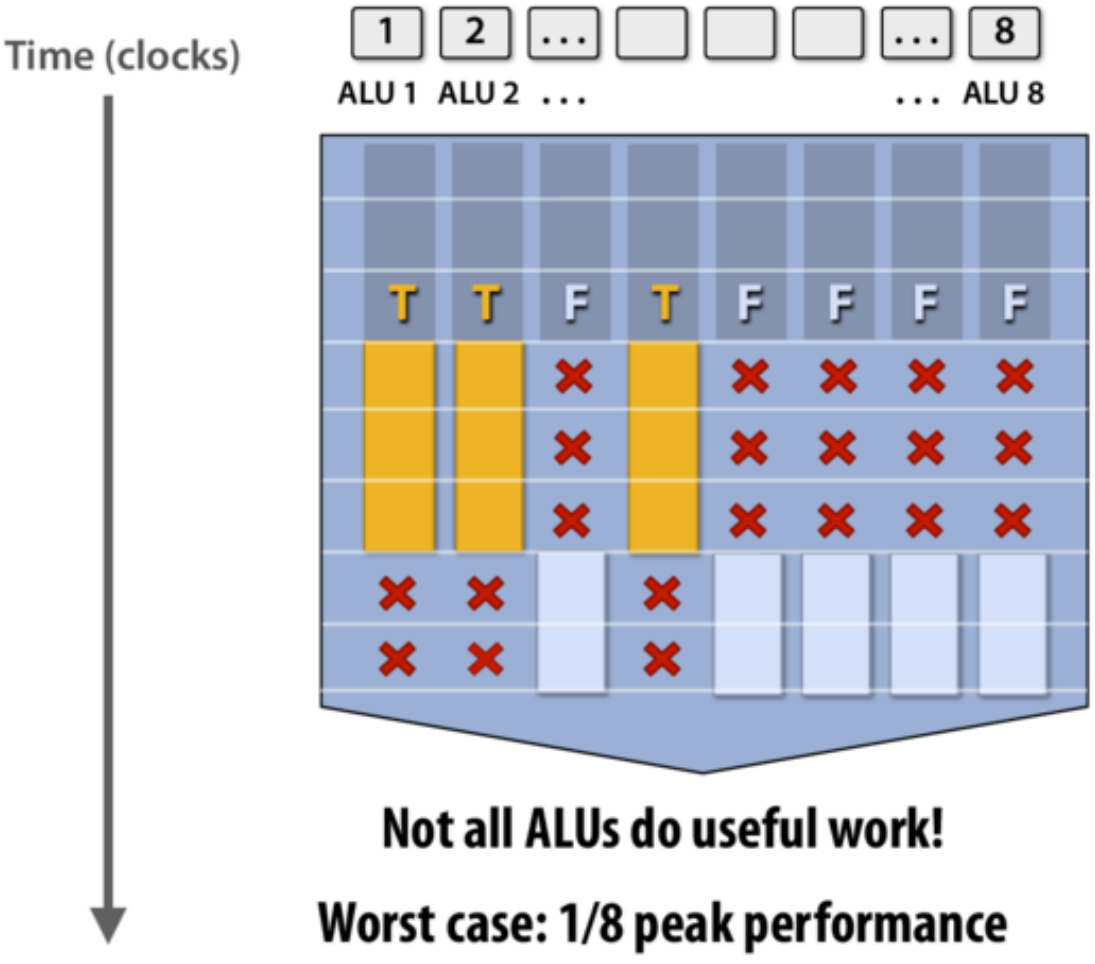


Reduction #1: Interleaved Addressing

```
__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) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

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

Warp Divergence



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

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		

Note: Block Size = 128 threads for all tests

Integers	BYTES	GB	Time (ms)	Time	1/Time	GB BW
4194304	16777216	0.015625	8.054	0.008054	124.1619	1.94003
		0.016777	8.054	0.008054	124.1619	2.083091

-Calculating achieved bandwidth and flops/Gflops, and evaluate CUDA kernel performance
http://stackoverflow.com/questions/12539300/calculating-achieved-bandwidth-and-flops-gflops-and-evaluate-cuda-kernel-perfor

Test on Jetson TK1

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ nvprof --print-gpu-trace ./a.out
CPU results = 4718580
==27381== NVPROF is profiling process 27381, command: ./a.out
The size of array is 1048576 and it is processedon # of Blocks: 2048
The size of array is 2048 and it is processedon # of Blocks: 4
GPU result = 4718580
==27381== Profiling application: ./a.out
==27381== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
178.39ms  4.4827ms                    -               -         -         -         -  4.1943MB  935.66MB/s        GK20A (0)         1         7  [CUDA memcpy HtoD]
183.16ms  44.295ms           (2048 1 1)       (512 1 1)         8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce0(int*, int*) [96]
228.78ms  24.668us              (4 1 1)       (512 1 1)         8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce0(int*, int*) [101]
229.62ms  5.1680us               (1 1 1)         (4 1 1)         8        0B       16B         -           -        GK20A (0)         1         7  reduce0(int*, int*) [106]
229.96ms  2.2500us                    -               -         -         -         -        4B  1.7778MB/s        GK20A (0)         1         7  [CUDA memcpy DtoH]
```

ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ **nvprof --help**
Usage: nvprof [options] [CUDA-application] [application-arguments]
Options:

-o, --output-profile <file name>
Output the result file which can be imported later or opened by the NVIDIA Visual Profiler.
…

-i, --import-profile <file name>
Import a result profile from a previous run.

-s, --print-summary Print a summary of the profiling result on screen.
…
…

--print-gpu-trace Print individual kernel invocations (including CUDA memcpy's/memset's) and sort them in chronological order. In event/metric profiling mode, show events/metrics for each kernel invocation.

nvprof --print-summary-per-gpu ./your_application

Reduction #2: Interleaved Addressing

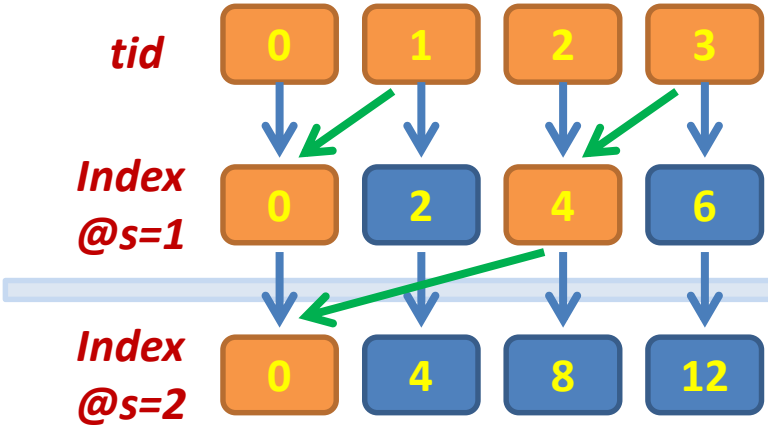
- Just replace divergent branch in inner loop:

```
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:

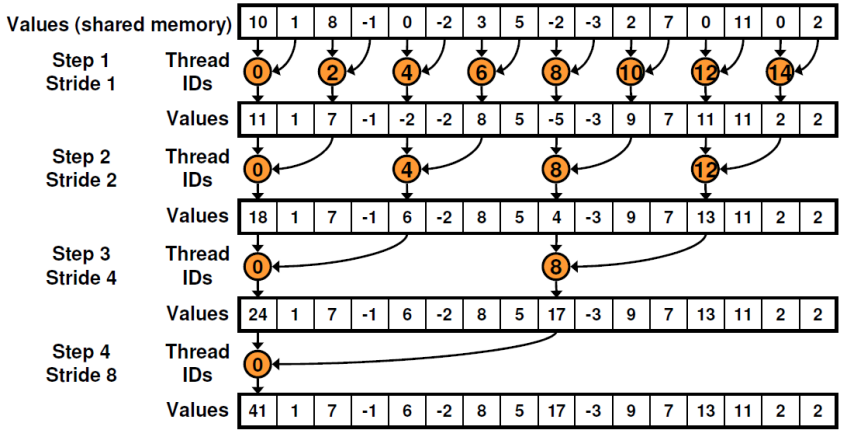
```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

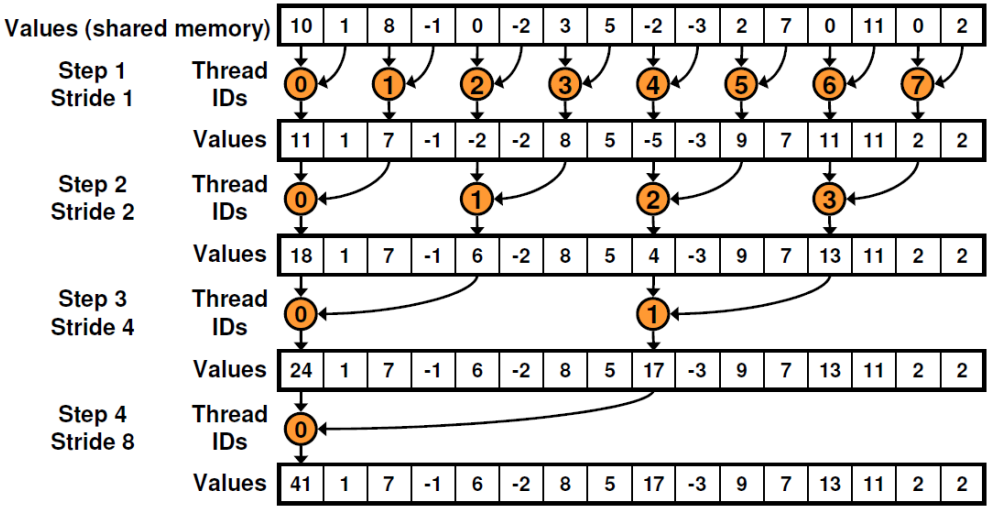
Reduction #2: Interleaved Addressing




```
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();
}
```

Parallel Reduction: Interleaved Addressing





New Problem: *Shared Memory Bank Conflicts*

Performance for 4M ellement reductiion

	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

Test on Jetson TK1

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ nvprof --print-gpu-trace ./a.out
CPU results = 4718580
==27473== NVPROF is profiling process 27473, command: ./a.out
The size of array is 1048576 and it is processedon # of Blocks: 2048
The size of array is 2048 and it is processedon # of Blocks: 4
GPU result = 4718580
==27473== Profiling application: ./a.out
==27473== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
186.37ms  4.4846ms                    -               -        -         -         -  4.1943MB  935.28MB/s        GK20A (0)         1         7  [CUDA memcpy HtoD]
191.14ms  34.433ms           (2048 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce1(int*, int*) [96]
226.74ms  16.584us              (4 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce1(int*, int*) [101]
227.44ms  4.6670us               (1 1 1)         (4 1 1)        8        0B       16B         -           -        GK20A (0)         1         7  reduce1(int*, int*) [106]
227.71ms  1.9160us                    -               -        -         -         -        4B  2.0877MB/s        GK20A (0)         1         7  [CUDA memcpy DtoH]
```

ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ **nvprof --help**
Usage: nvprof [options] [CUDA-application] [application-arguments]
Options:
-o, --output-profile <file name>
Output the result file which can be imported later or opened by the NVIDIA Visual Profiler.
…
-i, --import-profile <file name>
Import a result profile from a previous run.

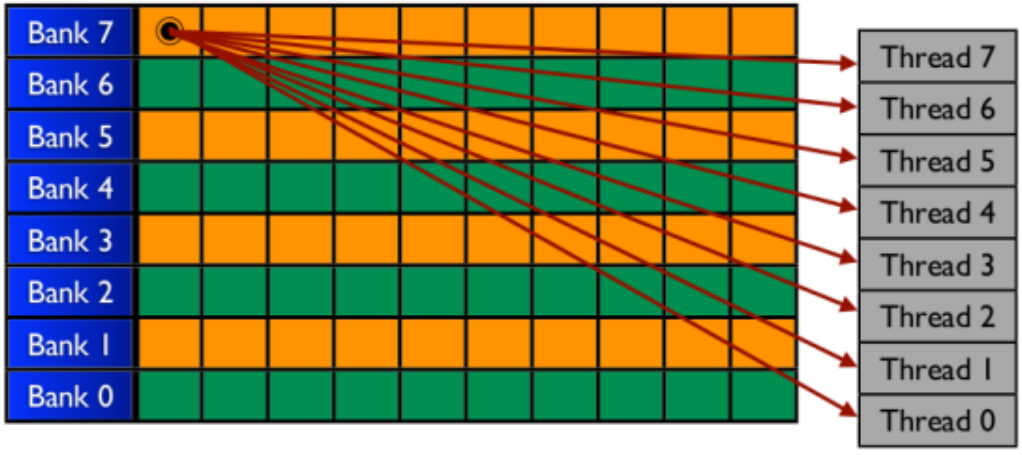
-s, --print-summary Print a summary of the profiling result on screen.
…
…
--print-gpu-trace Print individual kernel invocations (including CUDA memcpy's/memset's) and sort them in chronological order. In event/metric profiling mode, show events/metrics for each kernel invocation.

nvprof --print-summary-per-gpu ./your_application

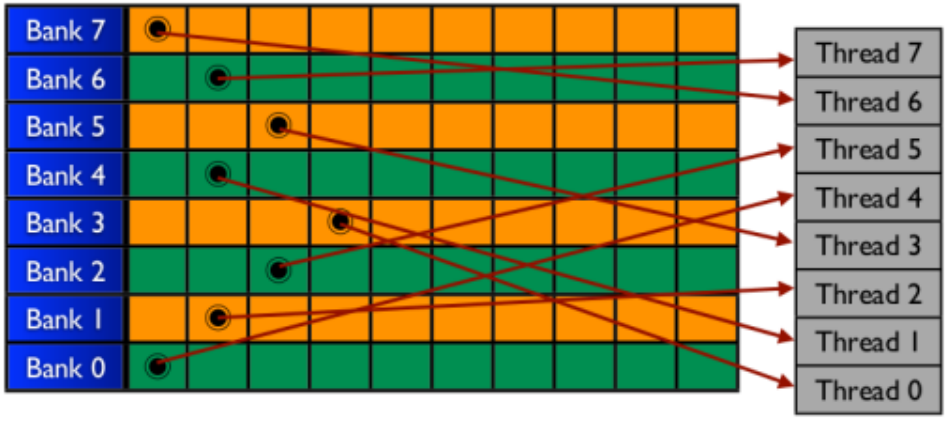
Shared Memory – interleaved banks

Bank 7	7	15	23						
Bank 6	6	14	22						
Bank 5	5	13	21						
Bank 4	4	12	20						
Bank 3	3	11	19						
Bank 2	2	10	18						
Bank 1	1	9	17						
Bank 0	0	8	16						

Shared Memory – interleaved banks

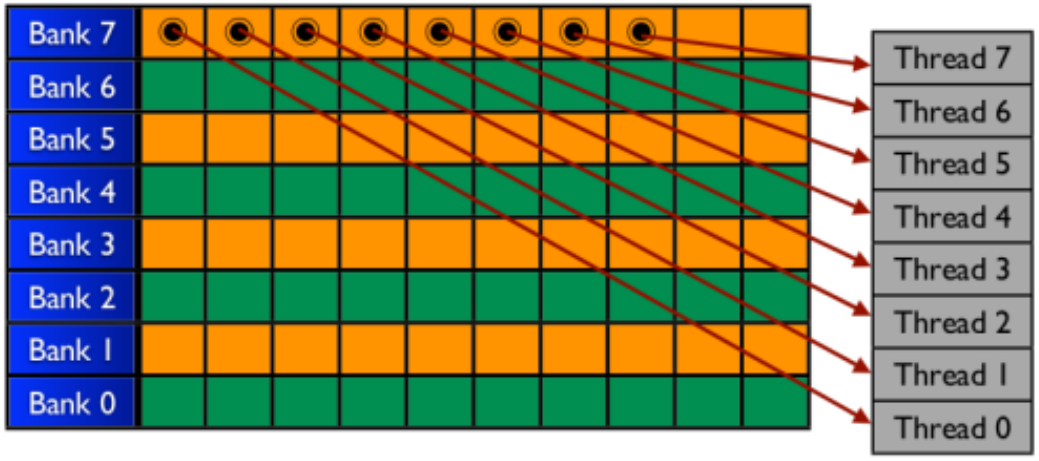


OK: one shared memory access is broadcast in parallel to all the threads in the half-warp.



OK: this access pattern satisfies the main rule as all threads in the half-warp access different shared memory banks.

Shared Memory – interleaved banks

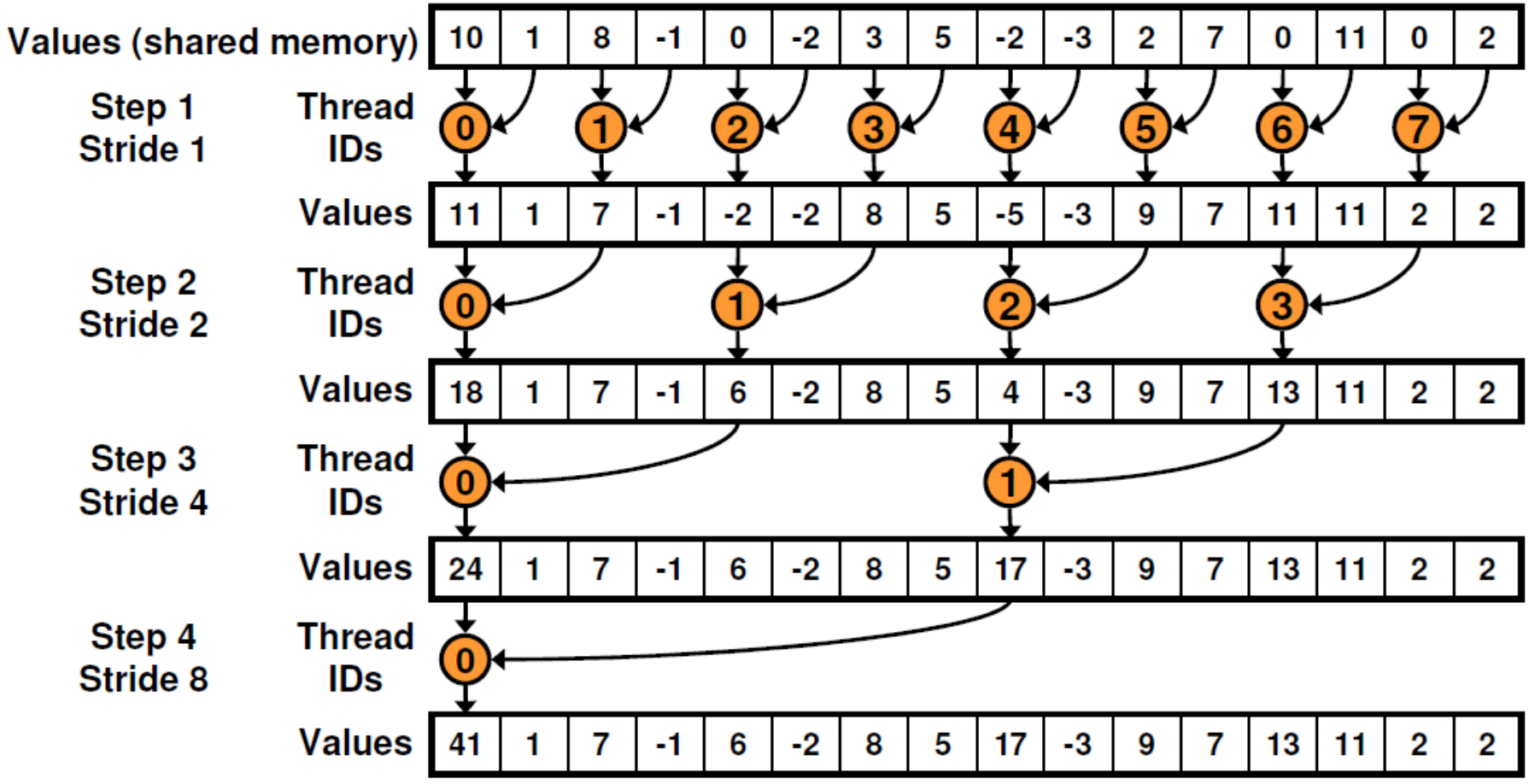


Not OK: in this case all threads in a half warp access the same bank.
The read/writes are bank conflicted, and are performed sequentially.
In this example the read/write performance would be 1/8 of maximum.

Shared Memory Bank Conflicts

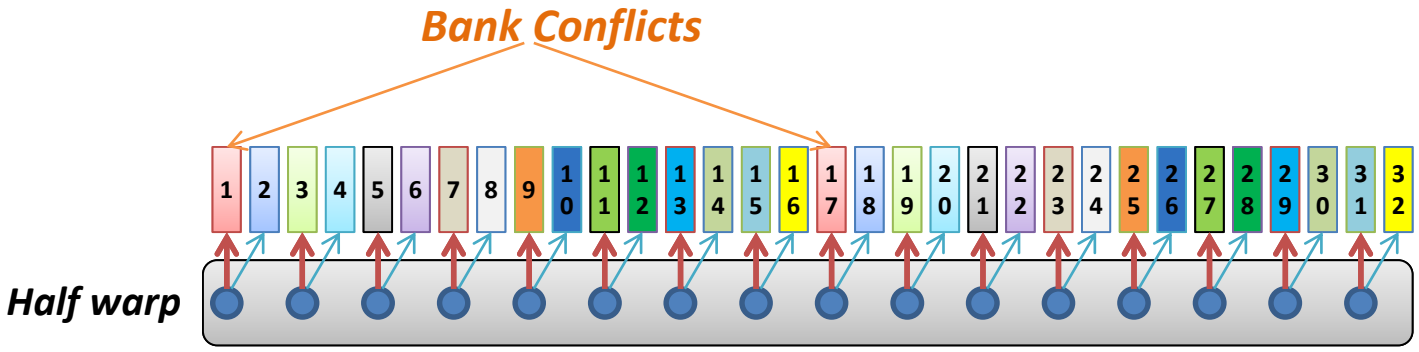
- Shared memory is split into equally sized memory banks (**16 banks** on devices of compute capability (CC) 1.x and **32 banks** on devices of compute capability 2.x).
- Any memory load or store of ‘n’ addresses that spans ‘b’ distinct memory banks can be serviced simultaneously,
  - Yielding an ***effective bandwidth that is ‘b’ times as high*** as the bandwidth of a single bank.

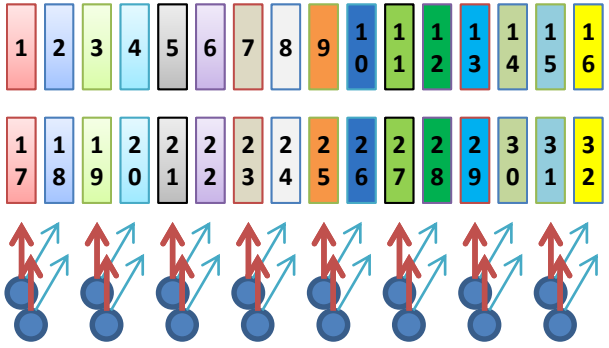
Parallel Reduction: Interleaved Addressing



New Problem: ***Shared Memory Bank Conflicts***

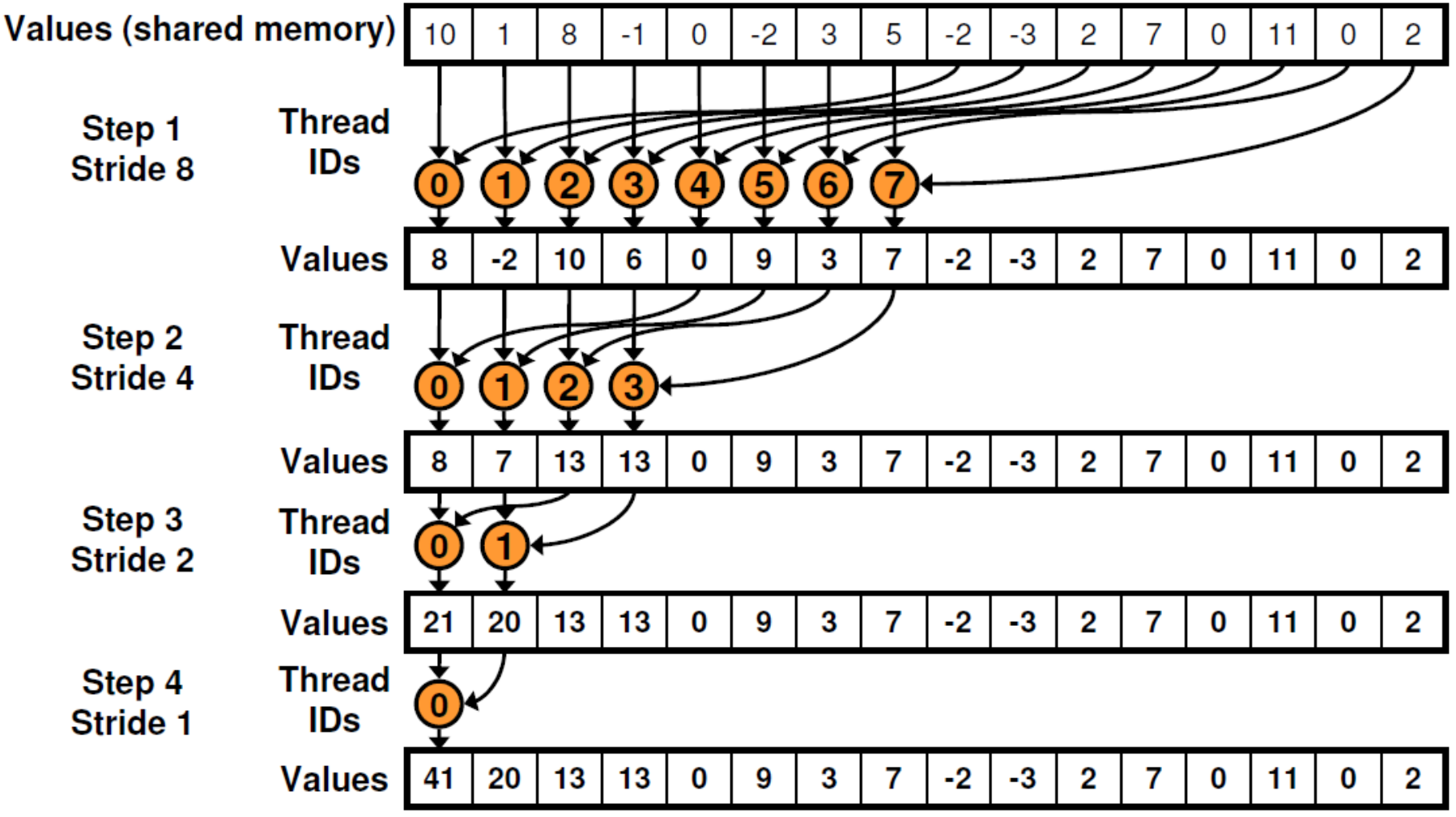
Parallel Reduction: Interleaved Addressing





New Problem: *Shared Memory Bank Conflicts*

Parallel Reduction: Sequential Addressing



Sequential addressing is conflict free

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();
}
```

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

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ nvprof --print-gpu-trace ./a.out
CPU results = 4718580
==27551== NVPROF is profiling process 27551, command: ./a.out
The size of array is 1048576 and it is processedon # of Blocks: 2048
The size of array is 2048 and it is processedon # of Blocks: 4
GPU result = 4718580
==27551== Profiling application: ./a.out
==27551== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
174.74ms  4.1868ms                  -               -        -         -         -  4.1943MB  1.0018GB/s        GK20A (0)         1         7  [CUDA memcpy HtoD]
179.22ms  20.879ms          (2048 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce2(int*, int*) [96]
201.29ms  12.500us             (4 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce2(int*, int*) [101]
202.00ms  4.3340us              (1 1 1)         (4 1 1)        8        0B       16B         -           -        GK20A (0)         1         7  reduce2(int*, int*) [106]
202.27ms  1.8330us                  -               -        -         -         -        4B  2.1822MB/s        GK20A (0)         1         7  [CUDA memcpy DtoH]
```

nvprof --print-summary-per-gpu ./your_application

Idle Threads

- Problem:

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

- Half of the threads are idle on first loop iteration!
- This is wasteful…

Reduction #4: First Add During Load

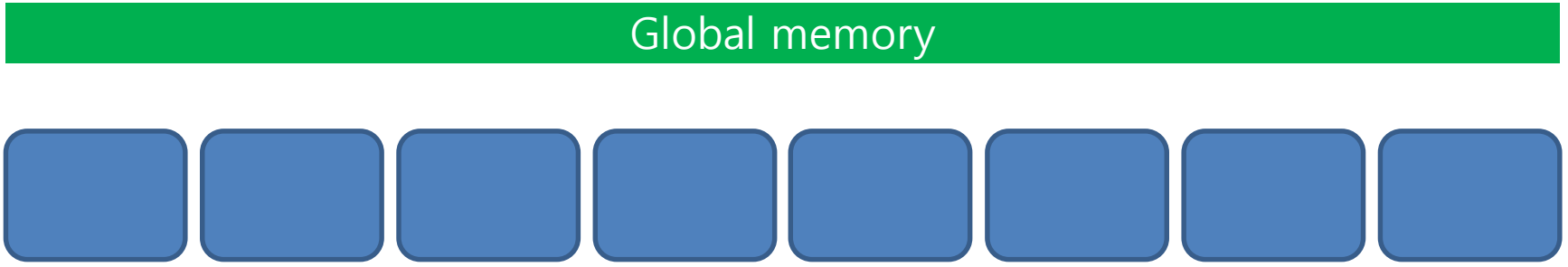
- Halve the number of blocks, and replace single load:

```
// 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,
// reading from global memory, 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();
```

Reduction #4: First Add During Load



Reduction #4: First Add During Load

i = **blockIdx.x*blockDim.x + threadIdx.x**;

threadIdx.x	0	1	2	3	0	1	2	3	0
blockIdx.x	0	0	0	0	1	1	1	1	2
blockDim.x	4	4	4	4	4	4	4	4	4
i	0	1	2	3	4	5	6	7	8

```
// 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();
```

i = **blockIdx.x*(blockDim.x*2) + threadIdx.x**;

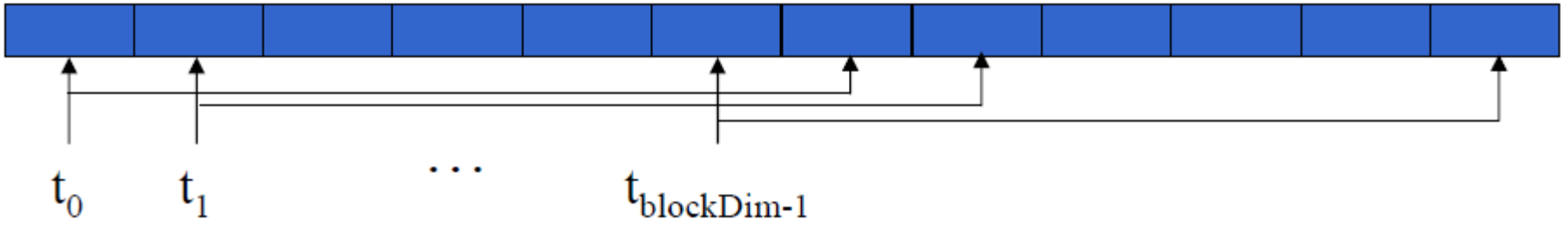
threadIdx.x	0	1	2	3	0	1	2	3	0
blockIdx.x	0	0	0	0	1	1	1	1	2
blockDim.x * 2	8	8	8	8	8	8	8	8	8
i	0	1	2	3	8	9	10	11	16

```
// perform first level of reduction,
// reading from global memory, 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();
```

Reduction #4: First Add During Load

- Reduction #3

- Reduction #4



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

- Half of the threads are idle on first loop iteration!

  ➔ Solved !

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

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ nvprof --print-gpu-trace ./a.out
CPU results = 4718580
==27763== NVPROF is profiling process 27763, command: ./a.out
The size of array is 1048576 and it is processed on # of Blocks: 1024
The size of array is 1024 and it is processed on # of Blocks: 1
GPU result = 4718580
==27763== Profiling application: ./a.out
==27763== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
177.36ms  4.4923ms                    -               -        -         -         -  4.1943MB  933.66MB/s        GK20A (0)         1         7  [CUDA memcpy HtoD]
182.14ms  10.778ms           (1024 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce3(int*, int*) [96]
194.30ms  9.1670us              (1 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce3(int*, int*) [101]
194.88ms  2.2500us                    -               -        -         -         -        4B  1.7778MB/s        GK20A (0)         1         7  [CUDA memcpy DtoH]
```

nvprof --print-summary-per-gpu ./your_application

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***

Unrolling the Last Warp

- As reduction proceeds, # “active” threads decreases
  - When s <= 32, we have only one warp left
- **Instructions are SIMD synchronous within a warp**
- 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
    - Whatever the “s” is, a warp is issued !
- Let’s unroll the last 6 iterations of the inner loop

waits until all threads in the thread block have reached this point **and all global and shared memory accesses made by these threads prior to __syncthreads() are visible to all threads in the block.** ➡ **volatile: ensure actual memory read/write !**

Reduction #5: Unroll the Last Warp

```
__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];
}
```

```
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);
```

- 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

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

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ nvprof --print-gpu-trace ./a.out
CPU results = 4718580
==28294== NVPROF is profiling process 28294, command: ./a.out
The size of array is 1048576 and it is processed on # of Blocks: 1024
The size of array is 1024 and it is processed on # of Blocks: 1
GPU result = 4718580
==28294== Profiling application: ./a.out
==28294== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
182.07ms  4.4717ms                   -               -        -         -         -  4.1943MB  937.96MB/s        GK20A (0)         1         7  [CUDA memcpy HtoD]
186.83ms  6.6488ms           (1024 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce4(int*, int*) [96]
194.59ms  34.334us              (1 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  reduce4(int*, int*) [101]
195.12ms  8.5840us                   -               -        -         -         -         4B  465.98KB/s        GK20A (0)         1         7  [CUDA memcpy DtoH]
```

nvprof --print-summary-per-gpu ./your_application

Complete Unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Luckily, the ***block size is limited by the GPU to 512 threads***
  - Also, we are sticking to power-of-2 block sizes

Technical specifications	Compute capability (version)						
	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Maximum number of threads per block	512				1024		

- 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?

- ***Templates*** to the rescue!
  - **CUDA supports C++ template parameters on device and host functions**

Unrolling with Templates

- Specify block size as a function template parameter:

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

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);
```

Volatile?

```
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];
}
```

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

Invoking Template Kernels

- Don’t we still need block size at compile time?
  - Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
{
    case 512:
        reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
        reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
       reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
        reduce5<64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
        reduce5<32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
        reduce5<16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 8:
        reduce5<8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 4:
        reduce5<4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 2:
        reduce5<2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 1:
        reduce5<1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```

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

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ nvprof --print-gpu-trace ./a.out
CPU results = 4718580
==28430== NVPROF is profiling process 28430, command: ./a.out
The size of array is 1048576 and it is processed on # of Blocks: 1024
The size of array is 1024 and it is processed on # of Blocks: 1
GPU result = 4718580
==28430== Profiling application: ./a.out
==28430== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
183.56ms  4.1890ms                   -               -        -         -         -  4.1943MB  1.0013GB/s        GK20A (0)         1         7  [CUDA memcpy HtoD]
188.04ms  5.9278ms          (1024 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  void reduce5<unsigned int=512>(int*, int*) [96]
195.08ms  31.835us            (1 1 1)       (512 1 1)        8        0B  2.0480KB         -           -        GK20A (0)         1         7  void reduce5<unsigned int=512>(int*, int*) [101]
195.59ms  8.7510us                   -               -        -         -         -        4B  457.09KB/s        GK20A (0)         1         7  [CUDA memcpy DtoH]
```

Parallel Reduction Complexity

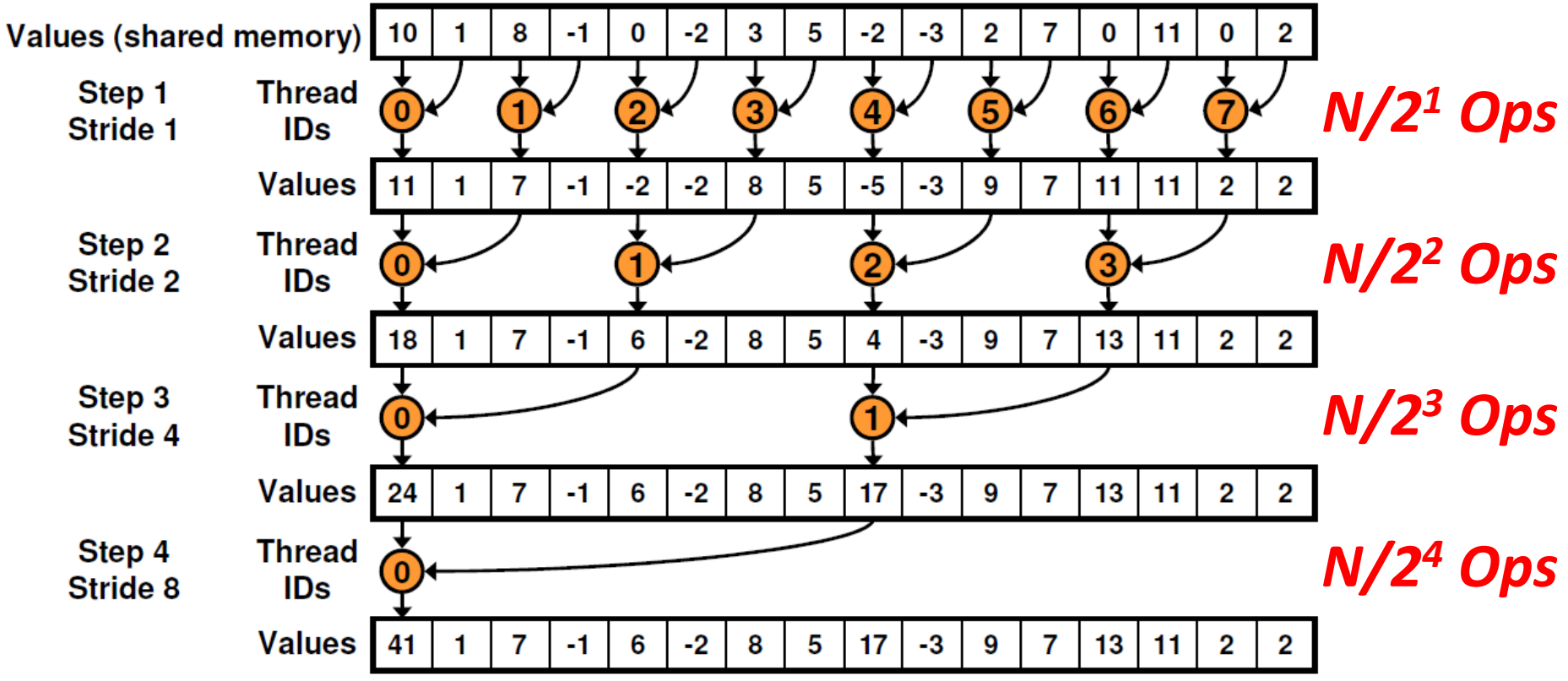
- **log(*N*)** parallel steps, each step S does $N/2^S$ independent ops
  - **Step Complexity** is O(log N)

- 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**
  - i.e. ***does not perform more operations than a sequential algorithm***

- With P threads physically in parallel (P processors), **time complexity** is ***O(N/P + log P)***
  - Compare to O(N) for sequential reduction
  - In a thread block, N=P, so **O(log N)**

Parallel Reduction Complexity

- **Log(*N*)** parallel steps, each step S does ***N/2^S*** independent ops
  - **Step Complexity** is O(log N) ~ Time Step~!



What About ***Cost***?

- ***Cost*** of a parallel algorithm is ***"#-of-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! – Sequential case : 1 proc. × O(N) = O(N)**
- ***Brent's theorem*** suggests **O(N/log N)** threads (processors)
  - **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

What About *Cost*?

- ***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) $\Rightarrow$ **cost efficient**

N = 1024 : Number of data elements
N/logN = 102.4 : # of Processors

Then,
Each processor has to work with "N / N/logN" data elements (= logN)

Then,
Parallel Processing with N/logN processors → log(N/logN) $\in$ O(logN)

Consequently, time complexity = Sequential O(logN) + Parallel O(logN) = O(logN)
Cost : O(N/logN) * O(logN) = O(N)

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
- In my experience, 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 perf with 64-256 blocks of 128 threads
  - 1024-4096 elements per thread

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:

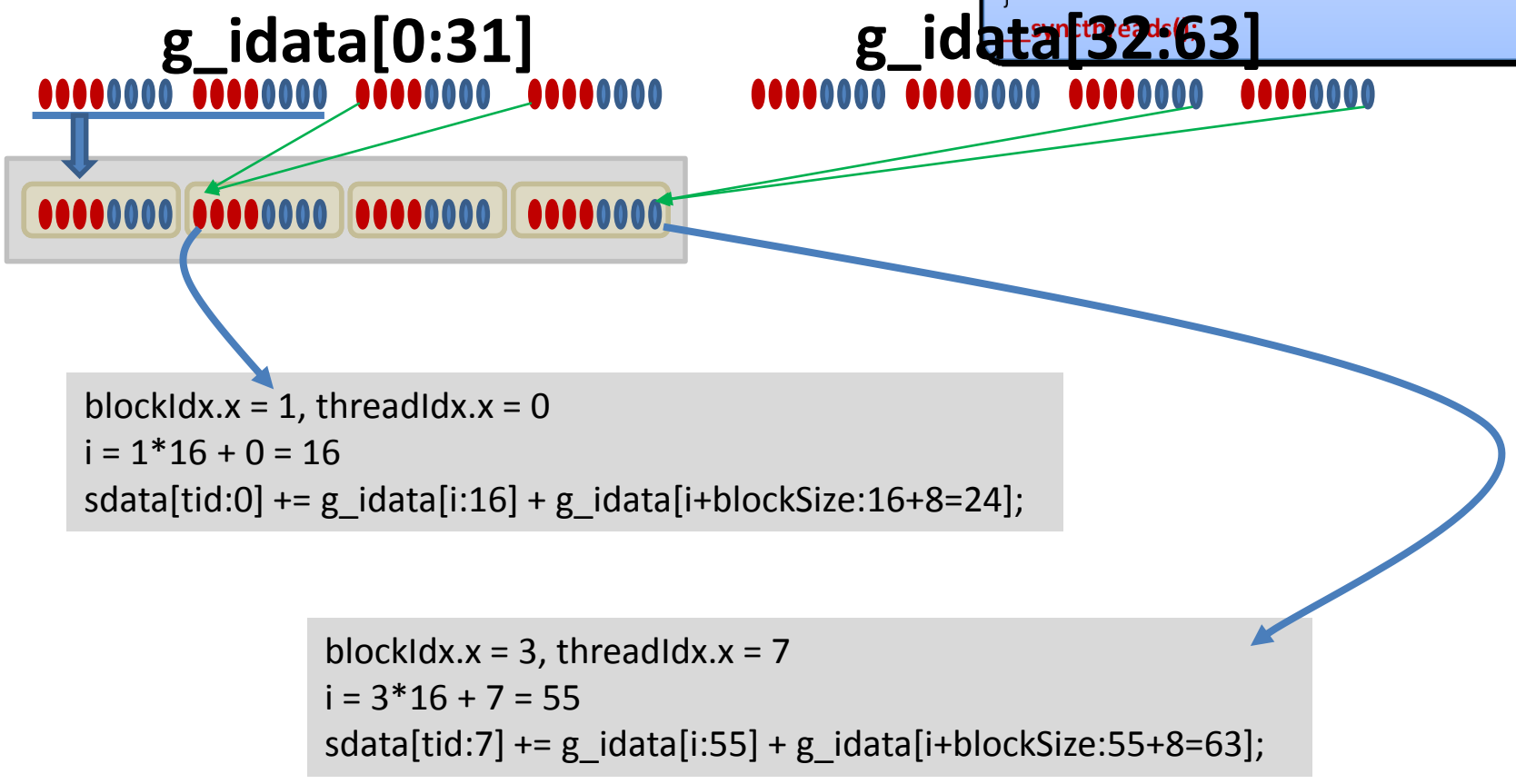
```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride to maintain coalescing!

Reduction #7: Multiple Adds / Thread

blockSize =8, gridDim.x = 4
I = blockIdx.x*16 + threadIdx.x
gridSize = 8*2*4 =64

```
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();
```



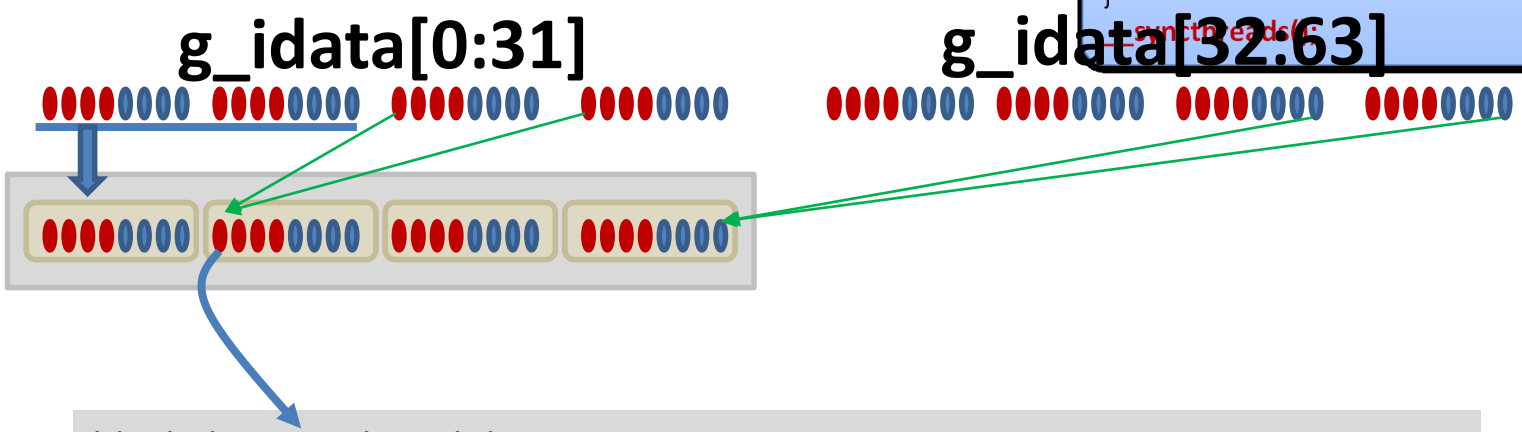
blockIdx.x = 1, threadIdx.x = 0
i = 1*16 + 0 = 16
sdata[tid:0] += g_idata[i:16] + g_idata[i+blockSize:16+8=24];

blockIdx.x = 3, threadIdx.x = 7
i = 3*16 + 7 = 55
sdata[tid:7] += g_idata[i:55] + g_idata[i+blockSize:55+8=63];

Reduction #7: Multiple Adds / Thread

blockSize =8, gridDim.x = 4
I = blockIdx.x*16 + threadIdx.x
gridSize = 8*2*4 =64

```
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();
```



blockIdx.x = 1, threadIdx.x = 0
i = 1*16 + 0 = 16
sdata[tid:0] += g_idata[i:16] + g_idata[i+blockSize:16+8=24];
i = 16 + 64 = 80
sdata[tid:0] += g_idata[i:16+84] + g_idata[i+blockSize:16+8+84=24+84];

Reduction #7: Multiple Adds / Thread

- gridSize, incremental global memory address
  - gridSize is multiples of 16
    - Aligned with next global memory load
    - **Memory coalescing!**

```
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();
```

Performance for 4M element reduction

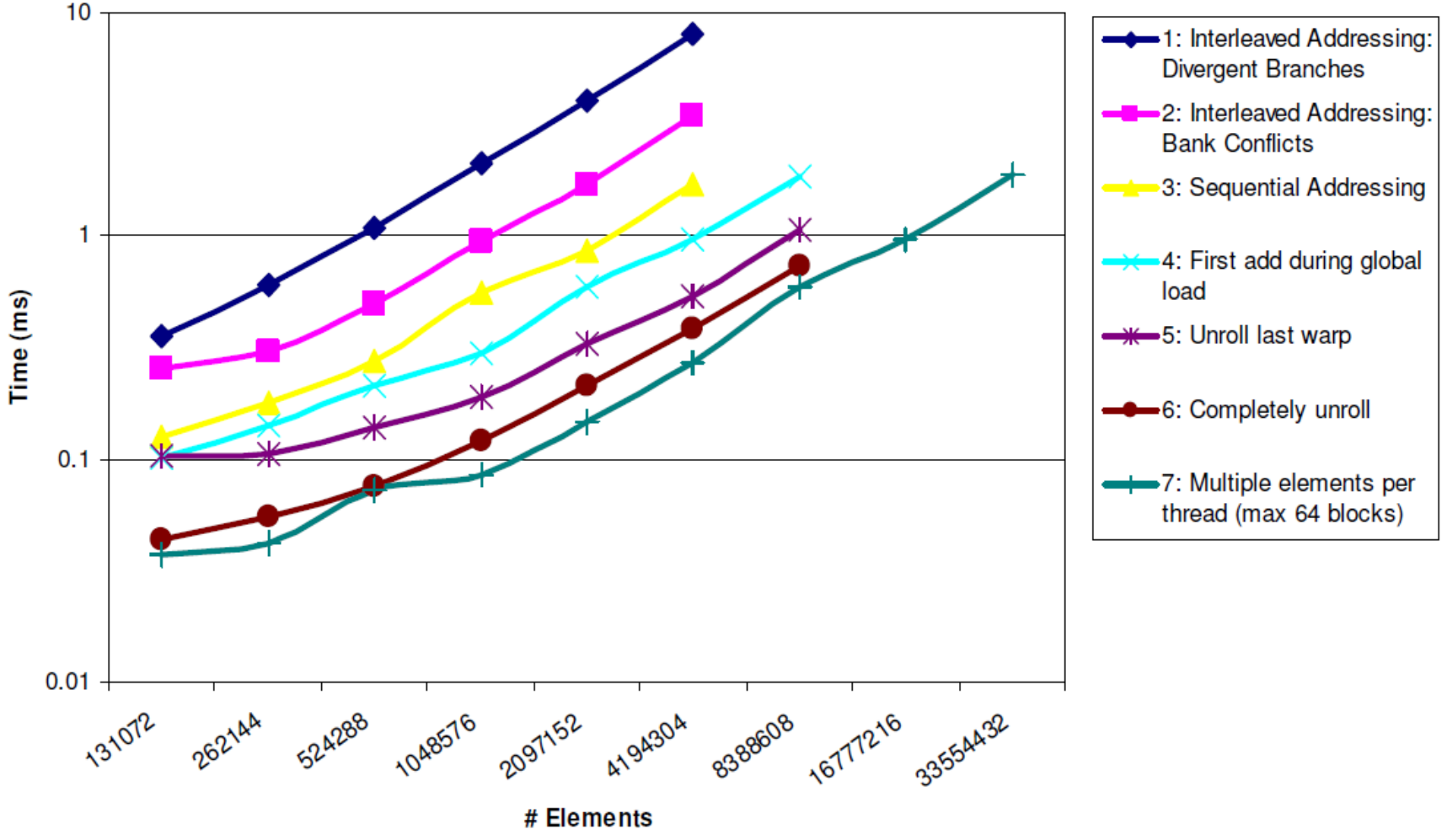
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: Interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first 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
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

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

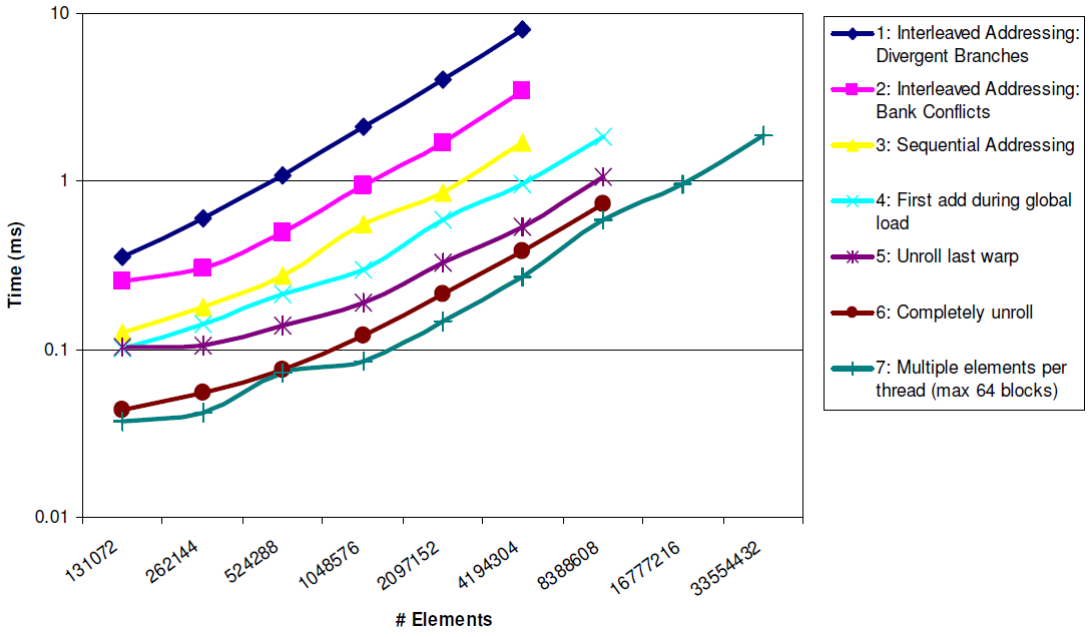
Final Optimized Kernel

```
template <unsigned int blockSize>
__global__ void reduce6(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<blockSize>(sdata, tid);

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

Performance Comparison



Performance Comparison



ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/6_Advanced/reduction$

```
"--shmoo":          Test performance for 1 to 32M elements with each of the 7 different kernels
"--n=<N>":          Specify the number of elements to reduce (default 1048576)
"--threads=<N>":    Specify the number of threads per block (default 128)
"--kernel=<N>":     Specify which kernel to run (0-6, default 6)
"--maxblocks=<N>": Specify the maximum number of thread blocks to launch (kernel 6 only, default 64)
"--cpufinal":       Read back the per-block results and do final sum of block sums on CPU (default false)
"--cputhresh=<N>": The threshold of number of blocks sums below which to perform a CPU final reduction
                    (default 1)
"-type=<T>":        The datatype for the reduction, where T is "int", "float", or "double" (default int)
```

Types of optimization

- Interesting observation:

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

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

Conclusion

- 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