



CUDA Optimization with Parallel Reduction

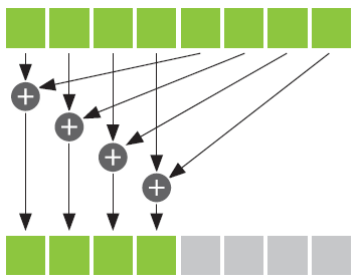

이 정 근 (Jeong-Gun Lee)

한림대학교 컴퓨터공학과, 임베디드 SoC 연구실
www.onchip.net
 Email: Jeonggun.Lee@hallym.ac.kr

CUDA Optimization

- “**Optimizing Parallel Reduction in CUDA**” written by Mark Harris for demonstrating how to optimize your CUDA applications



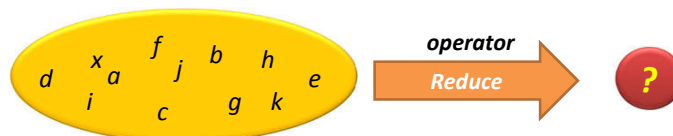
Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
- 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

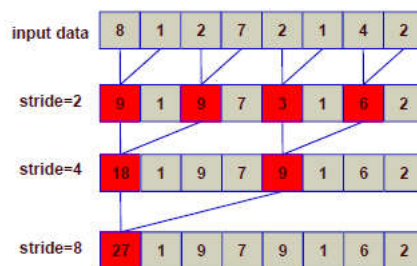


- Examples:
 - Minimum/maximum value
 - Average, sum, product, etc.
- Parallel Reduction: Do it in **parallel** obviously



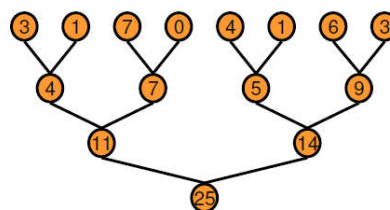
Parallel Reduction ??

- Calculating “ $x_1 + x_2 + x_3 + \dots + x_n$ ”
- 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.

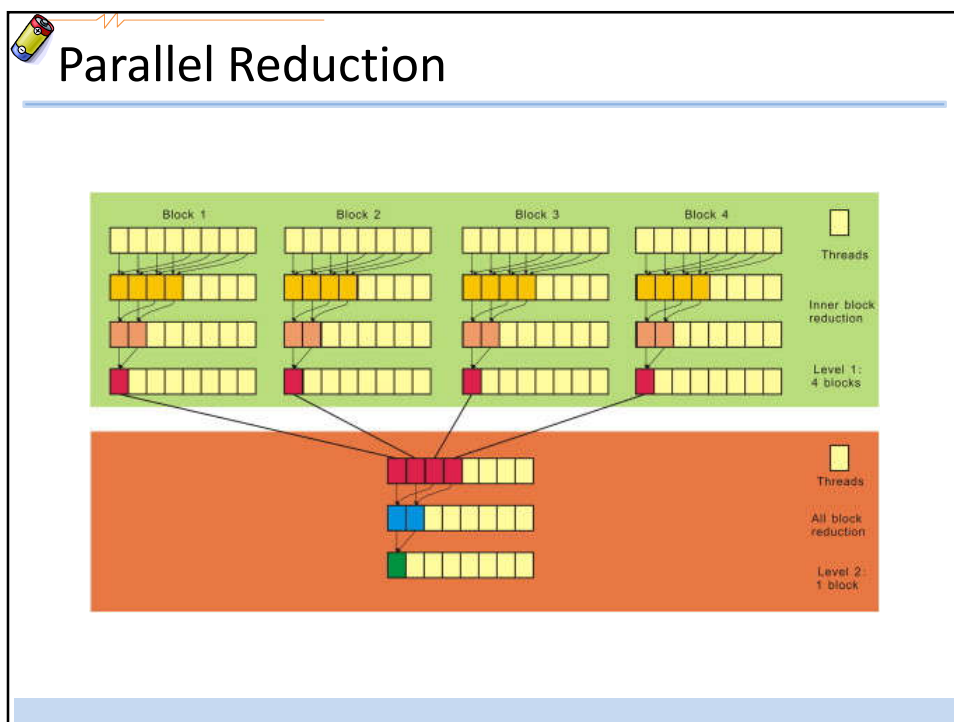
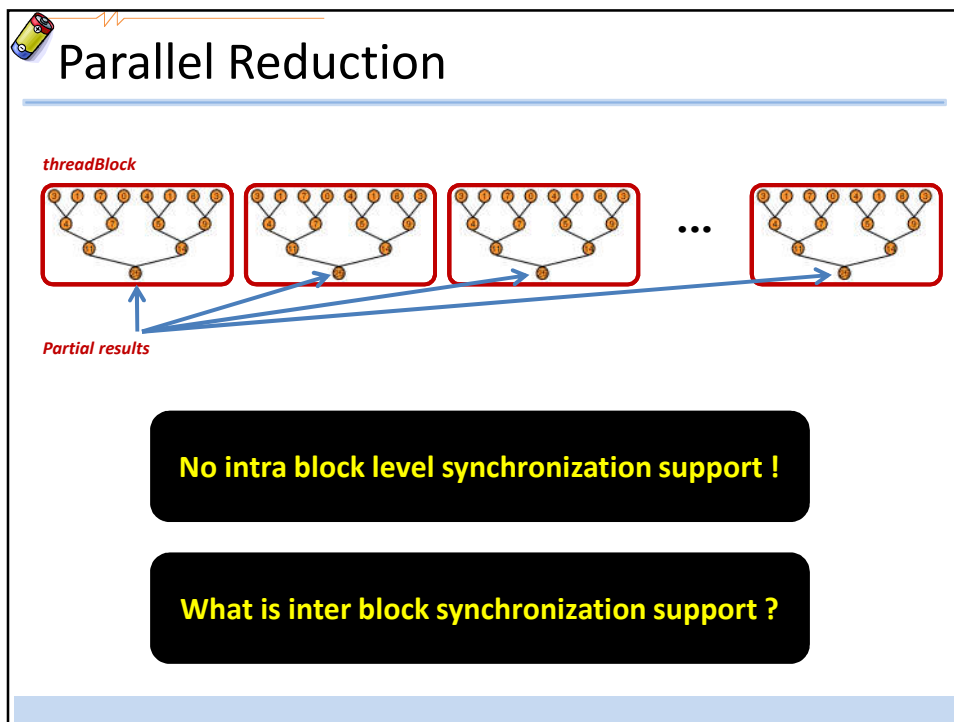


Parallel Reduction

- 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?





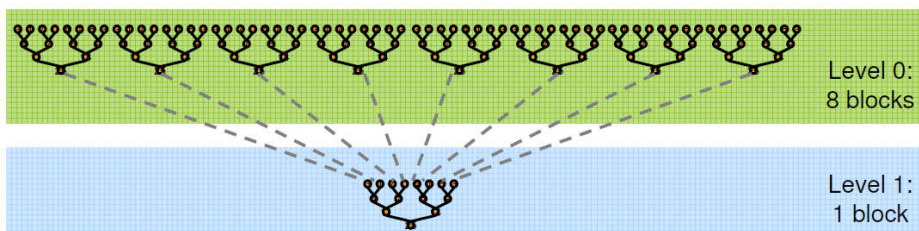
Problem: Global Synchronization

- 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()`**

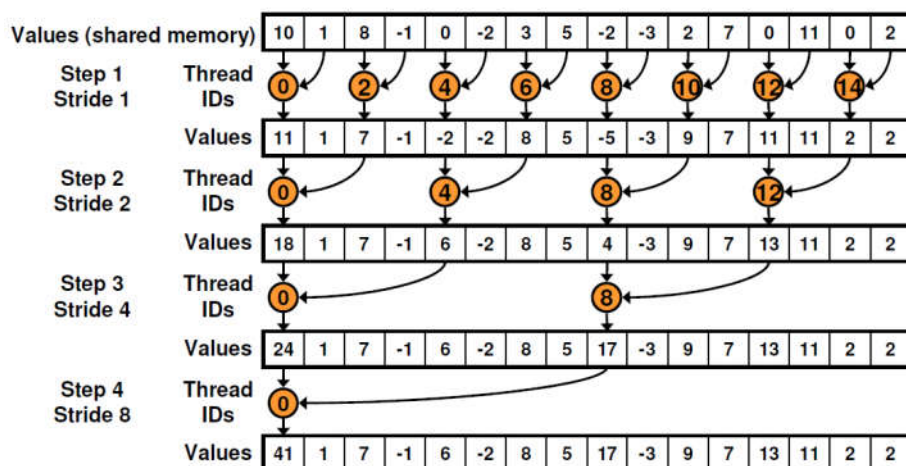


What is Our Optimization Goal?

- 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 (old) for this example
 - 384-bit memory interface, 900 MHz DDR (Double Data Rate)
 - $384 * (1800) / 8 = \mathbf{86.4\ GB/s}$



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];
}
```



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

Time (clocks)

1 2 ... 8
ALU 1 ALU 2 ... ALU 8

(assume logic below is to be executed for each element in input array 'X', 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;
```

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


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 = 4718000
==27381== NVPROF is profiling process 27381, command: ./a.out
The size of array is 20480 and it is processed on # of Blocks: 2048
The size of array is 2048 and it is processed on # of Blocks: 4
GPU result = 4718000
==27381== Profiling application: ./a.out
==27381== Profiling result:
Start: Duration      Grid Size    Block Size    Regs*    SSMem*    DSMem*    Size    Throughput    Device    Context    Stream    Name
170.50ms  4.487ms          (2048 1 1)    (512 1 1)    8        0B        0B        4.194MB    939.64MB/s    GK20A (0)    1        1        ? [CUDA memcpy [Src]]
183.14ms  44.290ms          (2048 1 1)    (512 1 1)    8        0B        0B        2.048MB    -            GK20A (0)    1        1        ? reduce0(int*, int*) [96]
228.70ms  24.600ms          (4 1 1)        (512 1 1)    8        0B        0B        2.048MB    -            GK20A (0)    1        1        ? reduce0(int*, int*) [101]
229.61ms  3.160ms          (1 1 1)        (4 1 1)      8        0B        0B        1KB       -            GK20A (0)    1        1        ? reduce0(int*, int*) [104]
229.94ms  2.250ms          -             -           -        -        -        4B       1.777MB/s    GK20A (0)    1        1        ? [CUDA memcpy [Dst]]


```

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();
}

```

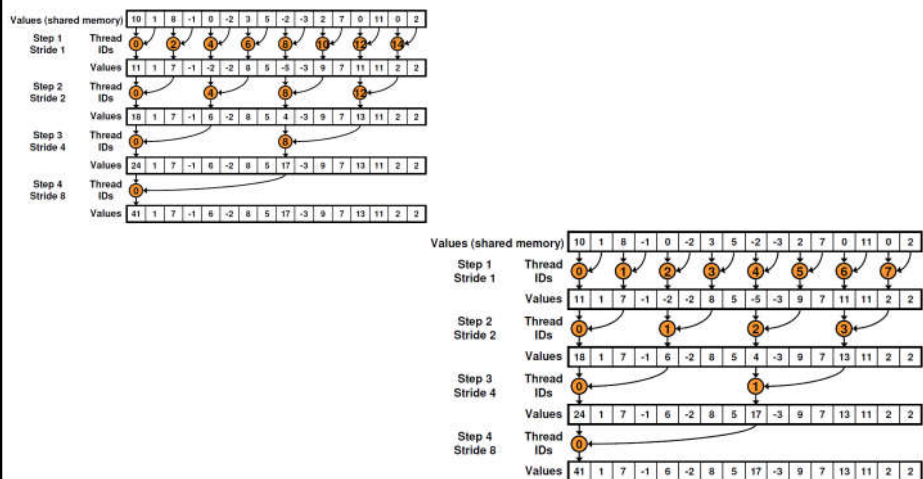


Reduction #2: Interleaved Addressing


```
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 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



Test on Jetson TK1

```

ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction$ nvprof --print-gpu-trace ./a.out
CPU results = 4718590
--27473-- nvprof is profiling process 27473, command: ./a.out
The size of array is 1048576 and it is processed on # of Blocks: 2048
The size of array is 2048 and it is processed on # of Blocks: 4
GPU results = 4718590
--27473-- Profiling application: ./a.out
--27473-- Profiling result:
  Start   Duration      Grid Size    Block Size    Regs*    SMem*    DSMem*    Size Throughput    Device    Context    Stream    Name
  186.7ms  4.4946ms          (2048 1 1)    (512 1 1)      8         0B    2.0480KB    -         -    GK20A (0)    1       7    (CUDA memcpy HtoB)
  191.14ms 34.433ms          (2048 1 1)    (512 1 1)      8         0B    2.0480KB    -         -    GK20A (0)    1       7    reduce1(int*, int*) (96)
  226.74ms 16.334us          (4 1 1)       (512 1 1)      8         0B    2.0480KB    -         -    GK20A (0)    1       7    reduce1(int*, int*) (104)
  227.14ms 1.4670us          (1 1 1)       (4 1 1)        8         0B        16B    -         -    GK20A (0)    1       7    reduce1(int*, int*) (106)
  229.71ms 1.9160us          -             -             -         -         -        4B    2.0877GB/s    GK20A (0)    1       7    (CUDA memcpy DtoH)

```

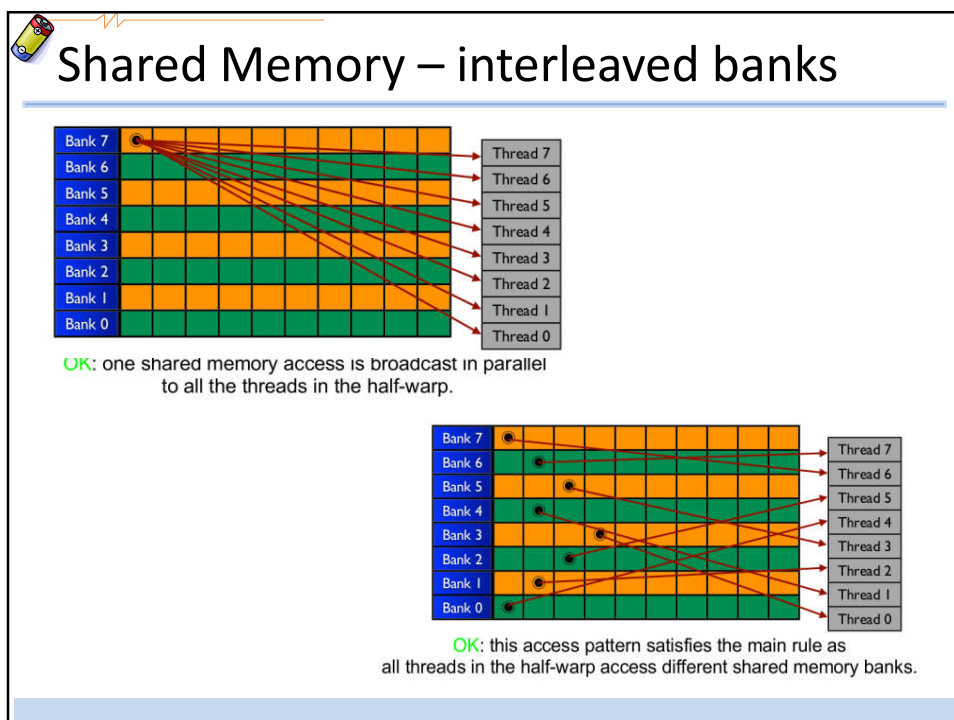
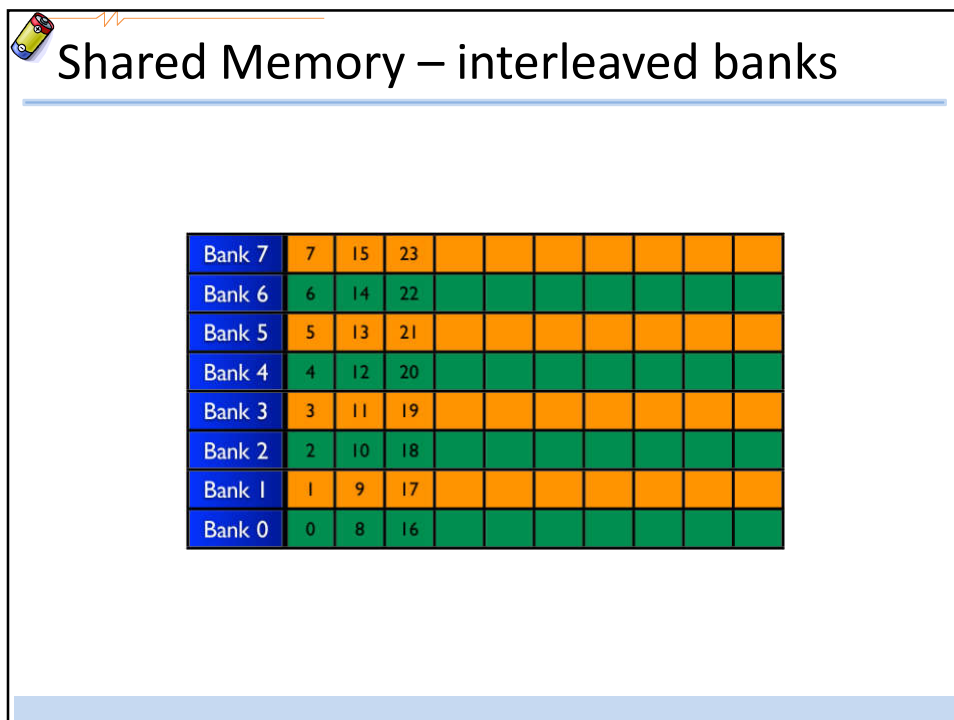
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/reduction\$ **nvprof --help**

Usage: nvprof [options] [CUDA-application] [application-arguments]

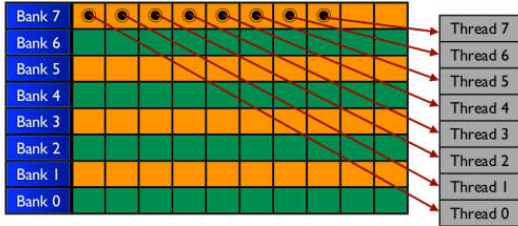
Options:

- 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



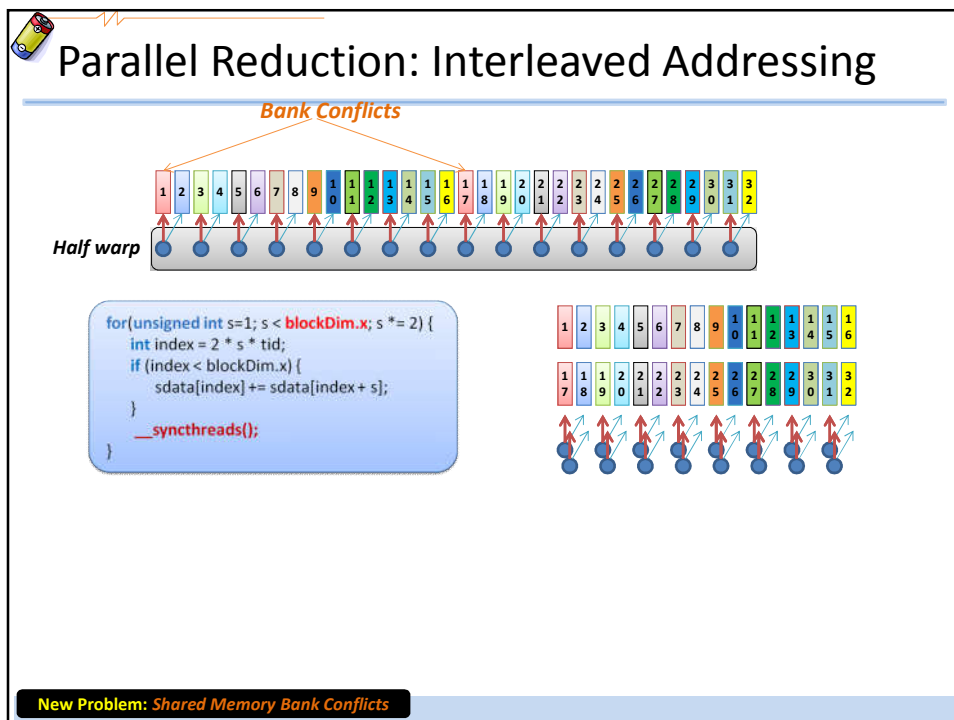
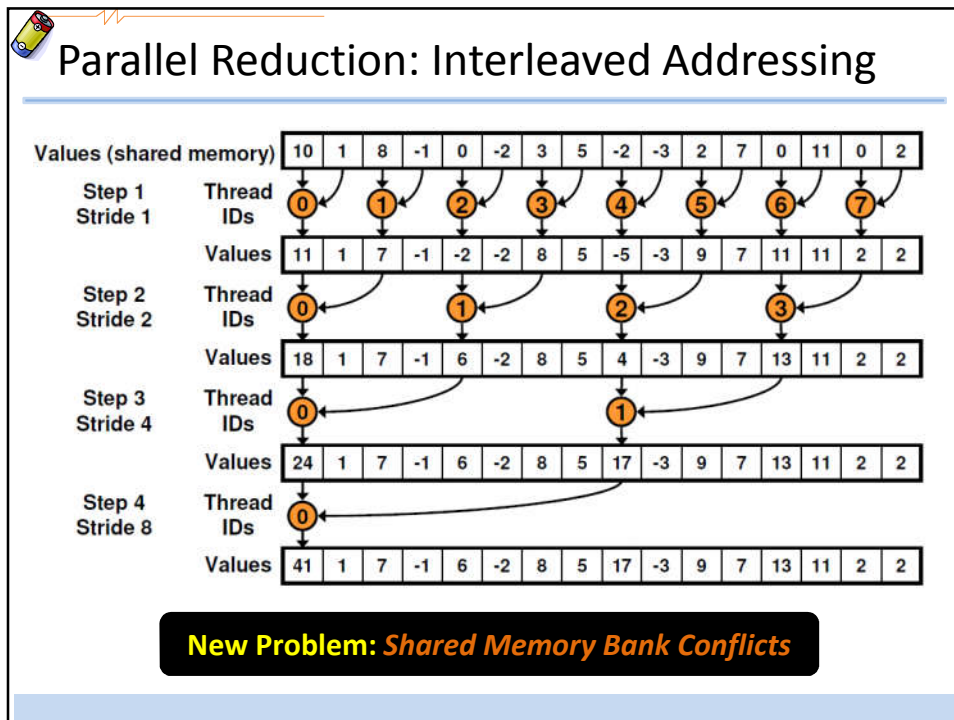
Shared Memory – interleaved banks

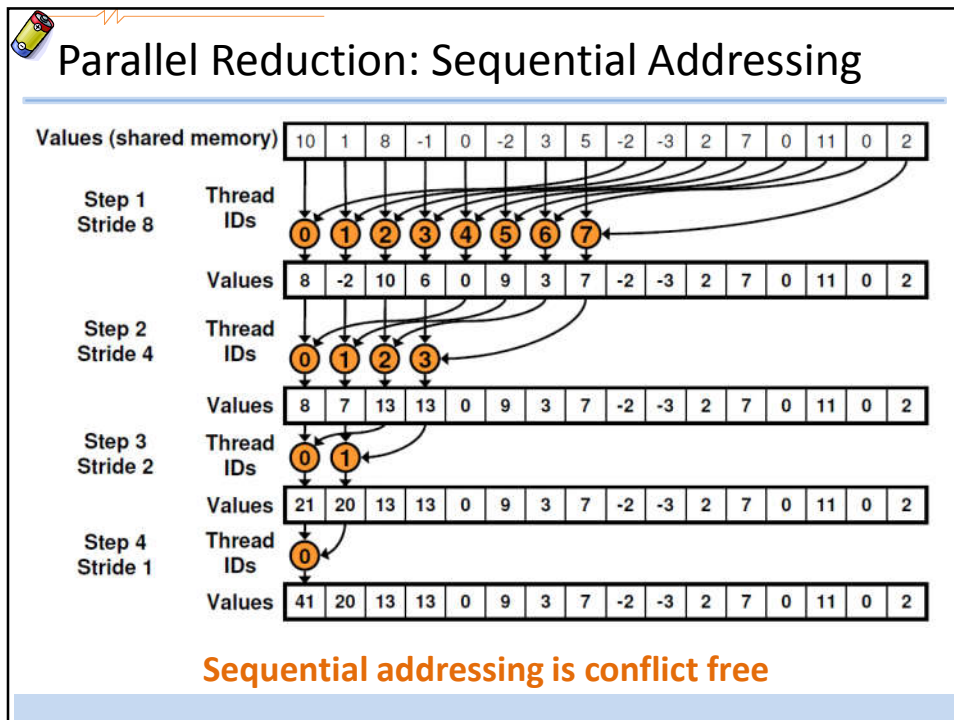


Not OK: in this case all threads in a half warp access the same bank. The read/writes are bank conflicted, and are performed sequentially. In this example the read/write performance would be 1/8 of maximum.

Shared Memory Bank Conflicts

- Shared memory is split into equally sized memory banks (**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.





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-8.5_Samples/REDUCTIONS$ 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* SMem* DMem* Size Throughput Device Context Stream Name
174.74ms 4.1868ms - - - - - 4.194MB 1.0018GB/s GK20A (0) 1 7 [CUDA memcpy HtoD]
179.22ms 20.879ms (2048 1 1) (512 1 1) 8 GB 2.0480KB - GK20A (0) 1 7 reduce2(int*, int*) [96]
201.29ms 12.500us (4 1 1) (512 1 1) 8 GB 2.0480KB - GK20A (0) 1 7 reduce2(int*, int*) [101]
202.00ms 4.3340us (1 1 1) (4 1 1) 8 GB 16B - GK20A (0) 1 7 reduce2(int*, int*) [106]
212.27ms 1.8330us - - - - - 4B 2.1822MB/s GK20A (0) 1 7 [CUDA memcpy HtoD]
  
```

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


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

Global memory



```
// each thread loads one element from global to shared mem
sdata[tid] = g_idata[i];
__syncthreads();
```
- With **two loads and first add** of the reduction:

Global memory



```
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Reduction #4: First Add During Load

$i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$

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();
```

$i = \text{blockIdx.x} * (\text{blockDim.x} * 2) + \text{threadIdx.x};$

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

```

dant@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/NVCC/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 8 of Blocks: 1024
The size of array is 1024 and it is processed on 8 of Blocks: 1
GPU result = 4718580
==27763== Profiling application: ./a.out
==27763== Profiling result:
Start Duration      Grid Size      Block Size      Regs*      SMem*      DSMem*      Size      Throughput      Device      Context      Stream      Name
127.56ms  4.482ms      -              -            -            -            -            4.1943MB  933.66MB/s      GK20A (0)    1          7          [CUDA memcpy Host]
152.14ms  10.778ms      (1024 1 1)    (512 1 1)      8            0B          2.0480KB    -          -              GK20A (0)    1          7          reduce3(int*, int*) [96]
154.30ms  9.1670us      (1 1 1)       (512 1 1)      8            0B          2.0480KB    -          -              GK20A (0)    1          7          reduce3(int*, int*) [104]
154.88ms  2.2500us      -              -            -            -            -            4B          1.7778MB/s      GK20A (0)    1          7          [CUDA memcpy Host]

```

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 $s \leq 32$, we have only one warp left
- **Instructions are SIMD synchronous within a warp**
- That means when $s \leq 32$:
 - 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/EXAMPLES/reduction$ nvprof --print-gpu-trace ./a.out
GPU 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    Reps*    SMem*    DSMem*    Size    Throughput    Device    Context    Stream    Name
182.07ms  4.4717ms          (1024 1 1)    (512 1 1)      8      0B    2.083GB    -      4.194MB    GE20A (0)    1      7    [CUDA memory HtoB]
186.93ms  6.6488ms          (1024 1 1)    (512 1 1)      8      0B    2.083GB    -      4.194MB    GE20A (0)    1      7    reduce(int*, int*) [86]
189.43ms  24.839ms          (1024 1 1)    (512 1 1)      8      0B    2.083GB    -      4.194MB    GE20A (0)    1      7    reduce(int*, int*) [101]
195.12ms  8.5840ms          (1024 1 1)    (512 1 1)      8      0B    2.083GB    -      4.194MB    GE20A (0)    1      7    [CUDA memory DtoB]
```

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, **block size is limited by the GPU to 512/1024 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;
}
```

- **$\log(N)$** parallel steps, each step S does $N/2^S$ independent ops

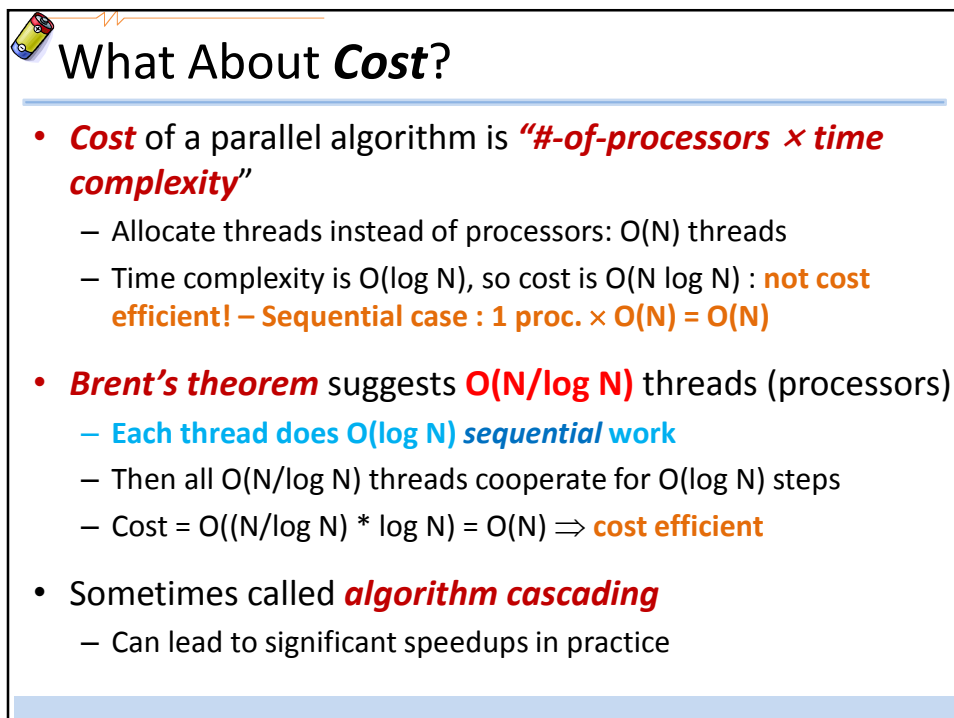
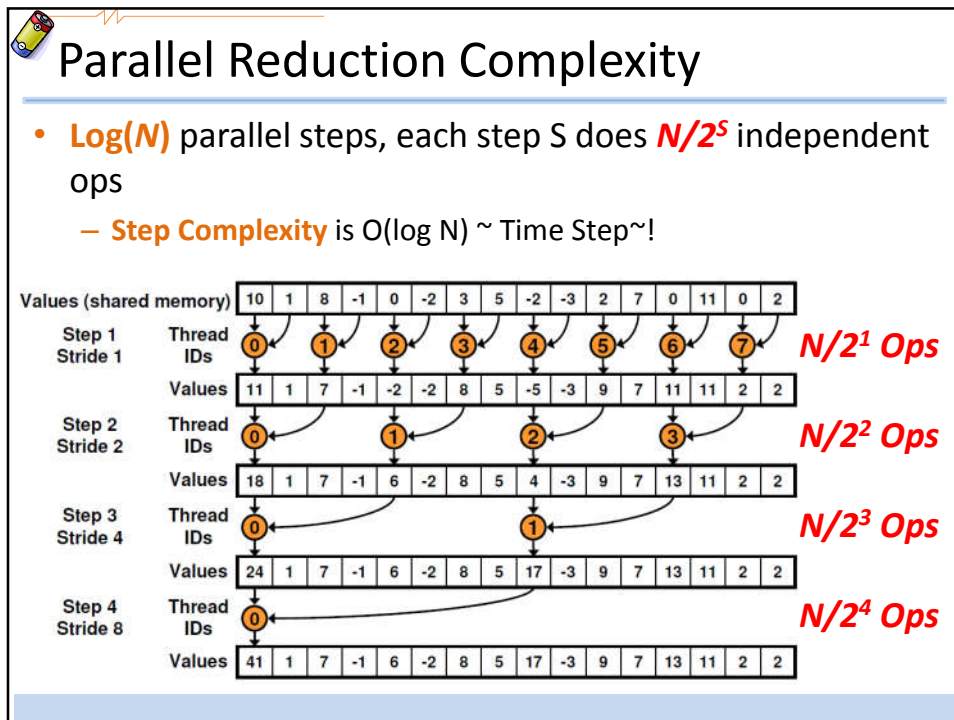
- For $N=2^D$, performs $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operations

- **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 $O(\log N)$





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

$N/\log N = 102.4$: # of Processors

Then,

Each processor has to work with " $N / N/\log N$ " data elements (= $\log N$)

Then,

Parallel Processing with $N/\log N$ processors $\rightarrow \log(N/\log N) \in O(\log N)$

Consequently, time complexity = Sequential $O(\log N)$ + Parallel $O(\log N) = O(\log N)$

Cost : $O(N/\log N) * O(\log N) = O(N)$



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
 # l = 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;
}
```

g_idata[0:31] g_idata[32:63]

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

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

Reduction #7: Multiple Adds / Thread

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

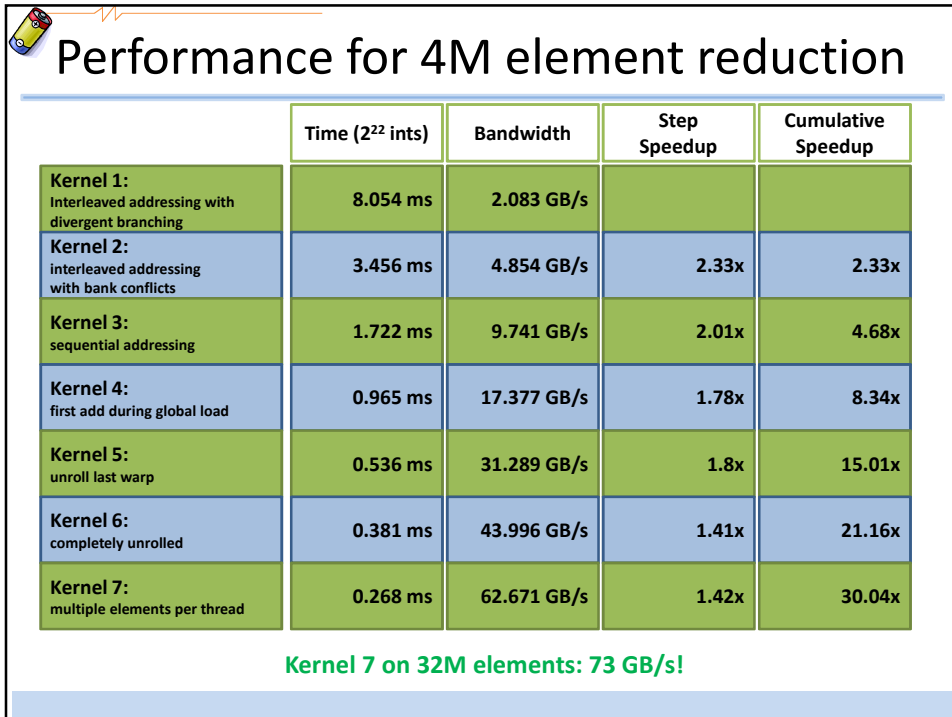
```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

```
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!**

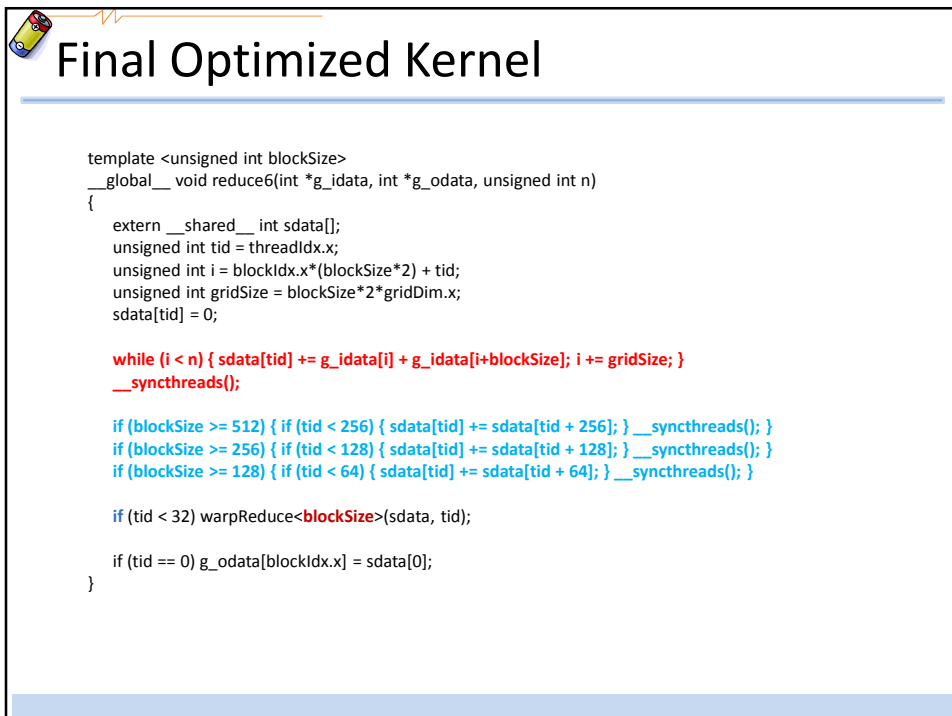
```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```



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;

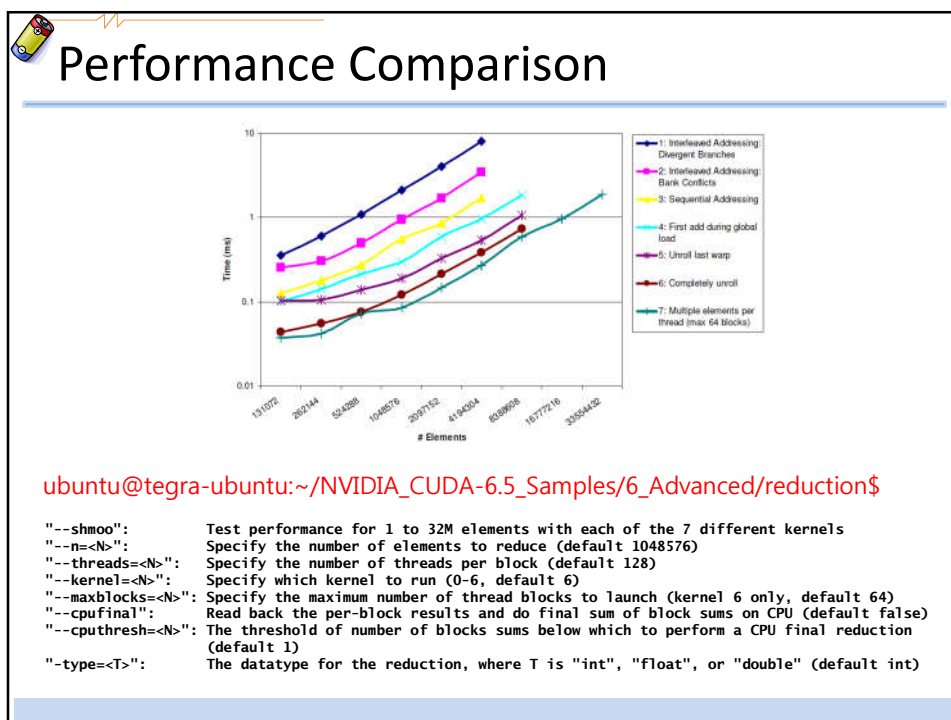
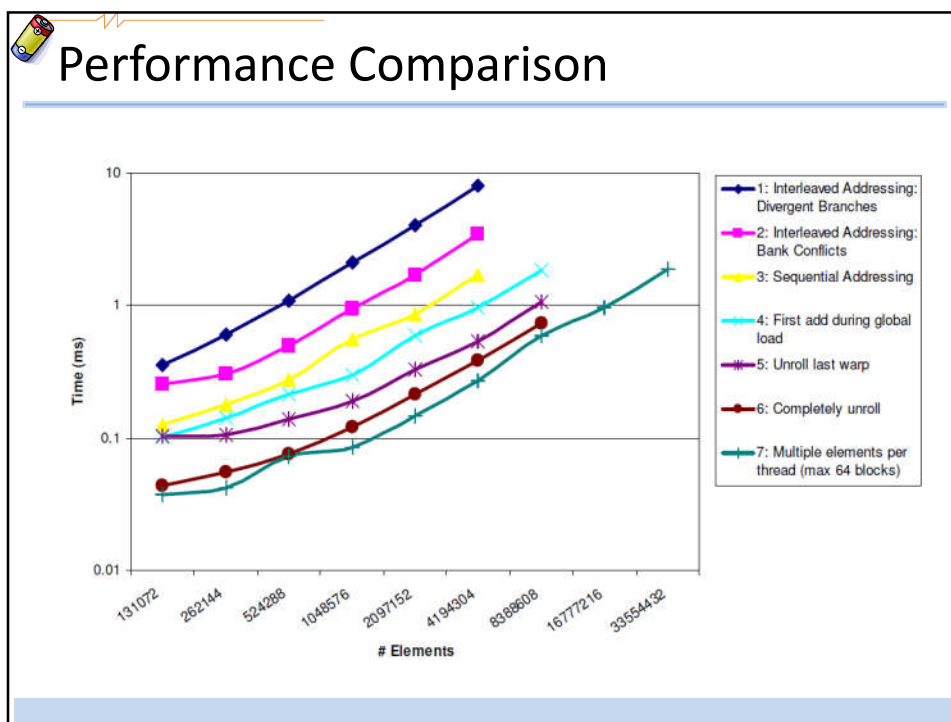
    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    __syncthreads();

    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];
}

```





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