并行与分布式计算

Ying Liu, Prof., Ph.D

School of Computer Science and Technology
University of Chinese Academy of Sciences
Data Mining and High Performance Computing Lab

Outline

- Case 1: Parallel Reduction

- Case 2: Sparse Matrix-Vector Multiplication

Parallel Reduction

- ■ Common and important data parallel primitive
  - ▪ Easy to implement in CUDA
    - • Harder to get it right
- ■ Serves as a great optimization example
  - ▪ Step by step through 7 different versions
  - ▪ Demonstrates several important optimization strategies

$$3 + 1 + 7 + 0 + 4 + 1 + 6 + 3 \ = \ ?$$
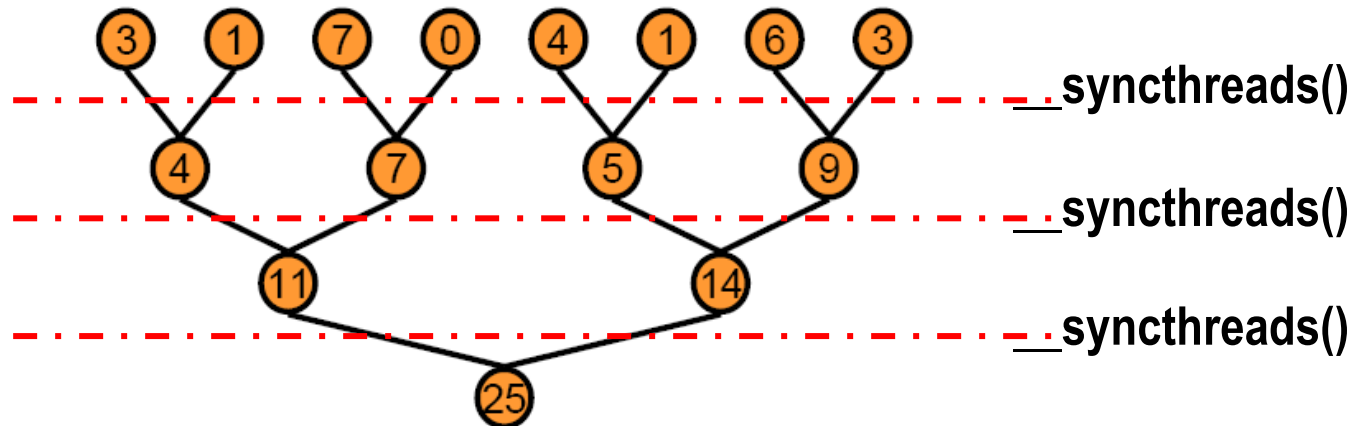
Parallel Reduction

- Tree-based approach used within each thread block
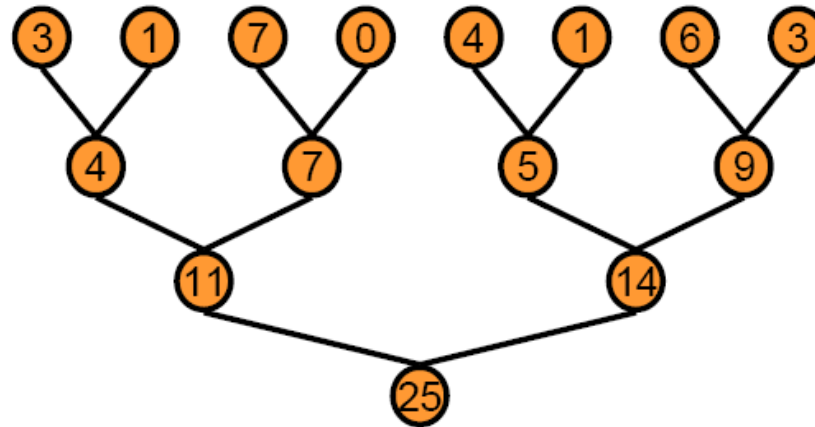
- Multiple thread blocks
  - Very large array
  - Each thread block processes a portion of the array
- How to manage communication between thread blocks?

Parallel Reduction: Global Synchronization

- Thread synchronization after pair-wise reduction

__syncthreads()

__syncthreads()

__syncthreads()

Parallel Reduction: Global Synchronization

- Problem of CUDA — not support global synchronization
  - Too many kernels, hardware synchronization is costly
  - Dead-lock

- Solution
  - Thread synchronization within thread block

Parallel Reduction: Decomposition

■ Partition the data into blocks/kernels

Level 0: 8 blocks

Level 1: 1 block
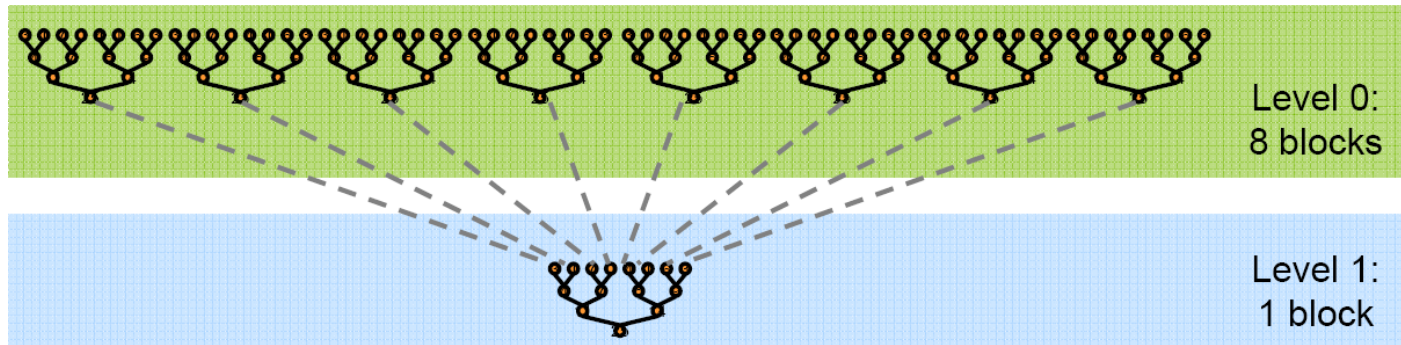
■ Code in each thread is identical

■ Iterative

Optimization Goal

- Philosophy
    - GFLOP/s: for compute-bound kernels
        - QR factorization, convolution, FIR filter, e.g.
    - Bandwidth: for memory-bound kernels
        - Database, video playback, …, e.g.
    - Both
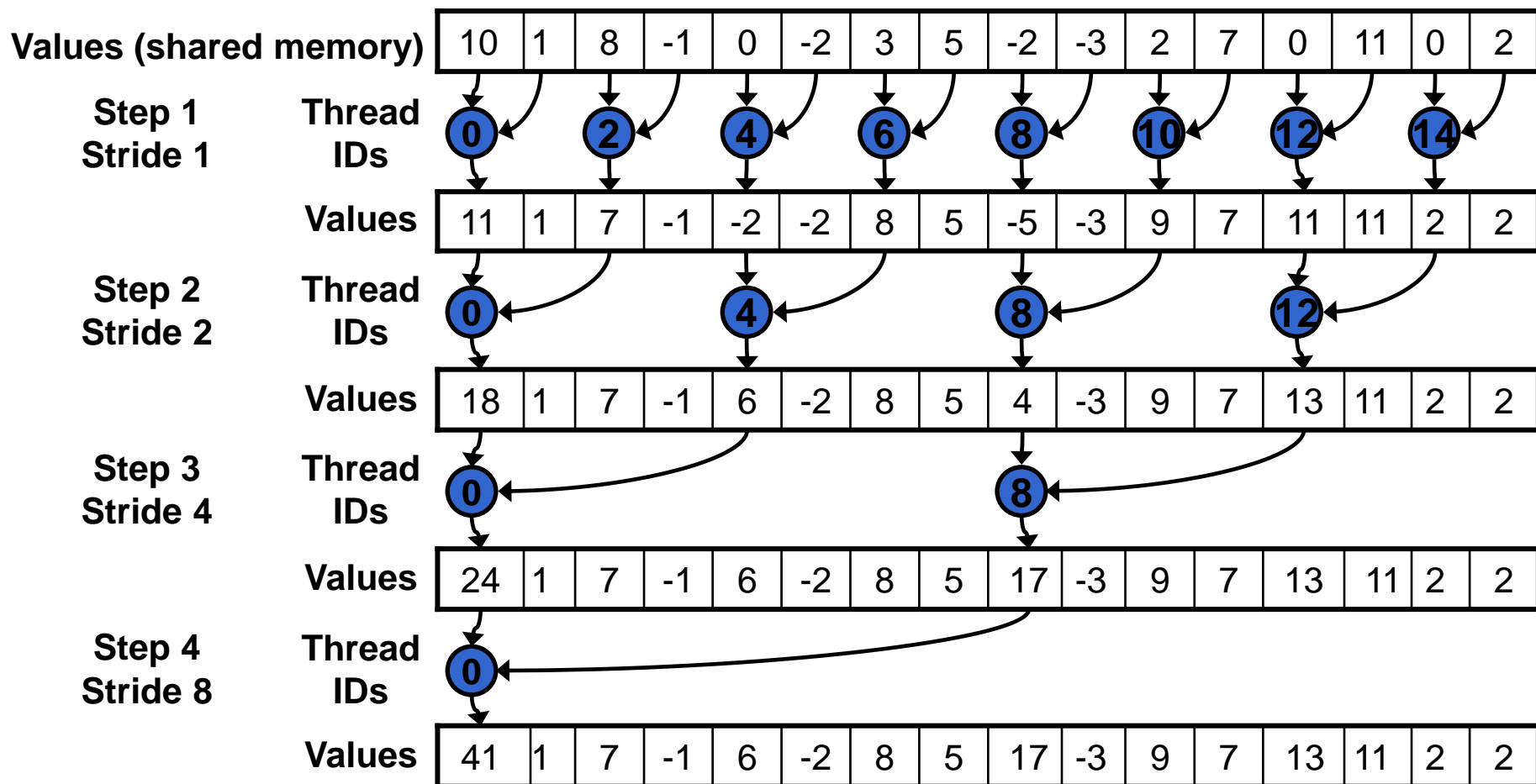   - Pattern matching, singular value decomposition, …, e.g.
- Reduction has very low computation intensity
    - 1 floating point operation / 2 elements
    - Maximize bandwidth!
- G80
    - 384-bit memory interface, 1.8GHz DDR
    - 384 * 1.8/8 = 86.4 GB/s

Reduction #1: Interleaved Addressing

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 ② ④ ⑥ ⑧ ⑩ ⑫ ⑭

Values

11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2

Step 2 Stride 2 — **Thread IDs**

⓪ ④ ⑧ ⑫

Values

18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2

Step 3 Stride 4 — **Thread IDs**

⓪ ⑧

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

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 branching results in very poor performance!
% operator is very slow

Performance for 4M Element Reduction

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

Note: Block Size = 128 threads for all tests

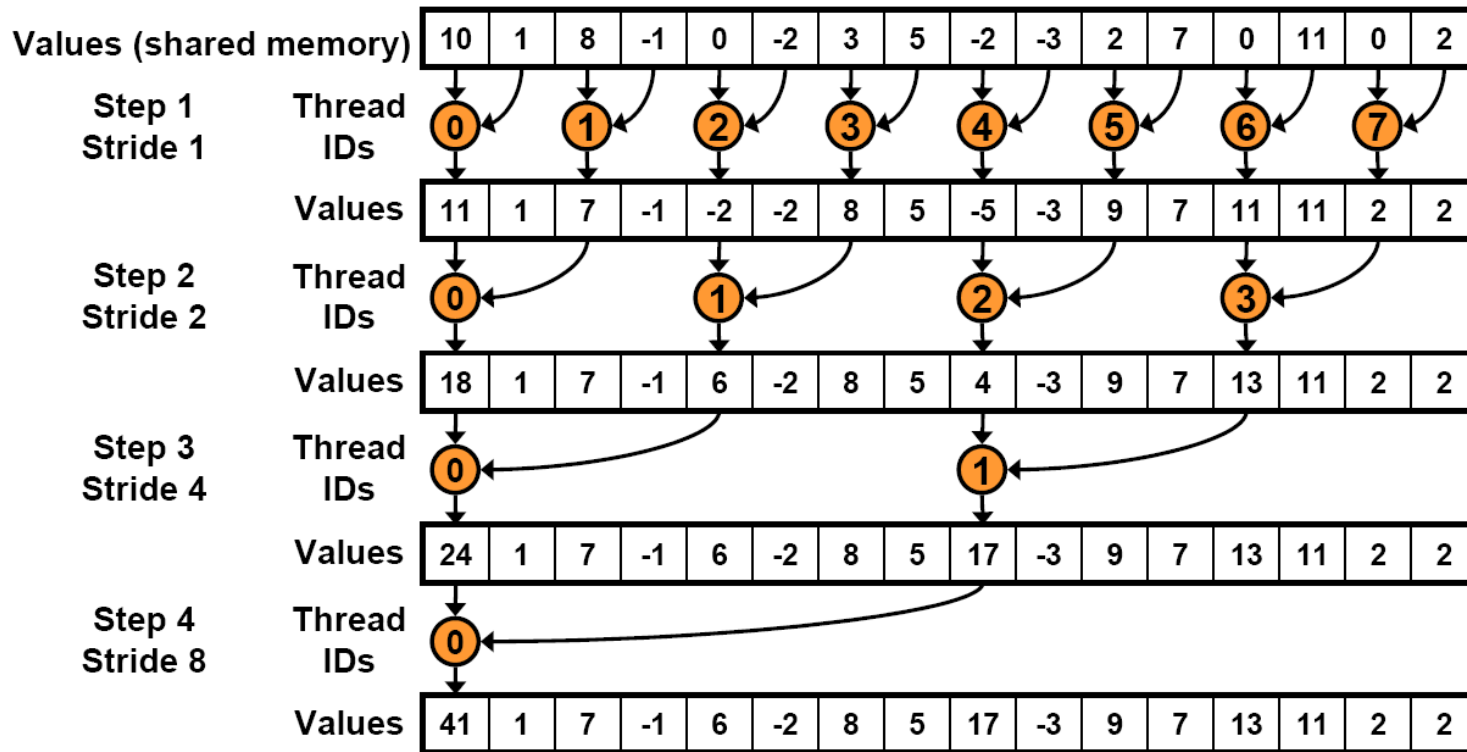# Reduction #2: Interleaved Addressing

- ## Divergence

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

- ## Avoid divergence

```
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

| 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

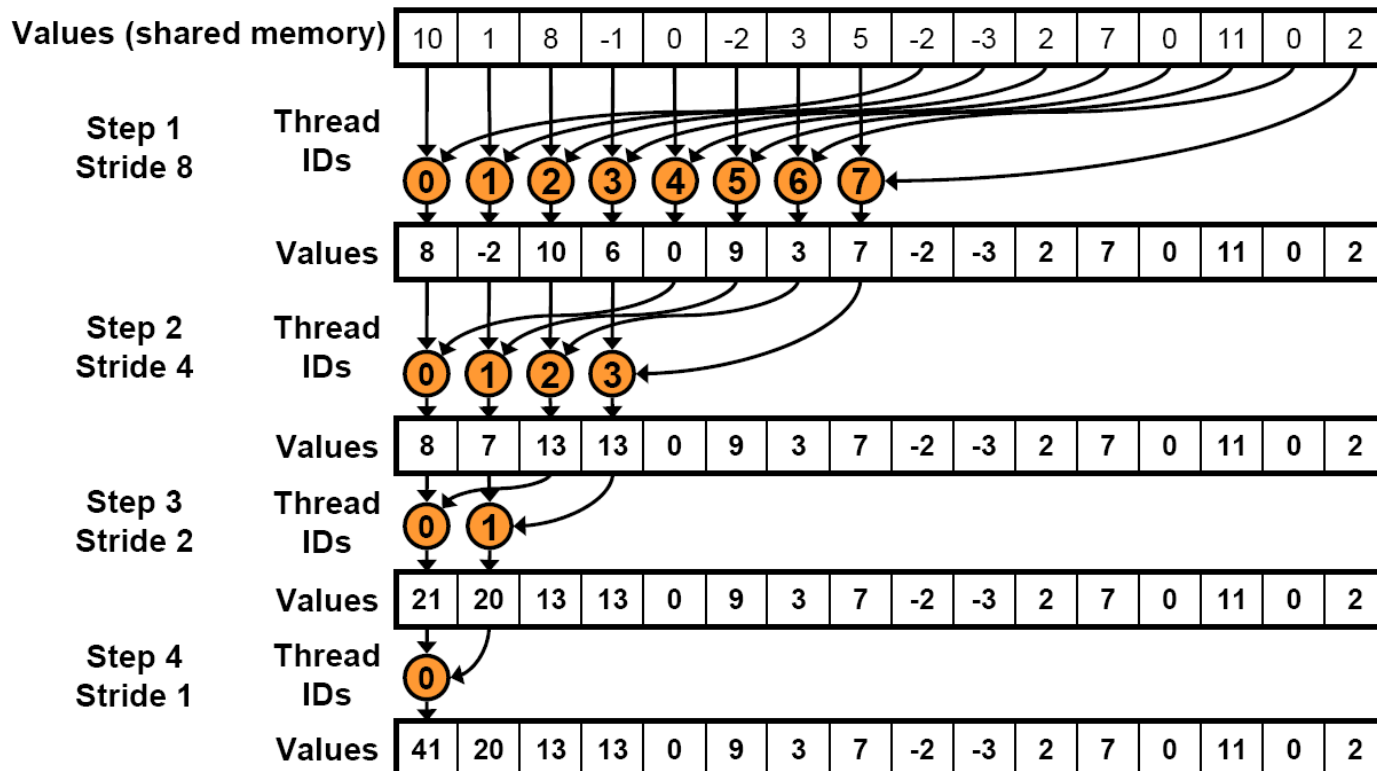| Values | 41 | 1 | 7 | -1 | 6 | -2 | 8 | 5 | 17 | -3 | 9 | 7 | 13 | 11 | 2 | 2 |

```
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();
}
```

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

Reduction #2: Interleaved Addressing

■ **Bank conflict!**

s=2

```
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();
}
```
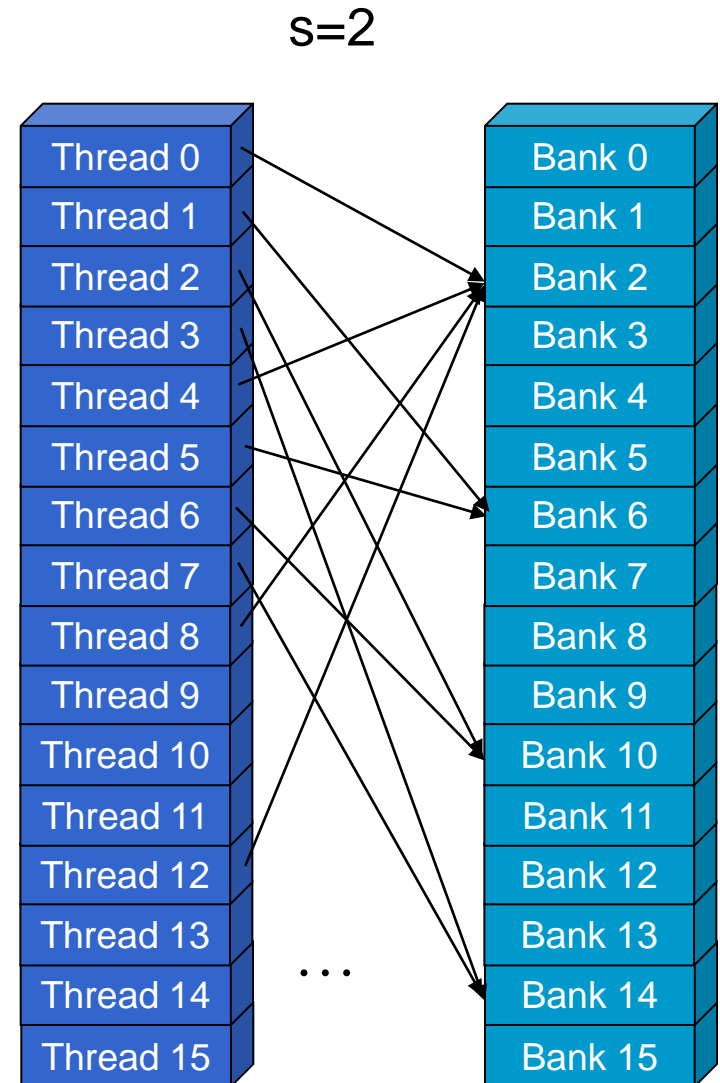
Parallel Reduction: Sequential Addressing

■ Bank 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

Observation: Idle Threads

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

- **In the first iteration, half threads are idle!**
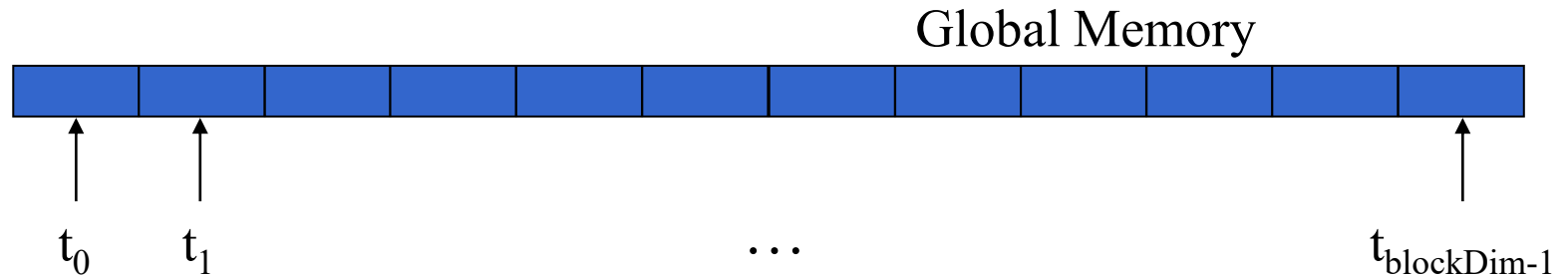 - Waste half resources …
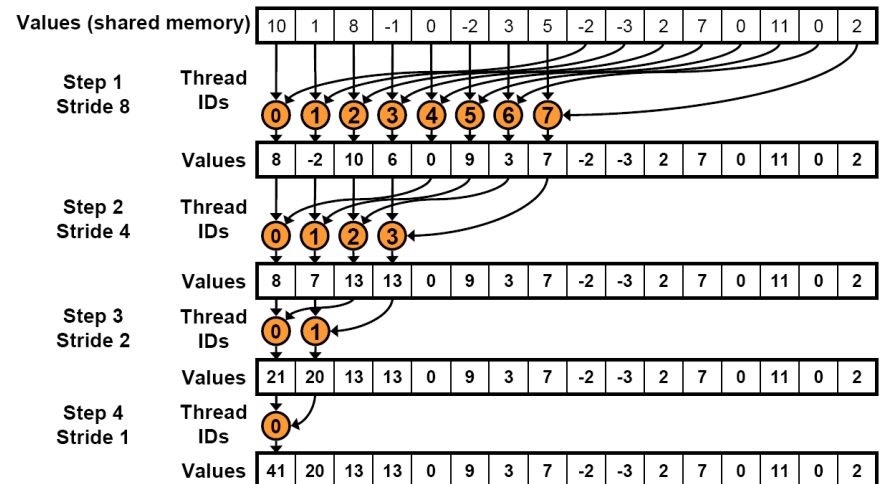
Reduction #4: First Add During Load

```
// 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();
```

- Reduce # blocks by half
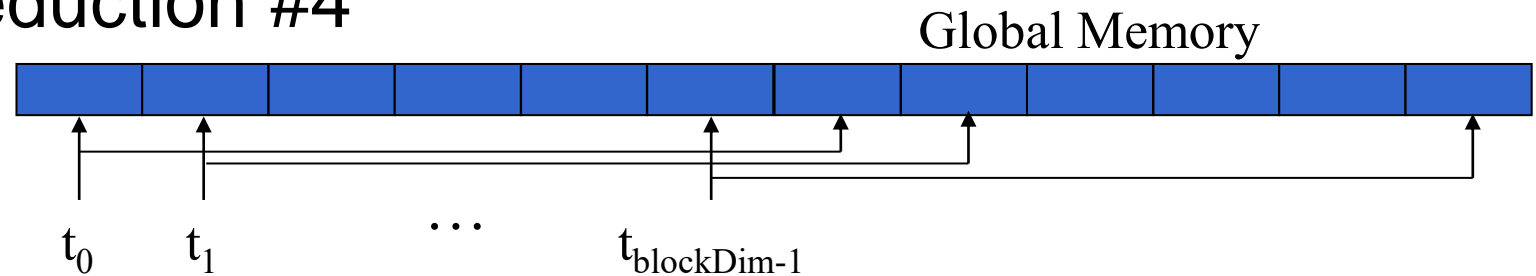- 2 global memory loads in the first add

```
// 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

Global Memory

t_0 t_1 $\ldots$ $t_{blockDim-1}$

- ## Reduction #4

Global Memory

t_0 t_1 $\ldots$ $t_{blockDim-1}$

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

Performance Bottleneck

- ## 17 GB/s << 86.4 GB/s
  - Algorithm has been optimized
  - Other operations may incur the cost
    - loads, stores
   - loops
- ## Strategy
  - Code optimization

Loop Unrolling

- # threads drops
  - When # threads <= 32, only one warp is working
    - Other warps are idle
  - We don't need "if (tid < s)" because it doesn't save any work
- Instructions are SIMD synchronous within a warp
- No synchronization is required in a warp
  - Scoreboarding automatically maintain synchronization
- Unroll the last 6 loops
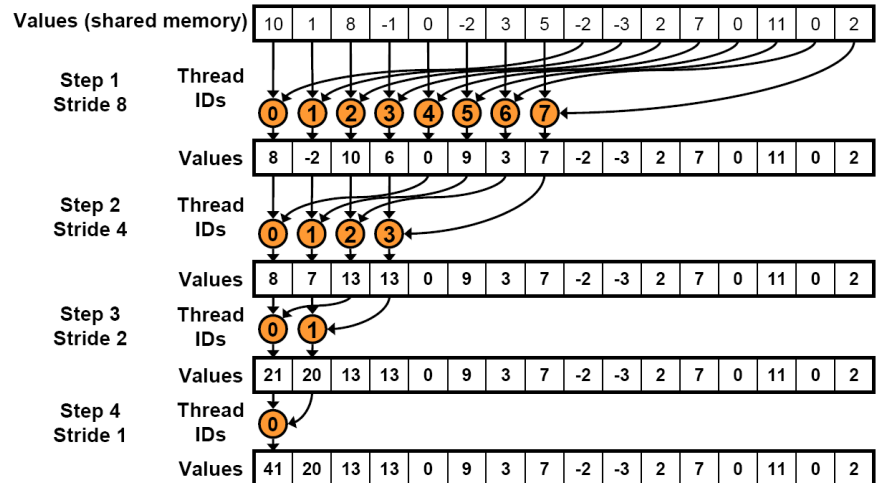
Reduction #5: Unroll the Last Warp

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```
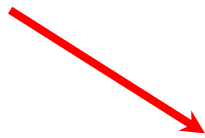
```
for (unsigned int s=blockDim.x/2; s>32; s>>=1){
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}
if (tid < 32){
    sdata[tid] += sdata[tid + 32]; sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8]; sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2]; sdata[tid] += sdata[tid + 1];
}
```

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

Completely Unrolled

- Loops may be completely unrolled
  - Assumption: the number of iterations is known!
  - Max # block threads = 512
  - 2^m threads in thread block
  - Unroll loops for 2^m threads, where $m = 0,1,2,\ldots$
- So we can easily unroll for a fixed block size
  - How can we unroll for block sizes that we don't know at compile time?
- How to make the code generic?
  - How to know # threads in a block when compiling?
- Template!
  - CUDA supports C++ template parameter

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) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

All code in RED will be evaluated at compile time

Unrolling with Templates

- Specify blocksize as a function template parameter:

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

Invoking Template Kernels

- ## Don't we still need block size at compile time?
  - ### No! 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

Reduction #7: Multiple Adds / Thread

- Replace adding 2 global memory loads in the first add

```
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();
```

- By adding multiple global memory loads

```
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();
```

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 16M elements: 72 GB/s!

```
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;
    do { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
        while (i < n);
    __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];        final version
        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];
}
```

Performance Comparison
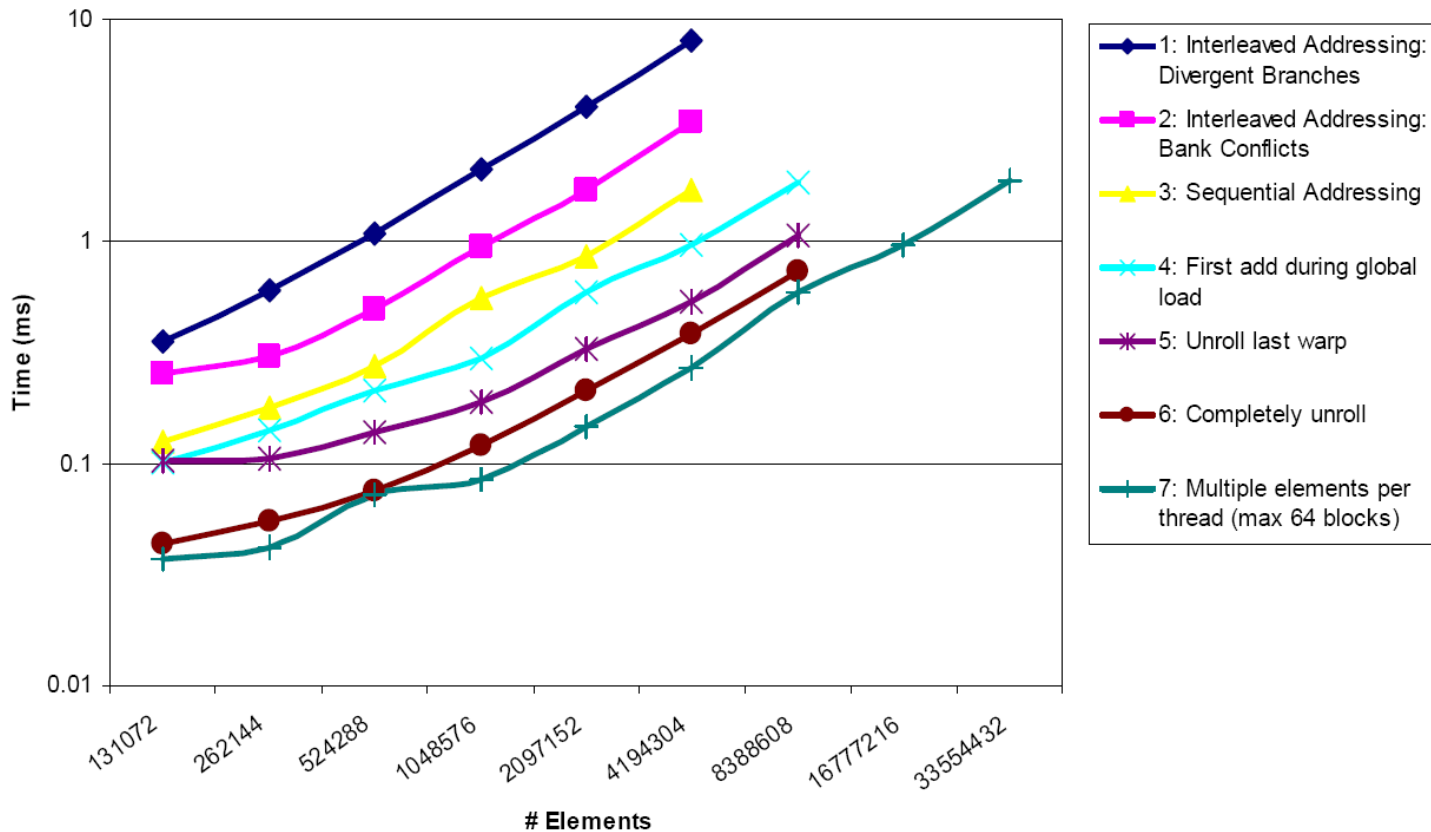
Parallel Reduction Summary

- Algorithm optimization
  - Changes to addressing, algorithm cascading
  - 8.34x speedup, combined!
- Code optimization
  - Loops Unrolling
  - 3.6x speedup, combined

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**

Outline

■ Case 1: Parallel Reduction

■ Case 2: Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication

- Sparse matrices have relatively few non-zero entries
- Frequently $O(n)$ rather than $O(n^2)$
- Only store & operate on these non-zero entries

Compressed Sparse Row (CSR) Format

Sparse Matrix-Vector Multiplication

```
float multiply_row(uint rowsize, // number of non-zeros in row
                   uint *Aj,      // column indices for row
                   float *Av,     // non-zero entries for row
                   float *x)      // the RHS vector
{
    float sum = 0;
    for (uint column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];
    return sum;
}
```

Sparse Matrix-Vector Multiplication

```
float multiply_row(uint size, uint *Aj, float *Av, float *x);


void csrmul_serial(uint *Ap, uint *Aj, float *Av, uint num_rows, float
    *x, float *y)
{
    for (uint row=0; row<num_rows; ++row)
    {
        uint row_begin = Ap[row];
        uint row_end = Ap[row+1];
        y[row] = multiply_row(row_end-row_begin,
                              Aj+row_begin,
                              Av+row_begin,
                              x);
    }
}
```

Sparse Matrix-Vector Multiplication

```
__device__ float multiply_row(uint size, uint *Aj, float *Av, float *x);


__global__ void csrmul_kernel(uint *Ap, uint *Aj, float *Av, uint
  num_rows, float *x, float *y)
{
    uint row = blockIdx.x*blockDim.x + threadIdx.x;
    if( row<num_rows )
    {
        uint row_begin = Ap[row];
        uint row_end = Ap[row+1];
        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

Using Shared Memory

```
__global__ void csrmul_shared(… … … … … …) {
    uint begin = blockIdx.x*blockDim.x, end = begin+blockDim.x;
    uint row = begin + threadIdx.x;

    __shared__ float cache[blocksize]; // array to cache rows

    if( row<num_rows) cache[threadIdx.x] = x[row]; // fetch to shared memory
        __syncthreads();

    if( row<num_rows ) {
        uint row_begin = Ap[row]; row_end = Ap[row+1]; float sum = 0;
        for(uint col=row_begin; col<row_end; ++col) {
            uint j = Aj[col];
            // Fetch from cached rows when possible
            float x_j = (j>=begin && j<end) ? cache[j-begin] : x[j];
            sum += Av[col] * x_j;
        }
        y[row] = sum;
    }
}
```