Parallel Patterns: Reduction

Prof. Seokin Hong

What is a reduction problem ?

- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
  - Average
  - Deviation

- a set of values → a single value
  - scores → maximum ?
  - prices → average price ?

Sequential Reduction Algorithm

- **Step1:** Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- **Step2:** Scan through the input and perform the reduction operation between the result value and the current input value
  - typically a single for-loop iteration !
  - O(*n*) operations !

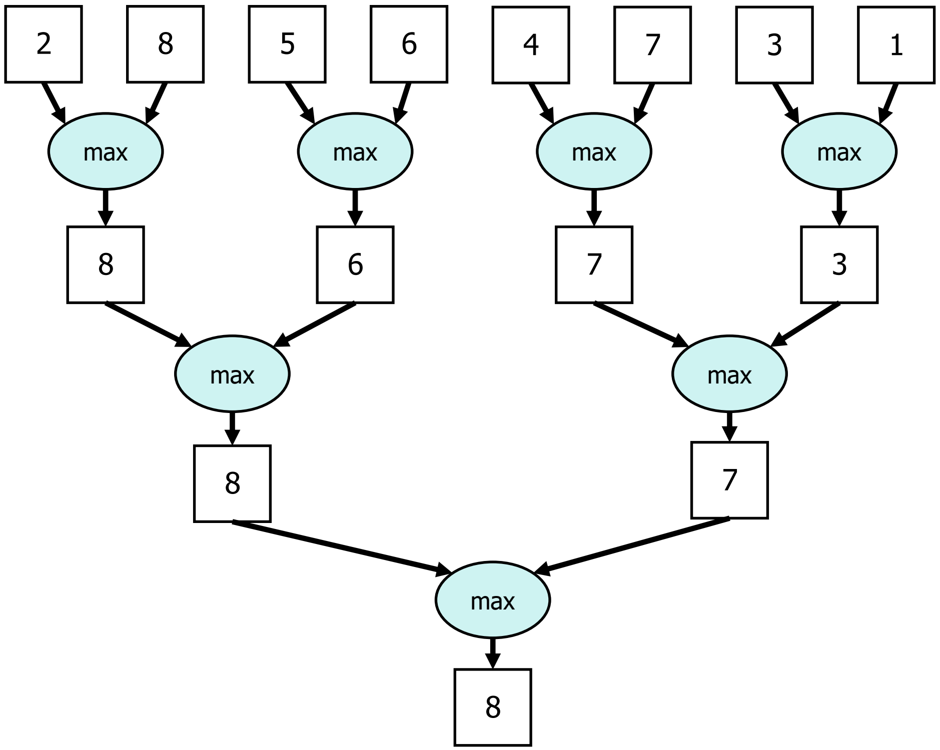
Example: Sequential Reduction

```
float  maximum = MIN_FLOAT;
for (i = 0; i < DATA_SIZE; ++i) {
    if (maximum < data[i]) {
        maximum = data[i];
    }
}
```

- Can you make any speed-up?
- Or, another way to compute it?

A Parallel Reduction Tree Algorithm

- in log n steps !



$\log_2 8 = 3$

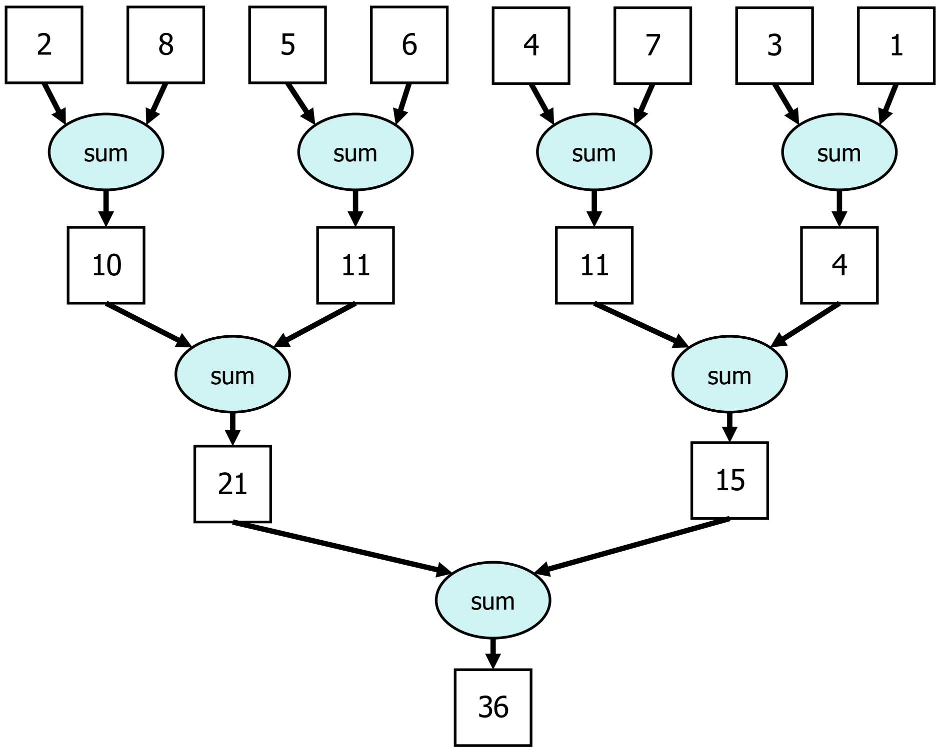
A Quick Analysis

- For N input values, the reduction tree performs
  - (1/2)N + (1/4)N + (1/8)N + … (1/N) = (1 – (1/N)) N = N – 1 operations

- $\log_2 n$ steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources

A Parallel Sum Reduction

- in log n steps !



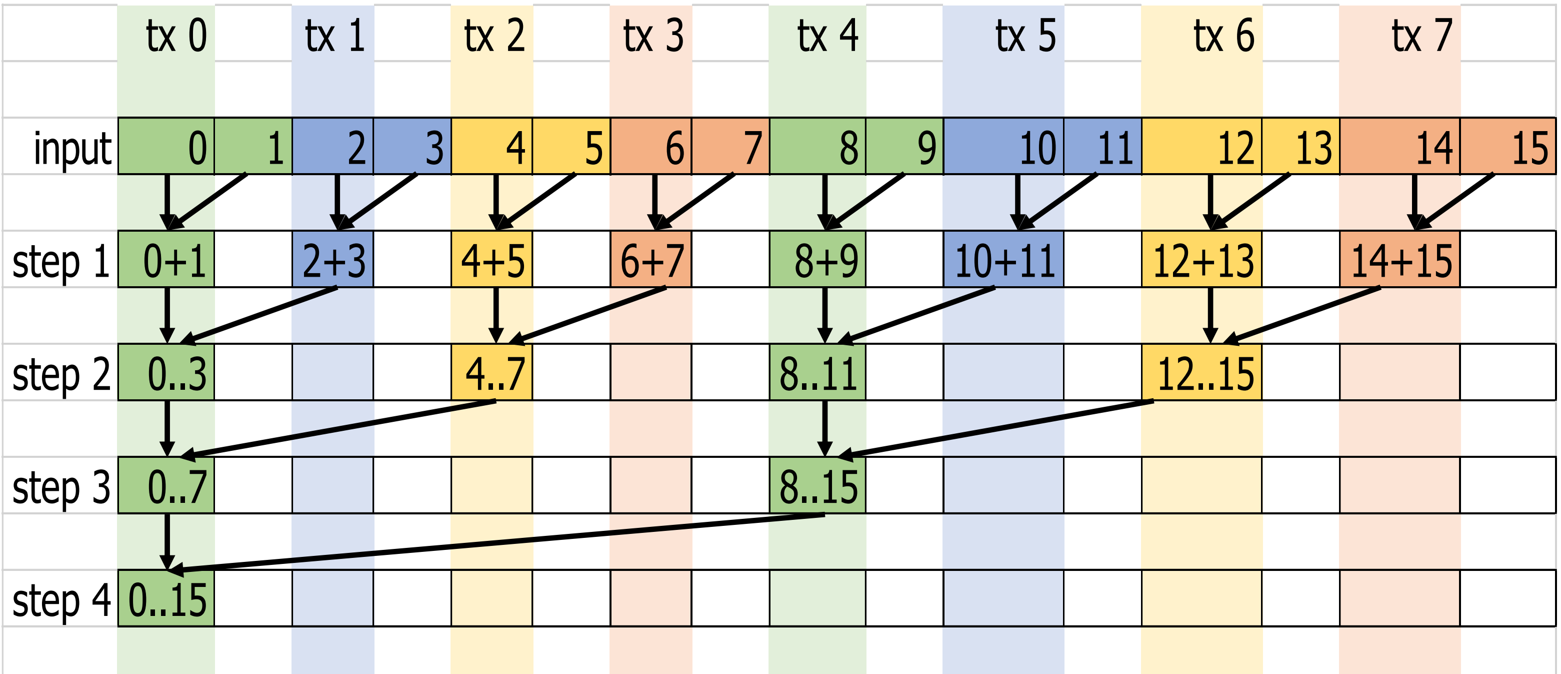
$\log_2 8 = 3$

A Sum Reduction Example

- Parallel implementation:
  - Recursively **halve # of threads**, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the final sum
  - The final sum will be in element 0
  - Reduces global memory traffic due to partial sum values

A Sum Reduction Example

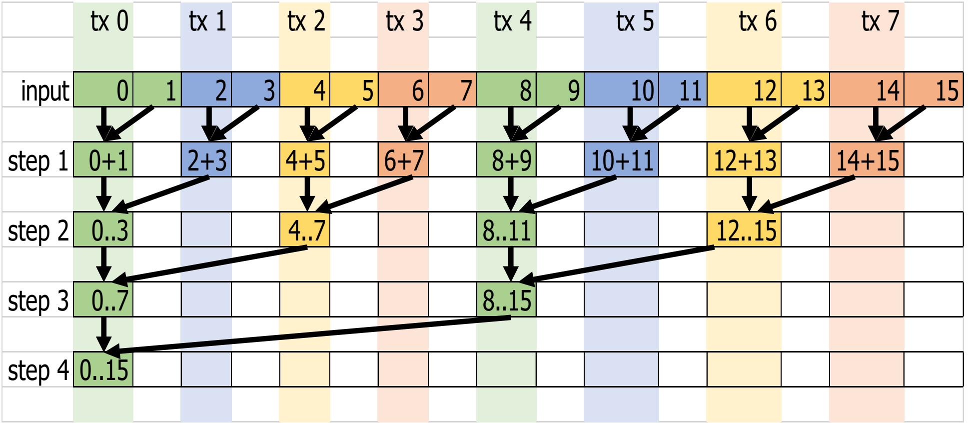


Simple Thread Index to Data Mapping

- Each thread is responsible of an even-index location of the partial sum vector

- After each step, **half of the threads are no longer needed**

- In each step, one of the inputs comes from an increasing distance away

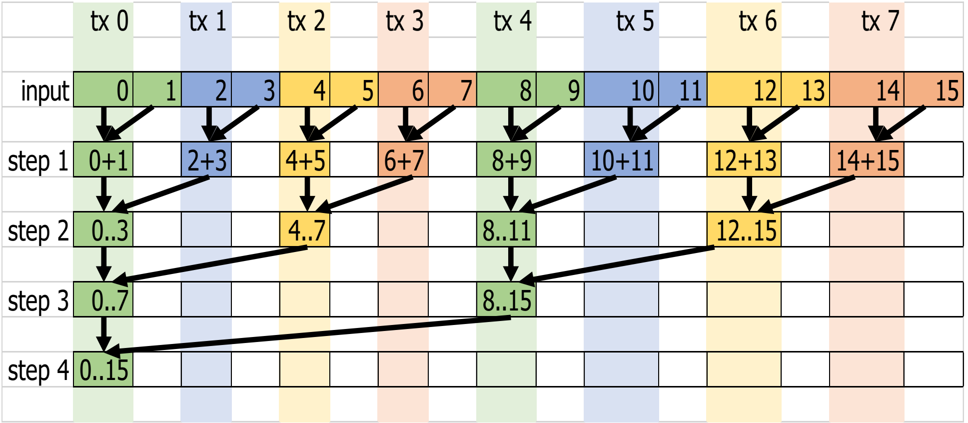


A Simple Thread Block Design

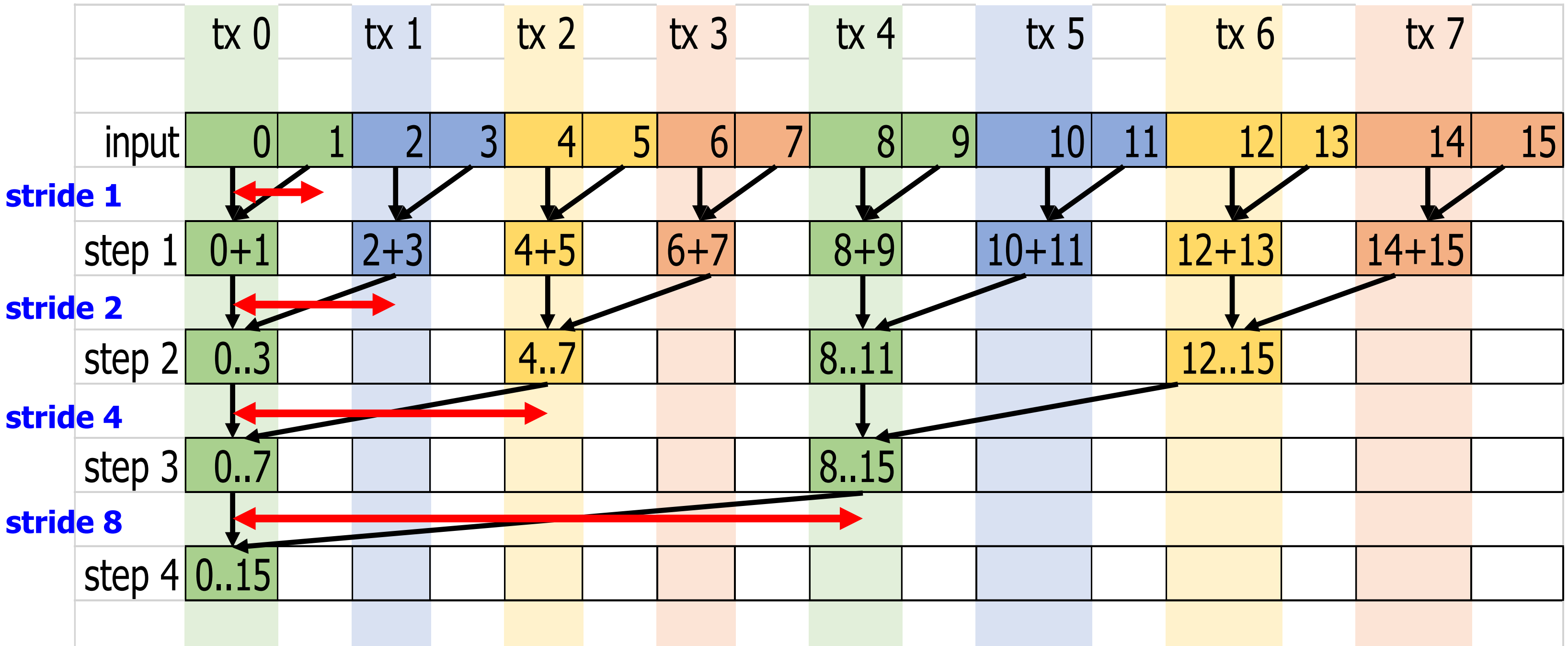
- Each thread block takes 2* BlockDim input elements
- Each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];
```

```
partialSum[tx] = input[tx];
partialSum[BLOCK_SIZE+tx] = input[BLOCK_SIZE + tx];
```

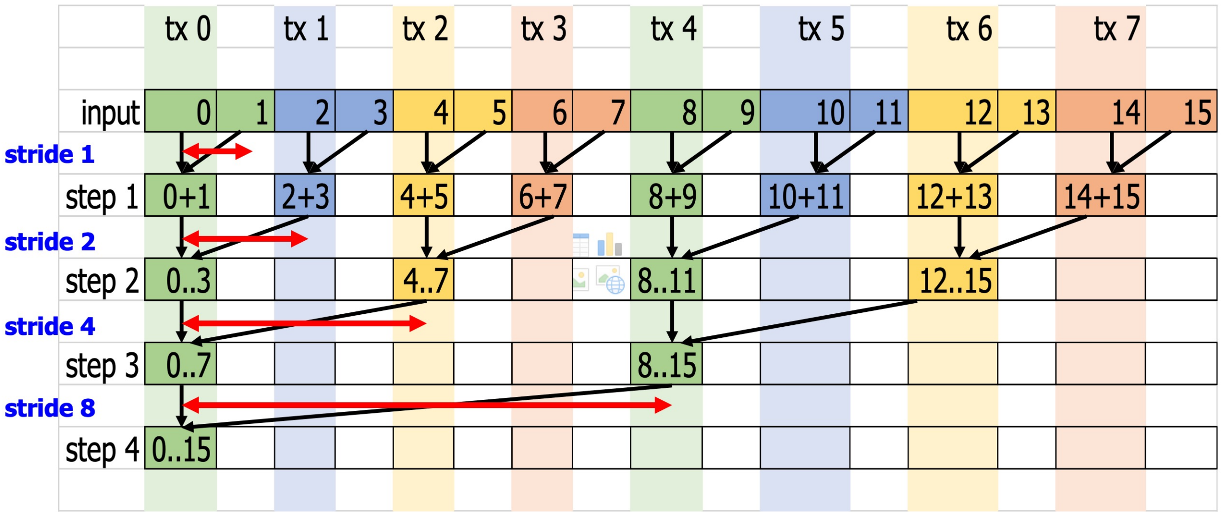


A Sum Reduction Example



The Reduction Steps

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



- Why do we need __syncthreads()?

A Sum Reduction

- **single thread-block** implementation
  - 1 block, with 1024 threads

```
#include <stdio.h>
#include <stdlib.h>
using namespace std;

#define GRIDSIZE        1
#define BLOCKSIZE       1024
#define TOTALSIZE       (GRIDSIZE * BLOCKSIZE)

void genData(…) { … }
```

A Sum Reduction

- kernel program

```
__global__ void kernel(unsigned* pData, unsigned* pAnswer) {
        __shared__ unsigned dataShared[2 * BLOCKSIZE];
        // each thread loads two elements from global to shared memory
        unsigned tx = threadIdx.x;
        dataShared[tx] = pData[tx];
        dataShared[BLOCKSIZE + tx] = pData[BLOCKSIZE + tx];
        // do reduction in the shared memory
        for (register unsigned stride = 1; stride <= BLOCKSIZE; stride *= 2) {
                __syncthreads();
                if (tx % stride == 0) {
                        dataShared[2*tx] += dataShared[2*tx + stride];
                }
        }
        // final synchronize
        __syncthreads();
        if (tx == 0) {
                pAnswer[tx] = dataShared[tx];
        }
}
```

A Sum Reduction

- Host code
  - data on the host and device

```
int main(void) {
        unsigned* pData = NULL;
        unsigned answer = 0;
        // malloc memories on the host-side
        pData = (unsigned*)malloc(2 * BLOCKSIZE * sizeof(unsigned));
        // generate source data
        genData(pData, 2 * BLOCKSIZE);
        // CUDA: allocate device memory
        unsigned* pDataDev;
        unsigned* pAnswerDev;
        cudaMalloc((void**)&pDataDev, 2 * BLOCKSIZE * sizeof(unsigned));
        cudaMalloc((void**)&pAnswerDev, 1 * sizeof(unsigned));
        cudaMemset(pAnswerDev, 0, 1 * sizeof(unsigned));
```

A Sum Reduction

- Host code
  - copy the data from host
  - kernel launch

```
// CUDA: copy from host to device
cudaMemcpy(pDataDev, pData, 2 * BLOCKSIZE * sizeof(unsigned), cudaMemcpyHostToDevice);
// start timer
system_clock::time_point start = system_clock::now();
// CUDA: launch the kernel
dim3 dimGrid(GRIDSIZE, 1, 1);
dim3 dimBlock(BLOCKSIZE, 1, 1);
kernel <<< dimGrid, dimBlock>>>(pDataDev, pAnswerDev);
// end timer
system_clock::time_point end = system_clock::now();
nanoseconds  du = end - start;
printf("%lld nano-seconds\n", du);
```

A Sum Reduction

- Host code
  - copy from device to host
  - print the result

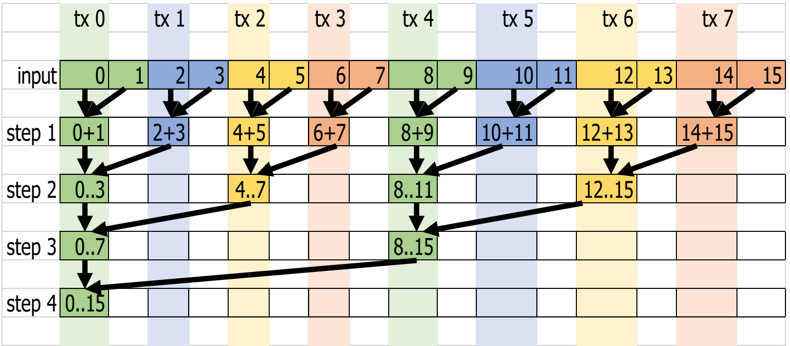
```
        // CUDA: copy from device to host
        cudaMemcpy(&answer, pAnswerDev, 1 * sizeof(unsigned),
                                        cudaMemcpyDeviceToHost);
        printf("answer = %u\n", answer);
        // CUDA: free the memory
        cudaFree(pDataDev);
        cudaFree(pAnswerDev);
        // free the memory
        free(pData);
}
```

Some Observations

- In each iteration, two control flow paths will be sequentially traversed
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources

- No more than half of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization
  - Some warps will still succeed, but with divergence since only one thread will succeed

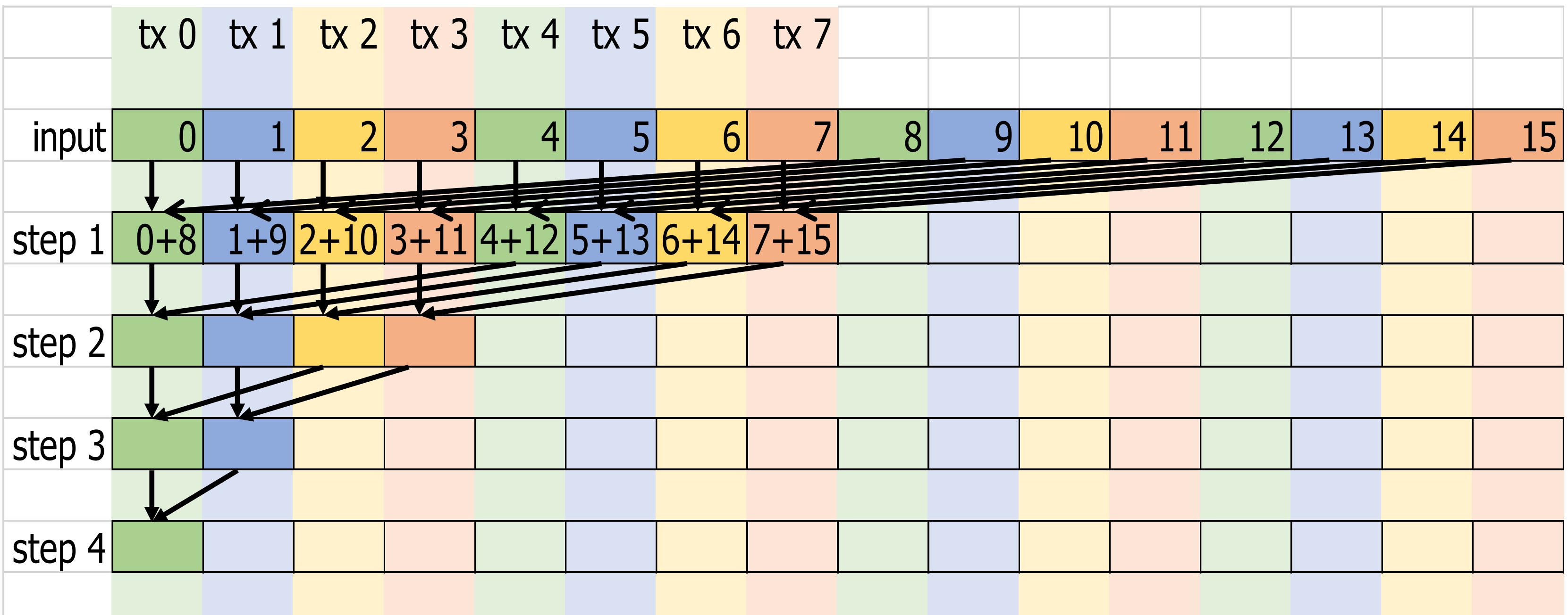
```
// do reduction in the shared memory
for (register unsigned stride = 1; stride <= BLOCKSIZE; stride *= 2) {
        __syncthreads();
        if (tx % stride == 0) {
                dataShared[2*tx] += dataShared[2*tx + stride];
        }
}
```



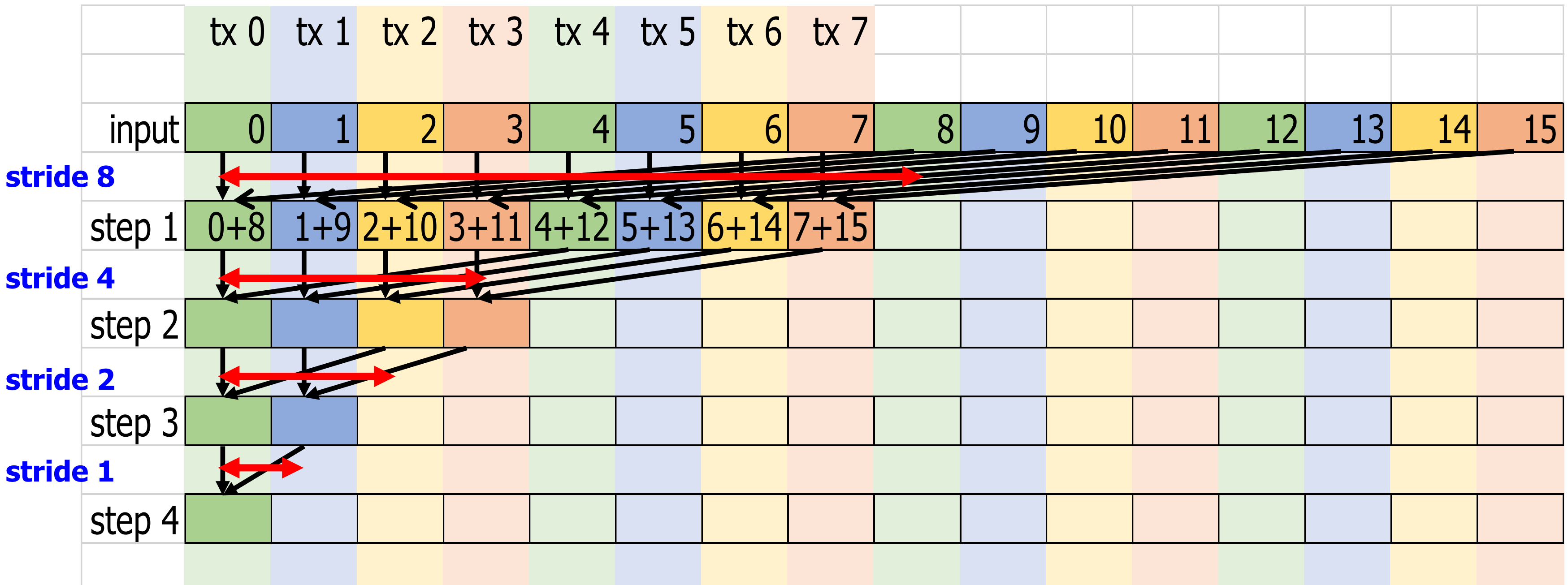
A Better Strategy for Sum Reduction

- Always compact the partial sums into the first locations in the partialSum[] array

- Keep the active threads consecutive



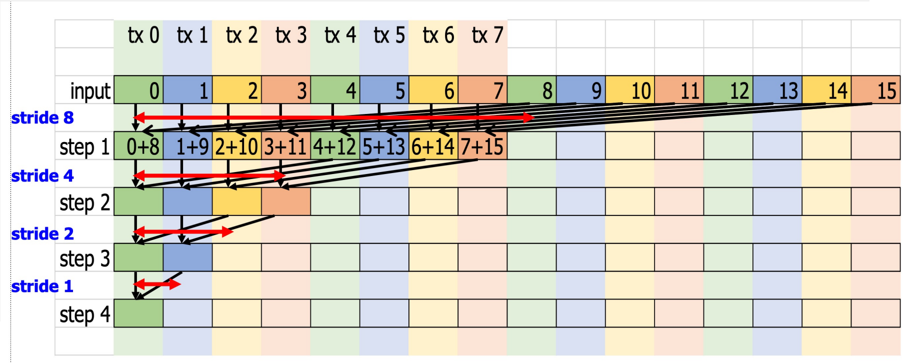
A Better Strategy for Sum Reduction (Cont'd)



A Better Strategy for Sum Reduction (Cont'd)

- Kernel

```
__global__ void kernel(unsigned* pData, unsigned* pAnswer) {
        __shared__ unsigned dataShared[2 * BLOCKSIZE];
        // each thread loads two elements from global to shared memory
        unsigned tx = threadIdx.x;
        dataShared[tx] = pData[tx];
        dataShared[BLOCKSIZE + tx] = pData[BLOCKSIZE + tx];
        // do reduction in the shared memory
        for (register unsigned stride = BLOCKSIZE; stride > 0; stride /= 2) {
                __syncthreads();
                if (tx < stride) {
                        dataShared[tx] += dataShared[tx + stride];
                }
        }       // final synchronize
        __syncthreads();
        if (tx == 0) {
                pAnswer[tx] = dataShared[tx];
        }
}
```



A Quick Analysis

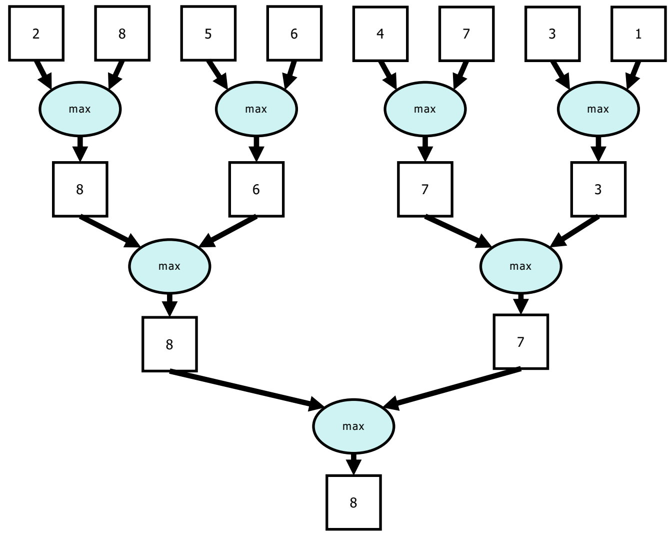
- For a 1024 thread block
  - No divergence in the first 5 steps
  - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step

- less than or equal to 32 threads → a warp
  - The final 5 steps will still have divergence
    - 16,8,4,2,1 threads within a warp

Big-Size Reduction

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

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

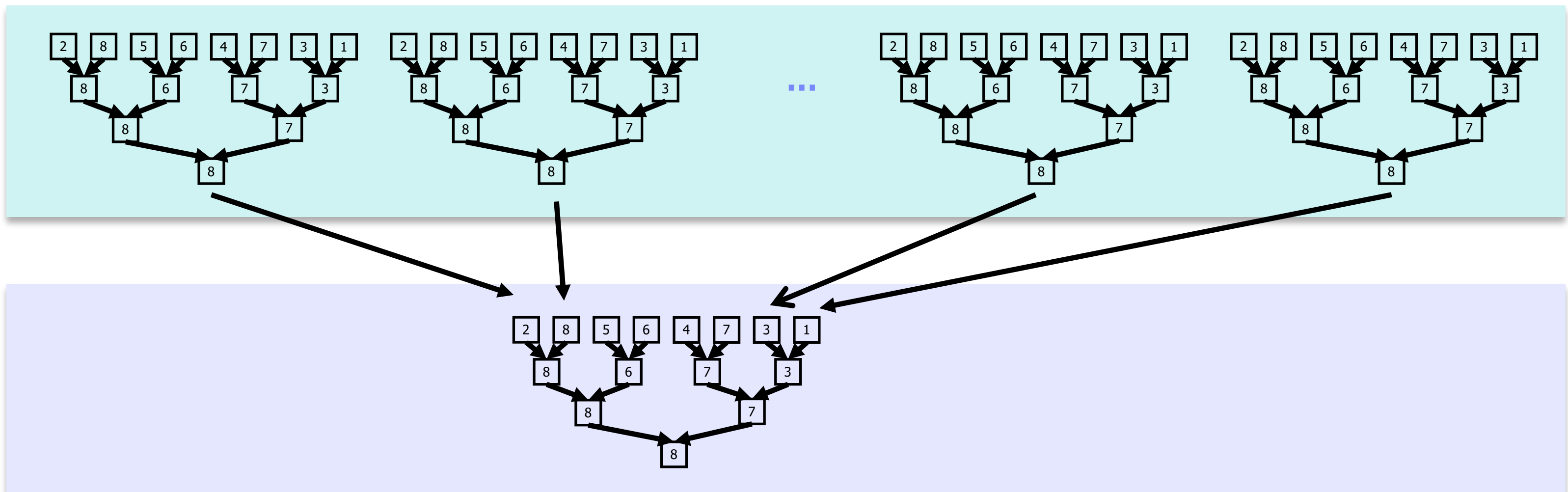
- **Other possible solution**
  - Use atomics at the end of thread block-level reduction

Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations

level 0
8 kernels

level 1
1 kernel



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

Sample Problem: Total Sum

- generate 32M integers, with values between 0 and 100.
- problem: sum up all 32M integers

```
#define GRIDSIZE      (32 * 1024)
#define BLOCKSIZE      1024
#define TOTALSIZE     (GRIDSIZE * BLOCKSIZE)
```

- a kind of reduction problem
  - CUDA will sum up in a tournament manner
  - for simplicity, CUDA will use **atomicAdd( )** for cross-thread-block communication

C++ version : sequential add

```
void genData(unsigned* ptr, unsigned size) {
    while (size--) {
        *ptr++ = (unsigned)(rand() % 100);
    }
}

void kernel(unsigned* pData, unsigned* pAnswer, unsigned size) {
    register unsigned answer = 0;
    while (size--) {
        answer += *pData++;
    }
    *pAnswer = answer;
}
```

Execution Result

```
int main(void) {
    unsigned* pData = NULL;
   …
    pData = (unsigned*)malloc(TOTALSIZE * sizeof(unsigned));
    genData(pData, TOTALSIZE);
    kernel(pData, &answer, TOTALSIZE);
   …
}
```

- C++ version

elapsed time = 69.662972 msec

answer = 1659731932

Total Sum – atomic operation

- update the global variable with atomic operation
- generate 32M threads
  - each thread **add the element to the global variable**
  - is it fast ?

```
__global__ void kernel(unsigned* pData, unsigned* pAnswer) {
    register unsigned i = blockIdx.x * blockDim.x + threadIdx.x;
    register unsigned fValue = pData[i];
    atomicAdd(pAnswer, fValue);
}
```

Total Sum – atomic operation

```
int main(void) {
    …
    unsigned* pDataDev;
    unsigned* pAnswerDev;
    cudaMalloc((void**)&pDataDev, TOTALSIZE * sizeof(unsigned));
    cudaMalloc((void**)&pAnswerDev, 4 * sizeof(unsigned));
    cudaMemset(pAnswerDev, 0, 4 * sizeof(unsigned));
    dim3 dimGrid(GRIDSIZE, 1, 1);
    dim3 dimBlock(BLOCKSIZE, 1, 1);
    kernel<<<dimGrid, dimBlock>>>(pDataDev, pAnswerDev);
    …
}
```

Execution Result

- atomic add version

elapsed time = 406.730682 msec

answer = 1659731932

- C++ version (current winner)

elapsed time = 69.662972 msec

answer = 1659731932

Total Sum – Shared Memory

- One thread block (TB) = 1,024 threads
- load and atomic add the 1,024 elements on the SM

```
global__ void kernel(unsigned* pData, unsigned* pAnswer) {
    __shared__ unsigned answerShared;
    if (threadIdx.x == 0) { answerShared = 0.0F; }

    __syncthreads();
    register unsigned i = blockIdx.x * blockDim.x + threadIdx.x;
    register unsigned value = pData[i];   // register ← global memory
    atomicAdd(&answerShared, value);      // update the shared variable
    __syncthreads();

    if (threadIdx.x == 0) {
            atomicAdd(pAnswer, answerShared);     // update the global variable
    }
}
```

Execution Result

- shared memory version

elapsed time = 316.056274 msec

answer = 1659731932

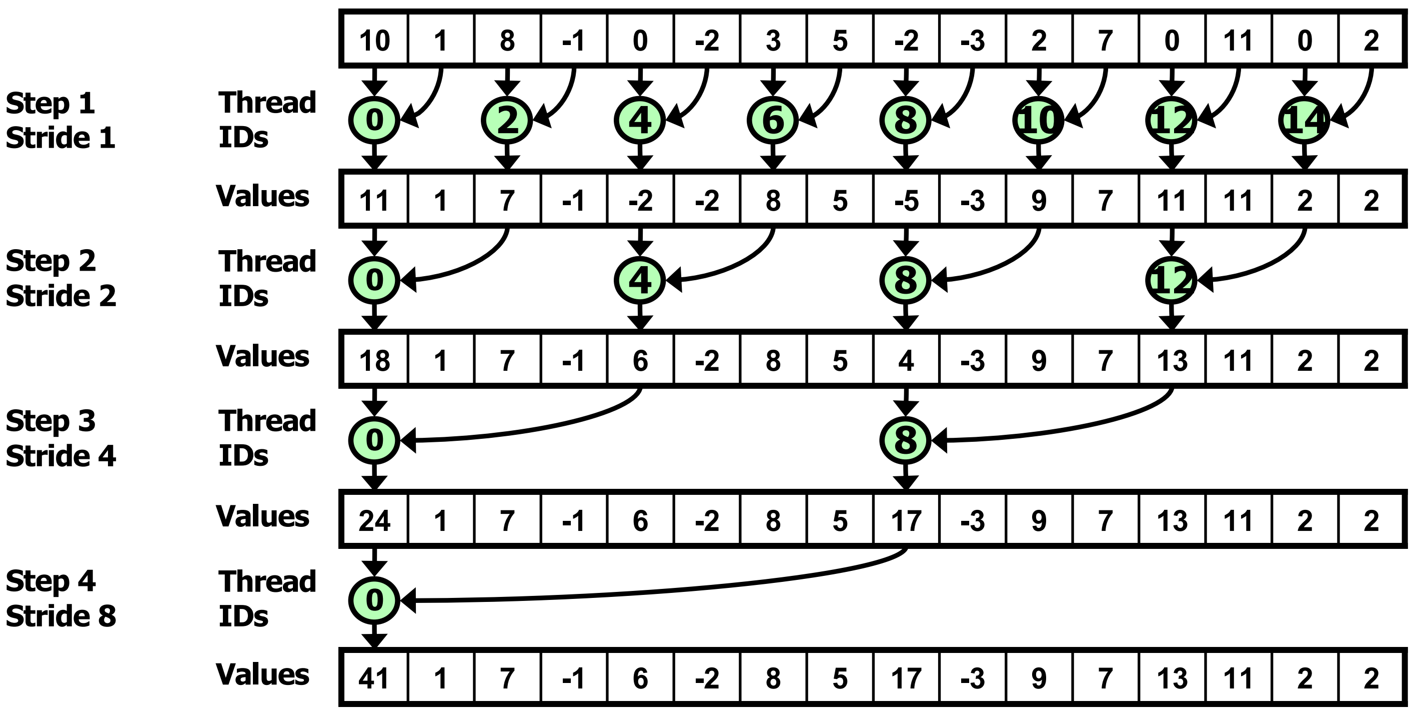
- C++ version (current winner)

elapsed time = 69.662972 msec

answer = 1659731932

Parallel Reduction: Interleaved Addressing

- summing up with a tree topology



Total Sum – Reduction

- parallel load 1,024 elements,
- then, sum up 1,024 elements in a tournament manner

```
__global__ void kernel(unsigned* pData, unsigned* pAnswer) {
    __shared__ unsigned dataShared[BLOCKSIZE];
    // each thread loads one element from global to shared memory
    register unsigned i = blockIdx.x * blockDim.x + threadIdx.x;
    register unsigned tid = threadIdx.x;
    dataShared[tid] = pData[i];
    __syncthreads();
    // do reduction in the shared memory
    for (register unsigned s = 1; s < BLOCKSIZE; s *= 2) {
            if (tid % (2*s) == 0) {
                    dataShared[tid] += dataShared[tid + s];
            }
            __syncthreads();
    }
    // add the partial sum to the global answer
    if (tid == 0) {
            atomicAdd(pAnswer, dataShared[0]);
    }
}
```

Execution Result

- reduction version (new winner !)

elapsed time = 18.387808 msec

answer = 1659731932

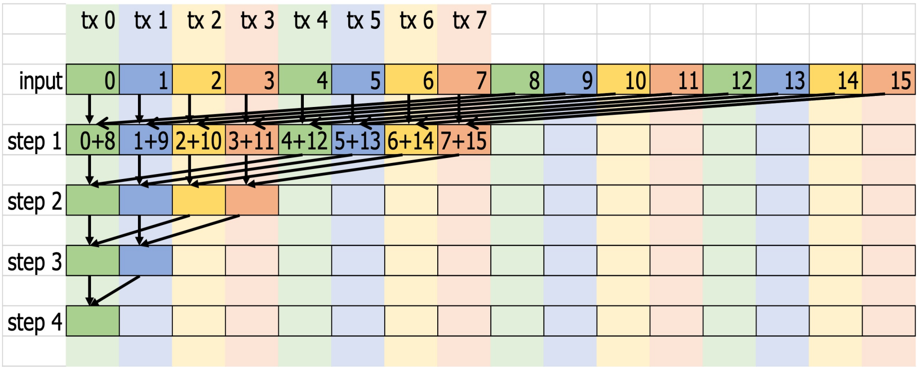
- C++ version (current winner)

elapsed time = 69.662972 msec

answer = 1659731932

Total Sum – Reversed Version

```
__global__ void kernel(unsigned* pData, unsigned* pAnswer) {
    __shared__ unsigned dataShared[BLOCKSIZE];
    // each thread loads one element from global to shared memory
    register unsigned i = blockIdx.x * blockDim.x + threadIdx.x;
    register unsigned tid = threadIdx.x;
    dataShared[tid] = pData[i];
    __syncthreads();
    // do reduction in the shared memory
    for (register unsigned s = BLOCKSIZE / 2; s > 0; s >>= 1) {
            if (tid < s) {
                    dataShared[tid] += dataShared[tid + s];
            }
            __syncthreads();
    }
    // add the partial sum to the global answer
    if (tid == 0) { atomicAdd(pAnswer, dataShared[0]); }
}
```



Execution Result

- reversed version (new winner !)

elapsed time = 15.488288 msec

answer = 1659731932

- reduction version (previous winner)

elapsed time = 18.387808 msec

answer = 1659731932

- C++ version

elapsed time = 69.662972 msec

Idle Threads

- Problem:

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

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

- This is wasteful…

First Add during Load

- With two loads and first add of the reduction:

```
__global__ void kernel(unsigned* pData, unsigned* pAnswer) {
    __shared__ unsigned dataShared[BLOCKSIZE];
    // each thread loads one element from global to shared memory
    register unsigned tid = threadIdx.x;
    register unsigned i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
    dataShared[tid] = pData[i] + pData[i + blockDim.x];
    __syncthreads();
    // do reduction in the shared memory
    for (register unsigned s = BLOCKSIZE / 2; s > 0; s >>= 1) {
            if (tid < s) {
                    dataShared[tid] += dataShared[tid + s];
            }
            __syncthreads();
    }
    // add the partial sum to the global answer
    if (tid == 0) { atomicAdd(pAnswer, dataShared[0]); }
}
```

First Add during Load

```
int main(void) {
  …
   // CUDA: launch the kernel
   dim3 dimGrid(GRIDSIZE / 2, 1, 1);
   dim3 dimBlock(BLOCKSIZE, 1, 1);
   kernel<<<dimGrid, dimBlock>>>(pDataDev, pAnswerDev);
  …
}
```

Execution Result

- first add version (new winner !)

elapsed time = 8.359968 msec

answer = 1659731932

- reversed version (previous winner)

elapsed time = 15.488288 msec

answer = 1659731932

- C++ version

elapsed time = 69.662972 msec

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

Unrolling the Last Warp (Cont'd)

```
__global__ void kernel(unsigned* pData, unsigned* pAnswer) {
    __shared__ unsigned dataShared[BLOCKSIZE];
    // each thread loads one element from global to shared memory
    register unsigned tid = threadIdx.x;
    register unsigned i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
    dataShared[tid] = pData[i] + pData[i + blockDim.x];
    __syncthreads();
    // do reduction in the shared memory
    for (register unsigned s = BLOCKSIZE / 2; s > 32; s >>= 1) {
            if (tid < s) {
                    dataShared[tid] += dataShared[tid + s];
            }
            __syncthreads();
    }
```

Unrolling the Last Warp (Cont'd)

```
    // unroll the last warp
    if (tid < 32) {
            warpReduce(dataShared, tid);
            // add the partial sum to the global answer
            if (tid == 0) {
                    atomicAdd(pAnswer, dataShared[0]);
            }
    }
}
```

Unrolling the Last Warp (Cont'd)

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

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

volatile keyword

- example: volatile int k;
  - meaning: variable k can be updated by an external device/process/thread.
  - effect: no optimization by the compiler
  - can be used in C / C++ / Java / CUDA

- C++/CUDA solution:
  - volatile int a[256];
- C++/CUDA example code:
  - a[i] = a[i] + a[i + 32];
  - a[i] = a[i] + a[i + 16];
- C++/CUDA compiler will optimize it as:
  - a[i] = a[i] + a[i + 32] + a[i + 16];
  - failure if a[i + 16] is updated by another process

Execution Result

- unrolled version (new winner !)

elapsed time = 6.175648 msec

answer = 1659731932

- first add version (previous winner)

elapsed time = 8.359968 msec

answer = 1659731932

- C++ version

elapsed time = 69.662972 msec

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 1,024 threads
  - Also, we are sticking to power-of-2 block sizes

Complete Unrolling

```
// do reduction in the shared memory
if (tid < 512) {
dataShared[tid] += dataShared[tid + 512];
__syncthreads();
if (tid < 256) {
        dataShared[tid] += dataShared[tid + 256];
        __syncthreads();
        if (tid < 128) {
                dataShared[tid] += dataShared[tid + 128];
                __syncthreads();
                if (tid < 64) {
                        dataShared[tid] += dataShared[tid + 64];
                        __syncthreads();
                        if (tid < 32) {
                                dataShared[tid] += dataShared[tid + 32];
                                dataShared[tid] += dataShared[tid + 16];
                                dataShared[tid] += dataShared[tid + 8];
                                dataShared[tid] += dataShared[tid + 4];
                                dataShared[tid] += dataShared[tid + 2];
                                dataShared[tid] += dataShared[tid + 1];
                                if (tid == 0) {
                                        atomicAdd(pAnswer, dataShared[0]);
                                }
                        }
                }
        }
}
}
```

Execution Result

- completely unrolled version (new winner !)

elapsed time = 5.603456 msec

answer = 1659731932

- unrolled version (previous winner)

elapsed time = 6.175648 msec

answer = 1659731932

- C++ version

elapsed time = 69.662972 msec

Types of optimization

- Algorithmic optimizations
  - Changes in addressing
  - 11.84x speedup

- Code optimizations
  - Loop unrolling
  - 2.54x speedup