Optimizing Reduction Kernels

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

March 25, 2020

Recap

- The Host-Kernel Model for CPU-GPU Systems
- The CUDA programming language
- Mapping multi-dimensional kernels to multi-dimensional data

Recap

- Querying device properties
- The concept of scheduling warps
- Performance bottlenecks
    - Branch Divergence
    - Global memory accesses

Parallel Patterns

- Matrix Multiplication (Gather Operation)
- Convolution (Stencil Operation)
- **Reduction**

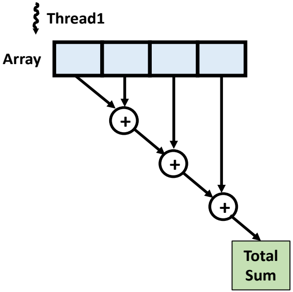
Reduction Algorithm

- Reduce vector to a single value via an associative operator
- Example: sum, min, max, average, AND, OR etc.
- Visits every element in the array
- Large arrays motivate parallel execution of the reduction
- Not compute bound but memory bound

Serial and Parallel Implementation

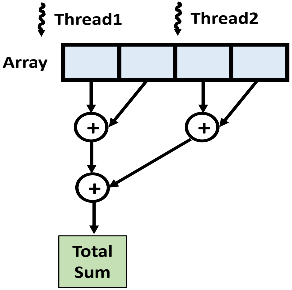
A sequential version

- O(n)
- for(int i = 0, i < n, ++i) ...



A parallel version

- $O(\log_2 n)$
- "tree"-based implementation



Parallel Reduction Algorithm

To process very large arrays:

- Multiple thread blocks required
- Each block reduces a portion of the array
- Need to communicate partial results between blocks
- Need global synchronization

Problem:

- CUDA does not support global synchronization

Solution:

- Kernel decomposition

Kernel Decomposition

- Decompose computation into multiple kernel invocations
- Kernel launch serves as a global synchronization point
- Negligible HW overhead, low SW overhead

Figure from 'Optimizing Parallel Reduction in CUDA' by Mark Harris

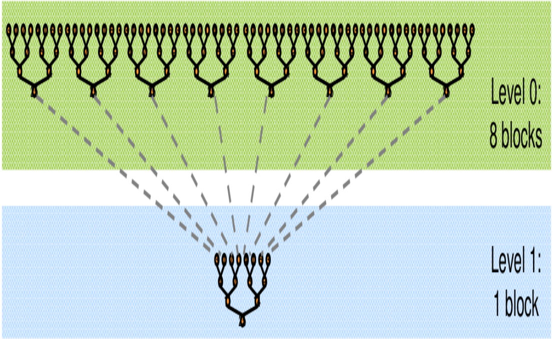


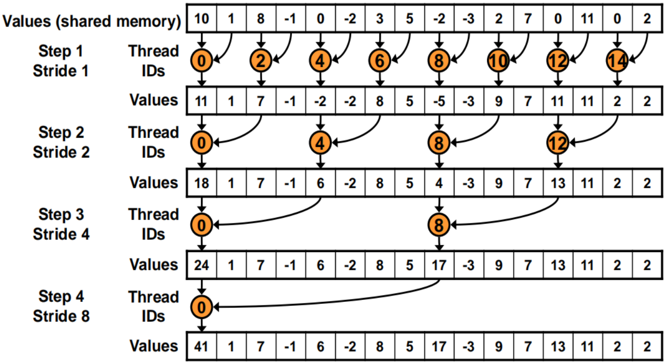
Figure: Multiple Kernel Invocations

Optimization In Reduction

- Metrics for GPU performance:
  - GFLOP/s for compute-bound kernels
    - One billion floating-point operations per second
  - Bandwidth for memory-bound kernels
    - Rate at which data can be read from or stored into memory by a processor
- Reduction has very low arithmetic intensity
  - Take 1 flop per element loaded
- Strive for peak bandwidth

Reduction 1: Interleaved Addressing

- Each thread loads one element from global memory to shared memory
- A thread adds two elements
- Half of the threads is deactivated at the end of each step overhead



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

Figure: Reduction with Interleaved Addressing and Divergent Branch

Reduction 1: Kernel

```
__global__ void reduce1(int *g_idata, int *g_odata, unsigned int n){
 extern __shared__ int sdata[];
 // load shared mem
 unsigned int tid = threadIdx.x;
 unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
 sdata[tid] = (i < n) ? g_idata[i] : 0;
 __syncthreads();
 // do reduction in shared mem
 for (unsigned int s=1; s < blockDim.x; s *= 2)
 { // modulo arithmetic is slow!
   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: Host

- The GPU kernel calculates data per block
- Partial sums computed by individual blocks
- Results will be stored in the first block elements of the global memory
- Final addition need to be done on this reduced data set
- By launching the same kernel again

Reduction 1: Host Code for Multiple Kernel Launch

```
...//cudaMemcpyHostToDevice...
int threadsPerBlock = 64;
int old_blocks, blocks = (N / threadsPerBlock) / 2;
blocks = (blocks == 0) ? 1 : blocks;
old_blocks = blocks;
while (blocks > 0) // call compute kernel
{
  sum<<<blocks, threadsPerBlock, threadsPerBlock*sizeof(int)>>>(devPtrA);
  old_blocks = blocks;
  blocks = (blocks / threadsPerBlock) / 2;
};
if (blocks == 0 && old_blocks != 1) // final kernel call, if still needed
  sum<<<1, old_blocks/2,  (old_blocks/2) * sizeof(int)>>>(devPtrA);
...//cudaMemcpyDeviceToHost...
```

Reduction 1: Analysis

Interleaved addressing with divergent branching

Problems:

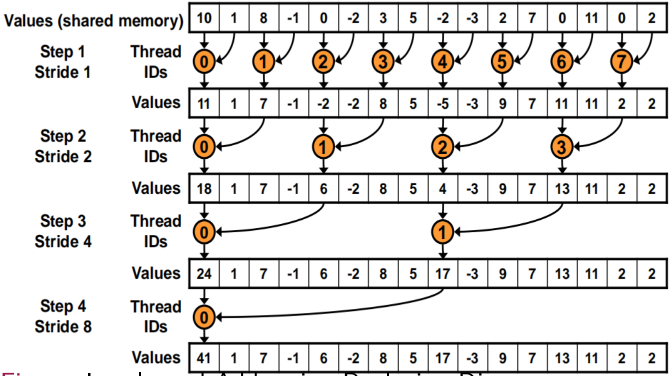
- highly divergent
- warps are very inefficient
- half of the threads does nothing!
- % operator is very slow
- loop is expensive

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951

Reduction 2: Interleaved Addressing

- Replace divergent branch in inner loop
- With strided index and non-divergent branch
- New Problem: Shared Memory Bank Conflicts



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

Figure: Interleaved Addressing Replacing Divergent Branch

Reduction 1: Kernel

```
__global__ void reduce1(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// load shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2)
{ //modulo arithmetic is slow!
   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 2: Kernel

```
__global__ void reduce2(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// load shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
// do reduction in shared mem
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();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 2: Analysis

Interleaved addressing with divergent branching

Problems:

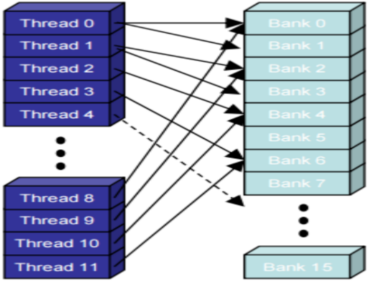
- ~~highly divergent~~
- ~~warps are very inefficient~~
- ~~% operator is very slow~~
- half of the threads does nothing!
- loop is expensive
- **shared memory bank conflicts**

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117

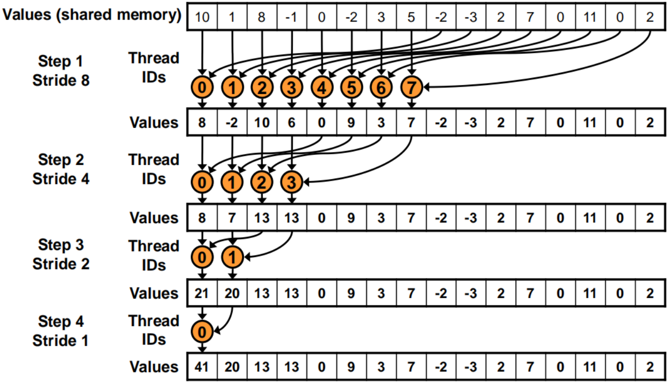
Shared Memory Bank Conflict

- Shared Memory is divided into banks and each bank has serial read/write access
- If more than one thread attempts to access same bank at same time, the accesses are serialized (Bank Conflict)
- The hardware splits a memory request decreasing the effective bandwidth



Reduction 3: Sequential Addressing

- Replace strided indexing in inner loop
- With reversed loop and threadID-based indexing
- New Problem: Idle Threads on first loop iteration



Values (shared memory): 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 1 Stride 8 — Thread IDs: 0 1 2 3 4 5 6 7

Values: 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 2 Stride 4 — Thread IDs: 0 1 2 3

Values: 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 3 Stride 2 — Thread IDs: 0 1

Values: 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 4 Stride 1 — Thread IDs: 0

Values: 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Figure: Reduction with Sequential Addressing

Reduction 2: Kernel

```
__global__ void reduce2(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// load shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
// do reduction in shared mem
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();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 3: Kernel

```
__global__ void reduce3(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// load shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s»=1)
{
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 3: Analysis

Interleaved addressing with divergent branching

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

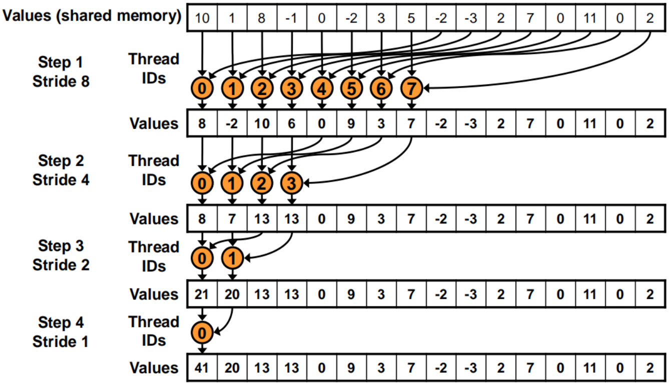
Problems:

- loop is expensive
- ~~shared memory bank conflicts~~
- **Half of the threads are idle on first loop iteration**

Reduction	**Time**	**Bandwidth**
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839

Reduction 4: First Add During Load

- Make busy all threads in the first step
- Halve the number of blocks
- Replace single load with two loads
- Allocation process performs the first reduction



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

Figure: Reduction with First Add During Load

Reduction 3: Kernel

```
__global__ void reduce3(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// load shared mem
unsigned int tid = threadIdx.x;
 unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
 sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s>>=1)
{
  if  (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 4: Kernel

```
__global__ void reduce4(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// 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();
// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s>>=1)  {
  if  (tid < s)
     sdata[tid] += sdata[tid + s];
  __syncthreads();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 4: Analysis

Memory bandwidth is still underutilized

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Problems:

- ~~Half of the threads are idle on first loop iteration~~
- **loop overhead**
- Another likely bottleneck is instruction overhead

Reduction	**Time**	**Bandwidth**
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098

Reduction 5: Unrolling the Last Warp

- Number of active threads decreases with the number of iteration
- When s <= 32, only one warp is left
- Warp runs the same instruction (SIMD)
- That means when s <= 32:
  - "__syncthreads()" is not needed
  - "if (tid < s)" is not needed
- Unroll last 6 iterations

Without unrolling, all warps execute every iteration of the for loop and if statement

Reduction 4: Kernel

```
 __global__ void reduce4(int *g_idata, int *g_odata, unsigned int n){
 extern __shared__ int sdata[];
 // 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();
 // do reduction in shared mem
 for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s)
   sdata[tid] += sdata[tid + s];
  __syncthreads();
 }
 // write result for this block to global mem
 if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 5: Kernel

```
__global__ void reduce5(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// 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();
// do reduction in shared mem
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);
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 5: warpReduce

```
_device__ void warpReduce(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];
}
```

Reduction 5: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Problems:

- Still have iterations
- **loop overhead**

Reduction	**Time**	**Bandwidth**
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098
Reduce 5	0.00836	32.1053

Reduction 6: Complete Unrolling

If number of iterations is known at compile time, could completely unroll the reduction.

- Block size is limited to 512 or 1024 threads
- Block size should be of power-of-2
- For a fixed block size, complete unrolling is easy
- For generic implementation, solution is-
  - CUDA supports C++ template parameters on device and host functions
  - Block size can be specified as a function template parameter

Reduction 5: Kernel

```
__global__ void reduce5(int *g_idata, int *g_odata, unsigned int n){
  extern __shared__ int sdata[];
  // 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();
  // do reduction in shared mem
  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);
  // write result for this block to global mem
  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

Reduction 6: Kernel

Specify block size as a **function template parameter** and all code highlighted in yellow will be evaluated at compile time.

```
template < unsigned int blockSize >
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n){
extern __shared__ int sdata[];
// 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();

// do reduction in shared memory
if (blockSize >= 512) {
  if (tid < 256)
    sdata[tid] += sdata[tid + 256];
  __syncthreads();
 }
```

Reduction 6: Kernel

```
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);

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

Reduction 6: Kernel

Modified warpReduce function:

```
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];
}
```

Reduction 6: Invoking Template Kernels

Use a switch statement for possible block sizes while invoking template kernels

```
switch (threadsPerBlock) {
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;
}
```

Reduction 6: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

- **Algorithm Cascading** can lead to significant speedups in practice

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098
Reduce 5	0.00836	32.1053
Reduce 6	0.00769	34.9014

Reduction 7: Multiple Adds / Thread

Algorithm Cascading:

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Replace load and add two elements
- With a loop to add as many as necessary

Reduction 6: Kernel

```
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n){
  extern __shared__ int sdata[];
  // 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();

  // do reduction in shared mem
  ...
  // write result for this block to global mem
  ...
}
```

Reduction 7: Kernel

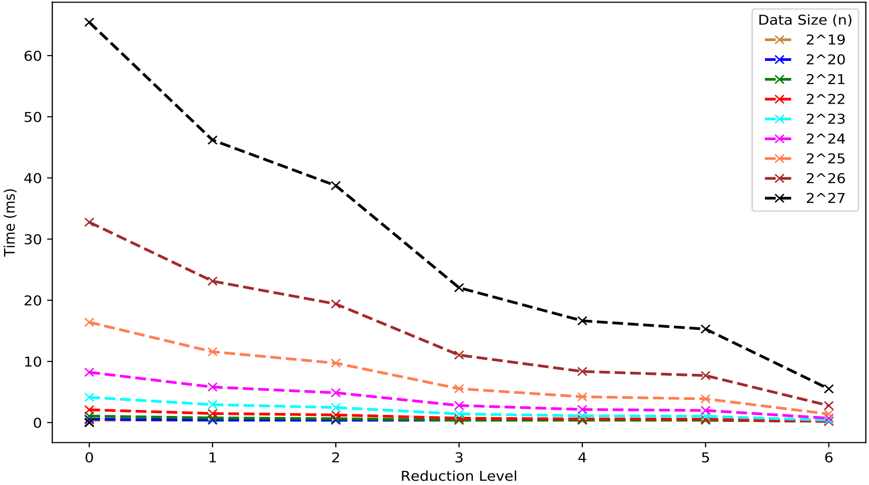
```
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n)
{
  extern __shared__ int sdata[];
  // reading from global memory, writing to shared memory
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sdata[tid] = 0;
  while (i < n) {
   sdata[tid] += g_idata[i] + g_idata[i+blockSize];
   i += gridSize;
  }
  __syncthreads();
  // do reduction in shared mem
  ...
  // write result for this block to global mem
  ...
}
```

Reduction 7: Analysis

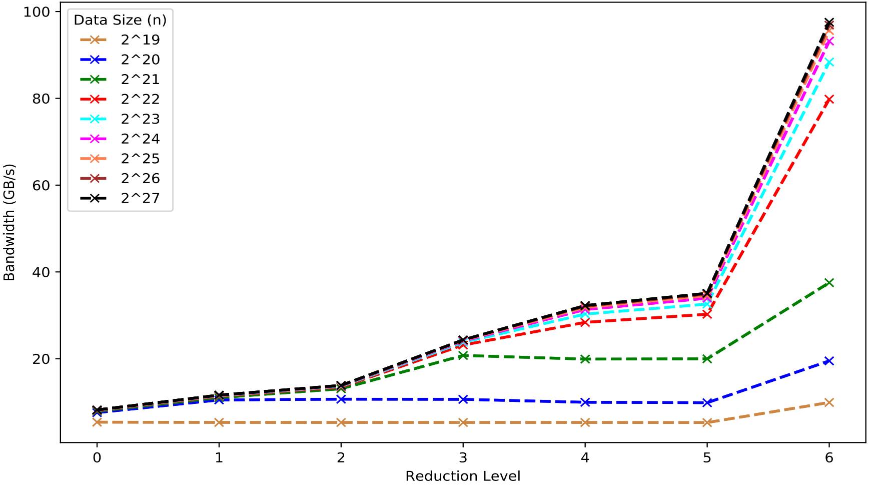
Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	**Time**	**Bandwidth**
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098
Reduce 5	0.00836	32.1053
Reduce 6	0.00769	34.9014
Reduce 7	0.00277	96.8672

Reduction Performance Comparison wrt time in Tesla K40m GPU



Reduction Performance Comparison wrt bandwidth in Tesla K40m GPU



Types of Optimization

- Algorithmic optimizations
  - Changes to addressing, algorithm cascading (Reduction No. 1 to 4, 7)
  - Approx 12x speedup
- Code optimizations
  - Loop unrolling (Reduction No. 5, 6)
  - Approx 3x speedup

Summary

Kernel	Optimization
Reduce1	Interleaved addressing (using modulo arithmetic) with divergent branching
Reduce2	Interleaved addressing (using contiguous threads) with bank conflicts
Reduce3	Sequential addressing, no divergence or bank conflicts
Reduce4	Uses n/2 threads, performs first level during global load
Reduce5	Unrolled loop for last warp, intra-warp synchronisation barriers removed
Reduce6	Completely unrolled, using template parameter to assert whether the number of threads is a power of two
Reduce7	Multiple elements per thread, small constant number of thread blocks launched. Requires very few synchronisation barriers

Example of Applications on Reduction

- Bitonic Sort
- Prefix sum

Problem: Sorting

- Sort any random permutation of numbers in ascending or descending order
- Basic introduction to sorting networks
- Focus on a comparison based sort - **Bitonic Sort**
- Discuss how operations can be parallelized using CUDA.

Sorting Networks

A sorting network is composed of two elements

- **Wires:** Wires run from left to right, carrying values (one per wire) that traverse the network all at the same time.
- **Comparators:** Comparators connect two wires. When a pair of values, traveling through a pair of wires, encounter a comparator, the comparator may or may not swap the values.

Comparator

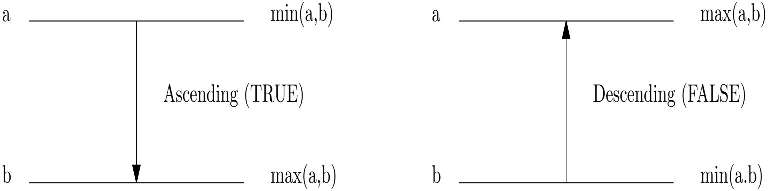


Figure: Comparator Function

A Simple Sorting Network

Sort four numbers a, b, c, d in ascending order where $a > b > c > d$

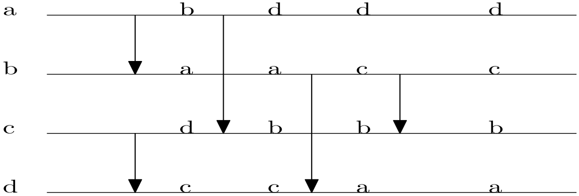
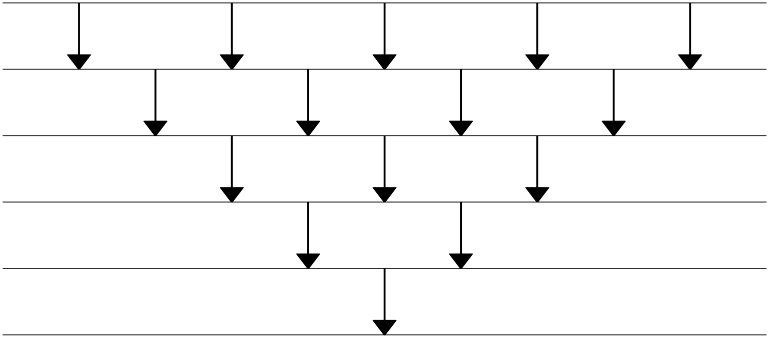


Figure: Sorting Four Numbers

Bubble Sort



Any comparison based sort can be done using a sorting network.

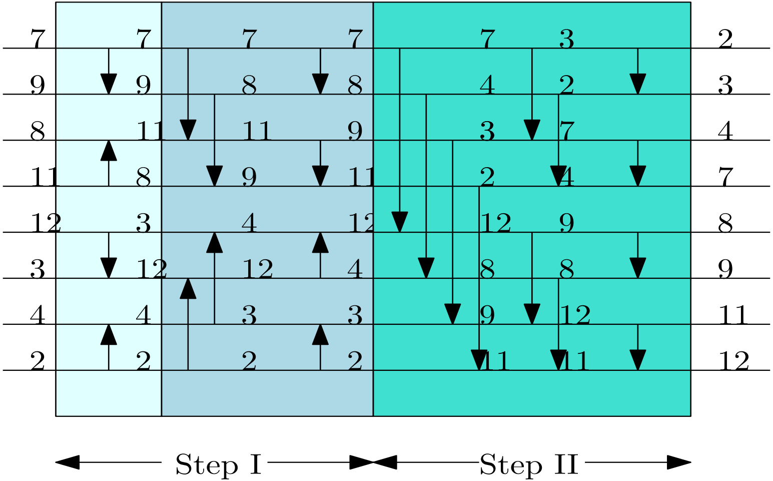
Bitonic Sort

Bitonic sort takes place using two fundamental steps:

- **Step I:** Convert an arbitrary sequence to a bitonic sequence.
- **Step II:** Convert a bitonic sequence to a sorted sequence.

A Bitonic Sequence is a sequence of numbers which is first strictly increasing then after a point strictly decreasing. $a_1 < a_2 < ... < a_m > b_1 > b_2 > ... > b_n$

Bitonic Sort



Recursive Structure

- If you look closely, Step I uses Step II recursively on smaller sequences.
- Step II can be used to sort in any order (ascending or descending). The order can be controlled using the comparator.
- Step I uses Step II in a way to construct subseqences that are bitonic in nature.

http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

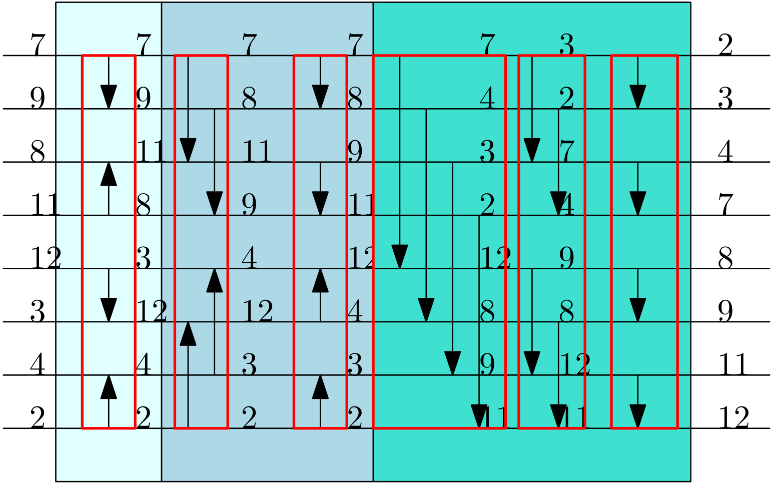
Recursive C Program

```
//Comparator
void compare(int i, int j, boolean dir){
  if (dir==(a[i]>a[j]))
    exchange(i, j);
}

//Step II
void bitonicMerge(int lo, int n, boolean dir){
  if (n>1){
    int m=n/2;
    for (int i=lo; i<lo+m; i++)
      compare(i, i+m, dir);
    bitonicMerge(lo, m, dir);
    bitonicMerge(lo+m, m, dir);
  }
}
```

Recursive C Program

```
//Step I
void bitonicSort(int lo, int n, boolean dir){
  if (n>1)
  {
    int m=n/2;
    bitonicSort(lo, m, ASCENDING);
    bitonicSort(lo+m, m, DESCENDING);
    bitonicMerge(lo, n, dir);
  }
}
```

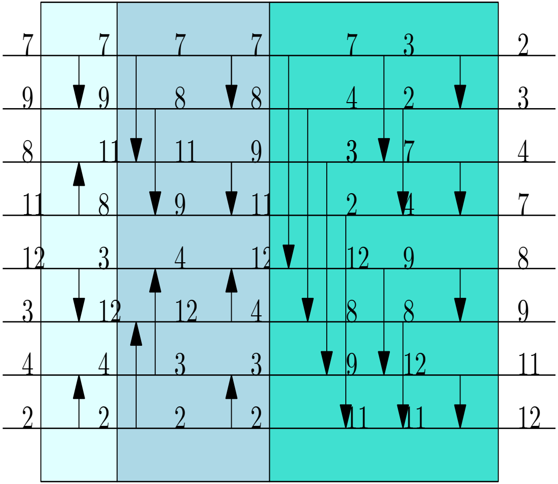
Scope for Parallelization



Formulate Parallel Solution

- Associate every cuda thread block with a sorting subproblem.
- Merge results from each SM to solve the original sorting problem.

Mapping Sorting Subproblem



Thread Block

4 threads for
eight elements

Shared Memory Size
=
2 * Number of Threads

Comparator

```
__device__ inline void Comparator(uint &keyA, uint &valA,uint &keyB,uint &valB
    ,uint dir){
  uint t;
  if ((keyA > keyB) == dir){
    t = keyA;
    keyA = keyB;
    keyB = t;
    t = valA;
    valA = valB;
    valB = t;
  }
}
```

NVIDIA CUDA SDK Benchmark Suite

Sort in Shared Memory

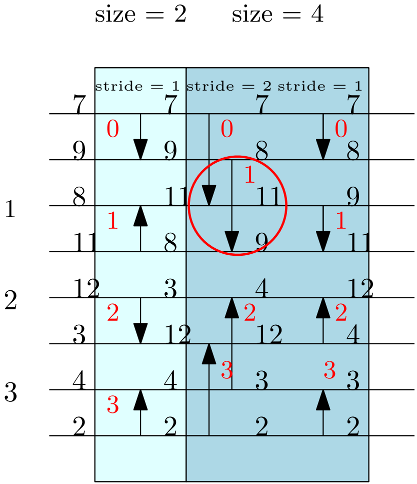
```
__global__ void bitonicSortShared1(uint *d_DstKey,uint *d_DstVal, uint *
    d_SrcKey,uint *d_SrcVal){
  //Shared memory storage for current subarray
  __shared__ uint s_key[SHARED_SIZE];
  __shared__ uint s_val[SHARED_SIZE];

  //Offset to the beginning of subarray and load data
  d_SrcKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
  d_SrcVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;
  d_DstKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
  d_DstVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;
```

```
s_key[threadIdx.x+0]=d_SrcKey[0];
s_val[threadIdx.x+0]=d_SrcVal[0];
s_key[threadIdx.x+SHARED_SIZE/2]=d_SrcKey[(SHARED_SIZE/2];
s_val[threadIdx.x+SHARED_SIZE/2]=d_SrcVal[(SHARED_SIZE/2];
      //strided load by threads, each thread loads two elements
for(size=2;size<SHARED_SIZE;size<<=1){
  //Bitonic merge
  uint ddd=(threadIdx.x&(size/2))!=0;
  for (stride=size/2;stride>0;stride>>=1){
    __syncthreads();
    pos=2*threadIdx.x-threadIdx.x&(stride-1);
    Comparator(s_key[pos+0],s_val[pos+0],
    s_key[pos+stride],s_val[pos+stride],ddd);
  }
}
```

Important Variables

- Size denotes the length of bitonic sequences being generated.
- Pos denotes the position of the first item being processed by the thread and is dependent on thread id and stride.
- Stride denotes the distance between the position of numbers being sorted by a thread.
- ddd represents direction and is dependent on thread id and size.

Bitonic Sequence Creation



$$ddd = (1\&(4/2)! = 0) = 1$$

$$pos = 2 * 1 - 1\&(2-1) = 1$$

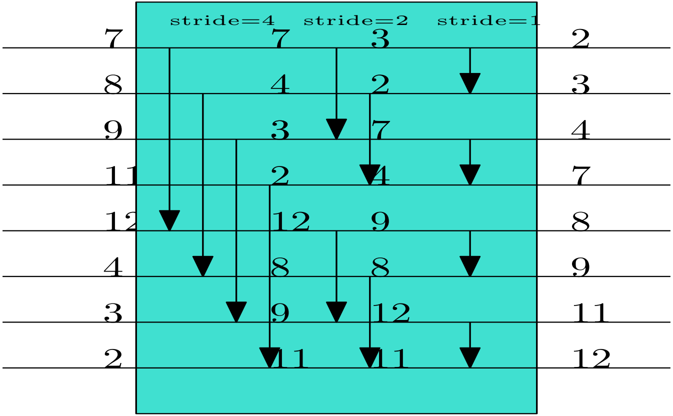
$$pos + stride = 1 + 1 = 2$$

```
  //sort in opposite directions odd/even block ids
  uint ddd = (blockIdx.x + 1) & 1;
  for(stride=SHARED_SIZE/2;stride>0;stride>>=1){
    __syncthreads();
    pos=2*threadIdx.x-threadIdx.x&(stride - 1));
    Comparator(s_key[pos+0],s_val[pos+0],
    s_key[pos+stride],s_val[pos+stride],ddd);
  }
  __syncthreads();
  d_DstKey[0]=s_key[threadIdx.x+0];
  d_DstVal[0]=s_val[threadIdx.x+0];
  d_DstKey[SHARED_SIZE/2]=s_key[threadIdx.x+SHARED_SIZE/2];
  d_DstVal[SHARED_SIZE/2]=s_val[threadIdx.x+SHARED_SIZE/2];
}
```

Sorting Bitonic Sequence

Block 0

$ddd = (0 + 1) \& 1 = 1$



Example Applications of reduction

- Bitonic Sort
- **Prefix Sum**

All-Prefix-Sums

The **all-prefix-sums** operation takes a binary associative operator $\oplus$, and an array of n elements.

$$[a_0, a_1, ..., a_{n-1}],$$

and returns the array

$$[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus, ... \oplus a_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array

[3 1 7 0 4 1 6 3],

would return

[3 4 11 11 15 16 22 25].

Inclusive and Exclusive scan

All-prefix-sums on an array of data is commonly known as ***scan***.

- **Inclusive scan** - a scan of an array generates a new array where each element j is the sum of all elements up to and including j.
- **Exclusive scan** - a scan of an array generates a new array where each element j is the sum of all elements excluding j.

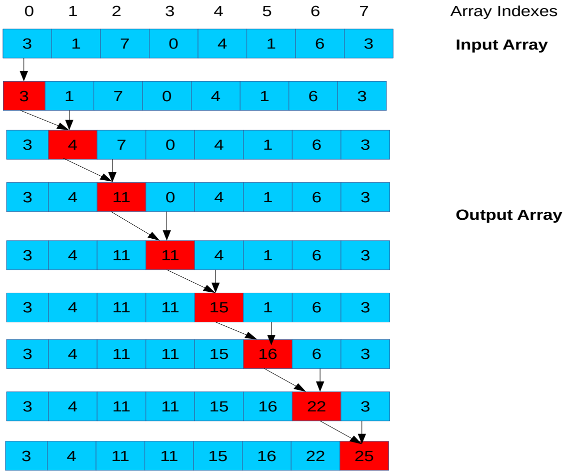
The exclusive scan operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements

$$[a_0, a_1, ..., a_{n-1}],$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus, ... \oplus a_{n-2})].$$

Example - Inclusive Scan



0	1	2	3	4	5	6	7	Array Indexes
3	1	7	0	4	1	6	3	**Input Array**
3	1	7	0	4	1	6	3	
3	4	7	0	4	1	6	3	
3	4	11	0	4	1	6	3	**Output Array**
3	4	11	11	4	1	6	3	
3	4	11	11	15	1	6	3	
3	4	11	11	15	16	6	3	
3	4	11	11	15	16	22	3	
3	4	11	11	15	16	22	25	

Sequential Code

Inclusive Scan

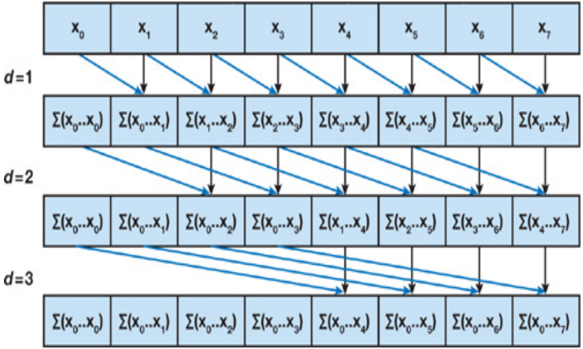
```
void scan( float* output, float* input, int length)
{
    output[0] = input[0]; // since this is a inclusive scan

    for(int j = 1; j < length; ++j)
     output[j] = output[j-1] + input[j];

}
```

Sequential Code - Complexity

- Code performs exactly $n-1$ adds for an array of length n
- Work complexity is $O(n)$
- Very large n, motivate parallel execution

Parallel Prefix Sum - Hillis/Steel Scan (Algorithm 1)

```
for $d = 1$ to $log_2 n$ do
    forall $k \geq$ in parallel do
        $x[k] = x[k - 2^{d-1}] + x[k]$
```



x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7

$d=1$

$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$

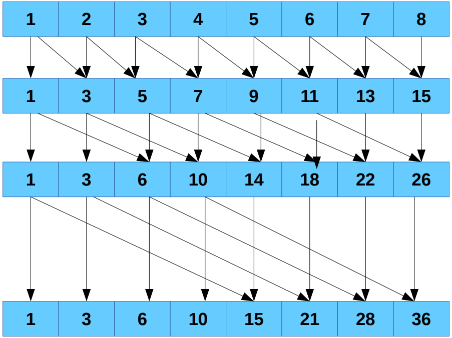
$d=2$

$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$

$d=3$

$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

Example- Hillis/Steel Inclusive Scan



1	2	3	4	5	6	7	8
1	3	5	7	9	11	13	15
1	3	6	10	14	18	22	26
1	3	6	10	15	21	28	36

Step-I
Add elements 2^0
step away

Step-II
Add elements 2^1
step away

Step-III
Add elements 2^2
step away

Nos of steps O(log n)
Work O (n * log n)

Analysis

- The algorithm performs $O(n log_2 n)$ additions operations.
- Remember that a sequential scan performs $O(n)$ adds. Therefore, this naïve implementation is **not work-efficient**.
- Algorithm 1 assumes that there are as many processors as data elements. On a GPU running CUDA, this is not usually the case.
- Instead, the *forall* is automatically divided into small parallel batches (called *warps*) that are executed sequentially on a multiprocessor.
- The algorithm 1 will not work because it performs the scan in place on the array. The results of one warp will be overwritten by threads in another warp.

A double-buffered version of the sum scan from Algorithm 1

Algorithm 2

```
for $d$ := 1 to $log_2 n$ do
    **forall** $k$ in **parallel** do
        if $k \geq 2^d$ then
            $x[out][k] := x[in][k\text{-}2^{d-1}] + x[in][k]$
        else
            $x[out][k] := x[in][k]$
    swap(in,out)
```

CUDA C code - Algorithm 1

```
global__ void scan(float *g_odata, float *g_idata, int n)
{
        extern  __shared__  float temp[]; // allocated on invocation
        int thid = threadIdx.x;
        int pout = 0, pin = 1;
        //For exclusive scan,shift right by one and set first elt to 0
        temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
        __syncthreads();
        for (int offset = 1; offset < n; offset *= 2)
        {
          pout = 1 - pout;
          pin  = 1 - pout;
          if (thid >= offset)
                temp[pout*n+thid] += temp[pin*n+thid - offset];
          else
                temp[pout*n+thid] =  temp[pin*n+thid];
          __syncthreads();
        }
        g_odata[thid] = temp[pout*n+thid1];
}
```

Parallel Prefix Sum - Blelloch Scan

- The idea is to build a *balanced binary tree* on the input data and sweep it to and from the root to compute the prefix sum.
- A binary tree with n leaves has $logn$ levels, and each level $d \in [0, n)$ has 2^d nodes.
- If we perform one add per node, then we will perform $O(n)$ adds on a single traversal of the tree.

Parallel Prefix Sum - Blelloch Scan

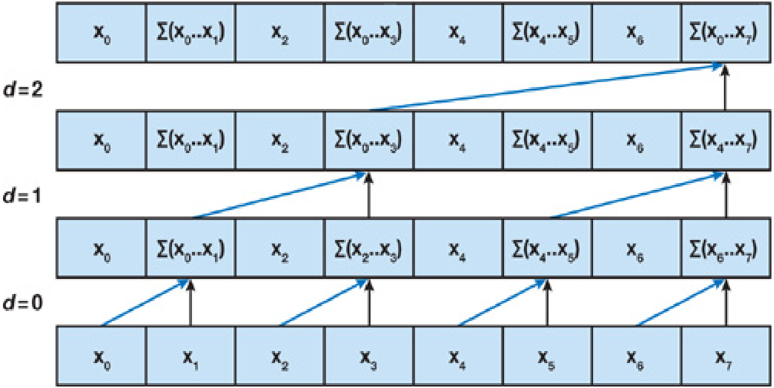
The algorithm consists of two phases:

- **Reduction Phase:** we traverse the tree from leaves to root computing partial sums at internal nodes of the tree
- **Down Sweep Phase:** We traverse back up the tree from the root, using the partial sums to build the scan in place on the array using the partial sums computed by the reduce phase.

Blelloch Scan - Reduce Phase

```
for $d := 0$ to $log_2 n - 1$ do
    for $k$ from $0$ to $n - 1$ by $2^{d+1}$ in paralle do
        $x[k + 2^{d+1} - 1] := x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$
```

Blelloch Scan - Reduce Phase



x_0	$\Sigma(x_0..x_1)$	x_2	$\Sigma(x_0..x_3)$	x_4	$\Sigma(x_4..x_5)$	x_6	$\Sigma(x_0..x_7)$

$d=2$

x_0	$\Sigma(x_0..x_1)$	x_2	$\Sigma(x_0..x_3)$	x_4	$\Sigma(x_4..x_5)$	x_6	$\Sigma(x_4..x_7)$

$d=1$

x_0	$\Sigma(x_0..x_1)$	x_2	$\Sigma(x_2..x_3)$	x_4	$\Sigma(x_4..x_5)$	x_6	$\Sigma(x_6..x_7)$

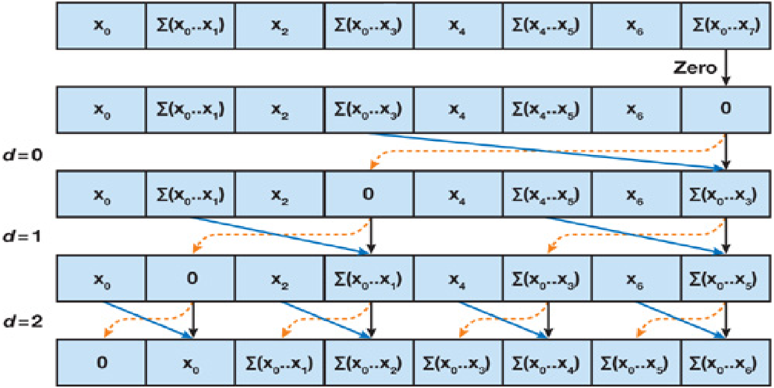
$d=0$

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Blelloch Scan - Down-Sweep Phase

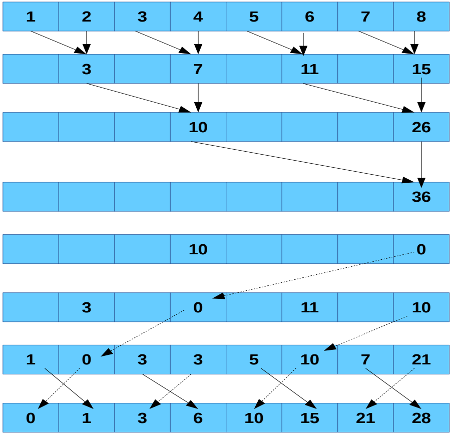
```
$x[n-1] := 0$
for $d := log_2 n$ down to 0 do
    for k from 0 to $n-1$ by $2^{d+1}$ in parallel do
        $t := x[k + 2^d - 1]$
        $x[k + 2^d - 1] := x[k + 2^{d+1} - 1]$
        $x[k + 2^{d+1} - 1] := t + x[k + 2^{d+1} - 1]$
```

Blelloch Scan - Down-Sweep Phase



x_0	$\Sigma(x_0..x_1)$	x_2	$\Sigma(x_0..x_3)$	x_4	$\Sigma(x_4..x_5)$	x_6	$\Sigma(x_0..x_7)$
x_0	$\Sigma(x_0..x_1)$	x_2	$\Sigma(x_0..x_3)$	x_4	$\Sigma(x_4..x_5)$	x_6	0
x_0	$\Sigma(x_0..x_1)$	x_2	0	x_4	$\Sigma(x_4..x_5)$	x_6	$\Sigma(x_0..x_3)$
x_0	0	x_2	$\Sigma(x_0..x_1)$	x_4	$\Sigma(x_0..x_3)$	x_6	$\Sigma(x_0..x_5)$
0	x_0	$\Sigma(x_0..x_1)$	$\Sigma(x_0..x_2)$	$\Sigma(x_0..x_3)$	$\Sigma(x_0..x_4)$	$\Sigma(x_0..x_5)$	$\Sigma(x_0..x_6)$

Example - Blelloch Exclusive Scan



1	2	3	4	5	6	7	8	Input
	3		7		11		15	Reduce Step 0
			10				26	Reduce Step 1
							36	Reduce Step 2
			10				0	Identity Elements
	3		0		11		10	Down Sweep 0
1	0	3	3	5	10	7	21	Down Sweep 1
0	1	3	6	10	15	21	28	Down Sweep 2

Cuda code

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
        extern __shared__ float temp[];  // allocated on invocation
        int thid = threadIdx.x;
        int offset = 1;

        temp[2*thid] = g_idata[2*thid]; // load input into shared memory
        temp[2*thid+1] = g_idata[2*thid+1];
        for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
        {
          __syncthreads();
          if (thid < d)
          {
                int ai = offset*(2*thid+1)-1;
                int bi = offset*(2*thid+2)-1;
                temp[bi] += temp[ai];
          }
          offset *= 2;
        }
```

Cuda code

```
        if (thid == 0)
         temp[n - 1] = 0;  // clear the last element
        for (int d = 1; d < n; d *= 2) //traverse down tree & build scan
        {
          offset >>= 1;
          __syncthreads();
          if (thid < d)
          {
                int ai = offset*(2*thid+1)-1;
                int bi = offset*(2*thid+2)-1;
                float t = temp[ai];
                temp[ai] = temp[bi];
                temp[bi] += t;
          }
        }
        __syncthreads();
        g_odata[2*thid] = temp[2*thid];//write results to device memory
        g_odata[2*thid+1] = temp[2*thid+1];
}
```

Reduction Conclusion

We have learnt how to-

- Understand CUDA performance characteristics
  - Memory coalescing,
  - Divergent branching,
  - Bank conflicts,
  - Latency hiding
- Use peak performance metrics to guide optimization
- Understand parallel algorithm complexity theory
- Identify type of bottleneck and
- Suitably optimize the algorithm

References

1. Optimizing Parallel Reduction in CUDA by Mark Harris
   White paper available at http://docs.nvidia.com.
2. Reductions and Low-Level Performance Considerations by David Tarjan
3. Parallel Prefix Sum (Scan) with CUDA by Mark Harris
   White paper available at
   https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html.