Lecture 8
CUDA Programming 3

张奇
复旦大学

Outline

Optimization & Example

CUDA Libraries

Control flow

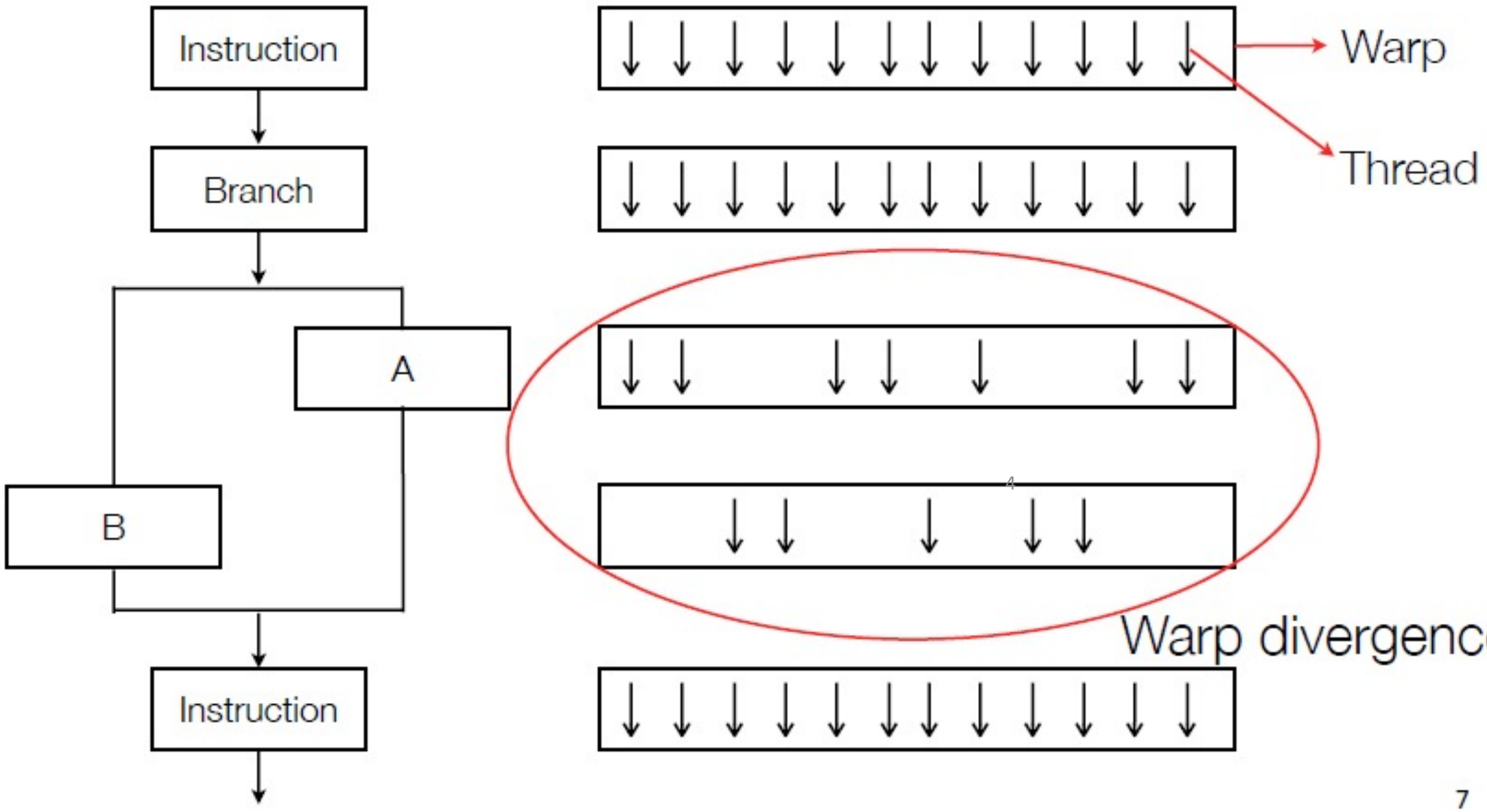
- If statement
  - Threads are executed in warps
  - Within a warp, the hardware is not capable of executing *if* and *else* statements at the same time!

```
__global__ void function();
{
     ....

     if (condition)
     {    ...
     }
     else
     {    ...
     }
}
```

3

Control flow

- How does the hardware deal with an if statement?



Control flow

- Hardware serializes the different execution paths
- Recommendations
  - Try to make every thread in the same warp do the same thing
    - If the if statement cuts at a multiple of the warp size, there is no warp divergence and the instruction can be done in one pass
  - Remember threads are placed consecutively in a warp (t0-t31, t32-t63, …)

    - But we cannot rely on any execution order within warps
  - If you can't avoid branching, try to make as many consecutive threads as possible do the same thing

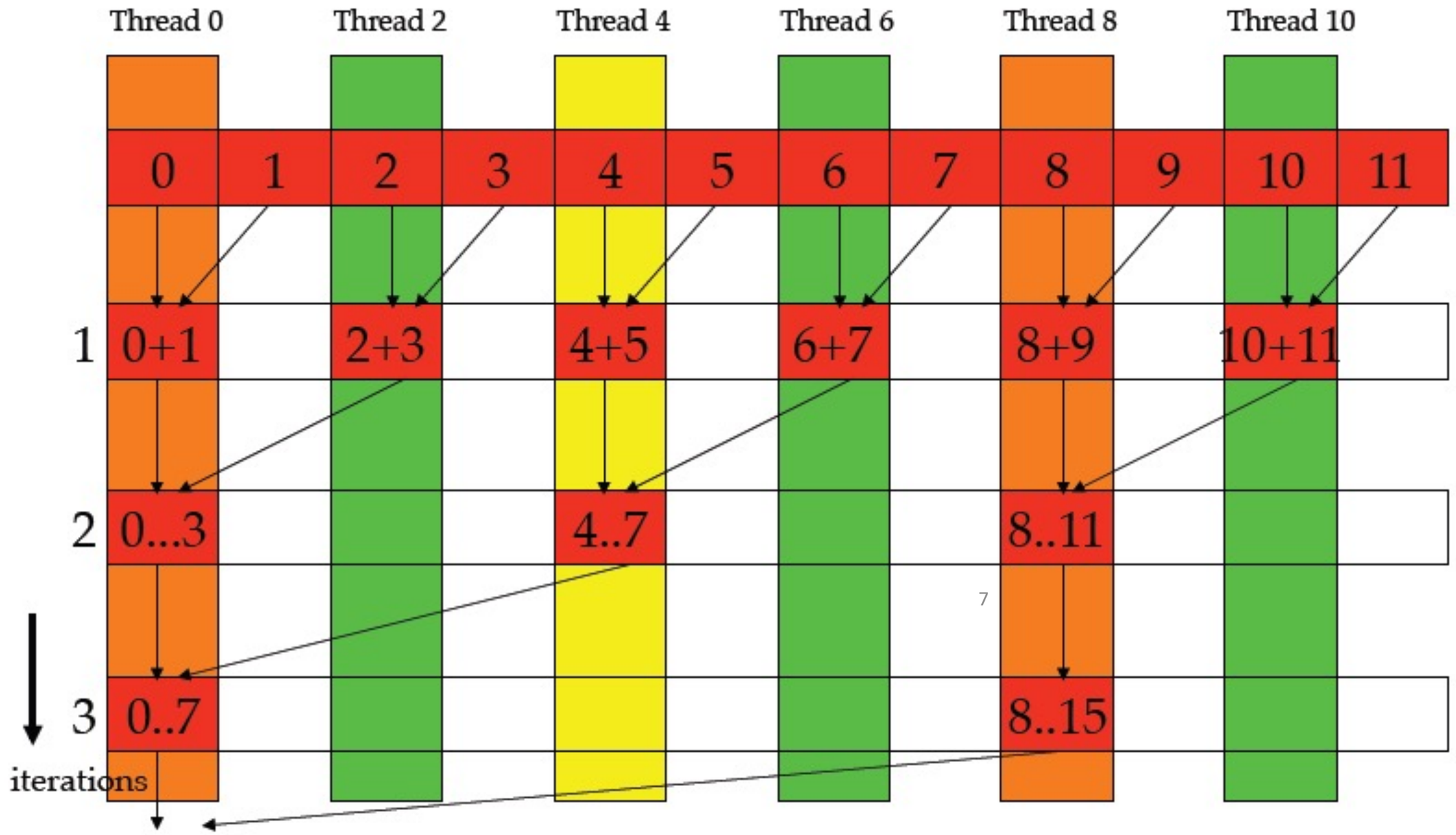
Control flow - An illustration

- Let's implement a sum reduction
- In a serial code, we would just loop over all elements and add
- Idea:
  - To implement in parallel, let's take every other value and add in place it to its neighbor
  - To the resulting array do the same thing until we have only one value

6

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

Control flow - An illustration



Control flow - An illustration

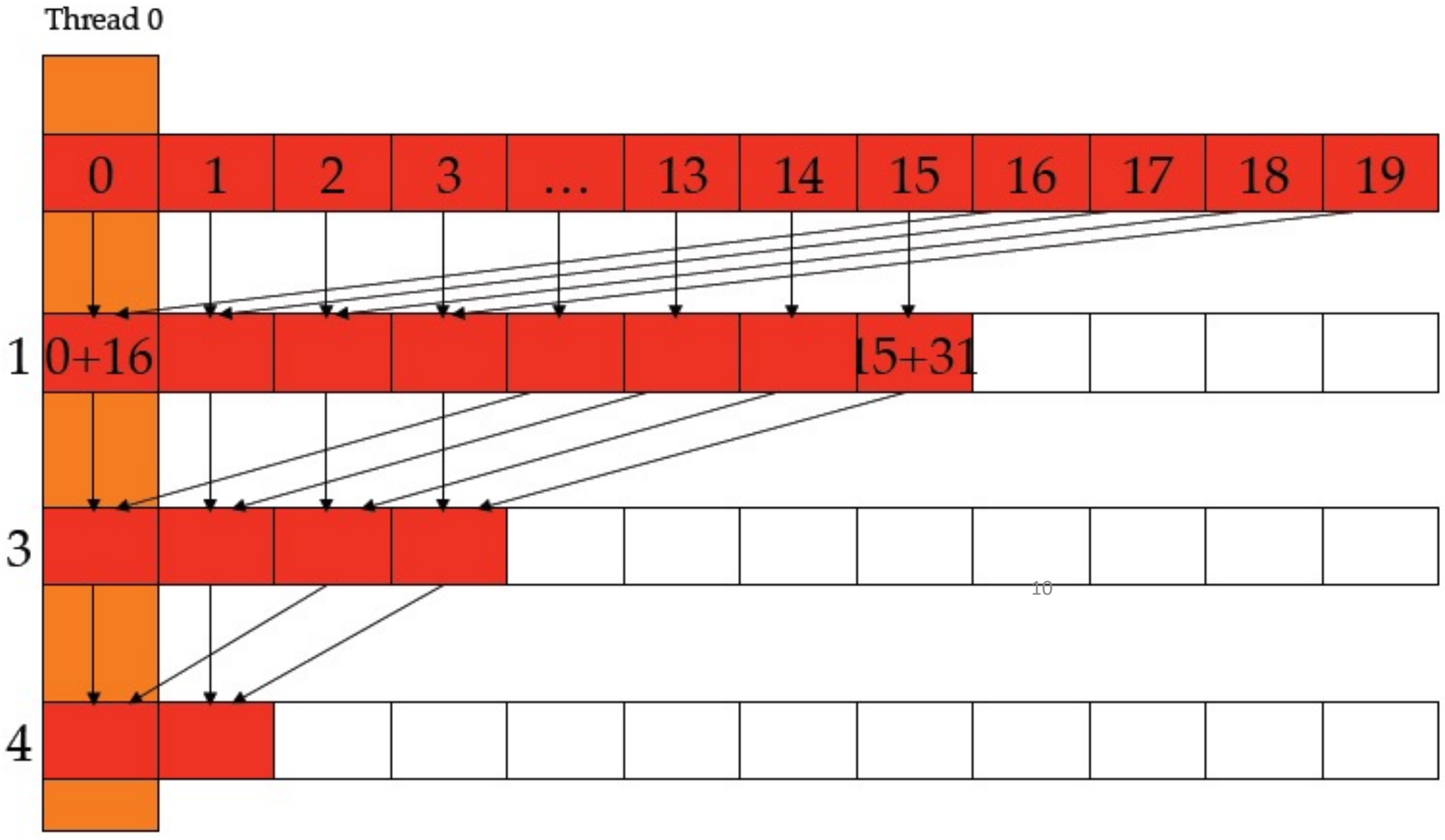
- Advantages
  - Right result
  - Runs in parallel
- Disadvantages
  - The number of threads decreases per iteration, but we're using much more warps than needed
    - There is warp divergence in every iteration [8]
  - No more than half the threads per thread are being executed per iteration
- Let's change the algorithm a little bit...

Control flow - An illustration

- Improved version
  - Instead of adding neighbors, let's add values with stride half a section away
  - Divide the stride by two after each iteration

```
__shared__ float partialSum[]
int t = threadIdx.x
for(int stride = blockDim.x; stride>1; stride>>=1)
{
  __syncthreads();
  if (t<stride)
  partialSum[t] += partialSum[t+stride];
}
```

Control flow - An illustration



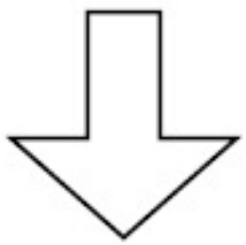
Control flow - An illustration

- 512 elements

Iteration	Exec. threads	Warps
1	256	16
2	128	8
3	64	4
4	32	2
5	16	1
6	8	1
7	4	1
8	2	1

Threads > Warp

Threads < Warp



Warp divergence!

11

Control flow - An illustration

- We get warp divergence only for the last 5 iterations
- Warps will be shut down as the iteration progresses
  - This will happen much faster than for the previous case
  - Resources are better utilized
  - For the last 5 iterations, only 1 warp is still active

Control flow - Loop divergence

- Work per thread data dependent

```
__global__ void per_thread_sum (int *indices, float *data, float *sums)
{
  ...
  for (int j=indices[i]; j<indices[i+1]; j++)
  {
    sum += data[j];
  }
  sums[i] = sum
}
```


David Tarjan - NVIDIA

Control flow - Loop divergence

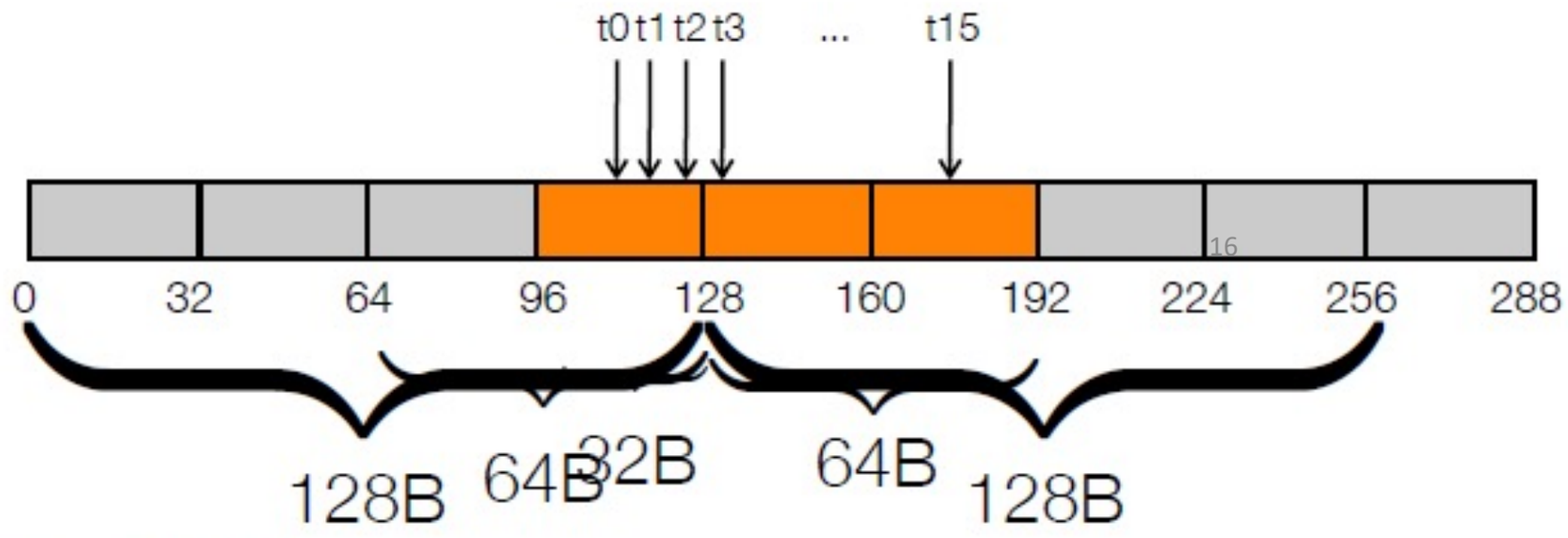
- Warp wont finish until the last thread finishes
  - Warp will be dragged
- Possible solution:
  - Try flatten peaks by making threads work in multiple data

Memory coalescing

- Global memory is accessed in chunks of aligned 32, 64 or 128 bytes
- Following protocol is used to issue a memory transaction of a half warp (valid for 1.X)
  - Find memory segment that contains address requested by the lowest numbered active thread
  - Find other active threads whose requested address lies in same segment, and reduce transaction size if possible
  - Do transaction. Mark serviced threads as inactive.
  - Repeat until all threads are serviced
- Worse case: fetch 32 bytes, use only 4 bytes: 7/8 wasted bandwidth!

Memory coalescing

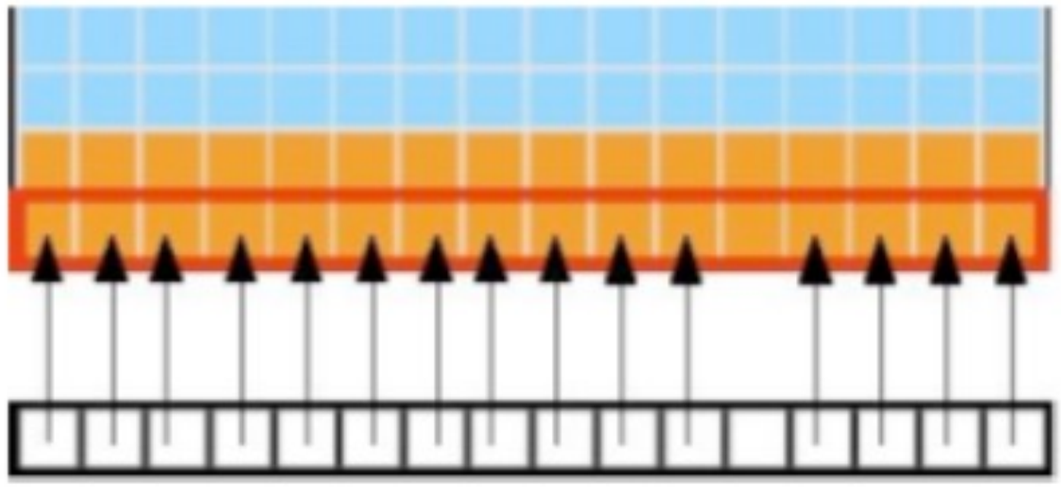
- Access pattern visualization
  - Thread 0 is lowest active, accesses address 116
    - Belongs to 128-byte segment 0-127



David Tarjan - NVIDIA

Memory coalescing

- Simple access pattern

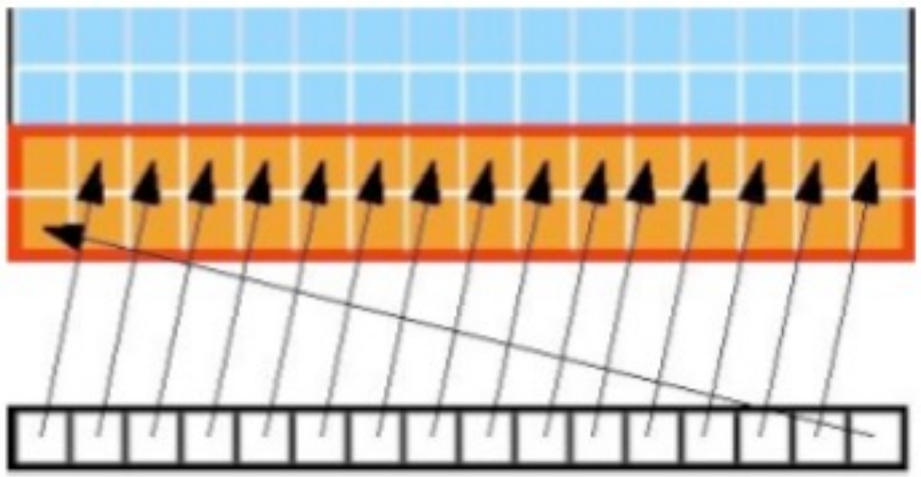


One 64 byte transaction

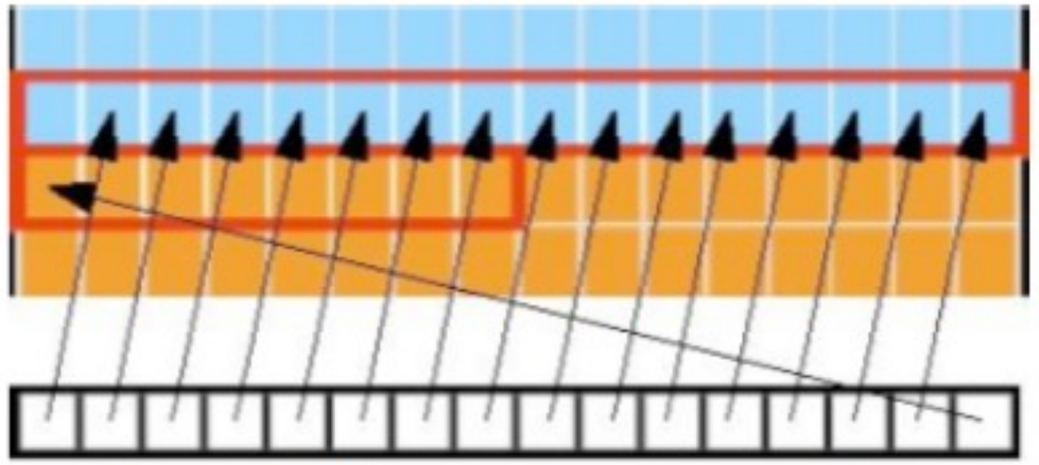
Will be looking at compute capability 1.X examples

Memory coalescing

- Sequential but misaligned



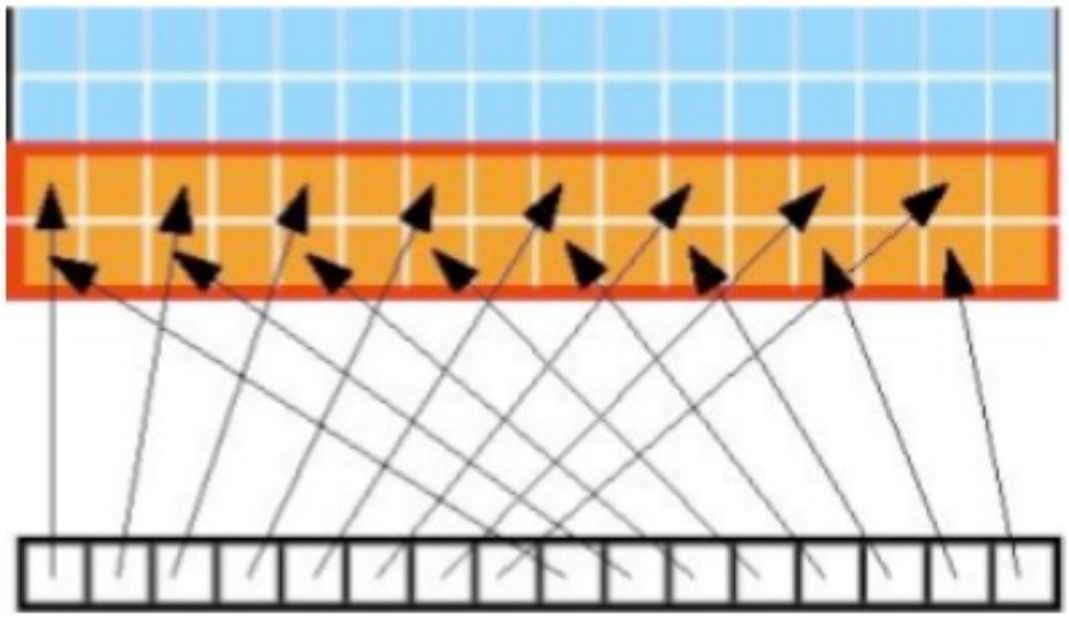
One 128 byte transaction



One 64 byte transaction
and one 32 byte transaction

Memory coalescing

- Strided access

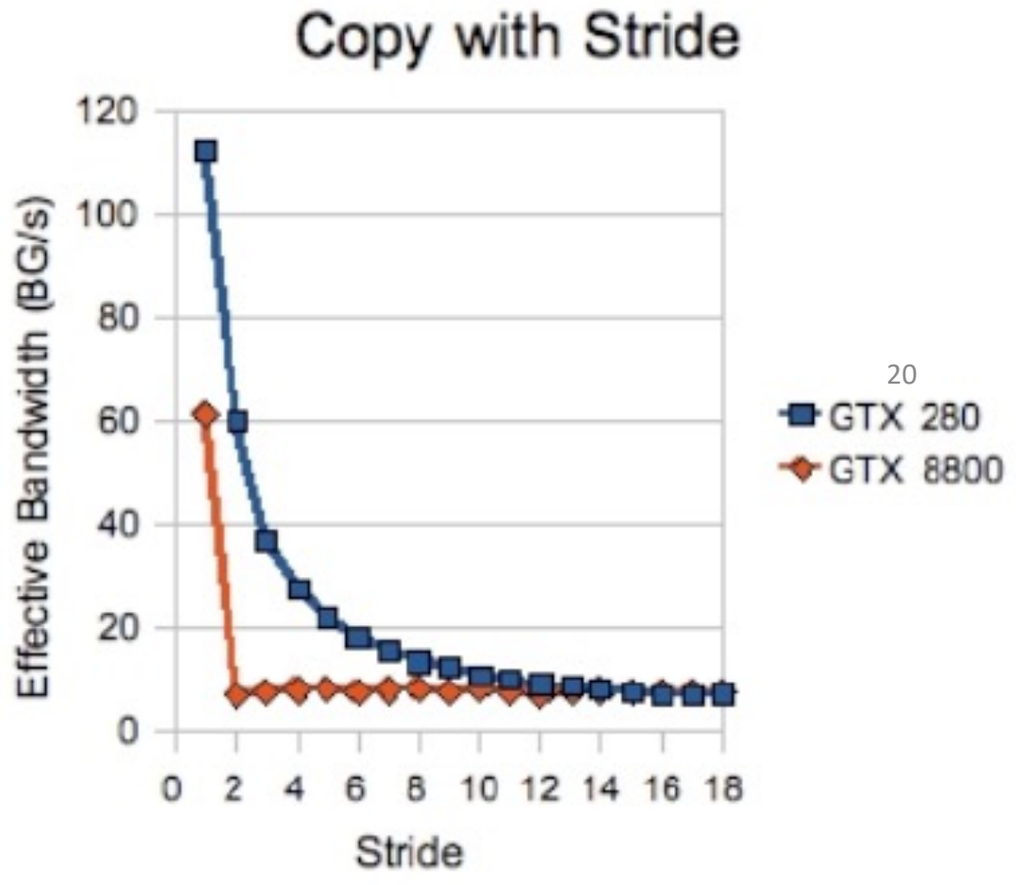


One 128 byte transaction, but half of bandwidth is wasted

Memory coalescing

- Example: Copy with stride

```
__global__ void strideCopy(float *odata, float *idata, int stride)
{
   int xid = (blockIdx.x*blockDim.x+threadIdx.x)*stride;
   odata[xid] = idata[xid];

}
```



Memory coalescing

- 2.X architecture
  - Global memory is cached
    - Cached in both L1 and L2: 128 byte transaction
    - Cached only in L2: 32 byte transaction
  - Whole warps instead of half warps

Memory coalescing - SoA or AoS?

- Array of structures

```
struct record
{
   int key;
   int value;
   int flag;
};

record *d_record;
cudaMalloc((void**) &d_records, ...);
```

22

David Tarjan - NVIDIA

Memory coalescing - SoA or AoS?

- Structure of array

```
struct SoA
{
   int *key;
   int *value;
   int *flag;
};

SoA *d_AoA_data;
cudaMalloc((void**) &d_SoA_data.keys, ...);
cudaMalloc((void**) &d_SoA_data.value, ...);
cudaMalloc((void**) &d_SoA_data.flag, ...);
```


David Tarjan - NVIDIA

Memory coalescing - SoA or AoS?

- cudaMalloc guarantees aligned memory, then accessing the SoA will be much more efficient

```
__global__ void bar (record *AoS_data, SoA SoA_data)
{
  int i = blockDim.x*blockIdx.x + threadIdx.x;
  // AoS wastes bandwidth
  int key = AoS_data[i].key;
  // SoA efficient use of bandwidth
  int key_better = SoA_data.keys[i];
}
```

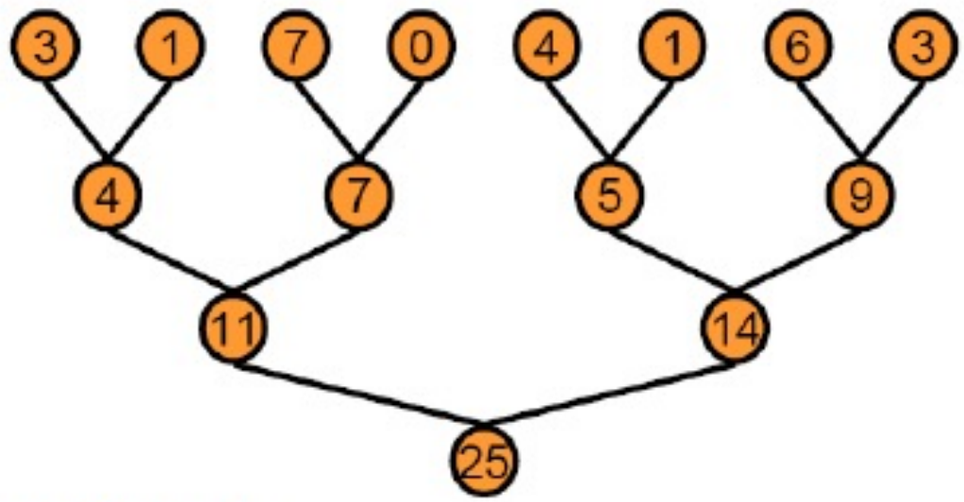

David Tarjan - NVIDIA

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

Parallel Reduction

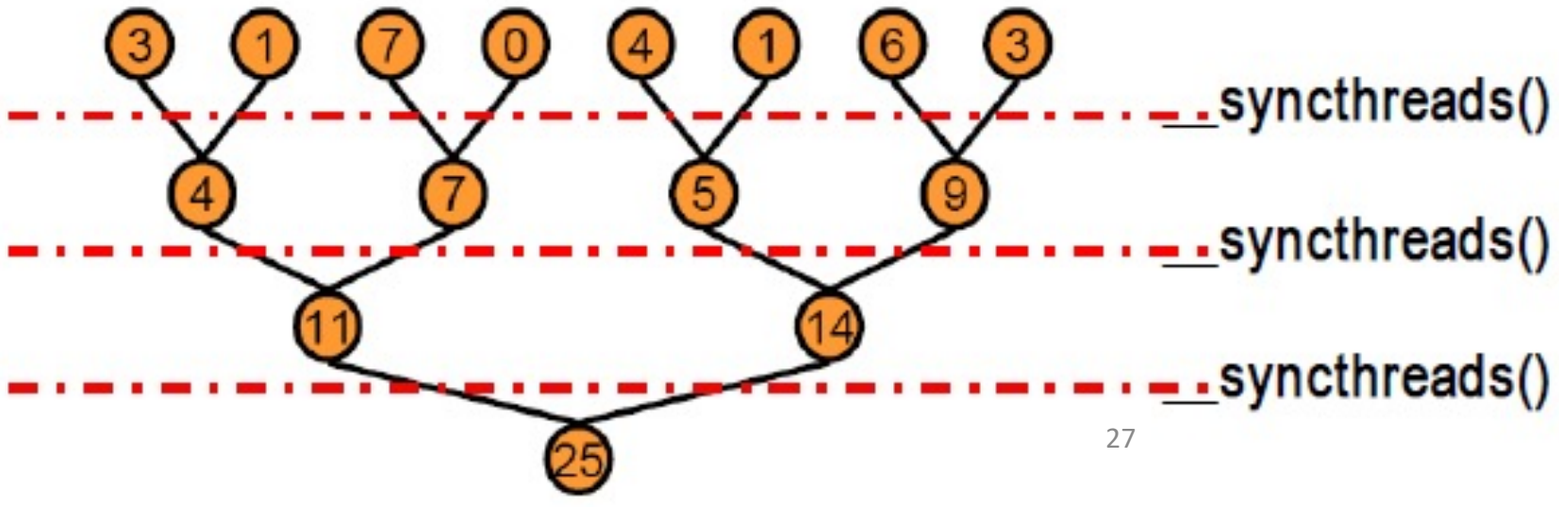
- Tree-based approach used within each thread block



- 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



27

Parallel Reduction: Global Synchronization

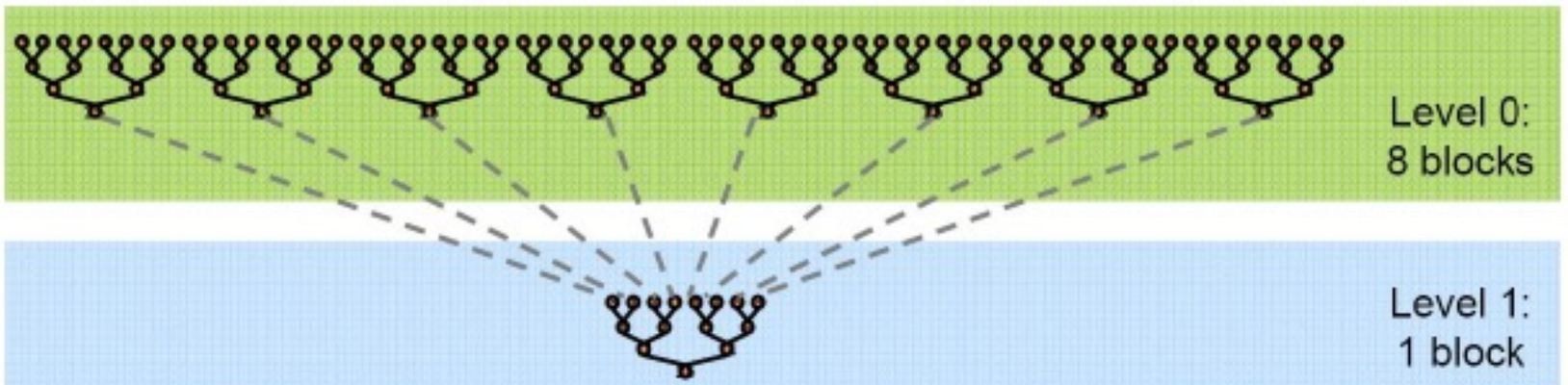
- Problem of CUDA — not support global synchronization
  - Too many kernels, hardware synchronization is costly
  - Dead-lock

- Solution
  - Thread synchronization within thread block

28

Parallel Reduction: Decomposition

- Partition the data into blocks/kernels



- Code in each thread is identical
- Iterative

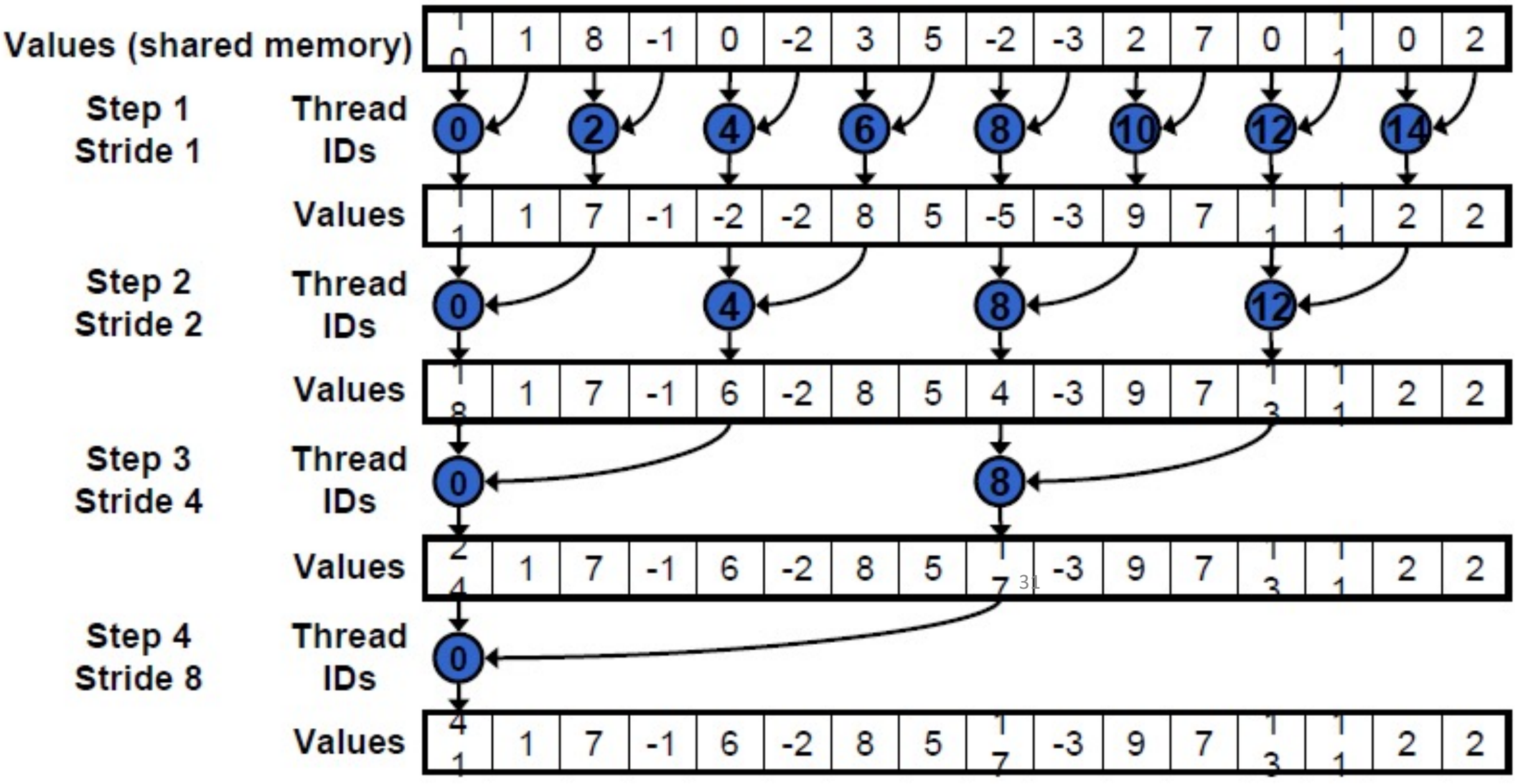
29

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, 900 MHz DDR
  - 384 * 900/4 = 86.4 GB/s

Reduction #1: 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];
}
```

32

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

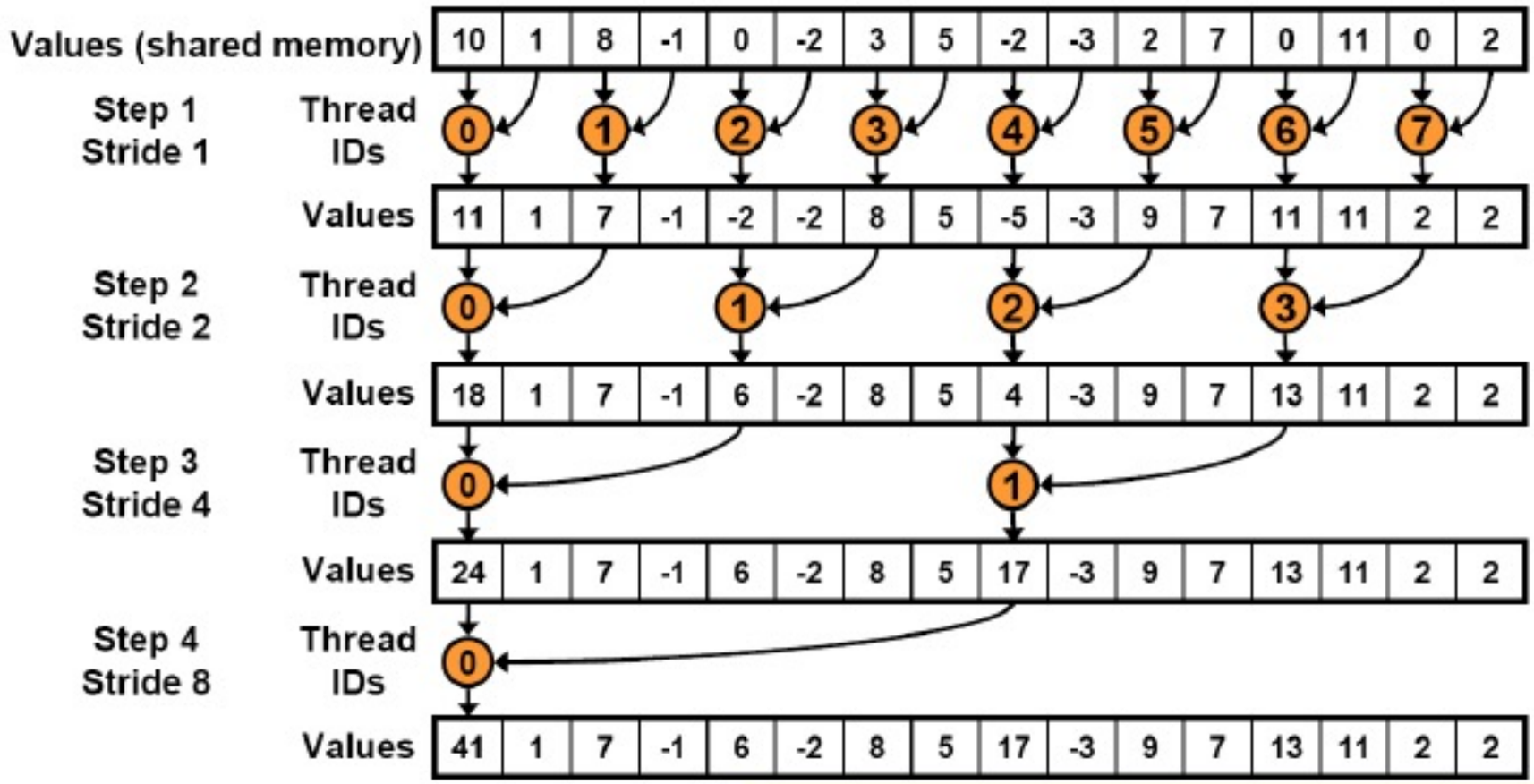
```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
      if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
}
```

- 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();
}
```

35

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();
}
```

Performance for 4M Element Reduction

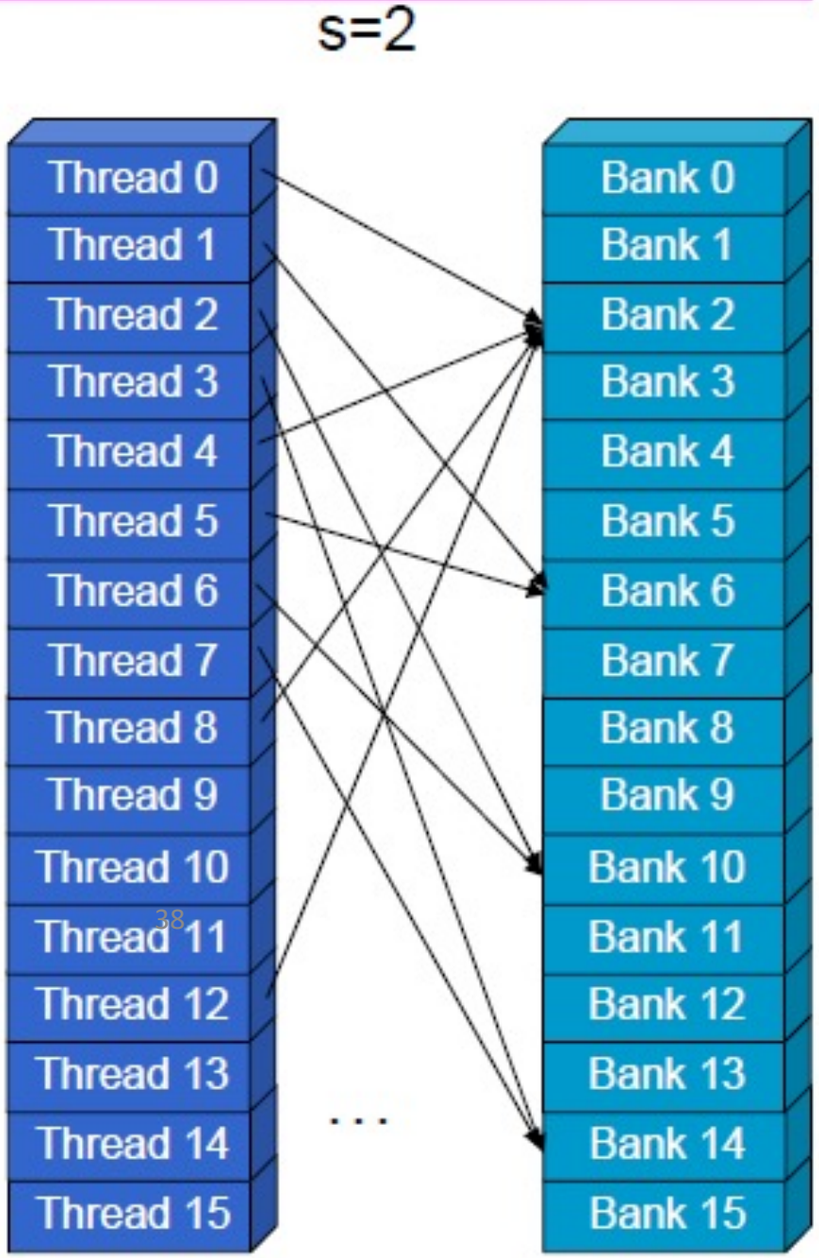
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

37

Reduction #2: Interleaved Addressing

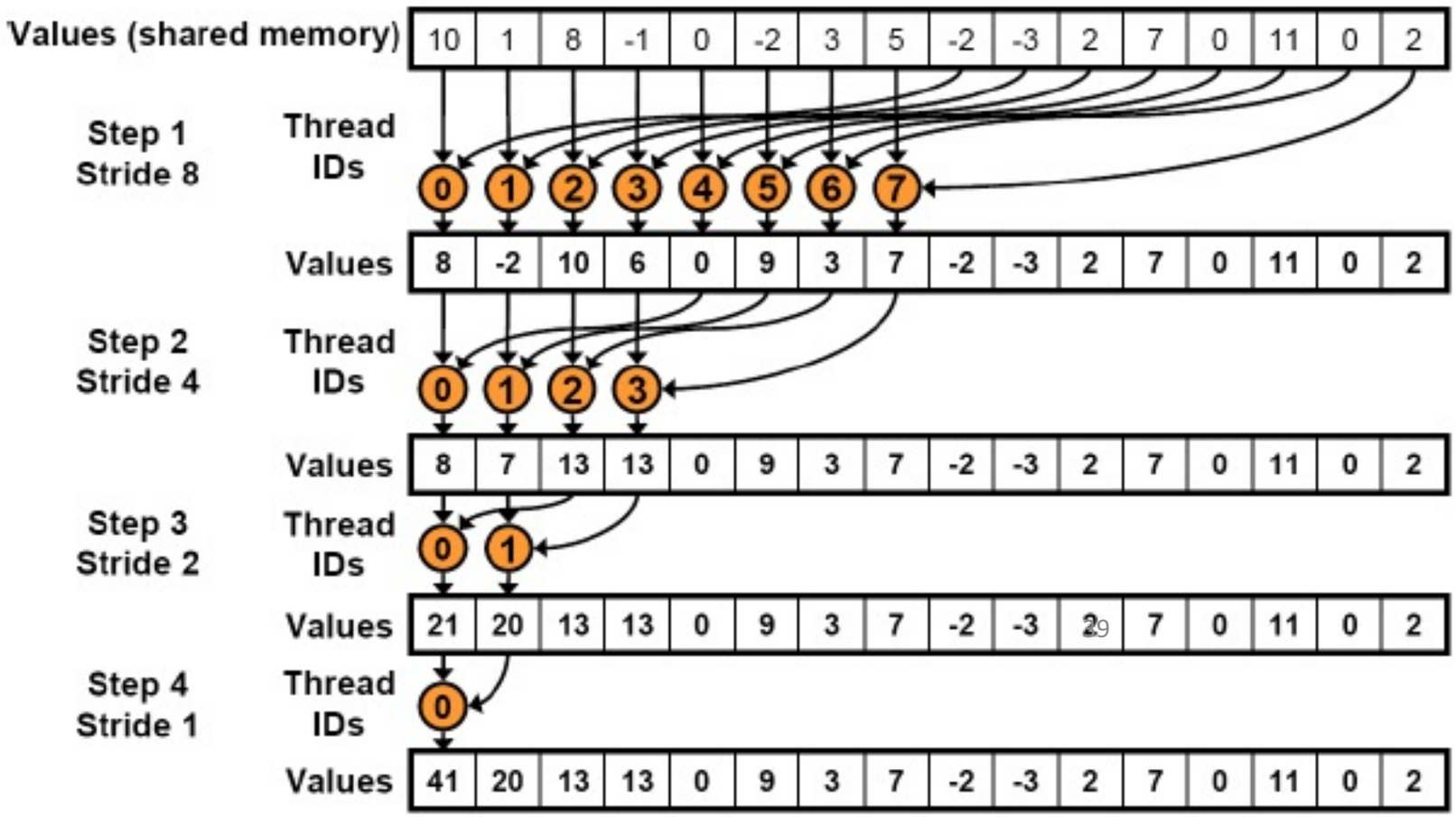
- Bank conflict!

```
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

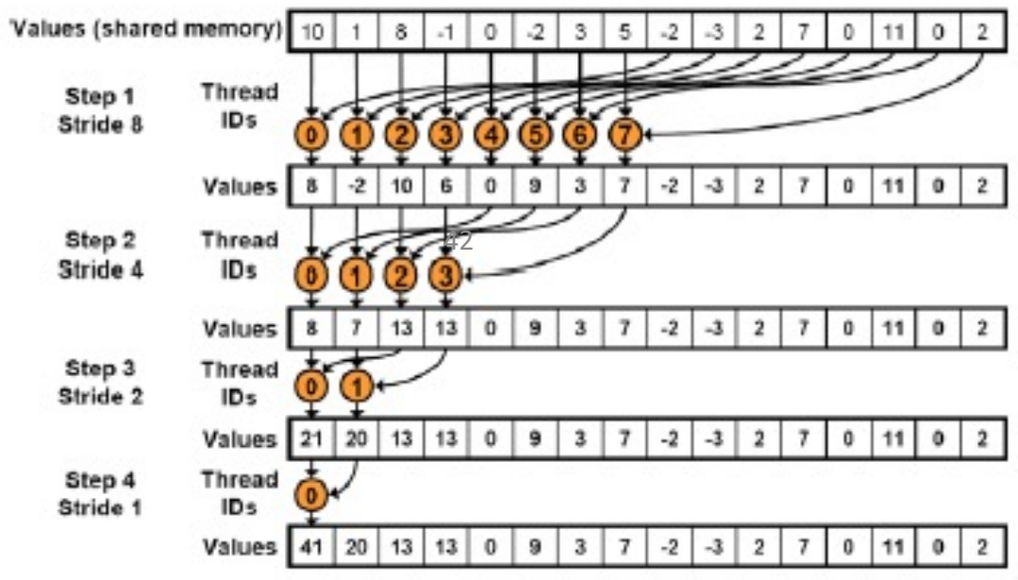
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

41

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

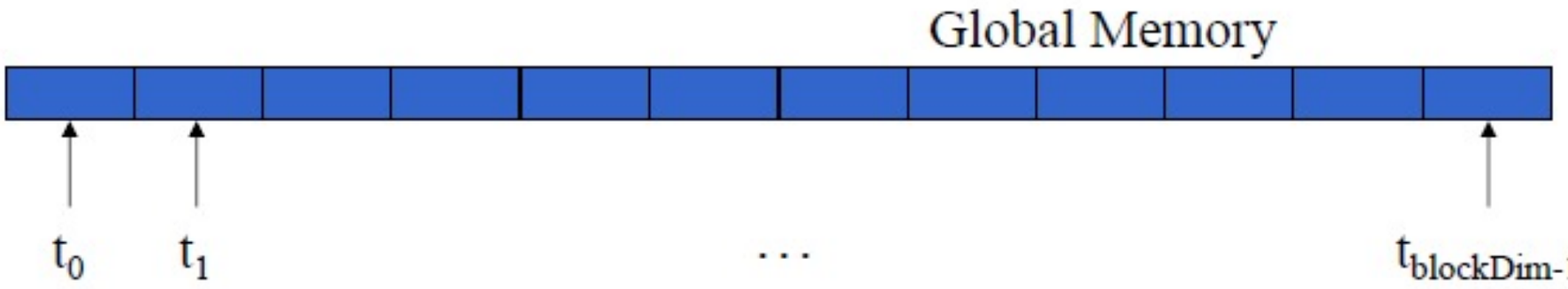
```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

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

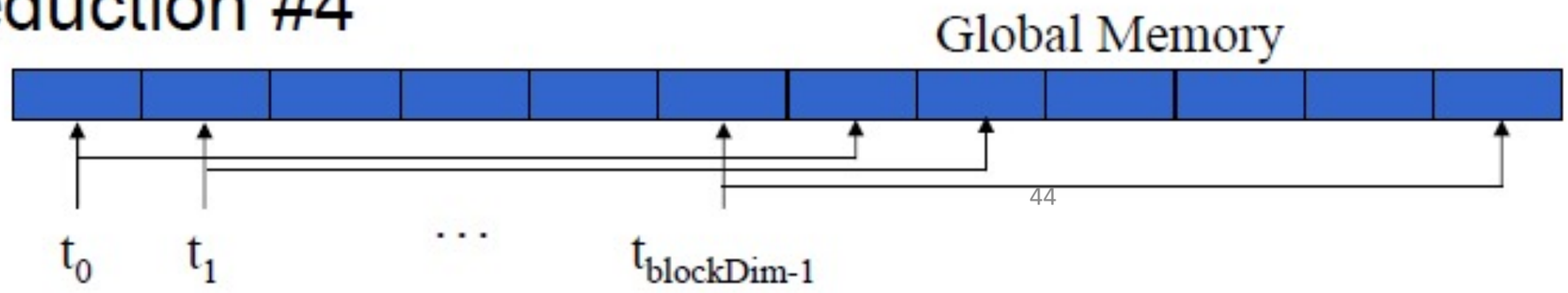
```
// perform first level of reduction
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Reduction #4: First Add During Load

- Reduction #3



- Reduction #4



Performance for 4M Element Reduction

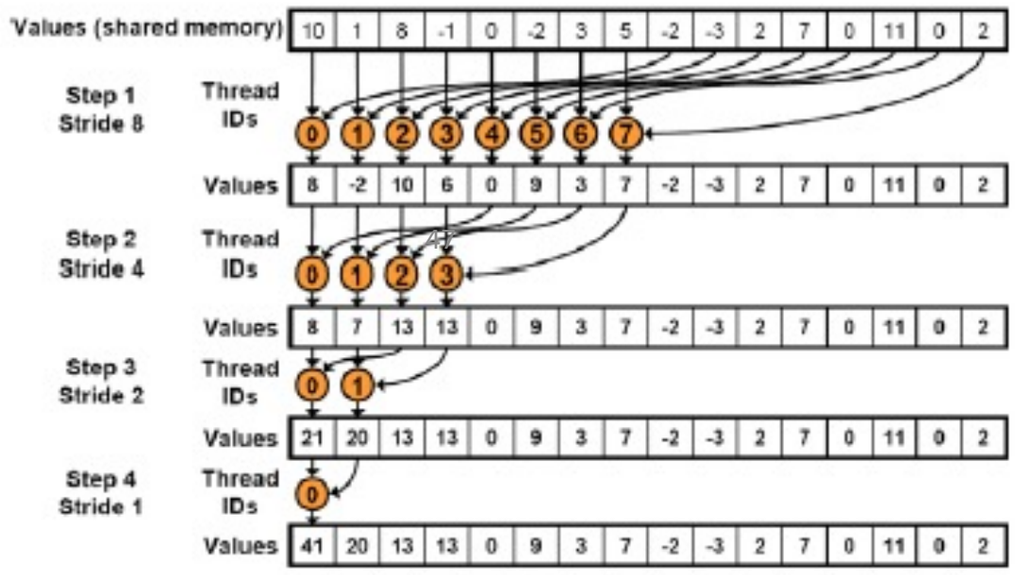
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

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];
}
```

48

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

49

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,…
- 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

Unrolling with Templates

- Specify blocksize 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) {
    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];
}
```

52

All code in RED will be evaluated at compile time

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

54

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();
```

55

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

57

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];
        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];
}
```

58

final version

Parallel Reduction Summary

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

GPU Computing with CUDA
Lecture 6 - CUDA Libraries - Thrust

Christopher Cooper
Boston University

August, 2011
UTFSM, Valparaíso, Chile

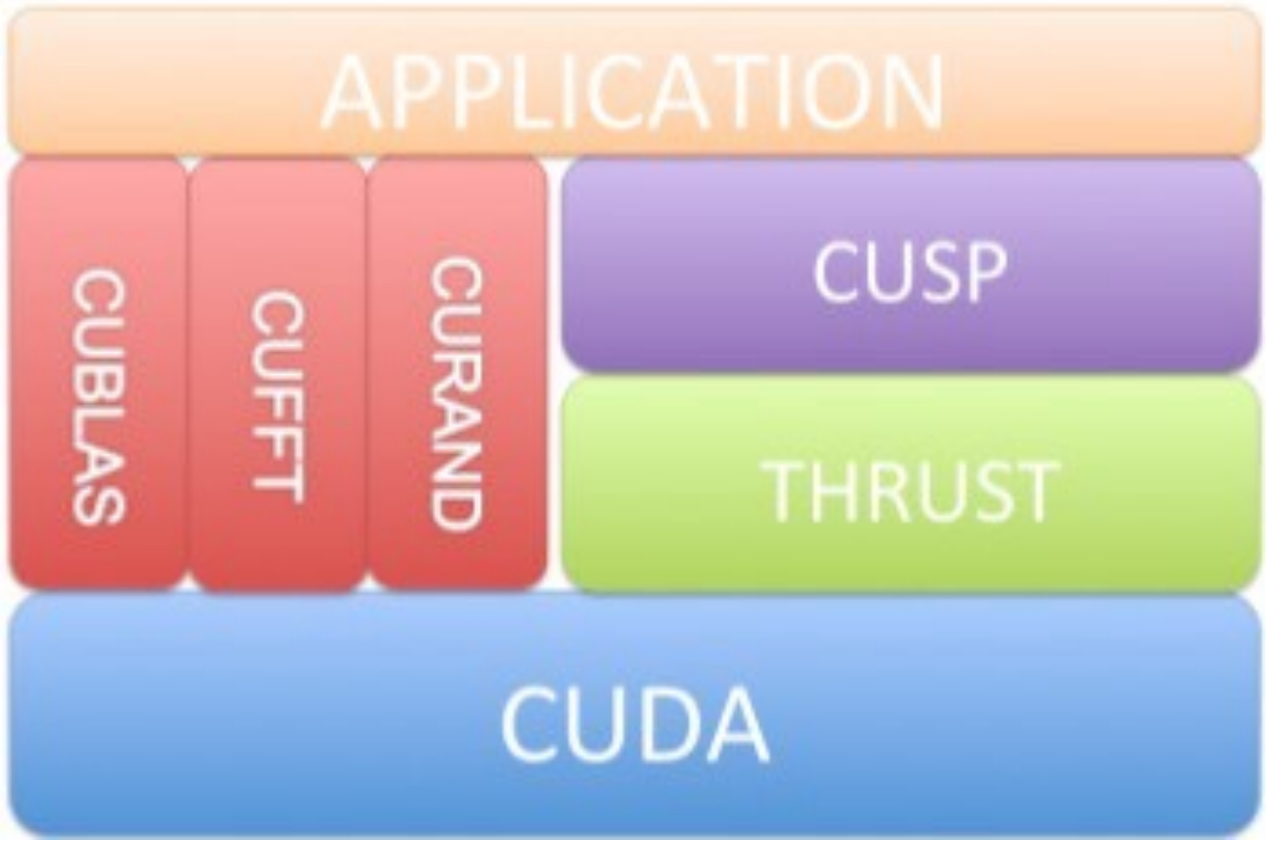
BU College of Engineering

Outline of lecture

- CUDA Libraries
- What is Thrust?
- Features of thrust
- Best practices in thrust

CUDA Libraries

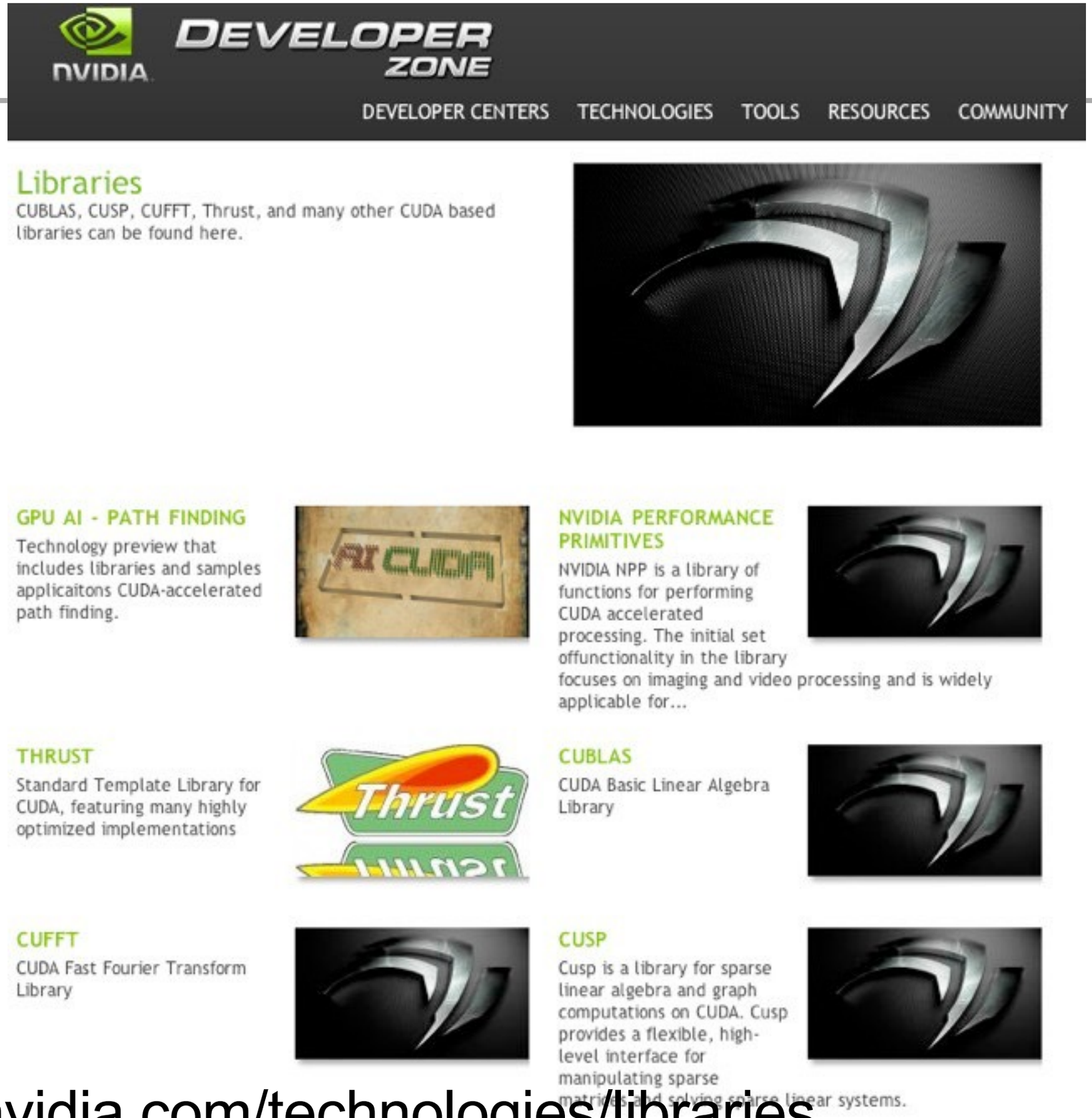
- NVIDIA has developed several libraries to abstract the user from CUDA



Bell, Dalton, Olson. Towards AMG on GPU

CUDA Libraries

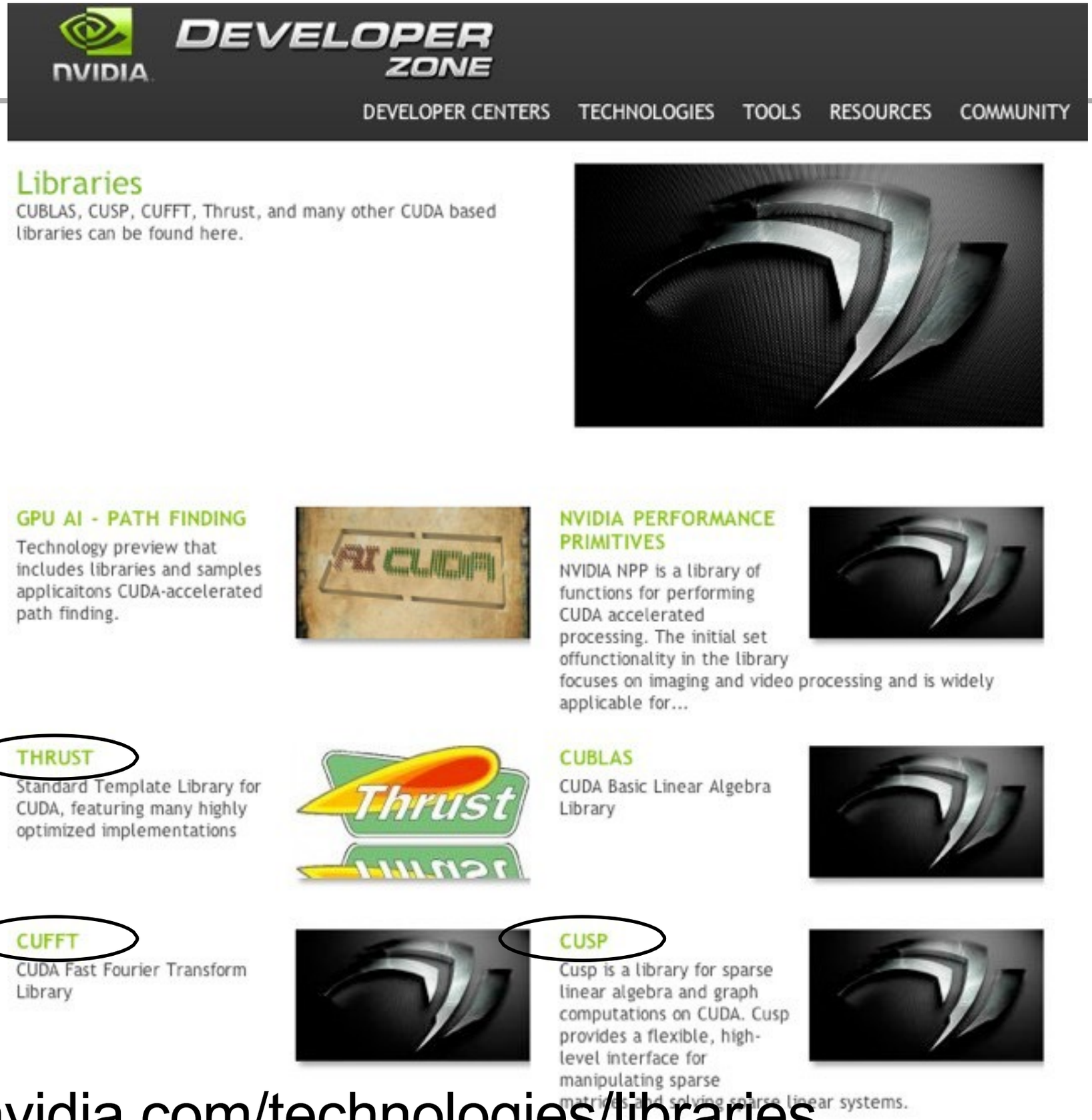
- CUDPP
- MAGMA
- IMSL
- VSIPL
- NVML
- CUPTI
- …..



http://developer.nvidia.com/technologies/libraries

CUDA Libraries

- CUDPP
- MAGMA
- IMSL
- VSIPL
- NVML
- CUPTI
- …..



http://developer.nvidia.com/technologies/libraries

Thrust - Introduction

- Template library for CUDA
  - Resembles C++ Standard Template Library (STL)
  - Collection of data parallel primitives
- Objectives
  - Programmer productivity
  - Encourage generic programming
  - High performance
  - Interoperability
- Comes with CUDA 4.0

Thrust - Introduction

- Containers
  - `thrust::host_vector<T>`
  - `thrust::device_vector<T>`
- Algorithms
  - `thrust::sort()`
  - `thrust::reduce()`
  - `thrust::inclusive_scan()`
- http://code.google.com/p/thrust/
- Slides from Nathan Bell and Jared Hoberock - NVIDIA

Thrust - Containers

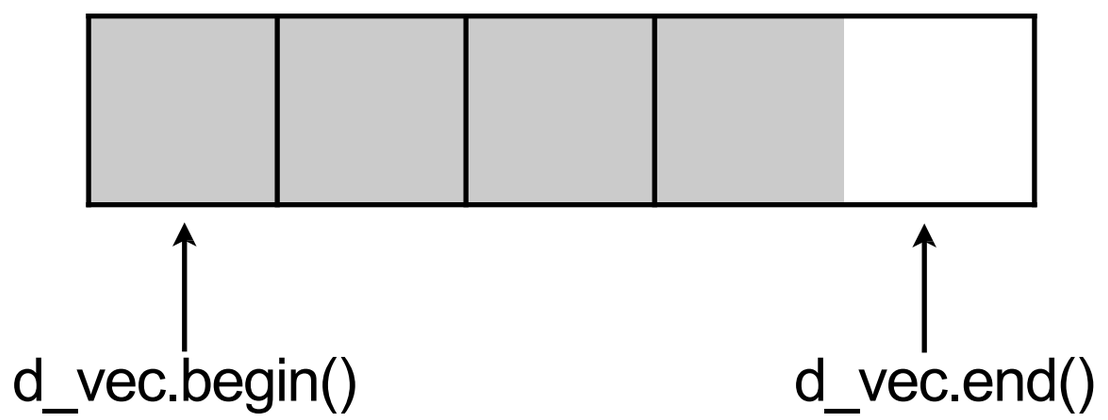
- Thrust provides two vector containers
  - `host_vector`: resides on CPU
  - `device_vector`: resides on GPU
- Hides cudaMalloc and cudaMemcpy

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;
// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;                    + d_vec[1] << std::endl;

std::cout << "sum: " << d_vec[0]
// vector memory automatically released w/  free() or cudaFree()
```

Thrust - Iterators

- Iterators can be thought as pointers to array elements
  - They carry other information

```
// allocate device vector
thrust::device_vector<int> d_vec(4);
d_vec.begin(); // returns iterator at first element of
d_vec
d_vec.end()//  returns iterator one past the last
element of d_vec
// [begin, end] pair defines a sequence of 4 elements
```



8

Thrust - Iterators

- Use iterators like pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);
thrust::device_vector<int>::iterator begin = d_vec.begin();
*begin = 13;          // same as d_vec[0] = 13;
int temp = *begin;    // same as temp = d_vec[0];
begin++;              // advance iterator one position
*begin = 25;          // same as d_vec[1] = 25;
```

Thrust - Iterators

- Important: keep track of your memory space

```
// initialize random values on host
thrust::host_vector<int> h_vec(1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
```

Thrust - Iterators

- Convertible to raw pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<<N/256, 256>>>(N, ptr);

// Note: ptr cannot be dereferenced on the host!
```

Thrust - Iterators

- Wrap raw pointers to use in thrust

```
int N = 10;

// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```

Thrust - Algorithms

- Standard algorithms
  - Reductions
  - Transformations
  - Prefix sums
  - Sorting
- Many have straight analog to STL
- You can use your own user defined types

Thrust - Algorithms

- Reductions

```
#include <thrust/reduce.h>

// declare storage
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers (equivalent calls)
reduce(i_vec.begin(), i_vec.end());
reduce(i_vec.begin(), i_vec.end(),    0, plus<int>());

// sum of floats (equivalent calls)
reduce(f_vec.begin(), f_vec.end());
reduce(f_vec.begin(), f_vec.end(), 0.0f, plus<float>());

// maximum of integers
reduce(i_vec.begin(), i_vec.end(),    0, maximum<int>());
```

Thrust - Algorithms

- Thrust comes with lots of important built in transformations

```
#include<thrust/device_vector.h>
#include<thrust/transform.h>
#include<thrust/sequence.h>
#include<thrust/copy.h>
#include<thrust/fill.h>
#include<thrust/replace.h>
#include<thrust/functional.h>
#include<iostream>

//allocate three device_vectors with 10 elements
thrust::device_vector<int>X(10);
thrust::device_vector<int> Y(10);
thrust::device_vector<int>Z(10);
//initialize X to 0, 1, 2, 3,....
thrust::sequence(X.begin(),X.end());
//compute Y=-X
thrust::transform(X.begin(),X.end(),Y.begin(),thrust::negate<int>());
//fill Z with two s
thrust::fill(Z.begin(),Z.end(),2);
//compute Y = X mod 2
thrust::transform(X.begin(),X.end(),Z.begin(),Y.begin(),thrust::modulus<int>());
//replace all the ones in Y with tens
thrust::replace(Y.begin(),Y.end(),1,10);
//print Y
thrust::copy(Y.begin(),Y.end(),std::ostream_iterator<int>(std::cout,"\n"));
return 0;
```

Thrust - Algorithms

- Prefix sums

```
#include<thrust/scan.h>
int data [6] = {1,0,2,2,1,3};
thrust::inclusive_scan(data, data + 6,  data);//in-place scan
//data is now {1,1,3,5,6,9}
```

data[2] = data[0] + data[1] + data[2]

```
#include<thrust/scan.h>
int data [6] = {1,0,2,2,1,3};
thrust::exclusive_scan(data, data + 6,  data);//in-place scan
//data is now {0,1,1,3,5,6}
```

data[2] = data[0] + data[1]

Thrust - Algorithms

- Sorting

```
#include<thrust/sort.h>
. . .
const int N=6;
int A [N] = {1,4,2,8,5,7};
thrust::sort(A,A+N);
// A is now {1,2,4,5,7,8}
```

Thrust - Fancy iterators

- Behave like “normal’” iterators
  - Also they can be seen as pointers
- Examples
  - `constant_iterator`
  - `counting_iterator`
  - `transform_iterator`
  - `permutation_iterator`
  - `zip_iterator`

Thrust - Fancy iterators

- `constant_iterator`

  - Mimics an infinite array with constant values

```
// create iterators
constant_iterator<int> begin(10);
constant_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 10
begin[100] // returns 10

// sum of (begin, end)
reduce(begin, end);   // returns 30 (i.e. 3 * 10)
```

Thrust - Fancy iterators

- `counting_iterator`

  - Mimics an infinite array with sequential values

```
// create iterators
counting_iterator<int> begin(10);
counting_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 11
begin[100] // returns 110

// sum of (begin, end)
reduce(begin, end);   // returns 33 (i.e. 10 +
11 + 12)
```

Thrust - Fancy iterators

- `transform_iterator`

  - Allows us to fuse separate algorithms into one

```
// initialize vector
device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// create iterator (type omitted)
begin = make_transform_iterator(vec.begin(), negate<int>());
end   = make_transform_iterator(vec.end(),   negate<int>());

begin[0]   // returns -10
begin[1]   // returns -20
begin[2]   // returns -30

// sum of [begin, end)
reduce(begin, end);   // returns -60 (i.e. -10 + -20 + -30)
```

Thrust - Fancy iterators

- permutation_iterator

  - Allows to fuse gather and scatter operations

```
#include<thrust/iterator/permutation_iterator.h>
...
//gather locations
thrust::device_vector<int> map(4);
map[0] = 3;
map[1] = 1;
map[2] = 0;
map[3] = 5;

//array to gather from
thrust::device_vector<int> source(6);
source[0] = 10;
source[1] = 20;
source[2] = 30;
source[3] = 40;
source[4] = 50;
source[5] = 60;

//fuse gather with reduction: sum = source[map[0]] + source[map[1]]+...
int sum = thrust::reduce(thrust::make_permutation_iterator(source.begin(),map.begin()),
thrust::make_permutation_iterator(source.begin(),map.end()));
```

Thrust - Fancy iterators

- `zip_iterator`

  - Looks like an array of structs

  - Stored in structure of arrays

```
// initialize vectors
device_vector<int>  A(3);
device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = ‘x’; B[1] = ‘y’; B[2] = ‘z’;

// create iterator (type omitted)
begin = make_zip_iterator(make_tuple(A.begin(), B.begin()));
end   = make_zip_iterator(make_tuple(A.end(),   B.end()));

begin[0]   // returns tuple(10, ‘x’)
begin[1]   // returns tuple(20, ‘y’)
begin[2]   // returns tuple(30, ‘z’)

// maximum of [begin, end)
maximum< tuple<int,char> > binary_op;
reduce(begin, end, begin[0], binary_op); //  returns tuple(30, ‘z’)
```

Thrust - Best practices

- Fusion
  - Combine related operations together
- Structure of Arrays
  - Ensure memory coalescing
- Implicit Sequences
  - Eliminate memory accesses

Thrust - Fusion

- Combine related operations together
  - Conserves memory bandwidth
- Example: norm of a vector
  - Square each element
  - Compute the sum of squares and take sqrt()

Thrust - Fusion

- Unoptimized example

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_slow(device_vector<float>& x)
{
  // without fusion
  device_vector<float> temp(x.size());
  transform(x.begin(), x.end(), temp.begin(), square());

  return sqrt( reduce(temp.begin(), temp.end()) );
}
```

Thrust - Fusion

- Optimized implementation (3.8x)

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_fast(device_vector<float>& x)
{
  // with fusion
  return sqrt( transform_reduce(x.begin(), x.end(), square(), 0.0f, plus<float>());
}
```

Thrust - Structure of Arrays (SoA)

- Array of structures (AoS)
  - Often does not obey coalescing rules

```
device_vector<float3>
```

- Structure of arrays (SoA)
  - Obeys coealescing rules
  - Components stored in separate arrays

```
device_vector<float> x,y,z;
```

- Example: rotate 3D vectors

Thrust - Structure of Arrays (SoA)

```
struct rotate_float3
{
  __host__ __device__
  float3 operator()(float3 v)
  {
    float x = v.x;
    float y = v.y;
    float z = v.z;
    float rx = 0.36f*x +  0.48f*y + -0.80f*z;
    float ry =-0.80f*x +  0.60f*y +  0.00f*z;
    float rz = 0.48f*x +  0.64f*y +  0.60f*z;
    return make_float3(rx, ry, rz);
  }
};
...
device_vector<float3> vec(N);
transform(vec.begin(), vec.end, vec.begin(), rotate_float3());
```

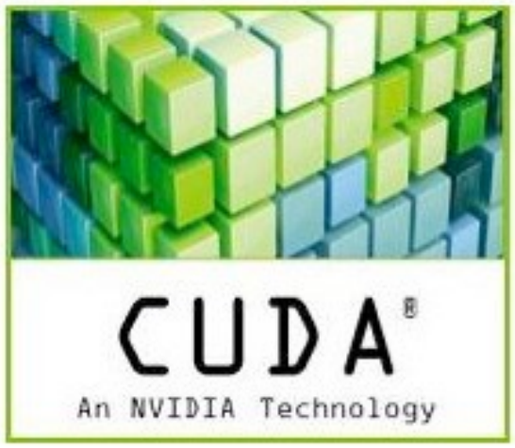
Thrust - Structure of Arrays (SoA)

```
struct rotate_tuple
{
  __host__ __device__
  tuple<float,float,float> operator()(tuple<float,float,float> v)
  {
    float x = get<0>(v);
    float y = get<1>(v);
    float z = get<2>(v);
    float rx = 0.36f*x +  0.48f*y + -0.80f*z;
    float ry =-0.80f*x +  0.60f*y +  0.00f*z;
    float rz = 0.48f*x +  0.64f*y +  0.60f*z;
    return make_tuple(rx, ry, rz);
  }
};
...
device_vector<float> x(N), y(N), z(N);
transform(make_zip_iterator(make_tuple(x.begin(), y.begin(),z.begin())),
          make_zip_iterator(make_tuple(x.end(),   y.end(),   z.end())),
          make_zip_iterator(make_tuple(x.begin(), y.begin(),z.begin())),
          rotate_tuple());
```

PyCUDA

- Python + CUDA = PyCUDA
- Python: scripting language ⟶ easy to code, but slow
- CUDA ⟶ difficult to code, but fast!
- PyCUDA wants to take the best of both worlds

- http://mathema.tician.de/software/pycuda



PyCUDA

- Scripting language
  - High level programming language that is interpret by another program at runtime rather than compiled
  - Advantages: ease on programmer
  - Disadvantages: slow (specially inner loops)
- PyCUDA
  - CUDA codes does not need to be a constant at compile time
  - Machine generated code: automatic manage of resources

PyCUDA

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

PyCUDA

- Transferring data

```
import numpy
a = a.astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)

cuda.memcpy_htod(a_gpu, a)
```

- Executing a kernel

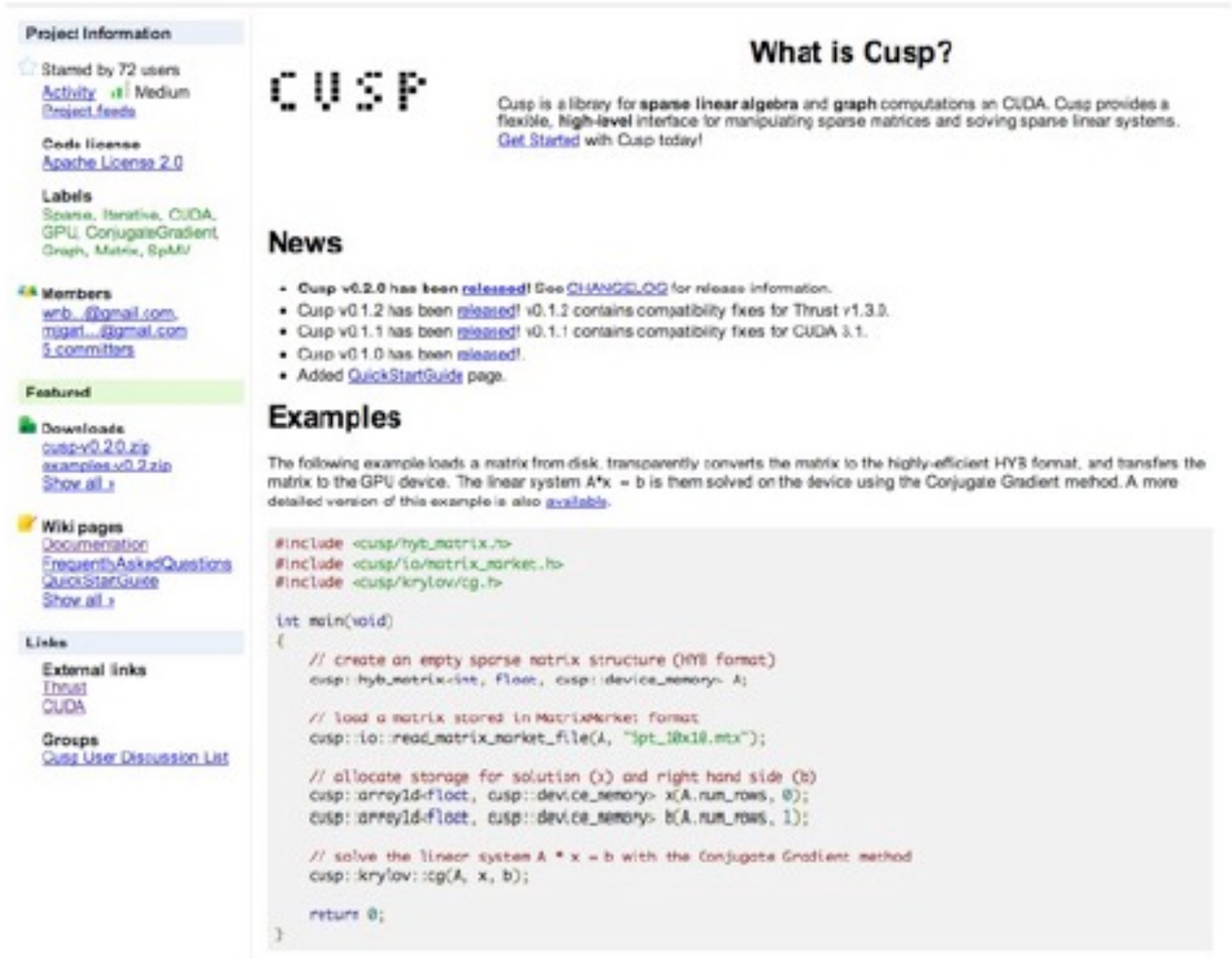
```
from pycuda.compiler import SourceModule
mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)
... # Allocate, generate and transfer
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

CUFFT

- CUFFT: CUDA library for FFTs on the GPU
- Supported by NVIDIA
- Features:
  - 1D, 2D, 3D transforms for complex and real data
  - Batch execution for multiple transforms
  - Up to 128 million elements (limited by memory)
  - In-place or out-of-place transforms
  - Double precision on GT200 or later
  - Allows streamed execution: simultaneous computation and data movement

Cusp

- Library for sparse linear algebra
- http://code.google.com/p/cusp-library/



Project Information

Starred by 72 users
Activity: Medium
Project feeds

Code license
Apache License 2.0

Labels
Sparse, Iterative, CUDA, GPU, ConjugateGradient, Graph, Matrix, SpMV

Members
wnb...@gmail.com, mgarl...@gmail.com
5 committers

Featured

Downloads
cusp-v0.2.0.zip
examples-v0.2.zip
Show all »

Wiki pages
Documentation
FrequentlyAskedQuestions
QuickStartGuide
Show all »

Links

External links
Thrust
CUDA

Groups
Cusp User Discussion List

CUSP

What is Cusp?

Cusp is a library for **sparse linear algebra** and **graph** computations on CUDA. Cusp provides a flexible, **high-level** interface for manipulating sparse matrices and solving sparse linear systems.
Get Started with Cusp today!

News

- **Cusp v0.2.0 has been released!** See CHANGELOG for release information.
- Cusp v0.1.2 has been released! v0.1.2 contains compatibility fixes for Thrust v1.3.0.
- Cusp v0.1.1 has been released! v0.1.1 contains compatibility fixes for CUDA 3.1.
- Cusp v0.1.0 has been released!.
- Added QuickStartGuide page.

Examples

The following example loads a matrix from disk, transparently converts the matrix to the highly-efficient HYB format, and transfers the matrix to the GPU device. The linear system A*x = b is then solved on the device using the Conjugate Gradient method. A more detailed version of this example is also available.

```
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>

int main(void)
{
    // create an empty sparse matrix structure (HYB format)
    cusp::hyb_matrix<int, float, cusp::device_memory> A;

    // load a matrix stored in MatrixMarket format
    cusp::io::read_matrix_market_file(A, "/pt_10x10.mtx");

    // allocate storage for solution (x) and right hand side (b)
    cusp::array1d<float, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<float, cusp::device_memory> b(A.num_rows, 1);

    // solve the linear system A * x = b with the Conjugate Gradient method
    cusp::krylov::cg(A, x, b);

    return 0;
}
```

©2011 Google - Terms - Privacy - Project Hosting Help