Parallel computing using GPU programming CUDA C

Mokhtar Essaid

Université de Haute Alsace – Faculté des Sciences et Techniques
Institut de Recherche en Informatique, Mathématiques, Automatique et Signal (IRIMAS)

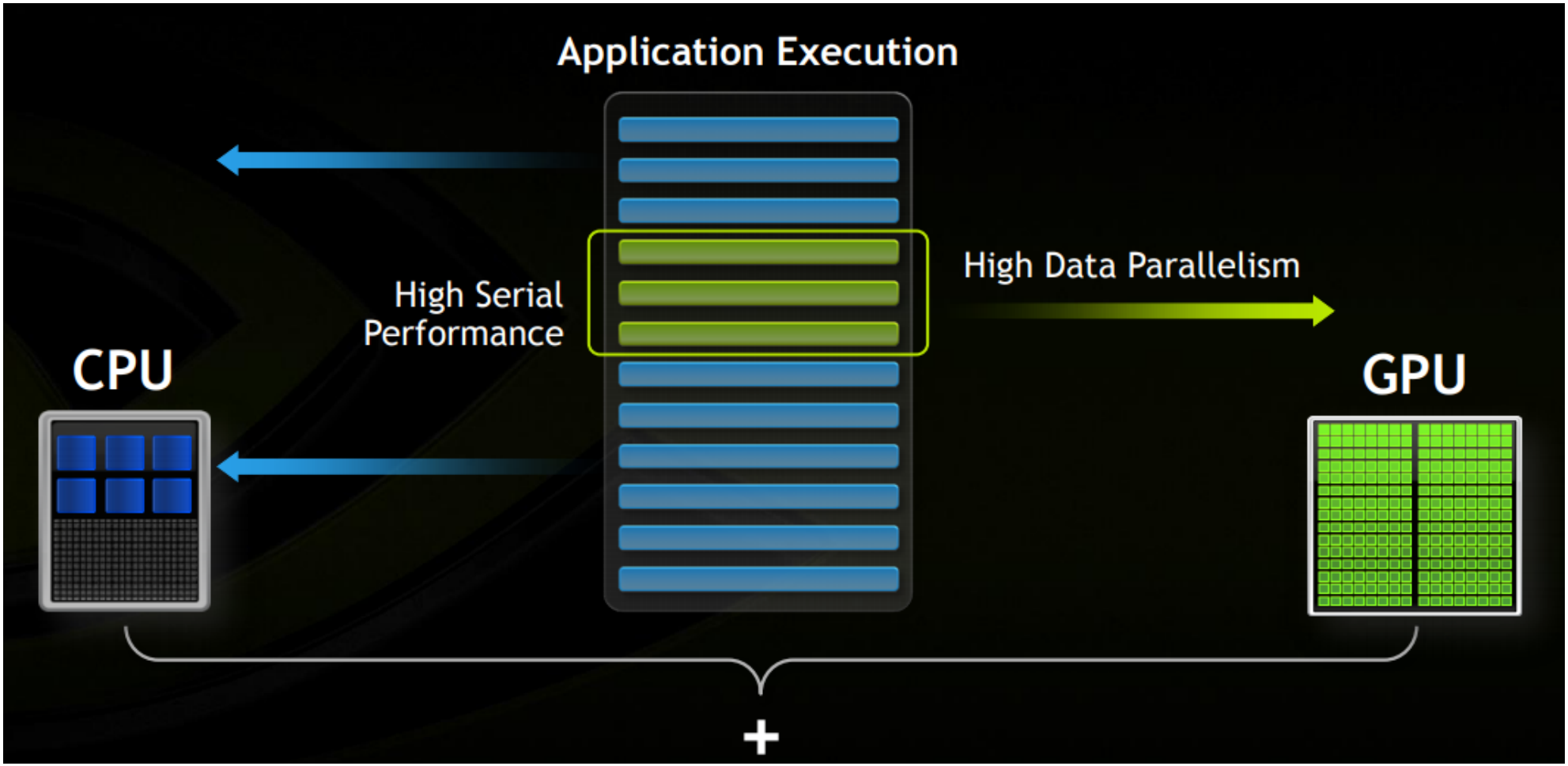
mokhtar.essaid@uha.fr

April 30, 2021

Sommaire

Heterogeneous Parallel Computing with CUDA

What is Heterogeneous Computing?



Low Latency or High Throughput?

F-22 Raptor

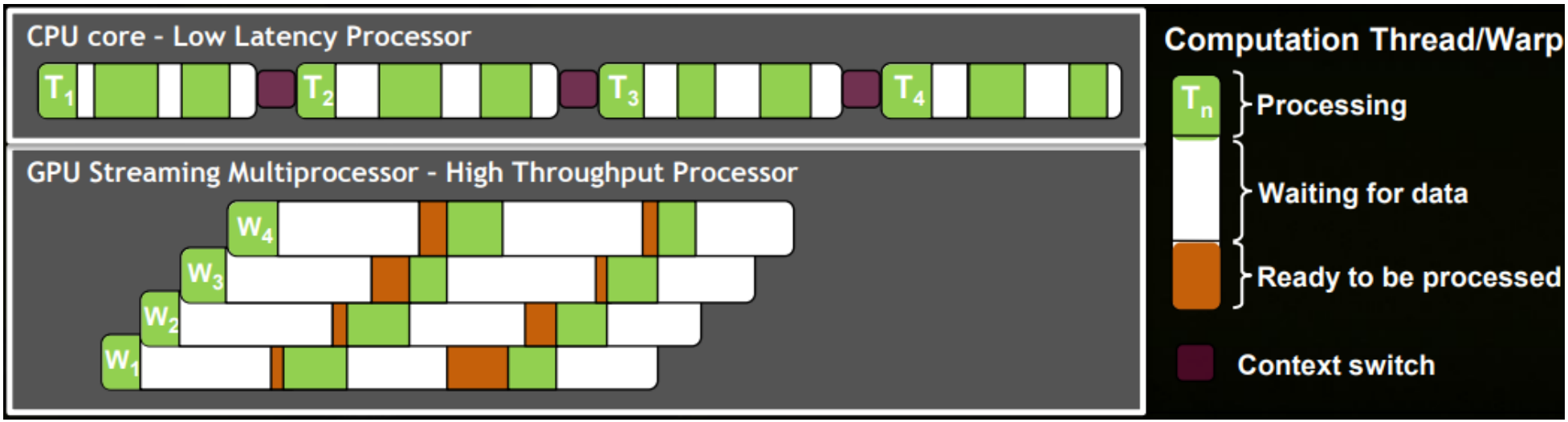
- 1500 mph
- Knoxville to San Jose in 1:25
- Seats 1

Boeing 737

- 485 mph
- Knoxville to San Jose in 4:20
- Seats 200

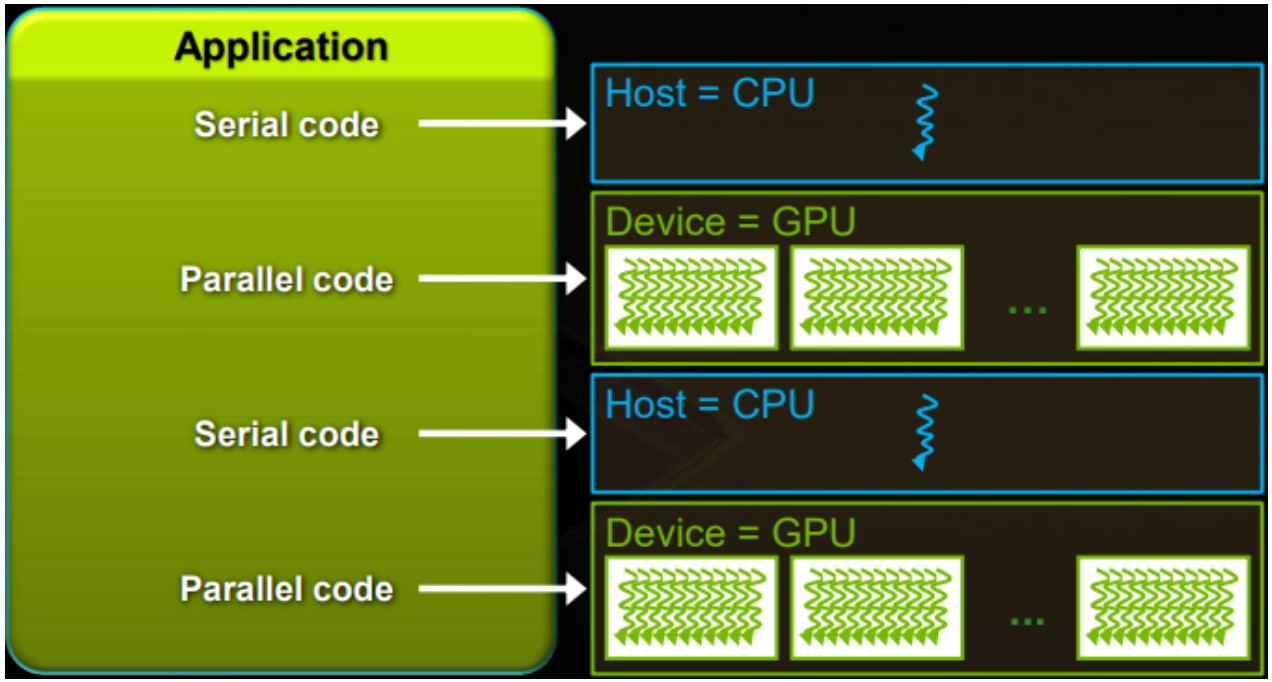
Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other threads

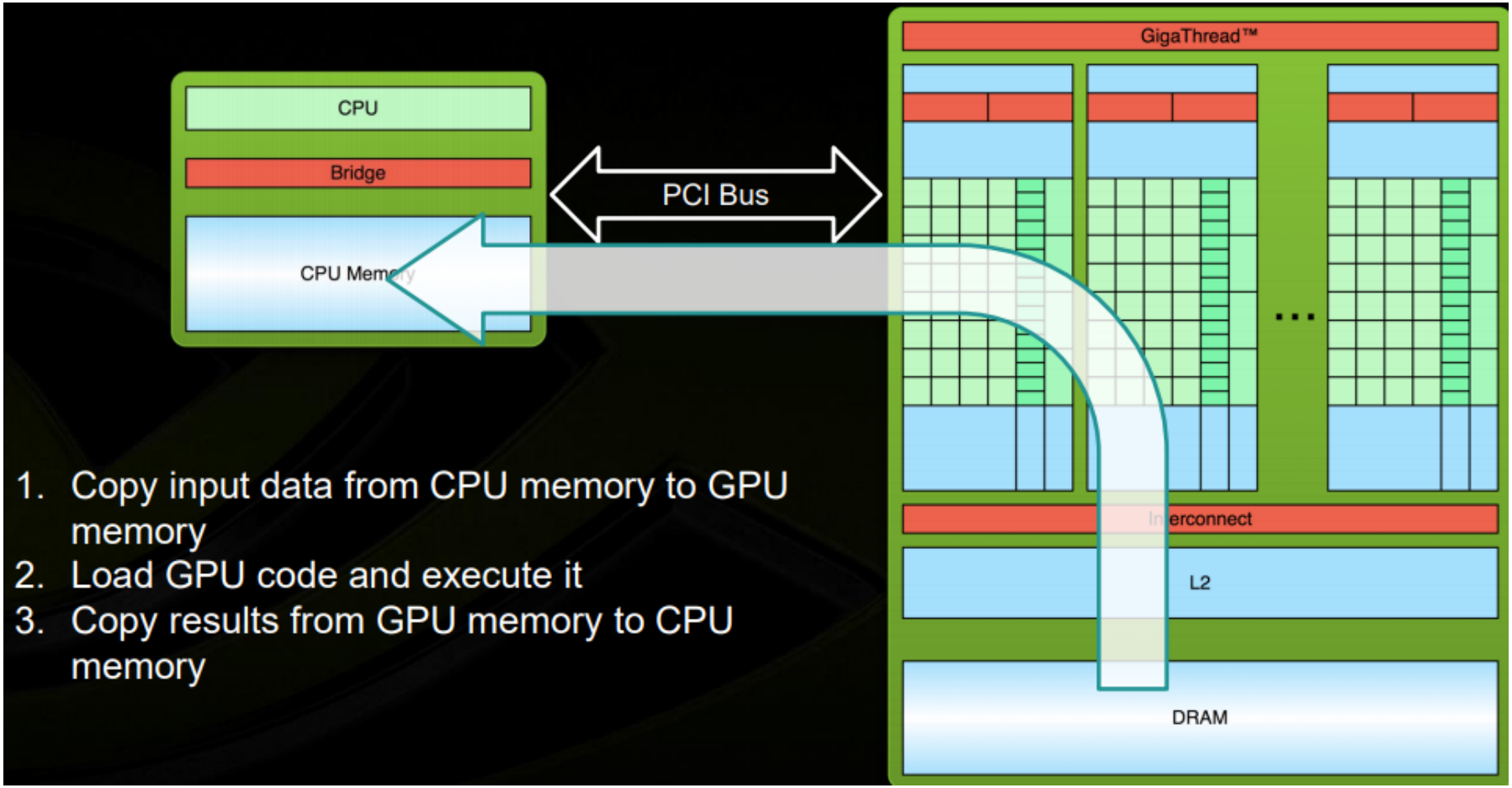


Heterogeneous Parallel Computing with CUDA

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements



Simple Processing Flow



GPU Programming Languages

- Numerical analytics: MATLAB, Mathematica, LabVIEW
- Fortran: CUDA Fortran
- C: CUDA C
- C++: CUDA C++
- Python: PyCUDA, Copperhead
- C#: GPU.NET

What is CUDA?

What is CUDA?

CUDA Platform and Programming Model

- Expose GPU computing for general purpose
- A model how to offload work to the GPU and how the work is executed on the GPU

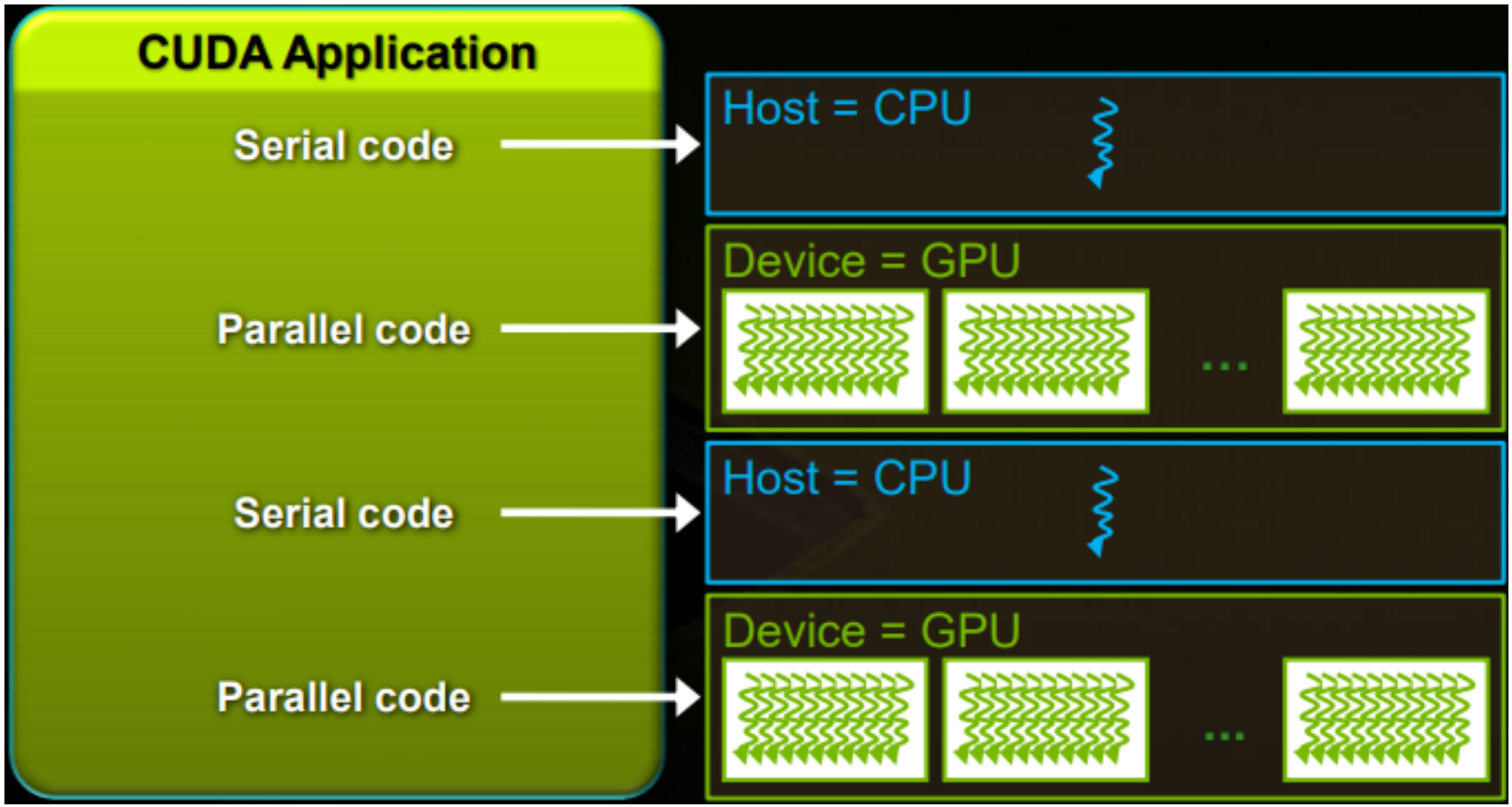
CUDA C/C++

- Based on industry-standard C/C++
- Small set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

What is CUDA?

Anatomy of a CUDA Application

- Serial code executes in a Host (CPU) thread (Functions)
- Parallel code executes in many Device (GPU) threads across multiple processing elements (Kernels)



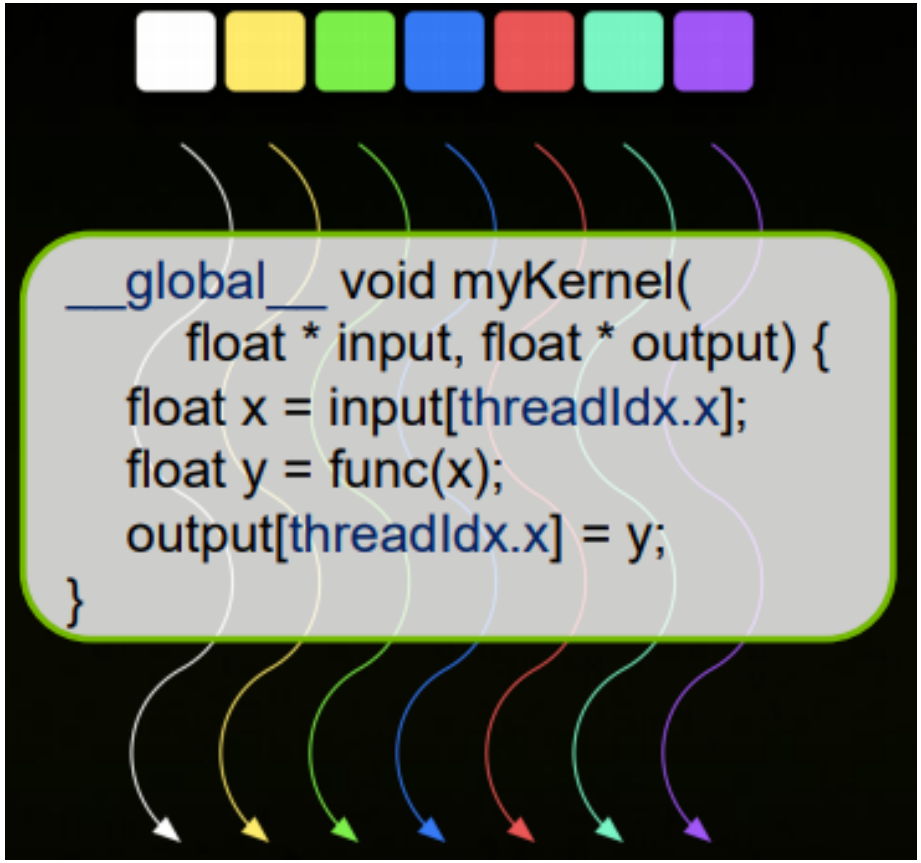
What is CUDA?: Kernels and Parallel Threads

- A kernel is a function executed on the GPU as an array of threads in parallel
All threads execute the same code

  - All threads execute the same code
  - can take different paths
  - but the fewer divergence between "neighboring" threads the better

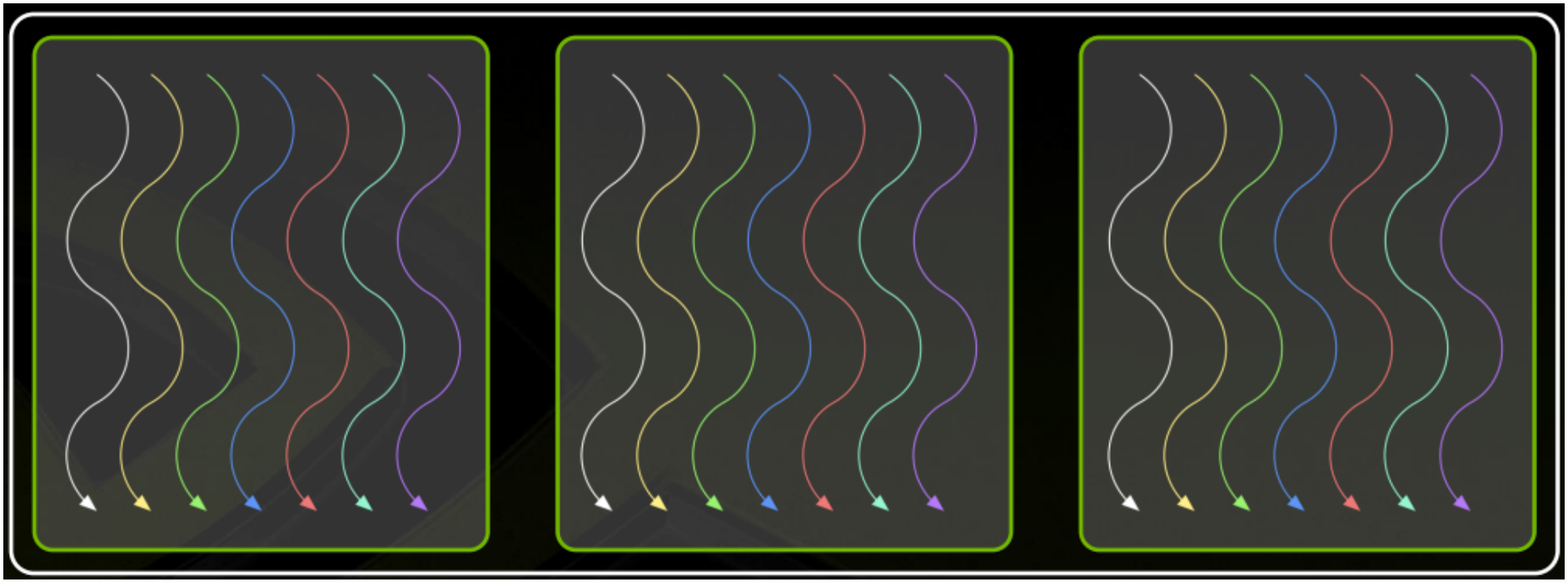
Each thread has an ID

  - Select input/output data
  - Control decisions



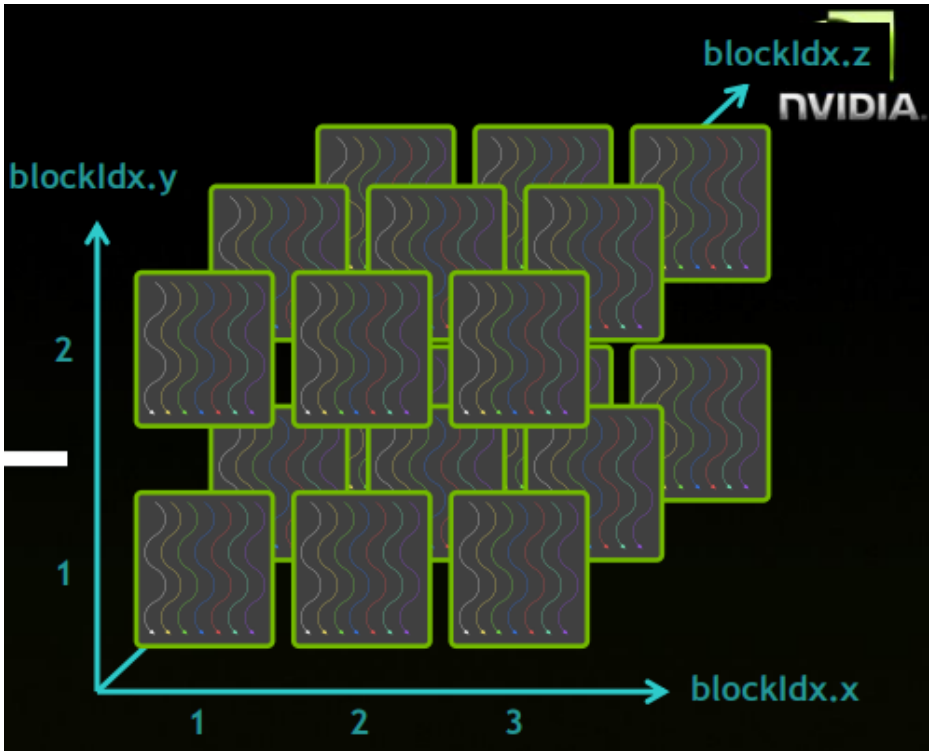
What is CUDA?: Kernels and Parallel Threads

CUDA Kernels: Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid

What is CUDA?: Kernels and Parallel Threads



- A grid is a 1D, 2D or 3D index space
- Threads query their position inside the index space using blockIdx.x,y,z and threadIdx.x,y,z and decide on the work they have to do
- Only threads in the same block can communicate directly (via shared memory)

HELLO WORLD!

```
int main(void) {
printf("Hello World!");
return 0; }
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output:

```
$ module load cudatoolkit
$ nvcc hello_world.cu
$ a.out
Hello World!
```

HELLO WORLD!

```
__global__ void mykernel(void) {     }

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

Two new syntactic elements. . .

HELLO WORLD!

__global__ void mykernel(void)
CUDA C/C++ keyword __global__ indicates a function that:

- Runs on the device
- Is called from host code

nvcc separates source code into host and device components

- Device functions (e.g. mykernel()) processed by NVIDIA compiler
- Host functions (e.g. main()) processed by standard host compiler - gcc, cl.exe

HELLO WORLD!

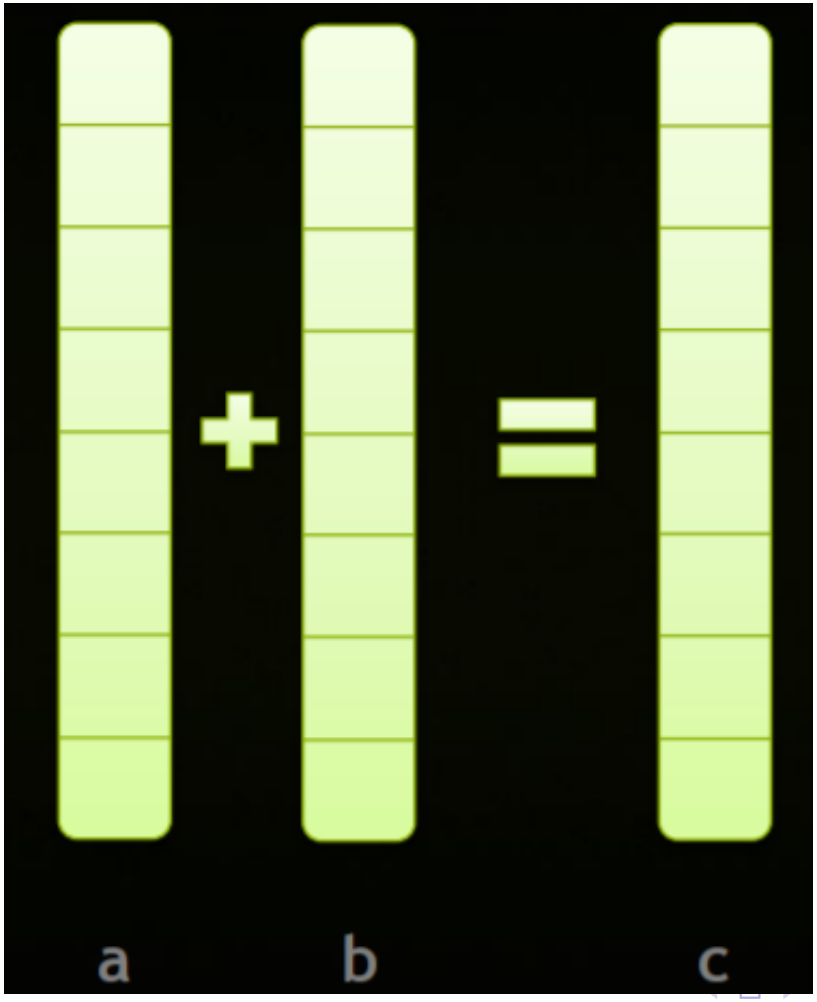
mykernel<<<1,1>>>();
Triple angle brackets mark a call from host code to device code

- Also called a "kernel launch"
- We will return to the parameters (1,1) in a moment

That's all that is required to execute a function on the GPU

Massive parallelism

- GPU computing is about massive parallelism!
- We need a more interesting example. . .
- We'll start by adding two integers and build up to vector addition



Massive parallelism

A simple kernel to add two integers

```
__global__ void add(int a, int b, int *c) {
            *c = a + b;
}
```

As before __global__ is a CUDA C/C++ keyword meaning

- add() will execute on the device
- add() will be called from the host
- Note that we are passing pointers: Necessary for c because we want the result back

Massive parallelism

Memory Management Host and device memory are separate entities

- Device pointers point to GPU memory
  - May be passed to/from host code
  - May not be dereferenced in host code
- Host pointers point to CPU memory
  - May be passed to/from device code
  - May not be dereferenced in device code

Simple CUDA API for handling device memory

- cudaMalloc(), cudaFree(), cudaMemcpy()
- Similar to the C equivalents malloc(), free(), memcpy()

Massive parallelism

Addition on the Device: add()

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_c;                       // device copy of c
        int size = sizeof(int);

        // Allocate space for device copy of c
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Massive parallelism

```
    // Launch add() kernel on GPU
    add<<<1,1>>>(a, b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Massive parallelism

Review

Difference between host and device

- Host CPU
- Device GPU

Using __global__ to declare a function as device code (kernel)

- Executes on the device
- Called from the host

Passing parameters from host code to a device kernel

Massive parallelism

GPU computing is about massive parallelism

- So how do we run code in parallel on the device?

Instead of executing add() once, execute N threads in parallel

Massive parallelism

- With add() running in parallel we can do vector addition
- Terminology: a block can be split into parallel threads
- Let's change add() to use parallel threads instead of a single thread

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use threadIdx.x
- Let's have a look at main()

Massive parallelism

```
// CUDA kernel for vector addition
// __global__ means this is called from the CPU, and runs on the GPU
__global__ void add(int* a, int* b, int* c, int N) {
    // Calculate global thread ID
    int tid =  threadIdx.x;
        // Each thread adds a single element
        c[tid] = a[tid] + b[tid];


}
```

Massive parallelism

```
36  int main()
37  {
38      // Array size of 2^16 (65536 elements)
39       int N = 5;
40      size_t bytes = sizeof(int) * N;
41      srand(time(NULL));
42      // Vectors for holding the host-side (CPU-side) data
43      int* h_a, * h_b, * h_c;
44
45      // Allocate pinned memory
46      cudaMallocHost(&h_a, bytes);
47      cudaMallocHost(&h_b, bytes);
48      cudaMallocHost(&h_c, bytes);
```

Massive parallelism

```
50      // Initialize random numbers in each array
51      for (int i = 0; i < N; i++) {
52          h_a[i] = rand() % 100;
53          h_b[i] = rand() % 100;
54      }
55      cout << "values of array a " << endl;
56      for (int i = 0; i < N; i++)
57      {
58          cout  << h_a[i] << " ";
59      }
60      cout << endl << "----------------- "<<endl ;
61      cout << "values of array b " << endl;
62      for (int i = 0; i < N; i++)
63      {
64          cout << h_b[i] << " ";
65      }
```

Massive parallelism

```
// Allocate memory on the device
int* d_a, * d_b, * d_c;
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

// Copy data from the host to the device (CPU -> GPU)
cudaMemcpy(d_a, h_a, sizeof(int) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, sizeof(int) * N, cudaMemcpyHostToDevice);

// Threads per CTA (1024 threads per CTA)
int NUM_THREADS = 5;
```

Massive parallelism

```
// Launch the kernel on the GPU
// Kernel calls are asynchronous (the CPU program continues execution after
// call, but no necessarily before the kernel finishes)
add<<<1, N>>> (d_a, d_b, d_c, N);

// Copy sum vector from device to host
// cudaMemcpy is a synchronous operation, and waits for the prior kernel
// launch to complete (both go to the default stream in this case).
// Therefore, this cudaMemcpy acts as both a memcpy and synchronization
// barrier.
cudaMemcpy(h_c, d_c, sizeof(int) * N, cudaMemcpyDeviceToHost);

cout << endl << "values of array c " << endl;
for (int i = 0; i < N; i++)
{
    cout << h_c[i] << " ";
}
```

Massive parallelism

```
// Free pinned memory
cudaFreeHost(h_a);
cudaFreeHost(h_b);
cudaFreeHost(h_c);

// Free memory on device
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

cout << endl<< "COMPLETED SUCCESSFULLY\n";

return 0;
```

Massive parallelism

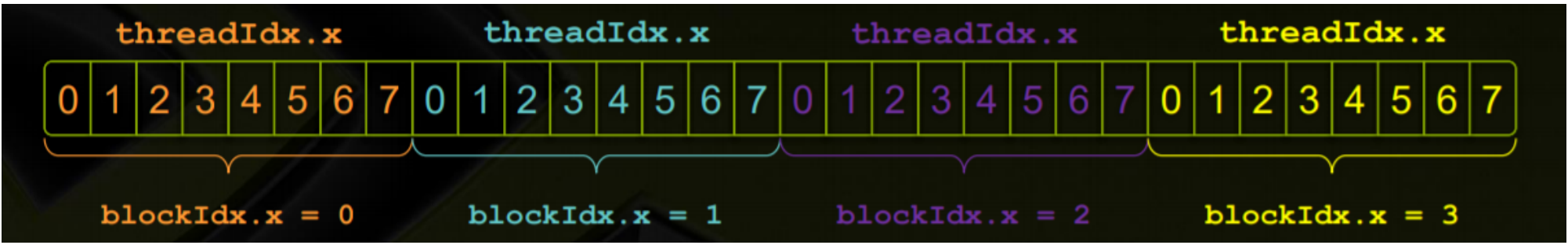
Review

- Launching parallel kernels
  - Kernels are launched with a given grid (1/2/3D index space) and block (1/2/3D index subspace) through the <<<grid, block>>> notation
  - Launch a N blocks with M threads with add<<<N,M>>>(...);
  - Use blockIdx.x to access block index, threadIdx.x to access thread index
- However, the number of threads per block is very limited (max 1024)
- Also, a block is always executed on a single streaming multiprocessor (SM), the other 14 (on K40) SMs are unused
- The grid allows a much bigger index space to fill the GPU

WELCOME TO THE GRID

Indexing Arrays with Blocks and Threads

- A thread calculates its position in the 1/2/3D grid using blockIdx.x and threadIdx.x
- Consider indexing an array with one element per thread (8 threads/block)



With M threads per block, a unique index for each thread is given by: int index = blockIdx.x * M + threadIdx.x;

WELCOME TO THE GRID

Indexing Arrays: Example
Which thread will operate on the red element?

int index = blockIdx.x * M + threadIdx.x;
= 2 * 8 + 5;
= 21;

WELCOME TO THE GRID

Vector Addition with Blocks and Threads
int index = blockIdx.x * blockDim.x + threadIdx.x;
Combined version of add() to use parallel threads and parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        c[index] = a[index] + b[index];
}
```

What changes need to be made in main()?

WELCOME TO THE GRID

Unified memory

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
        int *a, *b, *c;
        int size = N * sizeof(int);
        // Allocate space for device copies of a, b, c
        cudaMallocManaged(&a, size);
        cudaMallocManaged(&b, size);
        cudaMallocManged(&c, size);
        // Setup input values on host
        fillRandom(N, a, b);
        // Launch add() kernel on GPU with N blocks
        add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(a, b, c);
        //use result on host
        cudaDeviceSynchronize();
        printVector(N, c);
        // Cleanup
        cudaFree(a); cudaFree(b); cudaFree(c);
        return 0;
}
```

WELCOME TO THE GRID

Handling Arbitrary Vector Sizes

- Typical problems are not even multiples of blockDim.x
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        if (index < n)
            c[index] = a[index] + b[index];
}
```

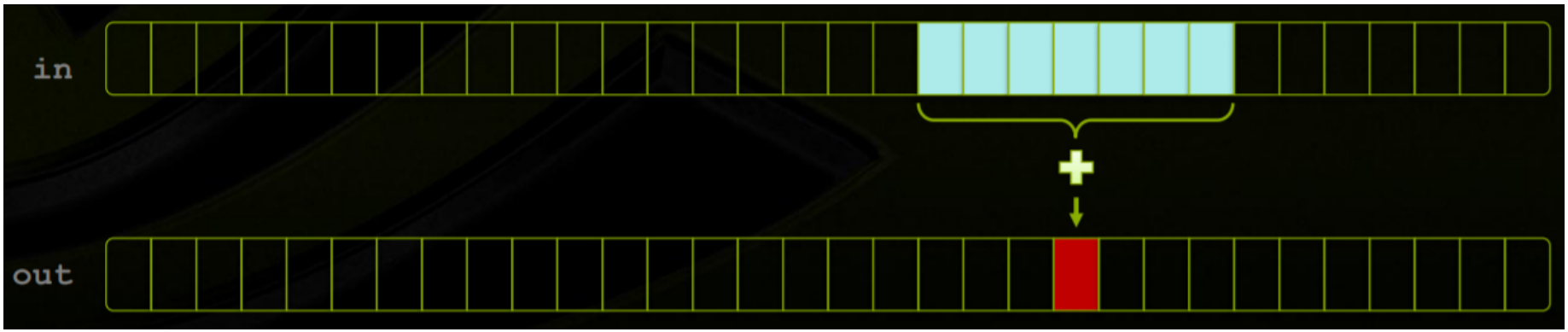
Update the kernel launch:
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);

COOPERATING THREADS

COOPERATING THREADS

1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
- Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



COOPERATING THREADS

Implementing Within a Block

- Each thread processes one output element (blockDim.x elements per block)
- Input elements are read several times

COOPERATING THREADS

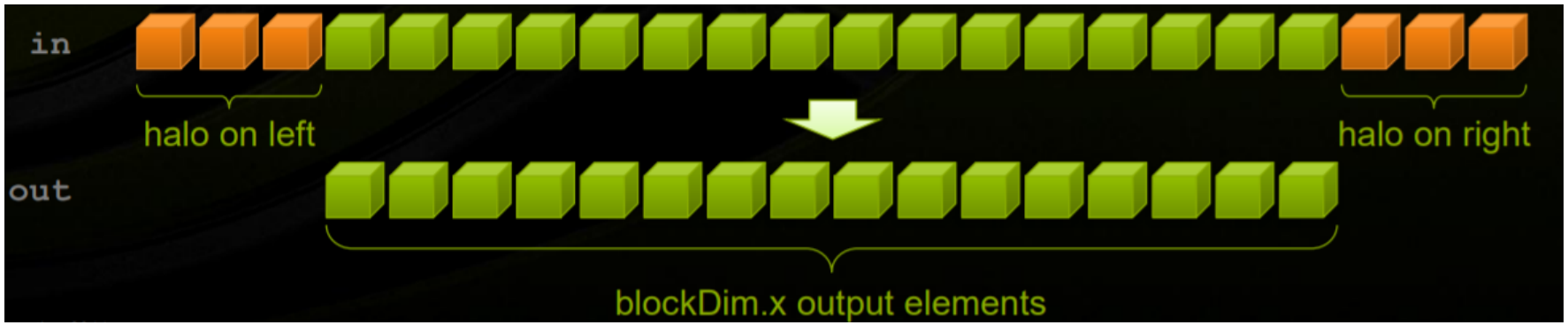
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory
- By opposition to device memory, referred to as global memory
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

COOPERATING THREADS

Implementing With Shared Memory

- Cache data in shared memory
- Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
- Compute blockDim.x output elements
- Write blockDim.x output elements to global memory
- Each block needs a halo of radius elements at each boundary



COOPERATING THREADS

```
__global__ void stencil_1d(int* in, int* out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
```

COOPERATING THREADS

```
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}
```

COOPERATING THREADS

__syncthreads()

- void __syncthreads();
- Synchronizes all threads within a block
- All threads must reach the barrier
- In conditional code, the condition must be uniform across the block

COOPERATING THREADS

Review

- Launching parallel threads
- Launch N blocks with M threads per block with kernel<<<N,M>>>(...);
- Use blockIdx.x to access block index within grid
- Use threadIdx.x to access thread index within block
- Allocate elements to threads:

Allocate elements to threads:

int index = threadIdx.x + blockIdx.x * blockDim.x;

COOPERATING THREADS

Review

- Use __shared__ to declare a variable/array in shared memory
- Data is shared between threads in a block
- Not visible to threads in other blocks
- Use __syncthreads() as a barrier to prevent data hazards

Coordinating Host & Device

- Kernel launches are asynchronous
- Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results
- cudaMemcpy() Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
- cudaMemcpyAsync() Asynchronous, does not block the CPU
- cudaDeviceSynchronize() Blocks the CPU until all preceding CUDA calls have completed

Reporting Errors

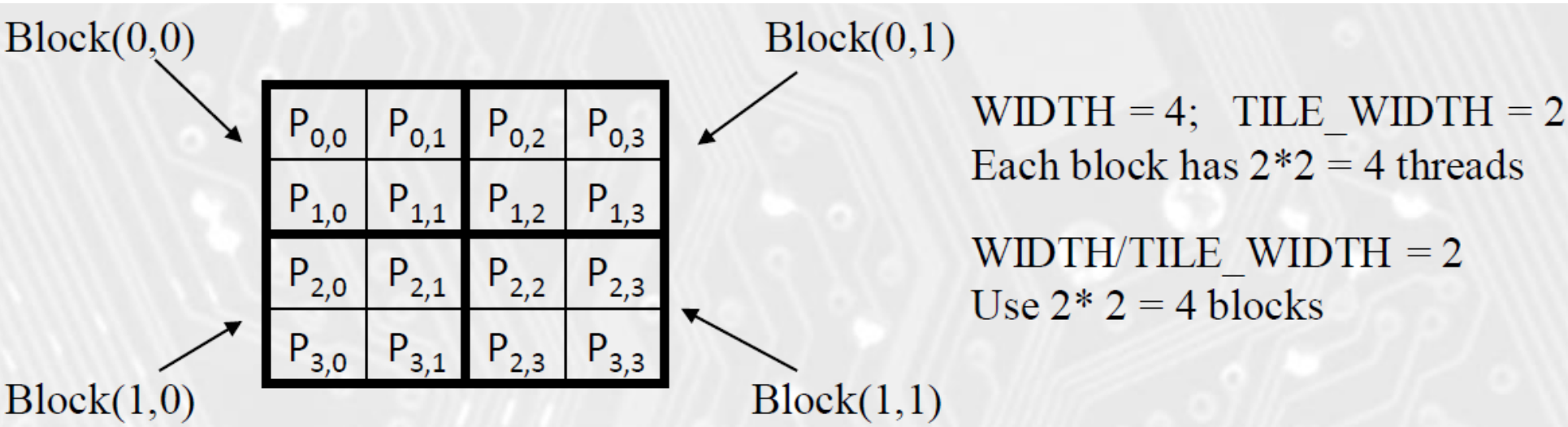
All CUDA API calls return an error code (cudaError_t)

- Error in the API call itself
- Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error: cudaError_t cudaGetLastError(void)
- Get a string to describe the error: char *cudaGetErrorString(cudaError_t)

printf("%s", cudaGetErrorString(cudaGetLastError()));

Matrix multiplication

- Have each 2D thread block to compute a (TILE_WIDTH)2 sub-matrix (tile) of the result matrix
- Each has (TILE_WIDTH)2 threads
- Generate a 2D Grid of (Width/TILE_WIDTH)2 blocks



Matrix multiplication

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use 4* 4 = 16 blocks

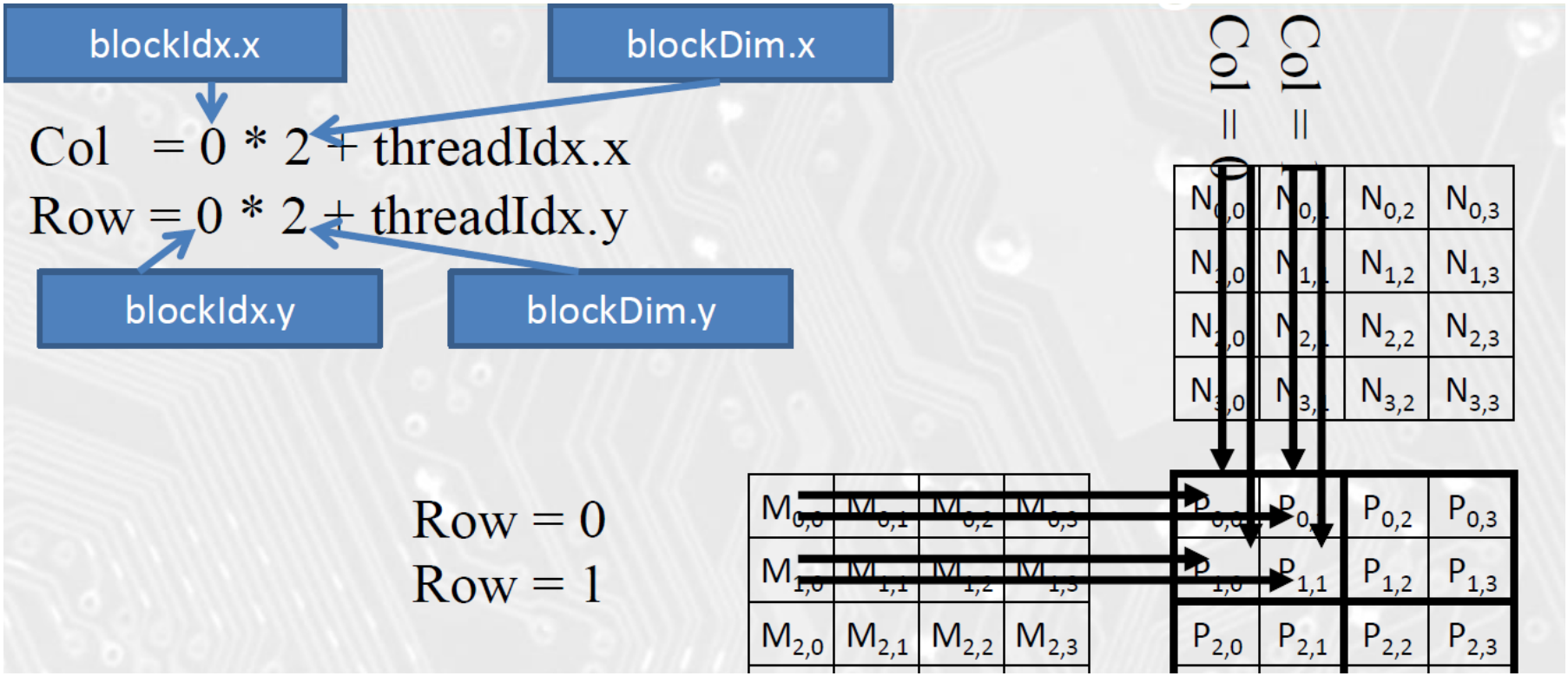
Matrix multiplication

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 =16 threads

WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks

Matrix multiplication



Parallel transpose

The transpose of a matrix is an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix.

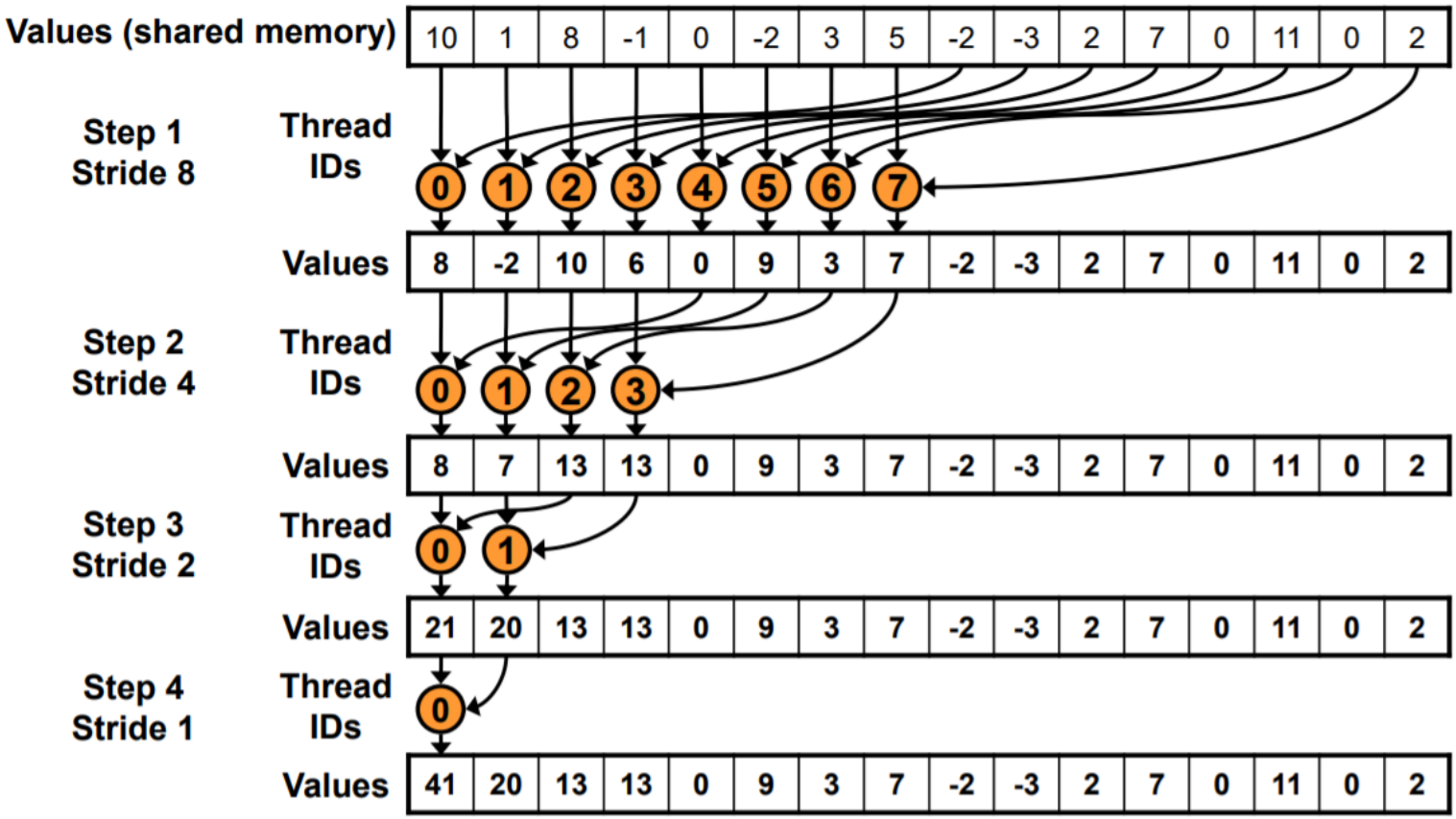
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Input

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Output

Parallel reduction



Parallel reduction

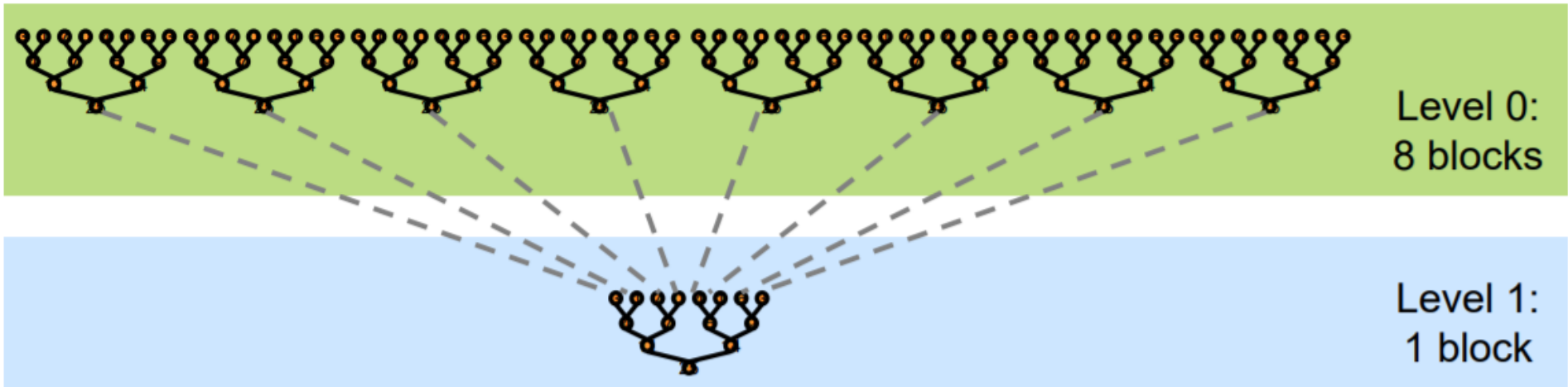
Problem: Global Synchronization

- 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

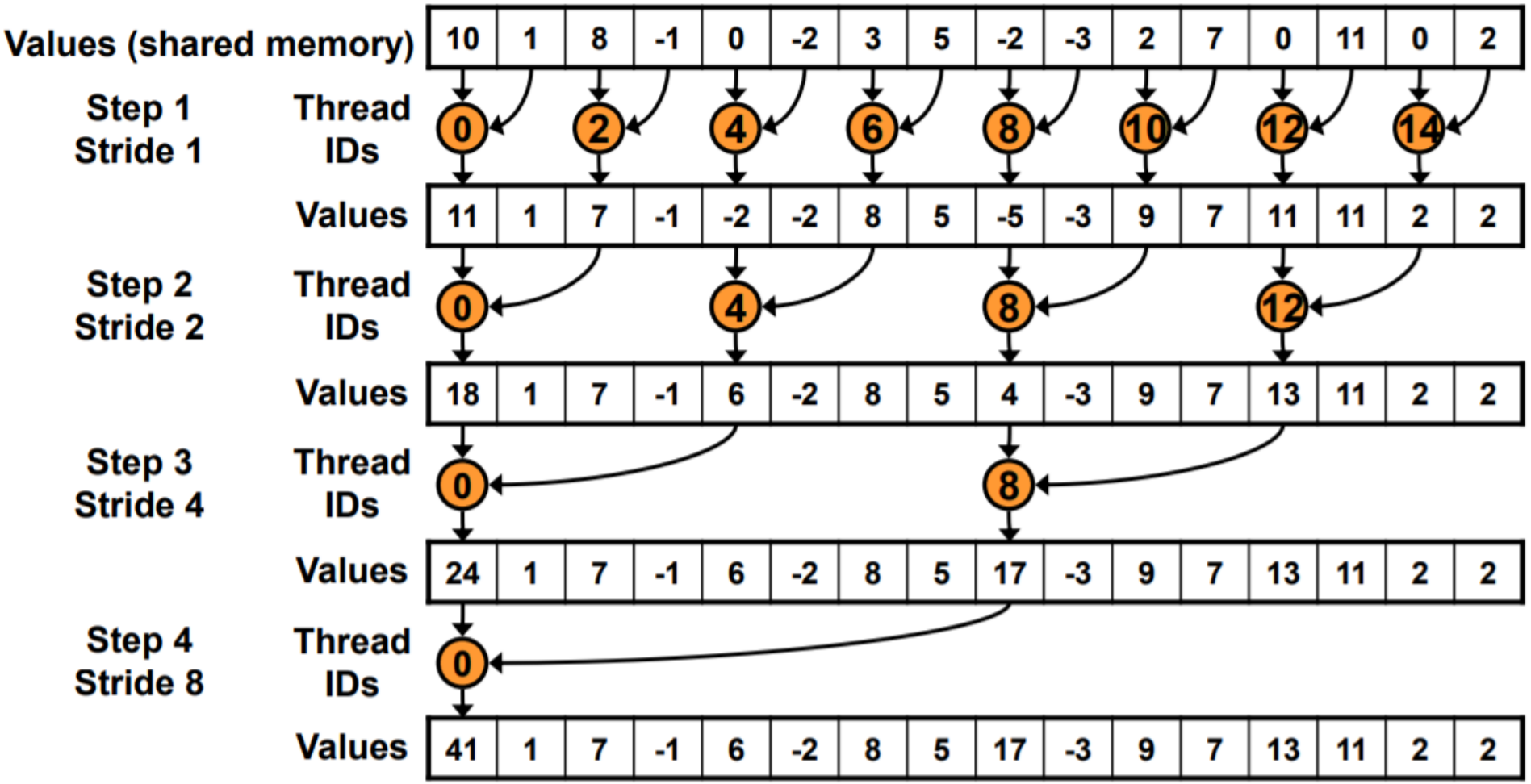
Parallel reduction

Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations
- In the case of reductions, code for all levels is the same: Recursive kernel invocation



Parallel reduction: Interleaved Addressing



Parallel reduction: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

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

  // do reduction in shared mem
  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

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

Problem: highly divergent warps are very inefficient, and % operator is very slow

Parallel reduction

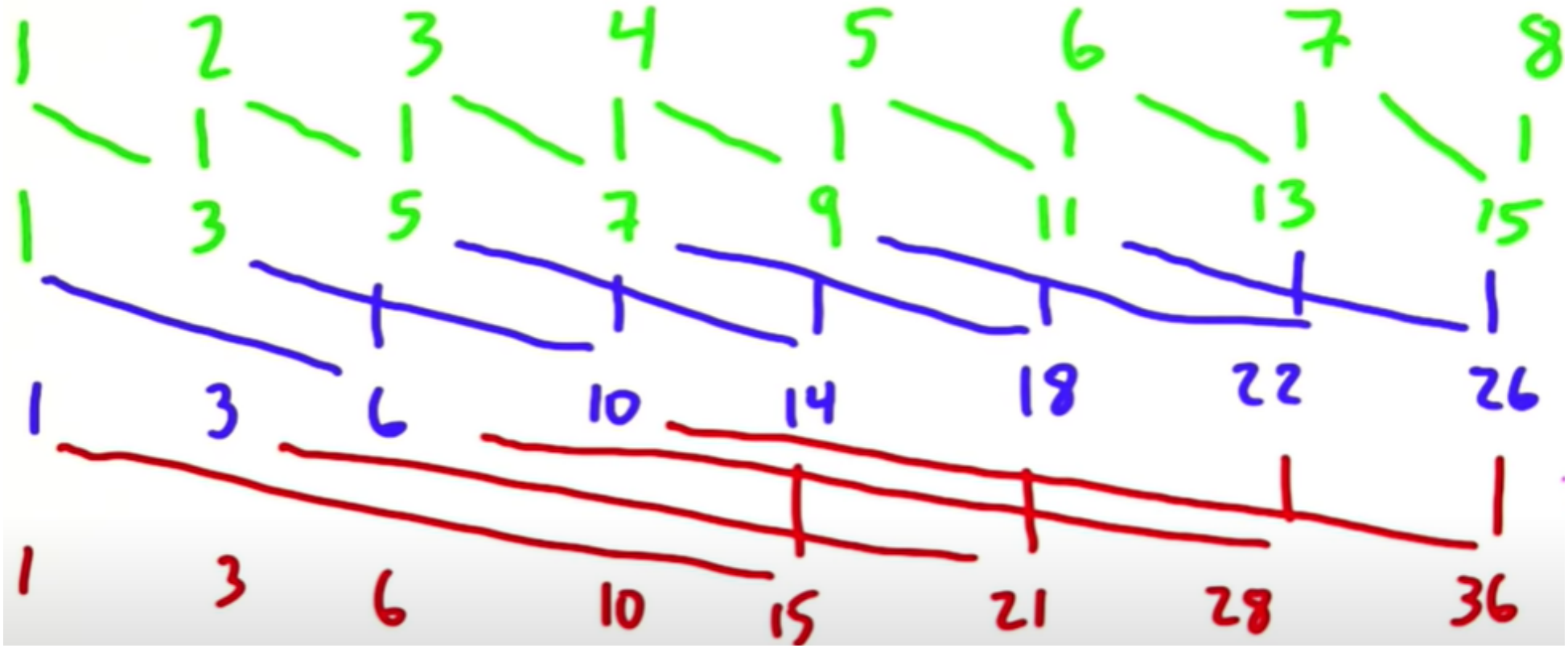
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first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

Scan operation

Example:

- INPUT: 1 2 3 4
- Operation: ADD
- OUTPUT: 1 3 6 10
- The scan operation addresses problems difficult to parallelize.
- Not useful in serial. However, very useful in parallel.

Scan operation



Computational time

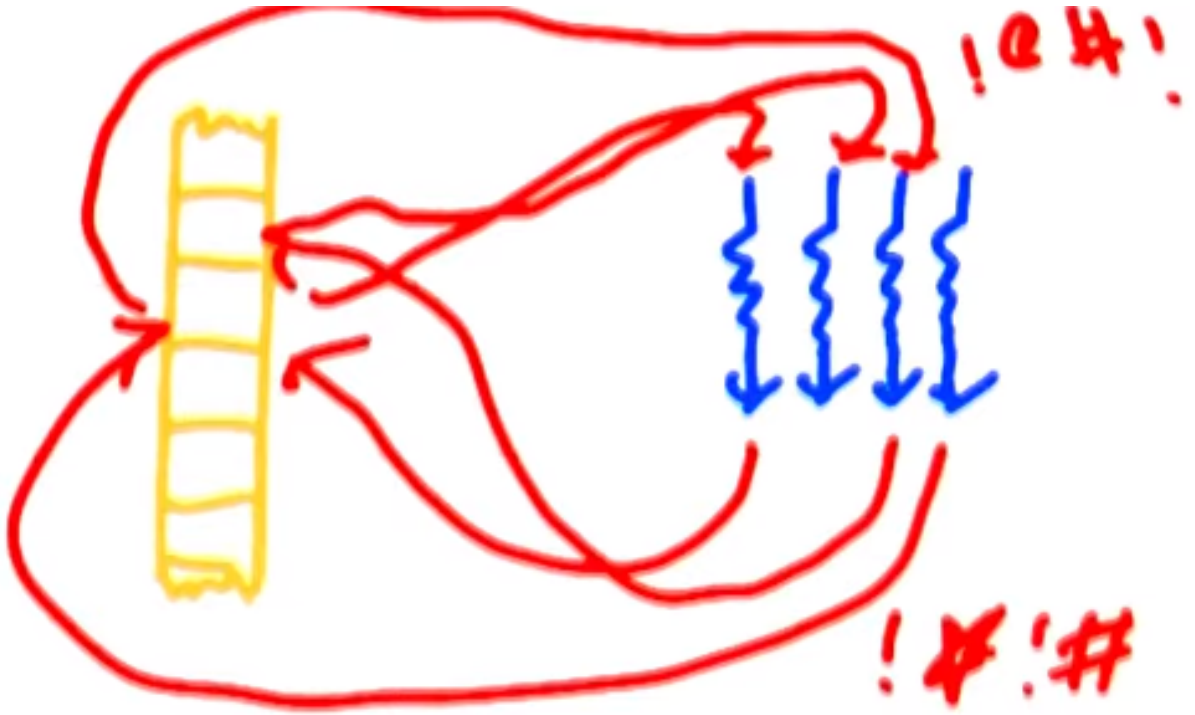
- cudaEvent_t start, stop;
- float milliseconds = 0;
- Before the kernel: cudaEventRecord(start);
- After the kernel: cudaEventRecord(stop);
- cudaEventElapsedTime(milliseconds, start, stop);

CUDA Atomic Operations

CUDA Atomic Operations

Problem:

- Lots of threads reading and writing same memory locations
- 10.000 threads incrementing an array of 10 elements



Let's see an example!

CUDA GPU ACCELERATED LIBRARIES

- Thrust: Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce, which can be composed together to implement complex algorithms with concise, readable source code.
- cuRAND: The cuRAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers.
- cuBLAS: a library for basic linear algebra sub-routines.
- cuSolver: cuSolver library is a high-level package based on the cuBLAS and cuSPARSE libraries for solving linear systems of the form: Ax=b

Thrust Library

```
int main(void)
{
    // H has storage for 4 integers
    thrust::host_vector<int> H(4);

    // initialize individual elements
    H[0] = 14;
    H[1] = 20;
    H[2] = 38;
    H[3] = 46;

    // H.size() returns the size of vector H
    std::cout << "H has size " << H.size() << std::endl;

    // print contents of H
    for(int i = 0; i < H.size(); i++)
        std::cout << "H[" << i << "] = " << H[i] << std::endl;

    // resize H
    H.resize(2);

    std::cout << "H now has size " << H.size() << std::endl;

    // Copy host_vector H to device_vector D
    thrust::device_vector<int> D = H;

    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;

    // print contents of D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    // H and D are automatically deleted when the function returns
    return 0;
}
```

Thrust Library

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>

#include <iostream>

int main(void)
{
    // initialize all ten integers of a device_vector to 1
    thrust::device_vector<int> D(10, 1);

    // set the first seven elements of a vector to 9
    thrust::fill(D.begin(), D.begin() + 7, 9);

    // initialize a host_vector with the first five elements of D
    thrust::host_vector<int> H(D.begin(), D.begin() + 5);

    // set the elements of H to 0, 1, 2, 3, ...
    thrust::sequence(H.begin(), H.end());

    // copy all of H back to the beginning of D
    thrust::copy(H.begin(), H.end(), D.begin());

    // print D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    return 0;
}
```

Thrust Library

Transformations are algorithms that apply an operation to each element in a set of (zero or more) input ranges and then stores the result in a destination range. One example we have already seen is thrust::fill, which sets all elements of a range to a specified value. Other transformations include thrust::sequence, thrust::replace, and of course thrust::transform.

Thrust Library

The following source code demonstrates several of the transformation algorithms. Note that thrust::negate and thrust::modulus are known as functors in C++ terminology. Thrust provides these and other common functors like plus and multiplies in the file thrust/functional.h.

```
#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>

int main(void)
{
    // 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 twos
    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 Library

Consider the vector operation y = a * x + y where x and y are vectors and a is a scalar constant.

Thrust Library

Consider the vector operation y = a * x + y where x and y are vectors and a is a scalar constant.

```
struct saxpy_functor
{
    const float a;

    saxpy_functor(float _a) : a(_a) {}

    __host__ __device__
        float operator()(const float& x, const float& y) const {
            return a * x + y;
        }
};

void saxpy_fast(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y)
{
    // Y <- A * X + Y
    thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));
}

void saxpy_slow(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y)
{
    thrust::device_vector<float> temp(X.size());

    // temp <- A
    thrust::fill(temp.begin(), temp.end(), A);

    // temp <- A * X
    thrust::transform(X.begin(), X.end(), temp.begin(), temp.begin(), thrust::multiplies<float>());

    // Y <- A * X + Y
    thrust::transform(temp.begin(), temp.end(), Y.begin(), Y.begin(), thrust::plus<float>());
}
```

Reduction using Thrust

A reduction algorithm uses a binary operation to reduce an input sequence to a single value. For example, the sum of an array of numbers is obtained by reducing the array with a plus operation. Similarly, the maximum of an array is obtained by reducing with an operator that takes two inputs and returns the maximum. The sum of an array is implemented with thrust::reduce as follows:

```
int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
```

Reduction using Thrust

Although thrust::reduce is sufficient to implement a wide variety of reductions, Thrust provides a few additional functions for convenience (like the STL). For example, thrust::count returns the number of instances of a specific value in a given sequence:

```
#include <thrust/count.h>
#include <thrust/device_vector.h>
...
// put three 1s in a device_vector
thrust::device_vector<int> vec(5,0);
vec[1] = 1;
vec[3] = 1;
vec[4] = 1;

// count the 1s
int result = thrust::count(vec.begin(), vec.end(), 1);
// result is three
```

Other reduction operations include count_if, min_element, max_element, is_sorted

Reduction using Thrust

With thrust::transform_reduce, we can also apply kernel fusion to reduction kernels. Consider the following example which computes the norm of a vector.

```
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

// square<T> computes the square of a number f(x) -> x*x
template <typename T>
struct square
{
    __host__ __device__
        T operator()(const T& x) const {
            return x * x;
        }
};

int main(void)
{
    // initialize host array
    float x[4] = {1.0, 2.0, 3.0, 4.0};

    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    // setup arguments
    square<float>        unary_op;
    thrust::plus<float> binary_op;
    float init = 0;

    // compute norm
    float norm = std::sqrt( thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, init, binary_op) );
```

Scan using Thrust

Parallel prefix-sums, or scan operations, are important building blocks in many parallel algorithms such as stream compaction and radix sort. Consider the following source code which illustrates an inclusive scan operation using the default plus operator:

```
#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}
```

Sort using Thrust

The thrust::sort and thrust::stable_sort functions are direct analogs of sort and stable_sort in the STL.

```
#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}
```

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

...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};

thrust::sort_by_key(keys, keys + N, values);

// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

Sort using Thrust

```
#include <thrust/sort.h>
#include <thrust/functional.h>

...
const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};

thrust::stable_sort(A, A + N, thrust::greater<int>());

// A is now {8, 7, 5, 4, 2, 1}
```