

CUDA C/C++ BASICS

NVIDIA Corporation

What is CUDA?

- CUDA Architecture
 - Expose GPU parallelism for general-purpose computing
 - Retain performance
- CUDA C/C++
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

Introduction to CUDA C/C++

- What will you learn with this slide set?
 - Start from “Hello World!”
 - Write and launch CUDA C/C++ kernels
 - Manage GPU memory
 - Manage communication and synchronization

Prerequisites

- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience

CONCEPTS



Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

HELLO WORLD!

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

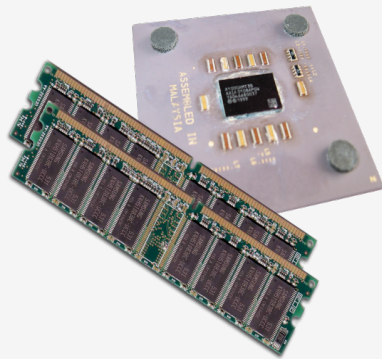
Asynchronous operation

Handling errors

Managing devices

Heterogeneous Computing

- Terminology:
 - *Host* The CPU and its memory (host memory)
 - *Device* The GPU and its memory (device memory)

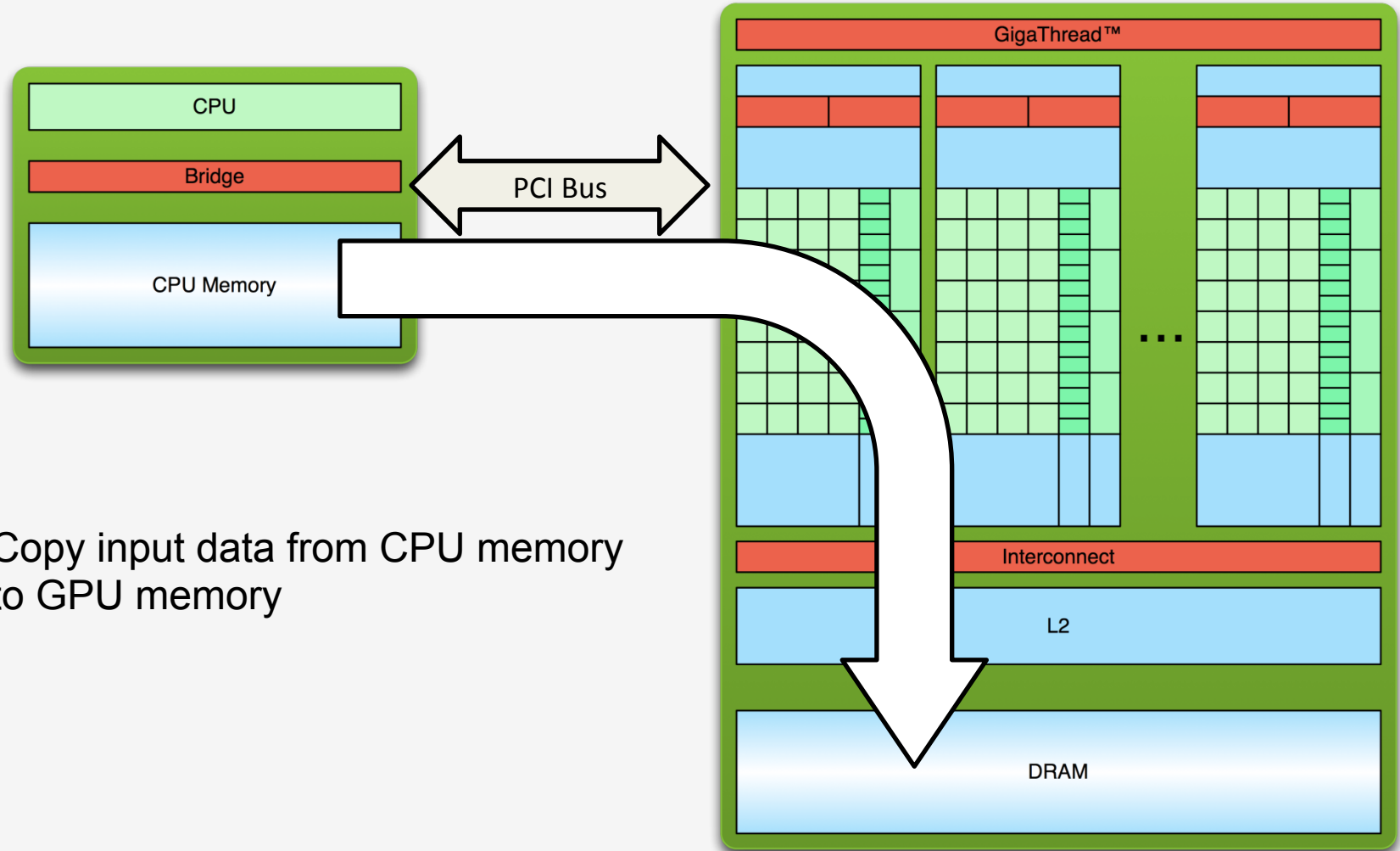


Host



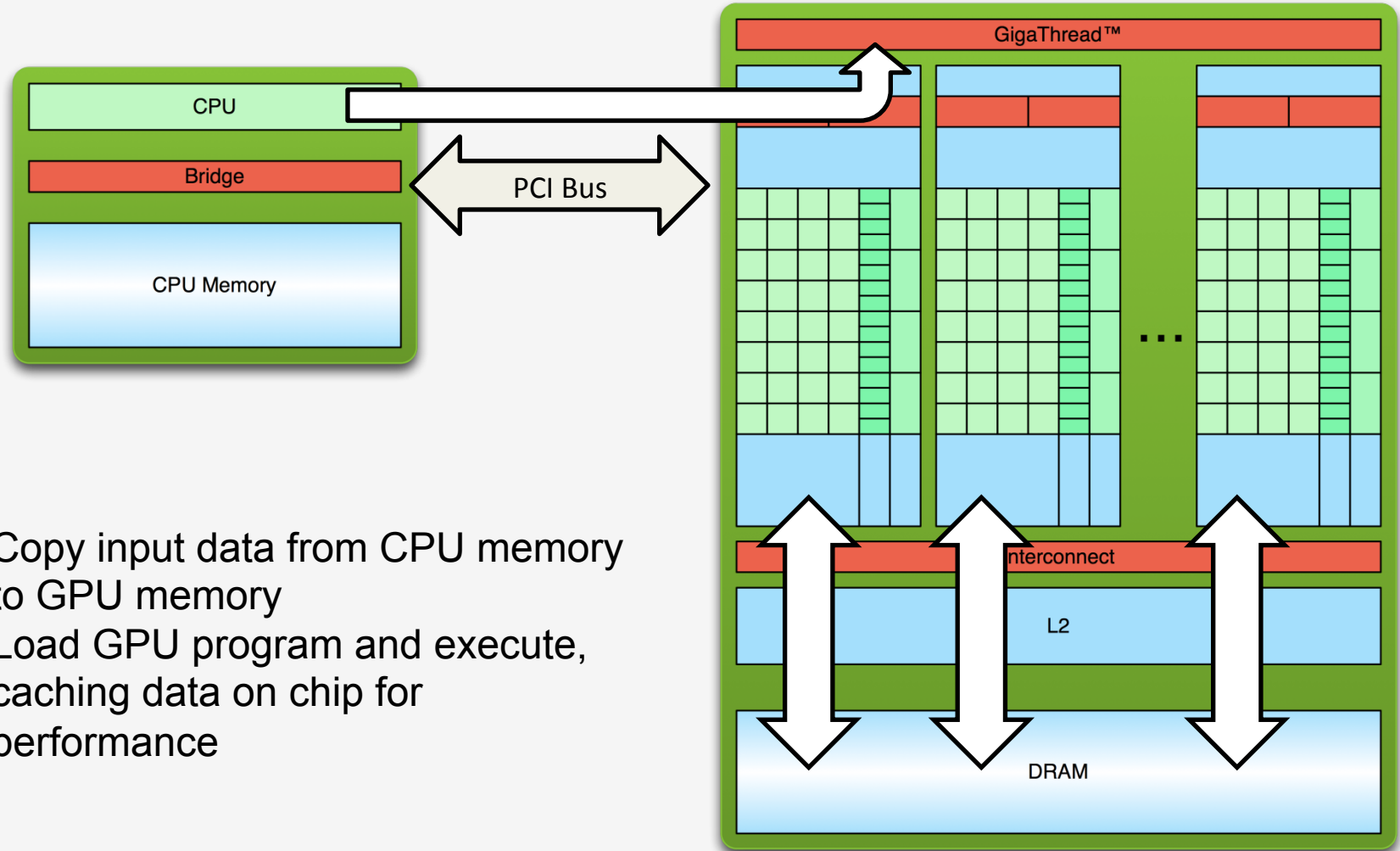
Device

Simple Processing Flow

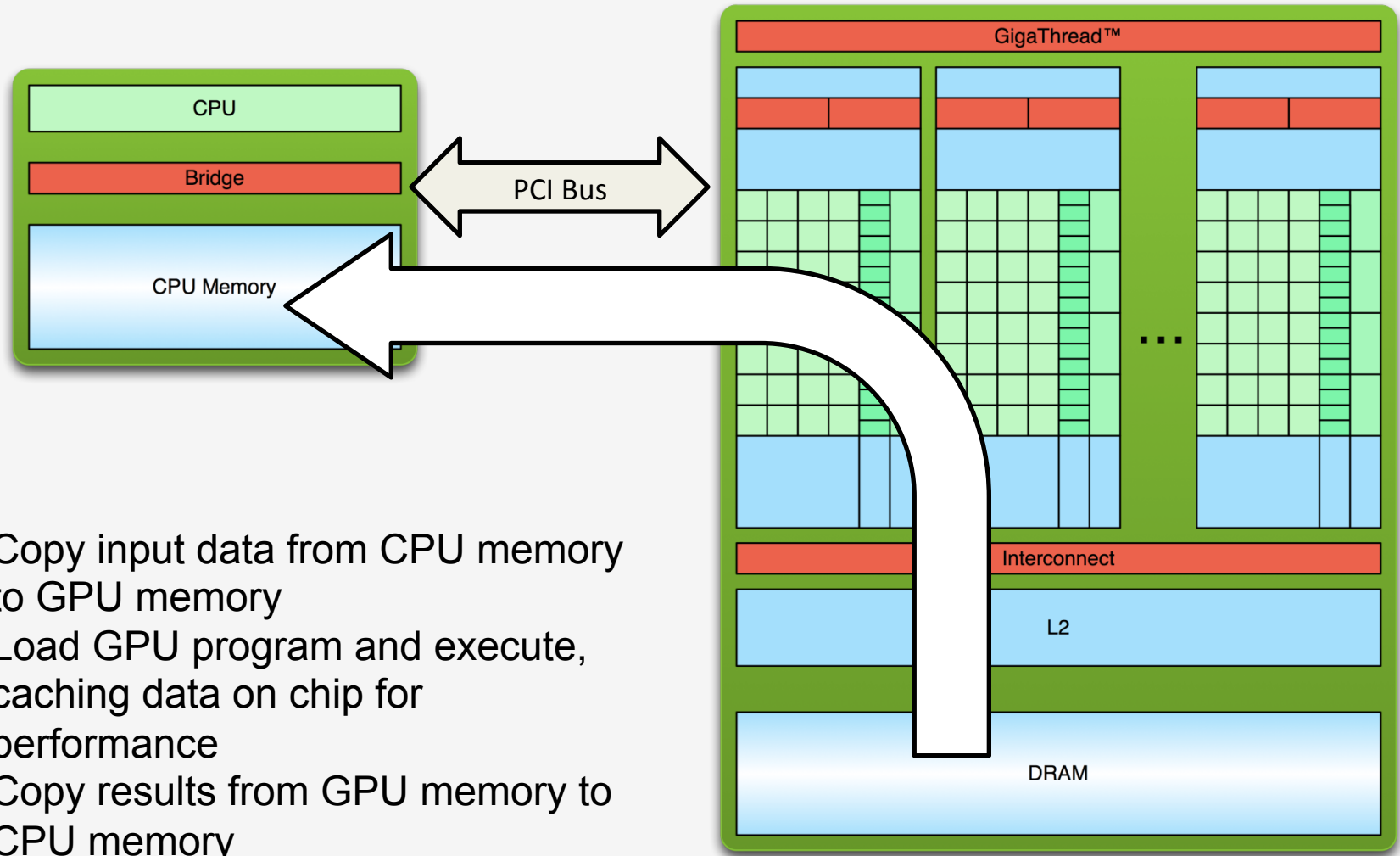


1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



Simple Processing Flow



Hello World!

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

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

Output:

```
$ nvcc  
hello_world.  
cu  
$ a.out  
Hello World!  
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {...}
```

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

- Two new syntactic elements...

Hello World! with Device Code

```
__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! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

```
__global__ void mykernel(void) {.....}
```

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

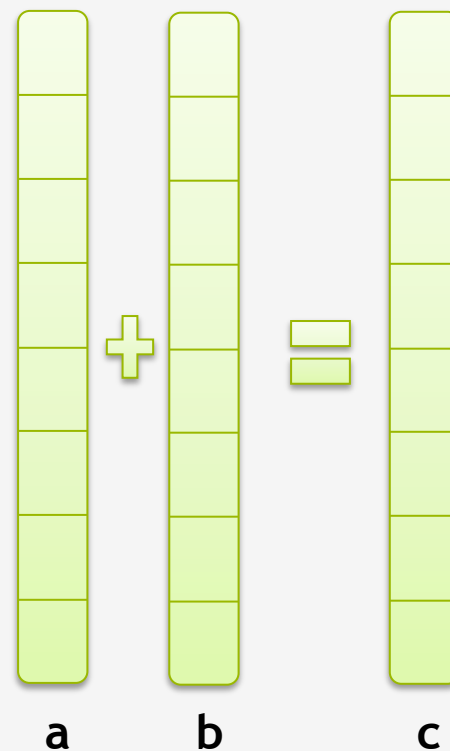
- `mykernel()` does nothing, somewhat anticlimactic!

Output:

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```

Parallel Programming in CUDA C/C++

- But wait... 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



Addition on the Device

- 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

Addition on the Device

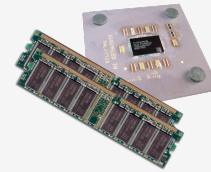
- Note that we use pointers for the variables

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

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

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()`



Addition on the Device: `add()`

- Returning to our `add()` kernel

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

- Let's take a look at `main()`...

Addition on the Device: `main()`

```
int main(void) {  
    int a, b, c;           // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

Addition on the Device: `main()`

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU
```

```
add<<<1,1>>>(d_a, d_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;
```

```
}
```

RUNNING IN PARALLEL

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

Managing devices

Moving to Parallel

- GPU computing is about massive parallelism
 - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>() ;
```



```
add<<< N, 1 >>>() ;
```

- Instead of executing `add()` once, execute `N` times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
 - The set of blocks is referred to as a **grid**
 - Each invocation can refer to its block index using `blockIdx.x`

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

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

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

- On the device, each block can execute in parallel:

Block 0

`c[0] = a[0] + b[0];`

Block 1

`c[1] = a[1] + b[1];`

Block 2

`c[2] = a[2] + b[2];`

Block 3

`c[3] = a[3] + b[3];`

Vector Addition on the Device: `add()`

- Returning to our parallelized `add()` kernel

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

- Let's take a look at `main()`...

Vector Addition on the Device: `main()`

```
#define N 512

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

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU with N blocks
```

```
add<<<N,1>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
free(a); free(b); free(c);
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

Review (1 of 2)

- Difference between *host* and *device*
 - *Host* CPU
 - *Device* GPU
- Using `__global__` to declare a function as device code
 - Executes on the device
 - Called from the host
- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch `N` copies of `add()` with `add<<<N,1>>>(...)` ;
 - Use `blockIdx.x` to access block index

INTRODUCING THREADS

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

Managing devices

CUDA Threads

- Terminology: a block can be split into parallel **threads**
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

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

- We use **threadIdx.x** instead of **blockIdx.x**
- Need to make one change in `main()`...

Vector Addition Using Threads: `main()`

```
#define N 512

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

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

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

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

COMBINING THREADS AND BLOCKS

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

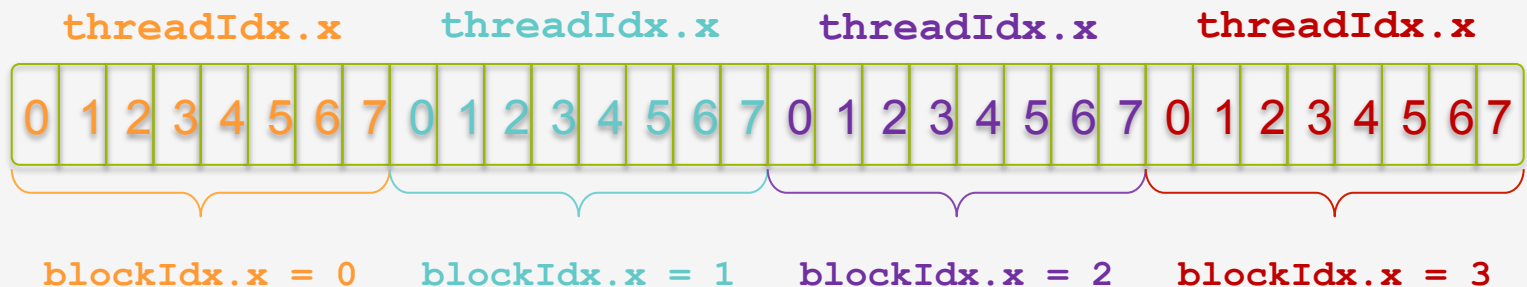
Managing devices

Combining Blocks and Threads

- We've seen parallel vector addition using:
 - Many blocks with one thread each
 - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/block)

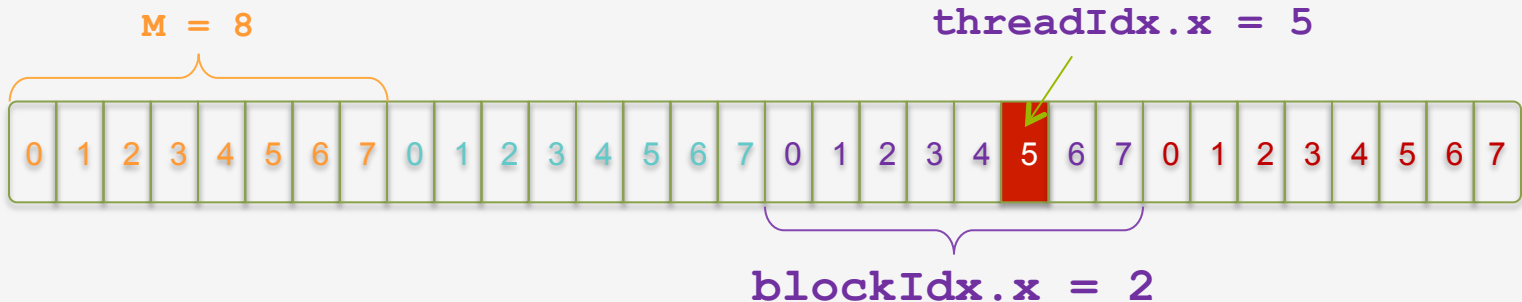
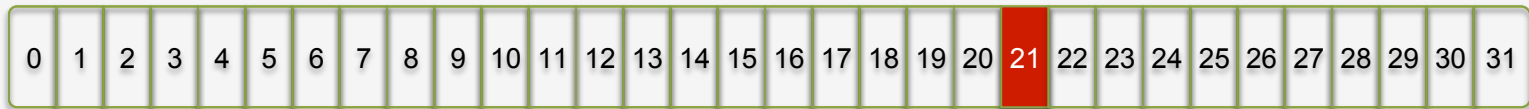


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?



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

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

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

- Combined version of `add()` to use parallel threads *and* parallel blocks

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

- What changes need to be made in `main()`?

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                // host copies of a, b, c
    int *d_a, *d_b, *d_c;          // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: `main()`

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU
```

```
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
free(a); free(b); free(c);
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly 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);
```

Why Bother with Threads?

- Threads seem unnecessary
 - They add a level of complexity
 - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
 - Communicate
 - Synchronize
- To look closer, we need a new example...

Review

- Launching parallel kernels
 - Launch N copies of `add()` with `add<<<N/M,M>>>(...)` ;
 - Use `blockIdx.x` to access block index
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

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

COOPERATING THREADS

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

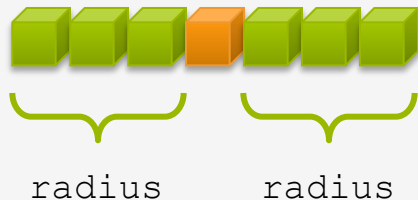
Asynchronous operation

Handling errors

Managing devices

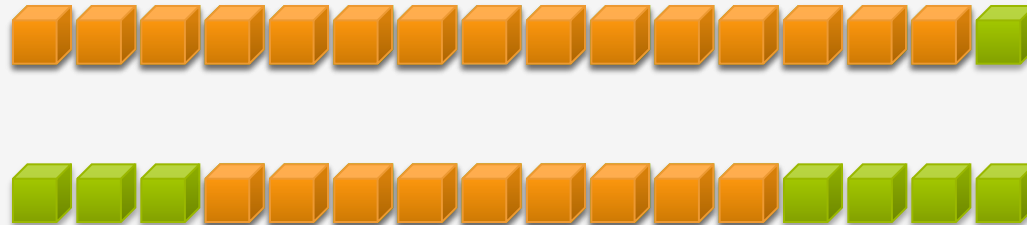
1D Stencil

- 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:



Implementing Within a Block

- Each thread processes one output element
 - blockDim.x elements per block
- Input elements are read several times
 - With radius 3, each input element is read seven times

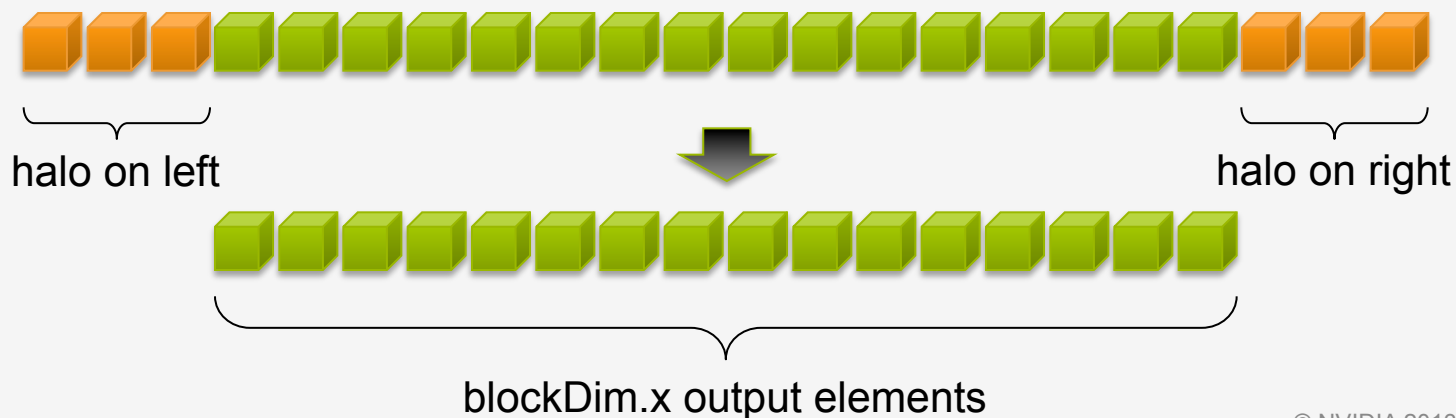


Sharing Data Between Threads

- Terminology: within a block, threads share data via `shared memory`
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
 - Read ($\text{blockDim.x} + 2 * \text{radius}$) 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



Stencil Kernel

```
__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];  
    }
```





Stencil Kernel

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

Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
temp[lindex] = in[gindex];           Store at temp[18]   
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];    Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
  
int result = 0;  
result += temp[lindex + 1];          Load from temp[19] 
```

__syncthreads()

- `void __syncthreads ();`
- Synchronizes all threads within a block
 - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
 - In conditional code, the condition must be uniform across the block

Stencil Kernel

```
__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];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

Stencil Kernel

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


Review (1 of 2)

- 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:

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

Review (2 of 2)

- 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
 - Use to prevent data hazards

MANAGING THE DEVICE

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

Managing devices

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
 - OR
 - 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\n", cudaGetErrorString(cudaGetLastError()));
```

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
```

```
cudaSetDevice(int device)
```

```
cudaGetDevice(int *device)
```

```
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

```
cudaSetDevice(i) to select current device
```

```
cudaMemcpy(...) for peer-to-peer copies†
```

[†] requires OS and device support

Introduction to CUDA C/C++

- What have we learned?
 - Write and launch CUDA C/C++ kernels
 - `__global__, blockIdx.x, threadIdx.x, <<<>>>`
 - Manage GPU memory
 - `cudaMalloc(), cudaMemcpy(), cudaFree()`
 - Manage communication and synchronization
 - `__shared__, __syncthreads()`
 - `cudaMemcpy() VS cudaMemcpyAsync(), cudaDeviceSynchronize()`

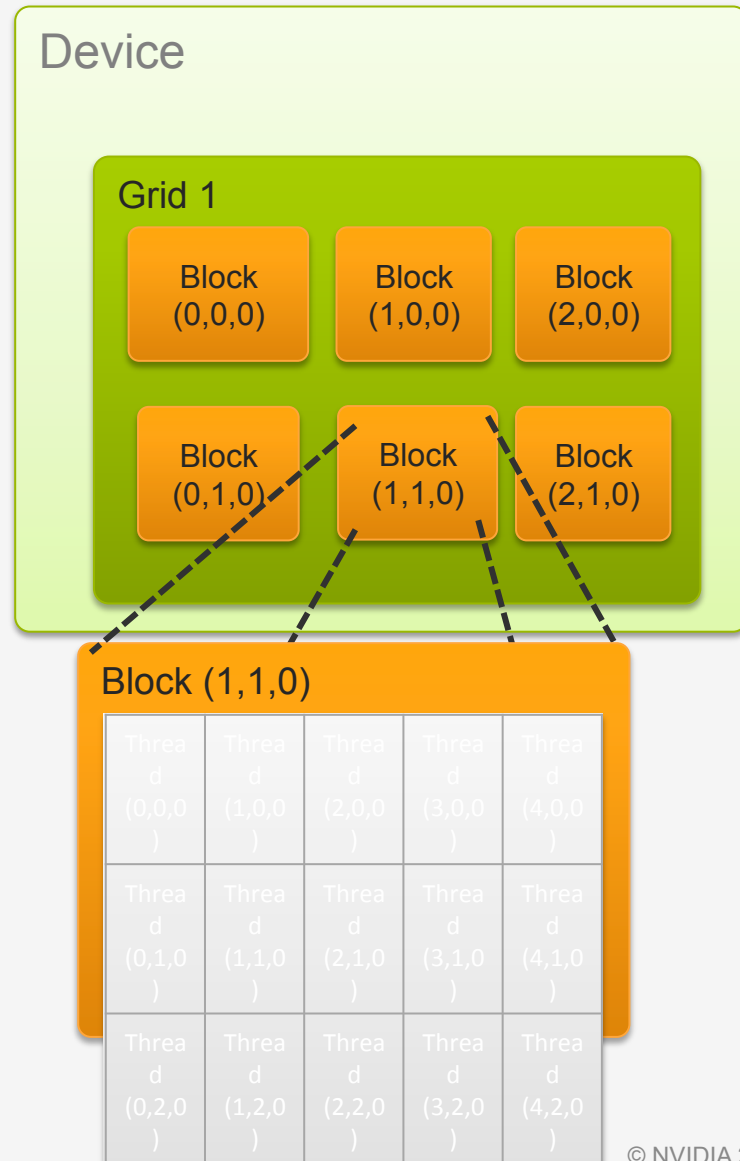
IDs and Dimensions

– A kernel is launched as a grid of blocks of threads

- `blockIdx` and `threadIdx` are 3D
- We showed only one dimension (x)

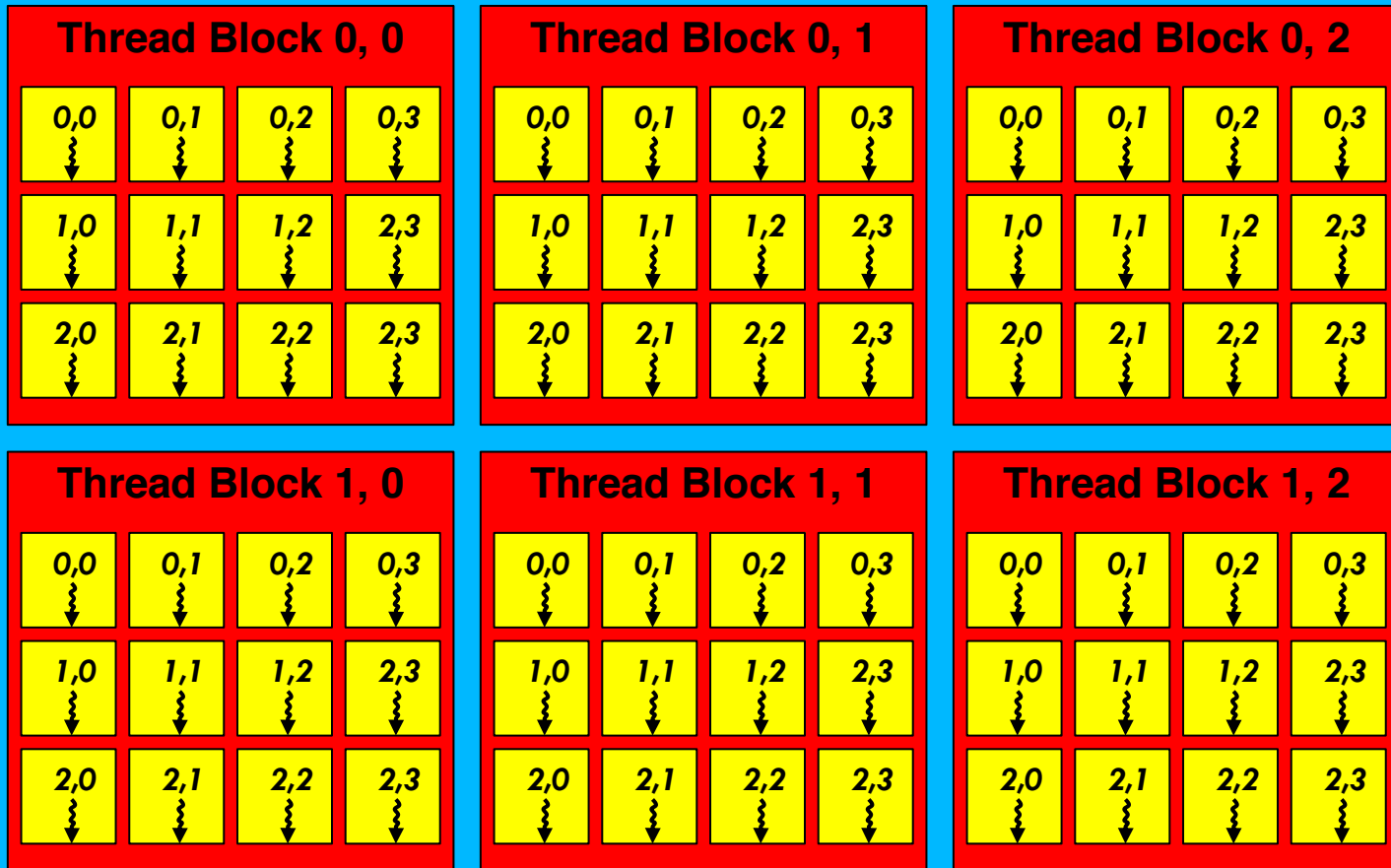
• Built-in variables:

- `threadIdx`
- `blockIdx`
- `blockDim`
- `gridDim`



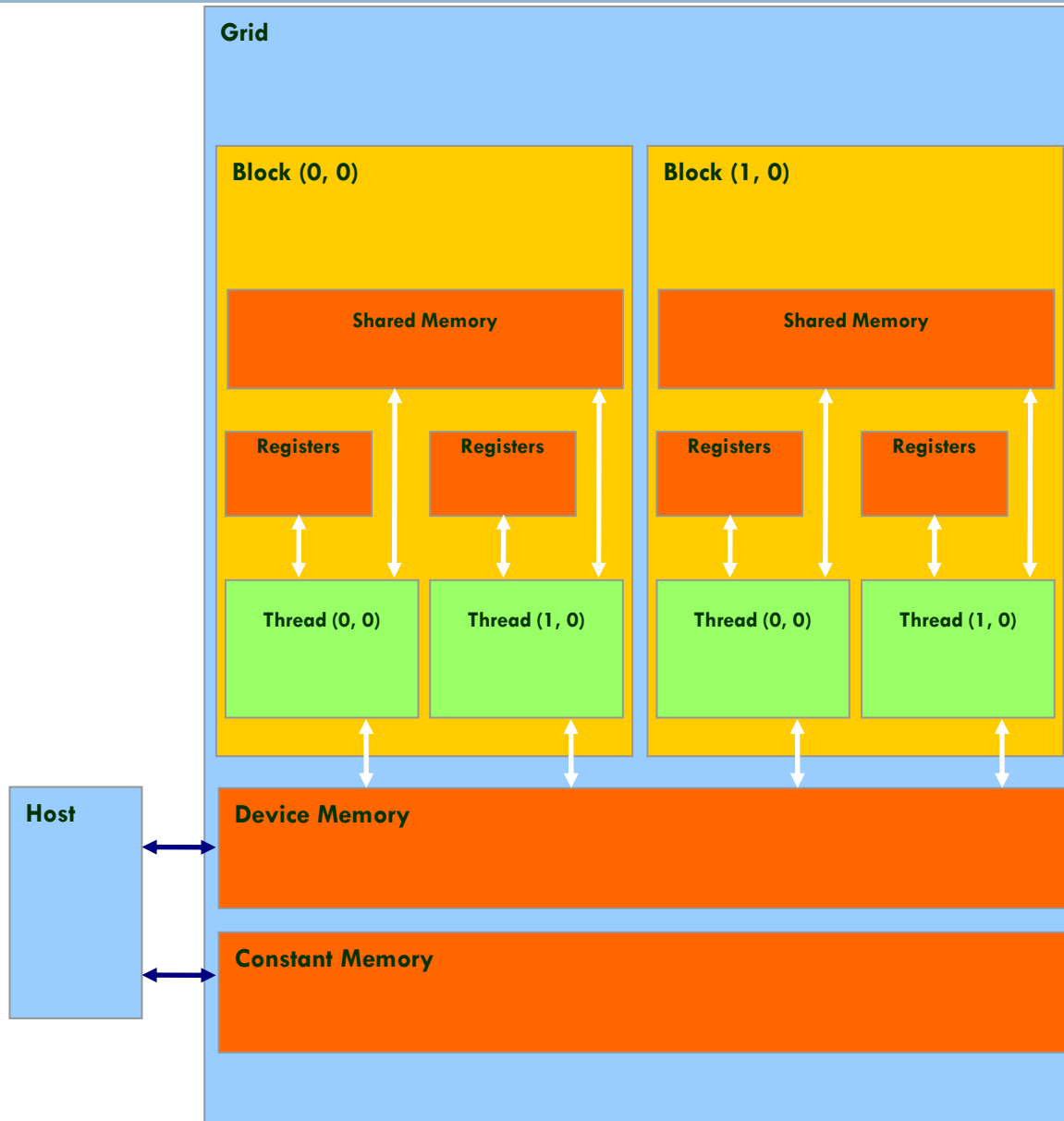
Grids, Thread Blocks and Threads

Grid



Hardware Memory Spaces in CUDA

4



Vector addition GPU code

5

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

GPU code

```
int main() {
    // initialization code here ...

    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);

    // cleanup code here ...
}
```

Host code

(can be in the same file)

- The host always initiates work for the device
 - For Kepler GPUs, device can generate work for itself (Dynamic Parallelism)
- Examples for common extensions are
 - `--global--` defines a GPU kernel that is callable by the CPU
 - `--device--` defines a GPU function that is callable by a GPU kernel or by another device function, but not callable by a CPU (host)
 - `--host--` or no qualifier marks a CPU function

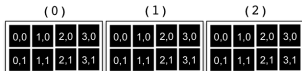
- A CUDA kernel is a **grid** of **thread blocks**
 - The grid can be 1D, 2D or 3D
- A **thread block** is, in turn, a 1D, 2D, or 3D **array of threads**
- A thread is executed by exactly one **stream processor** or **CUDA core**
- A thread block is executed by exactly one **Streaming Multiprocessor (SM)**
- However, CUDA cores and SM can interleave execution of multiple threads and thread blocks

- CUDA maintains special variables that store:
 - thread index within a thread block `threadIdx(.x, .y, .z)`
 - block index within a kernel grid `blockIdx(.x, .y, .z)`
 - dimension of a thread block `blockDim(.x, .y, .z)`
 - dimension of a kernel grid `gridDim(.x, .y, .z)`
- These variables are used without declaration
- Mainly used in assigning the work per block/thread
- The maximum values of the (x, y, z) in both `blockDim` and `gridDim` are GPU-dependent
- At least the x component needs to be declared. The default value for the y and z components is 1

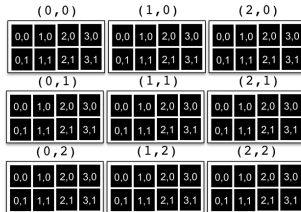
```
dim3 dimBlock(3)
dim3 dimGrid(4,2)
```



```
dim3 dimBlock(4, 2)
dim3 dimGrid(3)
```



```
dim3 dimBlock(4, 2)
dim3 dimGrid(3, 3)
```



Thread Scheduling

7

- Order in which thread blocks are scheduled is undefined!
 - ▣ any possible interleaving of blocks should be valid
 - ▣ presumed to run to completion without preemption
 - ▣ can run in any order
 - ▣ can run concurrently OR sequentially

- Order of threads within a block is also undefined!

Global synchronization

8

- Q: How do we do global synchronization with these scheduling semantics?

Global synchronization

9

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!

Global synchronization

10

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!
- A2: Finish a grid, and start a new one!

Global synchronization

11

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!
- A2: Finish a grid, and start a new one!

```
step1<<<grid1,blk1>>>(...);  
// CUDA ensures that all writes from step1 are complete.  
step2<<<grid2,blk2>>>(...);
```

- We don't have to copy the data back and forth!

Atomics

12

- Guarantee that only a single thread has access to a piece of memory during an operation
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
- Atomic Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Can be done on device memory and shared memory
- Much more expensive than load + operation + store

Example: Histogram

13

```
// Determine frequency of colors in a picture.  
// Colors have already been converted into integers  
// between 0 and 255.  
// Each thread looks at one pixel,  
// and increments a counter  
  
__global__ void histogram(int* colors, int* buckets)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int c = colors[i];  
    buckets[c] += 1;  
}
```

Example: Histogram

14

```
// Determine frequency of colors in a picture.  
// Colors have already been converted into integers  
// between 0 and 255.  
// Each thread looks at one pixel,  
// and increments a counter  
  
__global__ void histogram(int* colors, int* buckets)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int c = colors[i];  
    buckets[c] += 1;  
}
```

Example: Histogram

15

```
// Determine frequency of colors in a picture.  
// Colors have already been converted into integers  
// between 0 and 255.  
// Each thread looks at one pixel,  
// and increments a counter atomically  
  
__global__ void histogram(int* colors, int* buckets)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int c = colors[i];  
    atomicAdd(&buckets[c], 1);  
}
```


Example: Work queue

16

```
// For algorithms where the amount of work per item  
// is highly non-uniform, it often makes sense to  
// continuously grab work from a queue.
```

__global__

```
void workq(int* work_q, int* q_counter,  
          int queue_max, int* output)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int q_index = atomicInc(q_counter, queue_max);  
    int result = do_work(work_q[q_index]);  
    output[i] = result;  
}
```

17

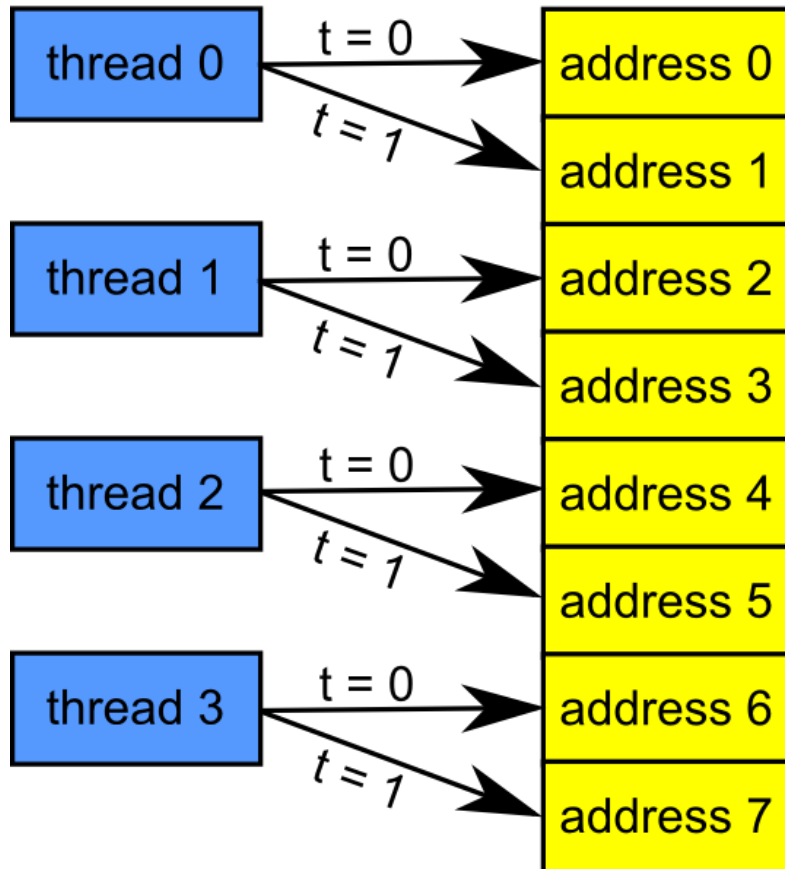
CUDA: optimizing your application

Coalescing

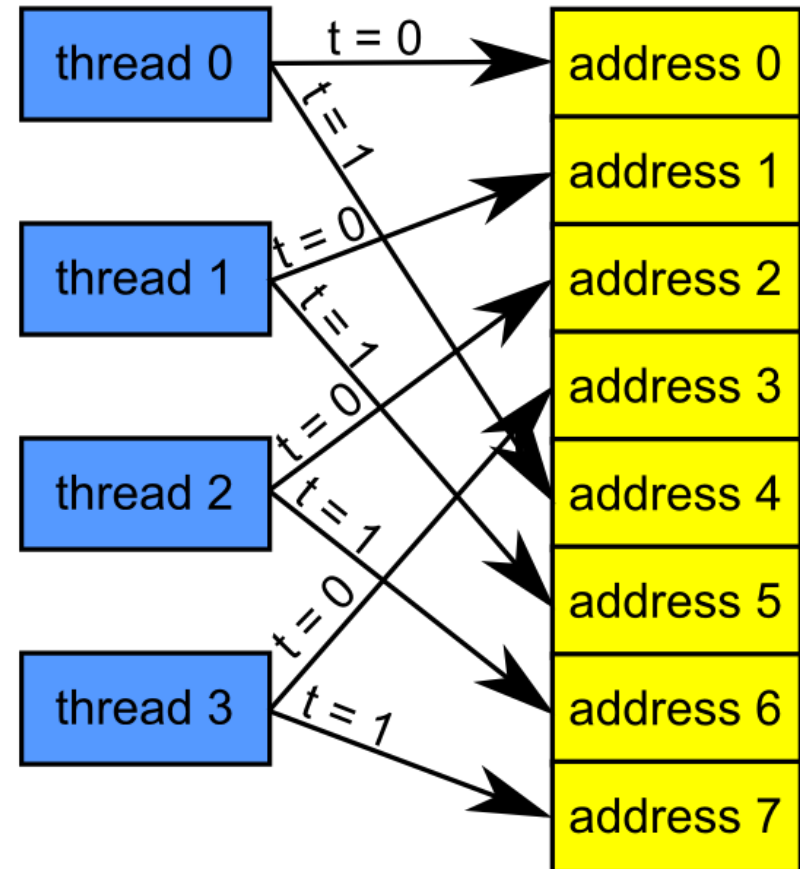
Coalescing

18

traditional multi-core
optimal memory access pattern



many-core GPU
optimal memory access pattern



Consider the stride of your accesses

19

```
__global__ void foo(int* input, float3* input2) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    // Stride 1, OK!  
    int a = input[i];  
  
    // Stride 2, half the bandwidth is wasted  
    int b = input[2*i];  
  
    // Stride 3, 2/3 of the bandwidth wasted  
    float c = input2[i].x;  
}
```

Example: Array of Structures (AoS)

20

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

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

Example: Structure of Arrays (SoA)

21

```
Struct SoA {  
    int* keys;  
    int* values;  
    int* flags;  
};
```

```
SoA d_SoA_data;  
cudaMalloc((void**) &d_SoA_data.keys, ...);  
cudaMalloc((void**) &d_SoA_data.values, ...);  
cudaMalloc((void**) &d_SoA_data.flags, ...);
```

Example: SoA vs AoS

22

```
__global__ void bar(record* AoS_data,  
                    SoA SoA_data) {  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    // AoS wastes bandwidth  
    int key1 = AoS_data[i].key;  
  
    // SoA efficient use of bandwidth  
    int key2 = SoA_data.keys[i];  
}
```

Memory Coalescing

23

- Structure of arrays is often better than array of structures
- Very clear win on regular, stride 1 access patterns
- Unpredictable or irregular access patterns are case-by-case
- Can lose a factor of 10 – 30!

24

CUDA: optimizing your application

Shared Memory

Using shared memory

25

```
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

Using shared memory

26

```
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

How do we use device memory bandwidth?

Using shared memory

27

```
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

How do we use device memory bandwidth?

The next thread also reads input[i]

Using shared memory

28

```
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
```

Using shared memory: coalescing

29

```
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

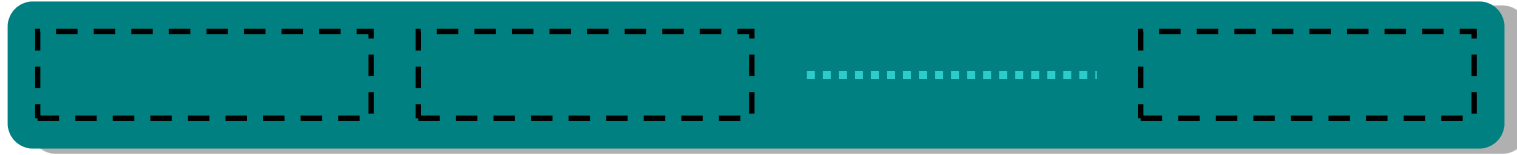
    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];    // COALESCED ACCESS!

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
```

A Common Programming Strategy

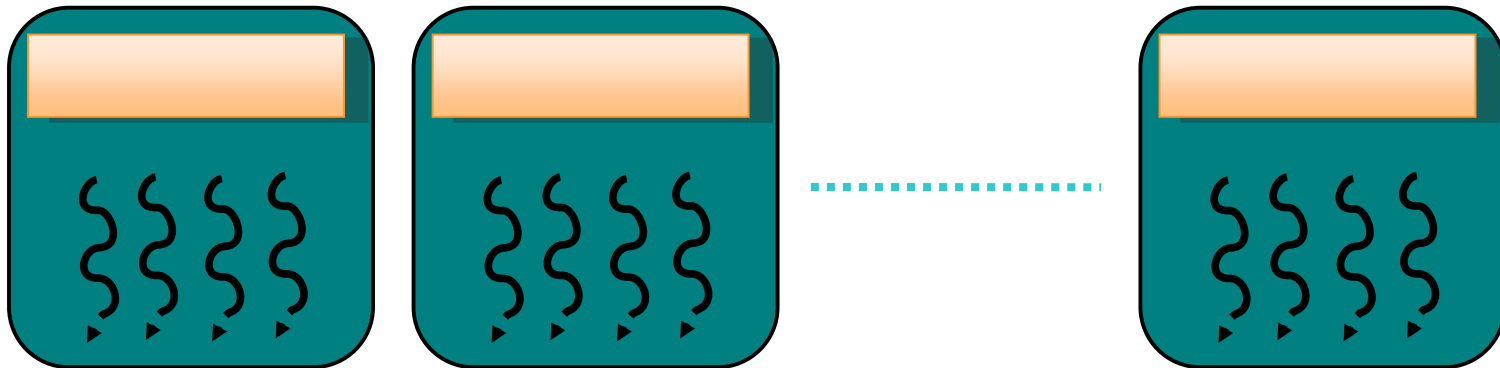
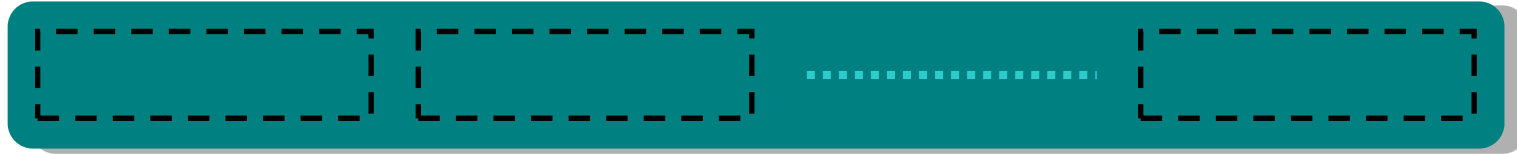
30



- Partition data into subsets that fit into shared memory

A Common Programming Strategy

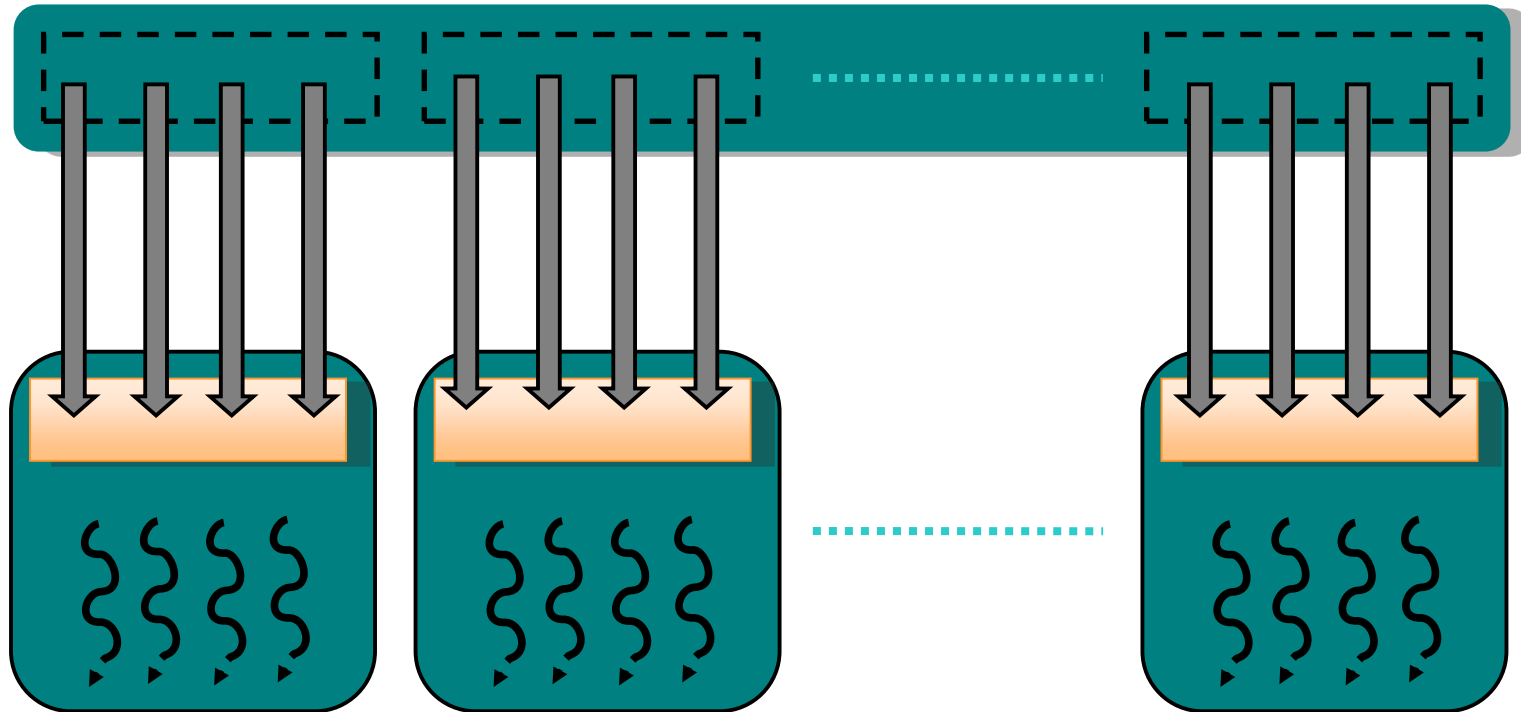
31



- Handle each data subset with one thread block

A Common Programming Strategy

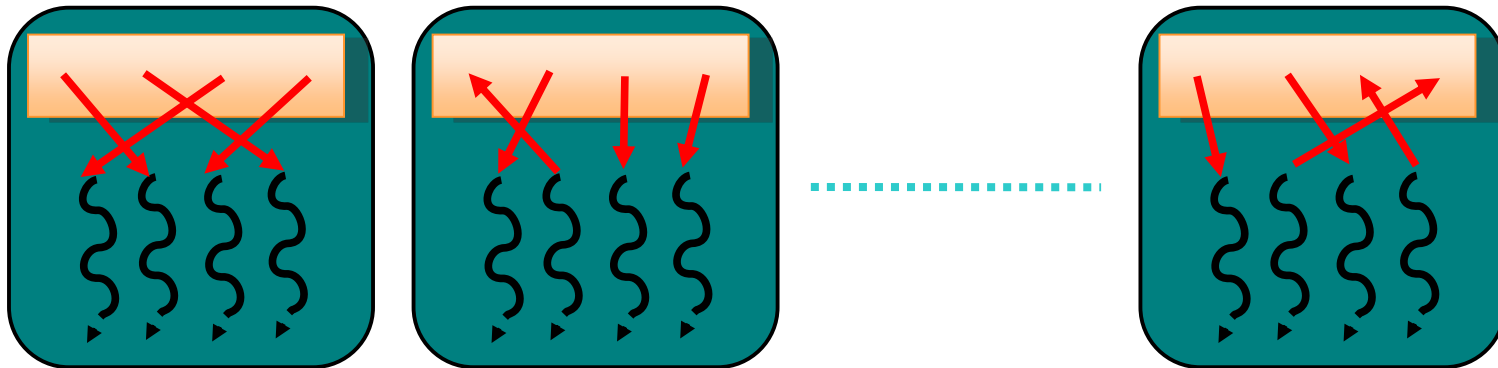
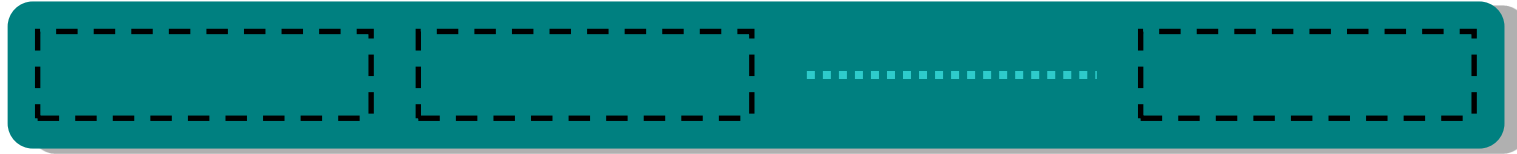
32



- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism

A Common Programming Strategy

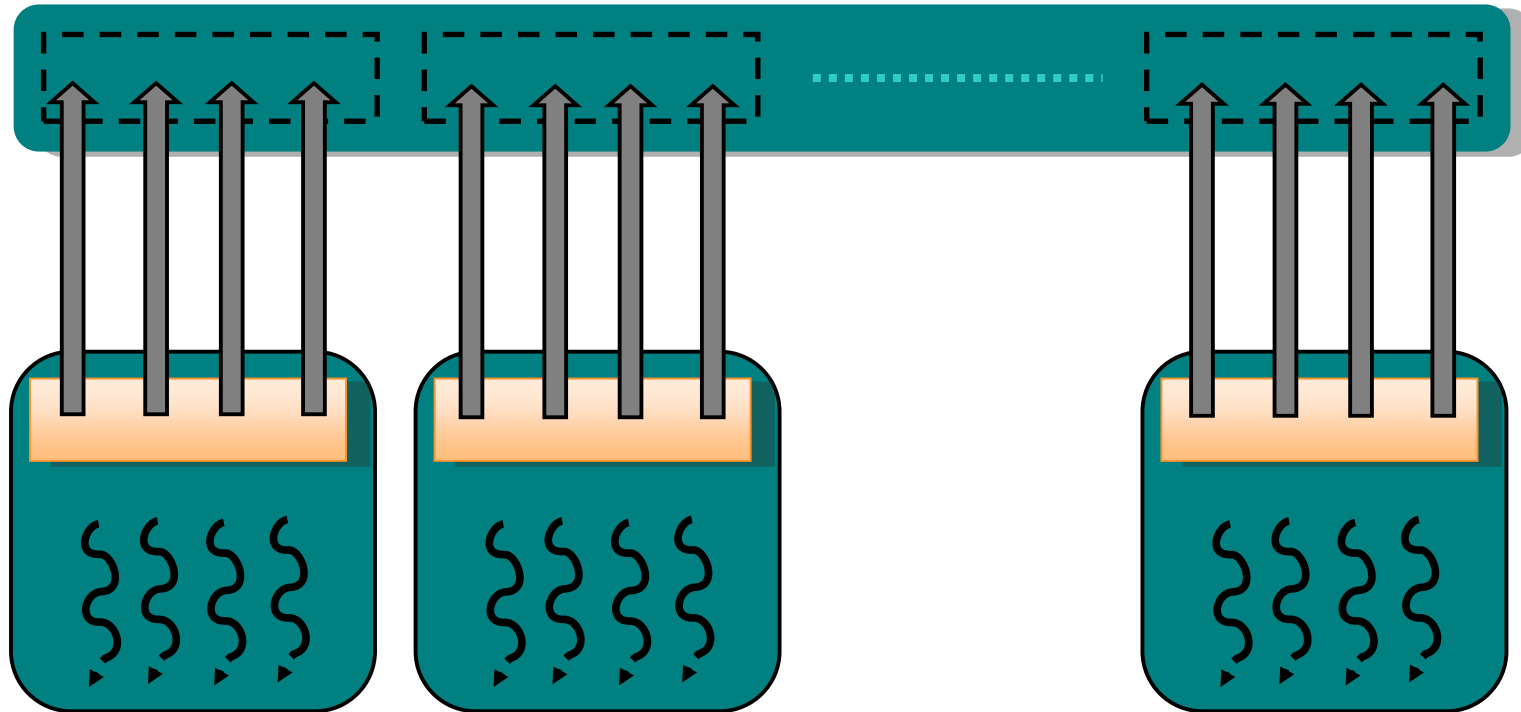
33



- Perform the computation on the subset from shared memory

A Common Programming Strategy

34



- Copy the result from shared memory back to device memory

35

CUDA: optimizing your application

Optimizing Occupancy

Thread Scheduling

36

- SM implements zero-overhead warp scheduling
 - ▣ A warp is a group of 32 threads that runs concurrently on a SM
 - ▣ At any time, only one of the warps is executed by SM
 - ▣ Warps whose next instruction has its inputs ready for consumption are eligible for execution
 - ▣ Eligible Warps are selected for execution on a prioritized scheduling policy
 - ▣ All threads in a warp execute the same instruction when selected

Stalling warps

37

- What happens if all warps are stalled?
 - ▣ No instruction issued → performance lost
- Most common reason for stalling?
 - ▣ Waiting on global memory
- If your code reads global memory every couple of instructions
 - ▣ You should try to maximize occupancy

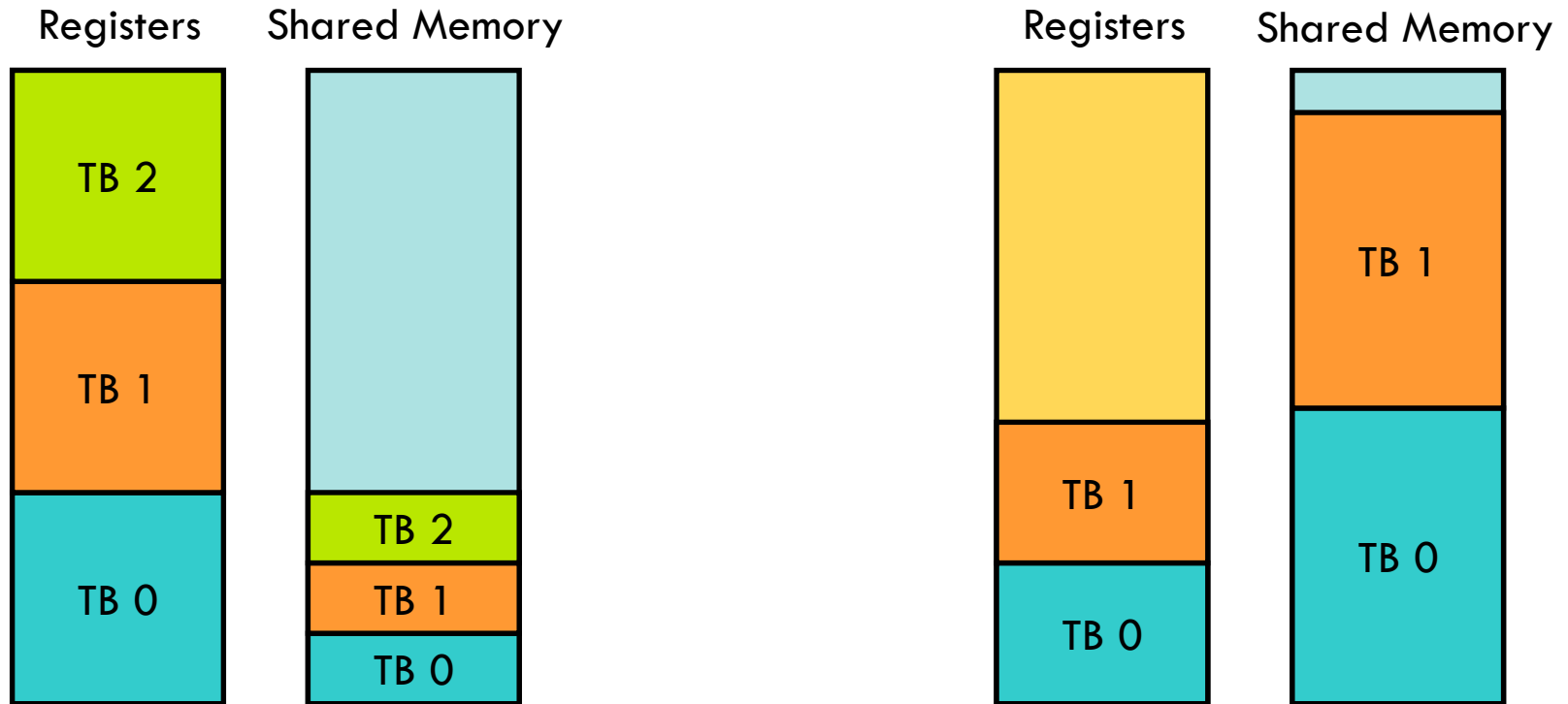
Occupancy

38

- What determines occupancy?
- Limited resources!
 - ▣ Register usage per thread
 - ▣ Shared memory per thread block

Resource Limits (1)

39



- Pool of registers and shared memory per SM
 - ▣ Each thread block grabs registers & shared memory
 - ▣ If one or the other is fully utilized ➡ no more thread blocks

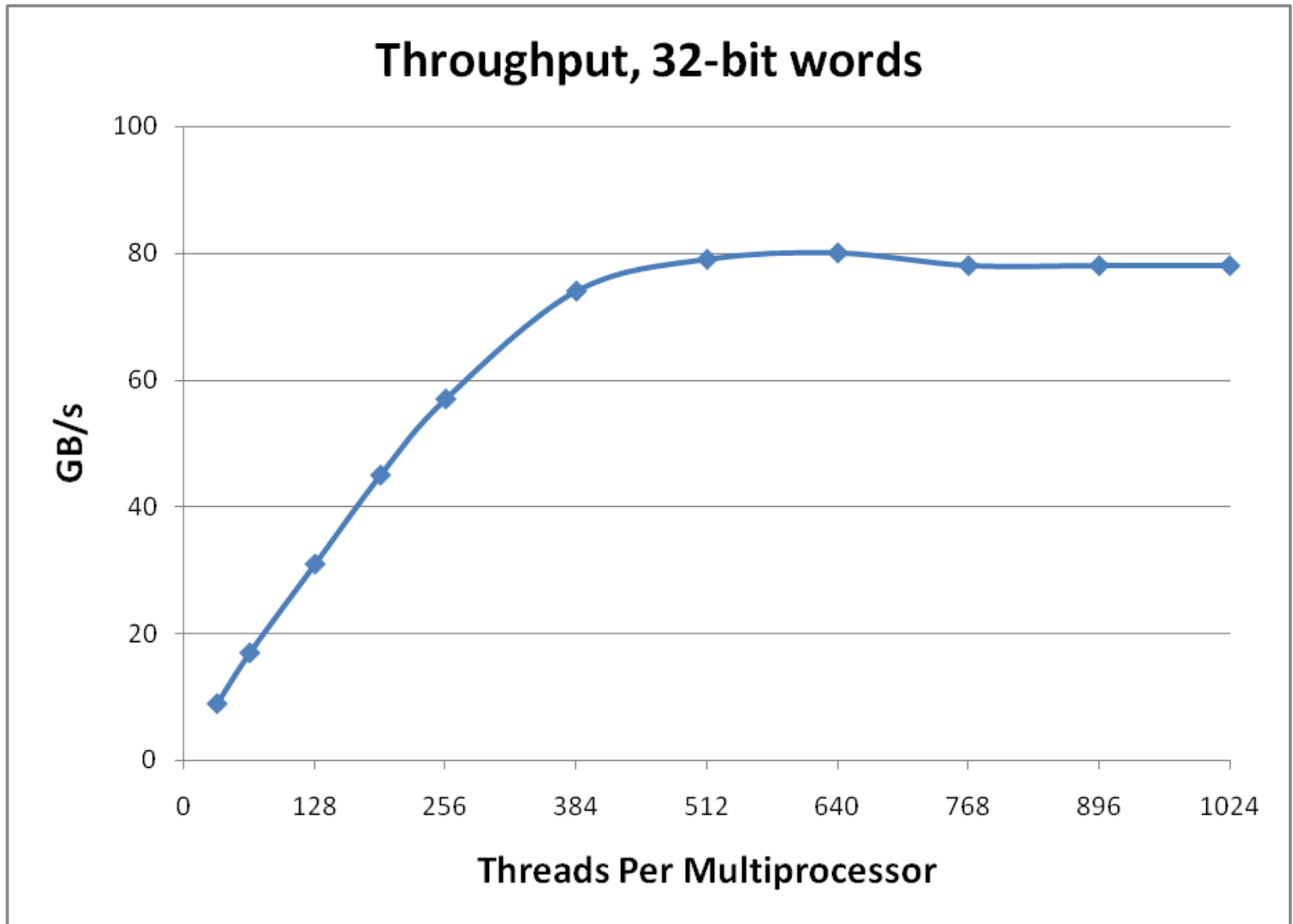
Resource Limits (2)

40

- Can only have a limited number of blocks per SM
 - ▣ If they're too small, can't fill up the SM
- Higher occupancy has diminishing returns for hiding latency

Hiding Latency with more threads

41



How do you know what you're using?

42

- Use “ `--ptxas-options=-v` ” to get register and shared memory usage
- You can plug those numbers into CUDA Occupancy Calculator

cmem[0]:kernel arguments

cmem[2]:user defined constant objects

cmem[16]:compiler generated constants (some of which may correspond to literal constants in the source code)