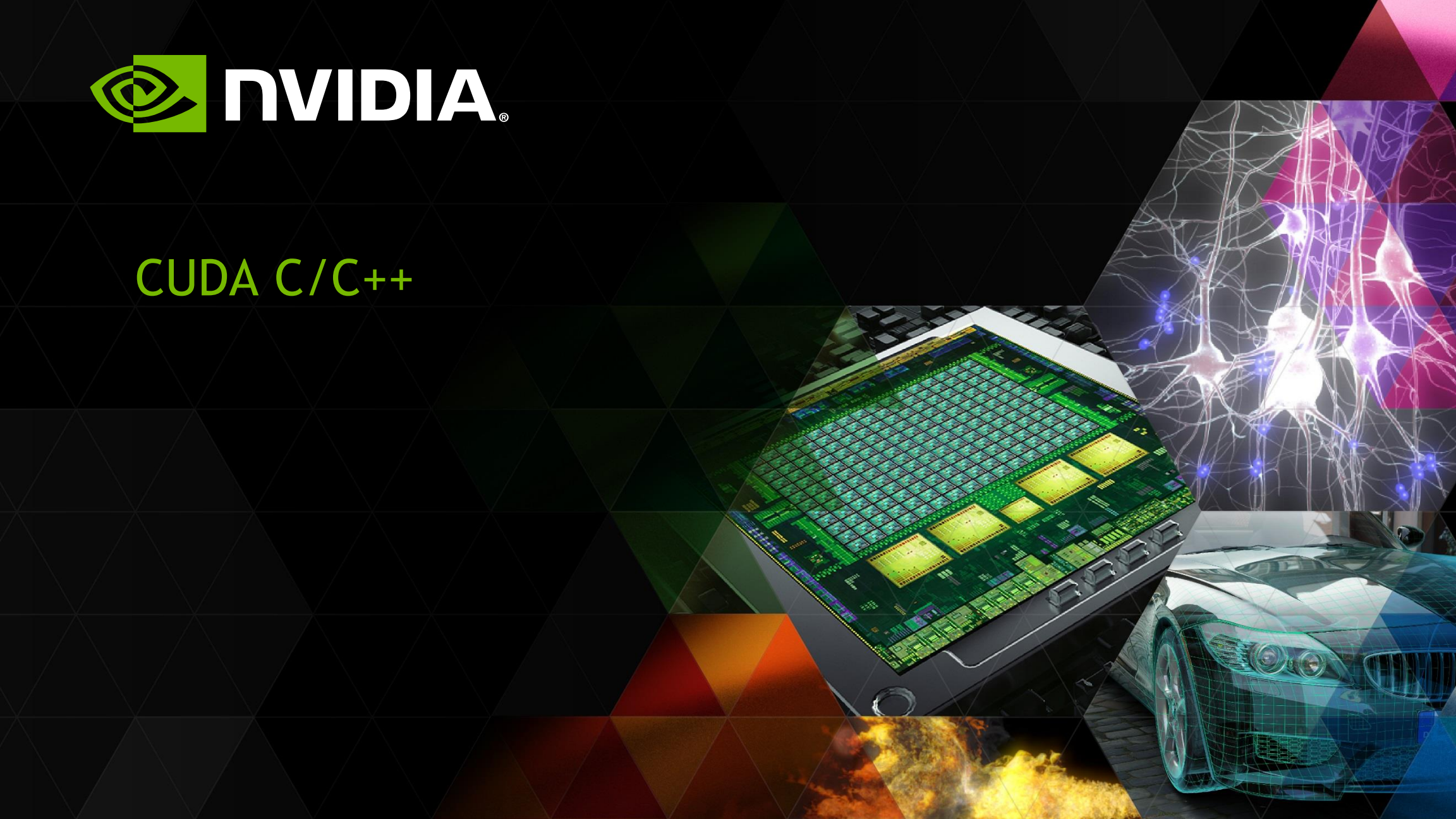

NVIDIA
CUDA C/C++

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 in this session?
  - 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

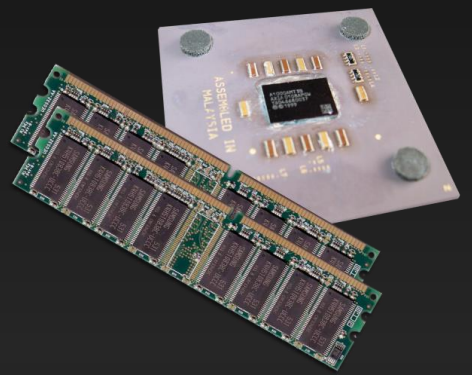
HELLO WORLD!

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

Heterogeneous Computing

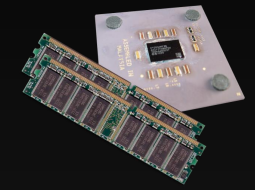
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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();

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

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

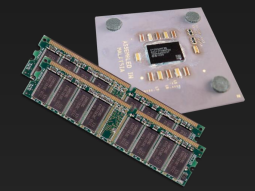
parallel fn

serial code

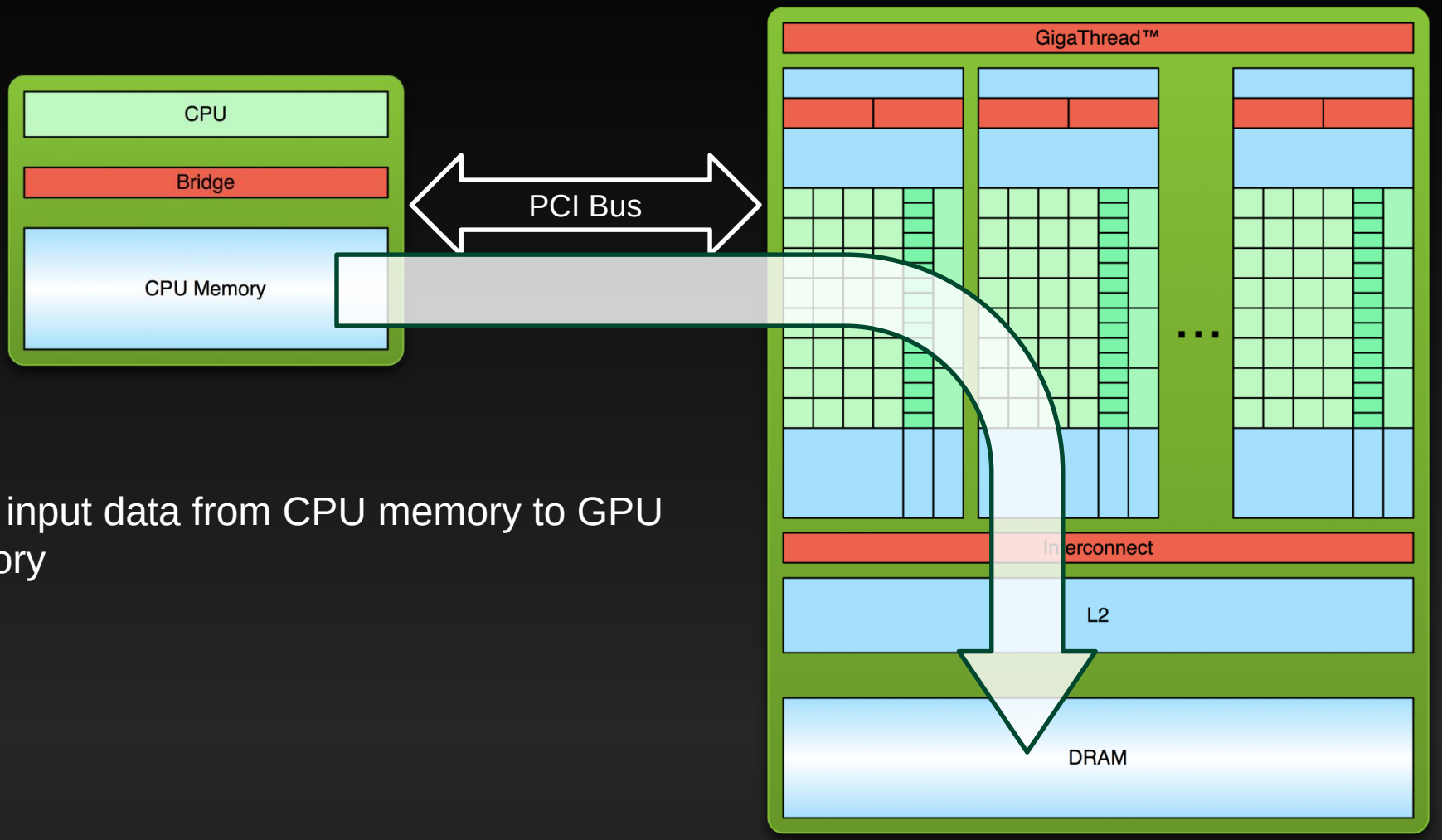
parallel code

serial code



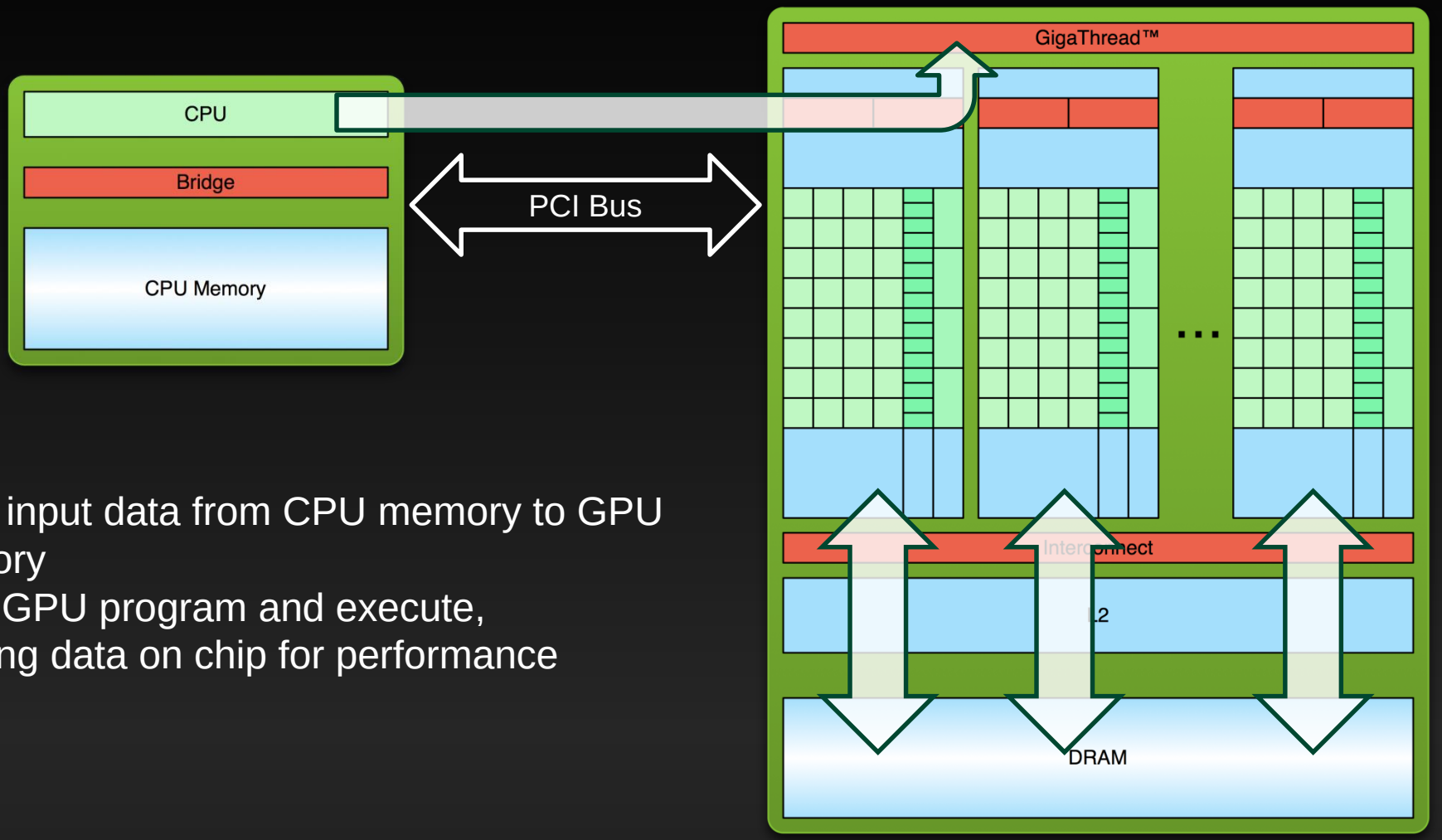


Simple Processing Flow



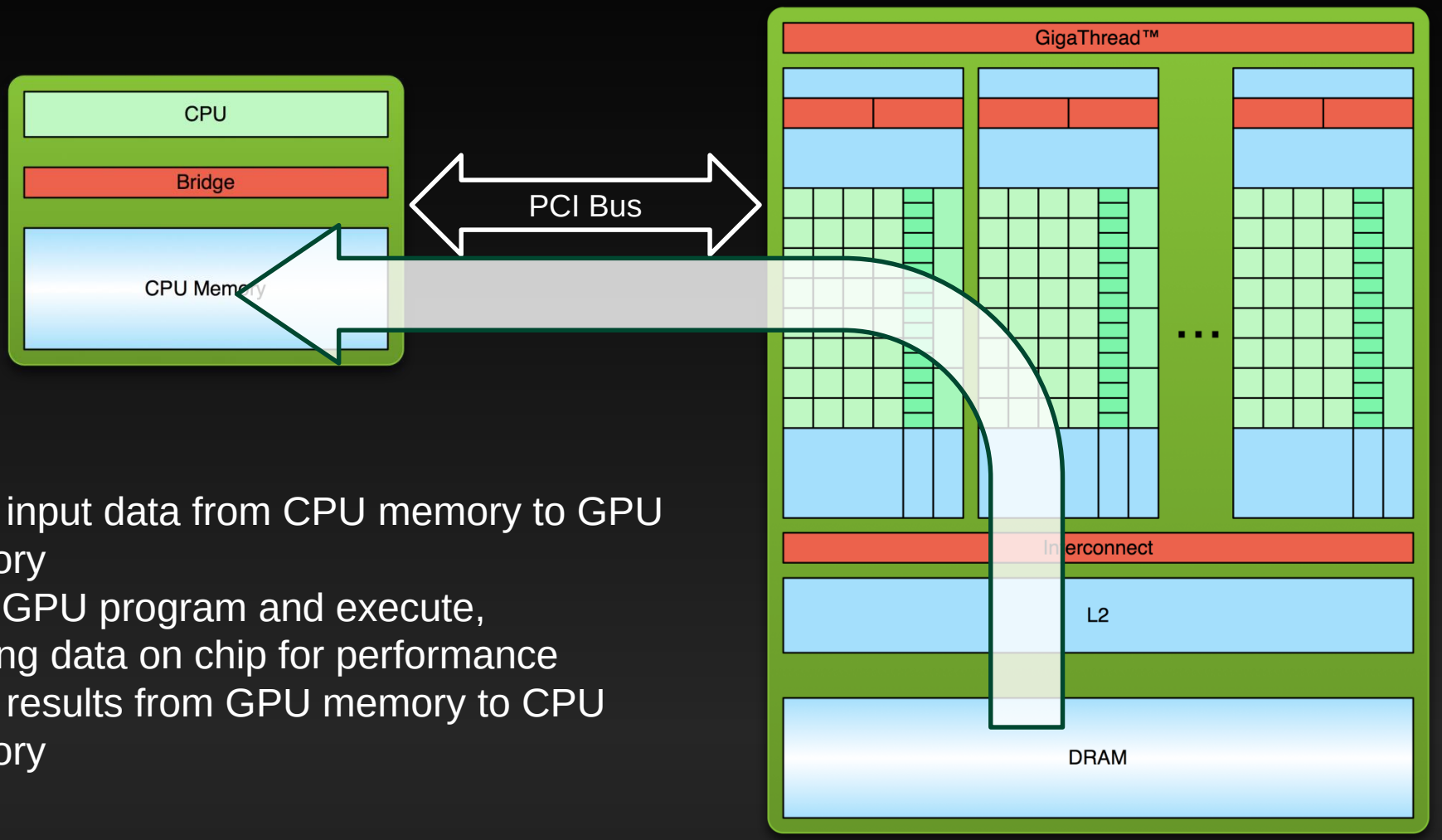
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

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) {
      printf("Hello World from device!\n");
}

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

- Two new syntactic elements…

Hello World! with Device Code


```
__global__ void mykernel(void) {
      printf("Hello world from device!\n");
}
```

- 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

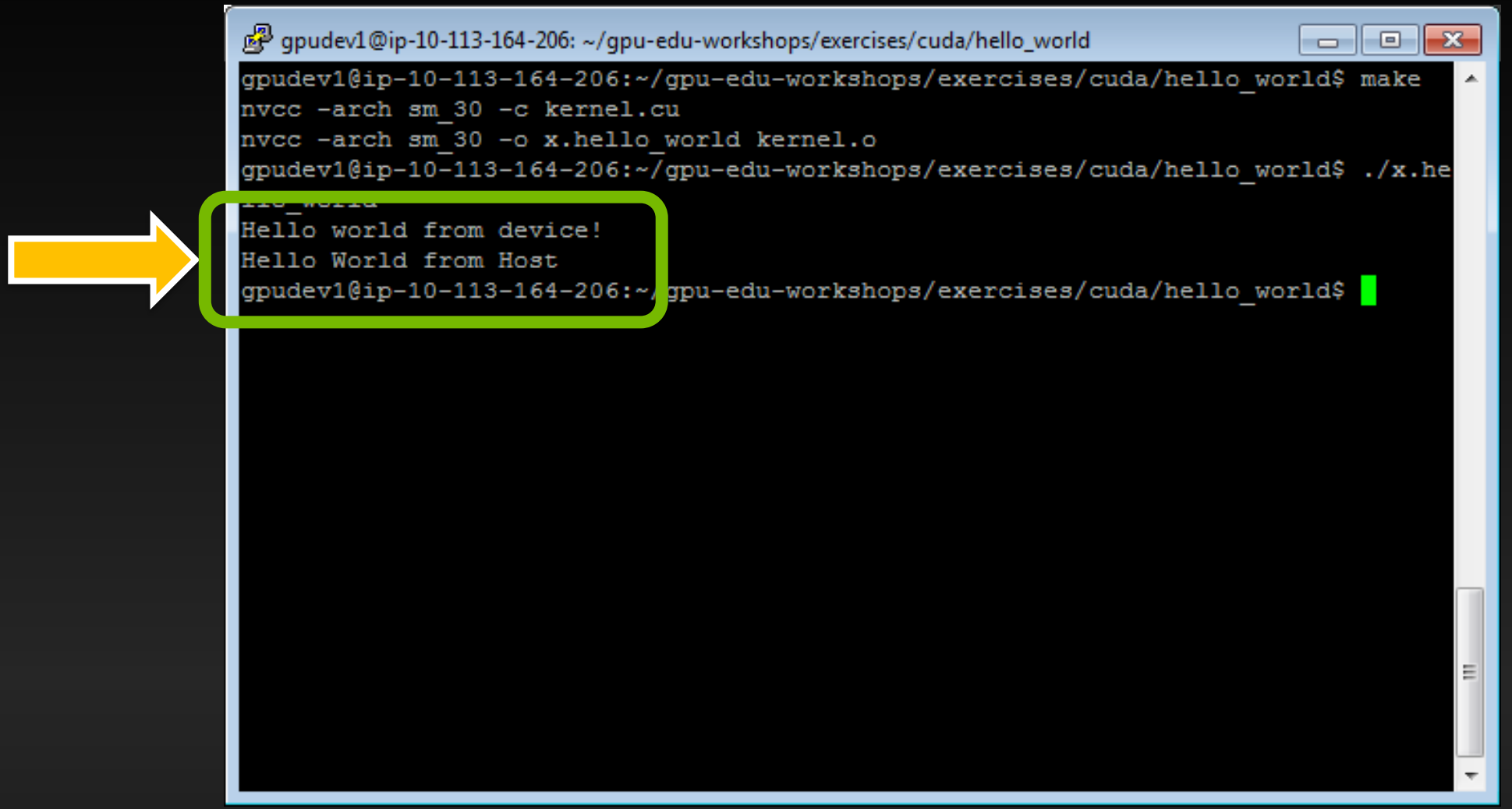

```
mykernel<<<1,1>>>();
```

- 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

- Login to your AWS instance
- Each coding project in a separate folder in the following dir
  - gpu-edu-workshops/exercises/
- `cd gpu-edu-workshops/exercises/cuda/hello_world`
- All dirs have Makefiles for you
- Try building/running the code
  - `make`
  - `./x.hello_world`

Screenshot


gpudev1@ip-10-113-164-206: ~/gpu-edu-workshops/exercises/cuda/hello_world
gpudev1@ip-10-113-164-206:~/gpu-edu-workshops/exercises/cuda/hello_world$ make
nvcc -arch sm_30 -c kernel.cu
nvcc -arch sm_30 -o x.hello_world kernel.o
gpudev1@ip-10-113-164-206:~/gpu-edu-workshops/exercises/cuda/hello_world$ ./x.he
Hello world from device!
Hello World from Host
gpudev1@ip-10-113-164-206:~/gpu-edu-workshops/exercises/cuda/hello_world$

Hello World! with Device Code


```
__global__ void mykernel(void) {
      printf("Hello from device!\n");
}

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

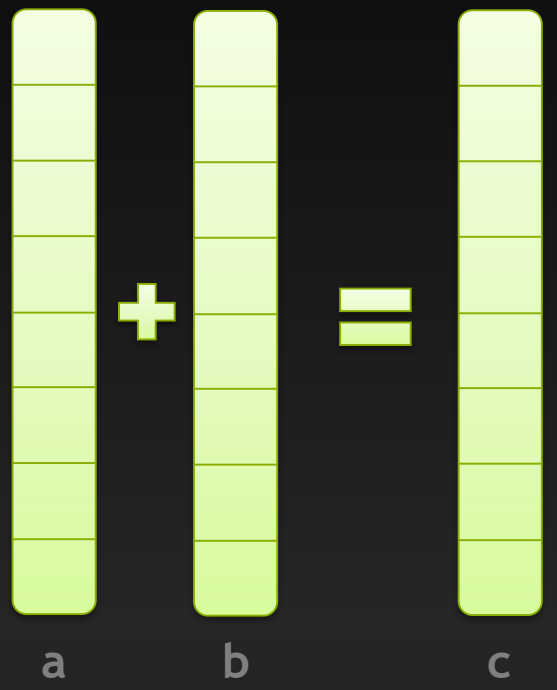
Output:

```
$ nvcc hello.cu
$ a.out
Hello from
device!
Hello from
Host!
$
```

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

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

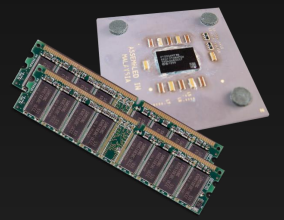
```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- `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()`

Error checking

- Kernels
  - `kernel<<<>>>(…)`
  - `checkKERNEL()`
- All other CUDA calls
  - `checkCUDA( cuda…() )`
- Add `–DDEBUG` to your compile line in Makefile

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()…
- Open exercises/cuda/simple_add/kernel.cu
- Fill-in missing code as indicated.
  - Need to replace "FIXME" with code. Comments should help.
  - If something isn't clear, PLEASE ASK! ☺
  - PASS/FAIL will be printed to the screen

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`

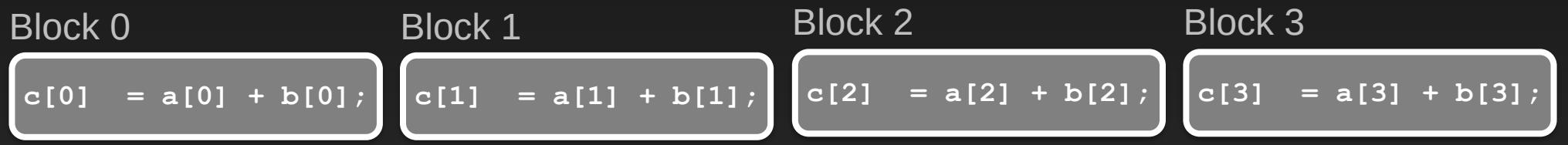
```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- 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:



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()…
- Open exercises/cuda/simple_add_blocks/kernel.cu
- Fill-in missing code as indicated.
  - Should be clear from comments where you need to add some code
  - Need to replace "FIXME" with the proper piece of code.

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()`…
- Open exercises/cuda/simple_add_threads/kernel.cu

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;
}
```

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- **Indexing**
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

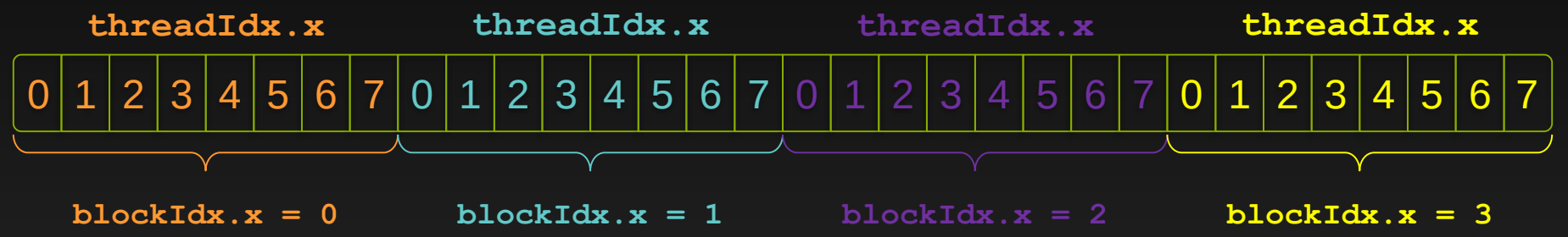
COMBINING THREADS AND BLOCKS

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)

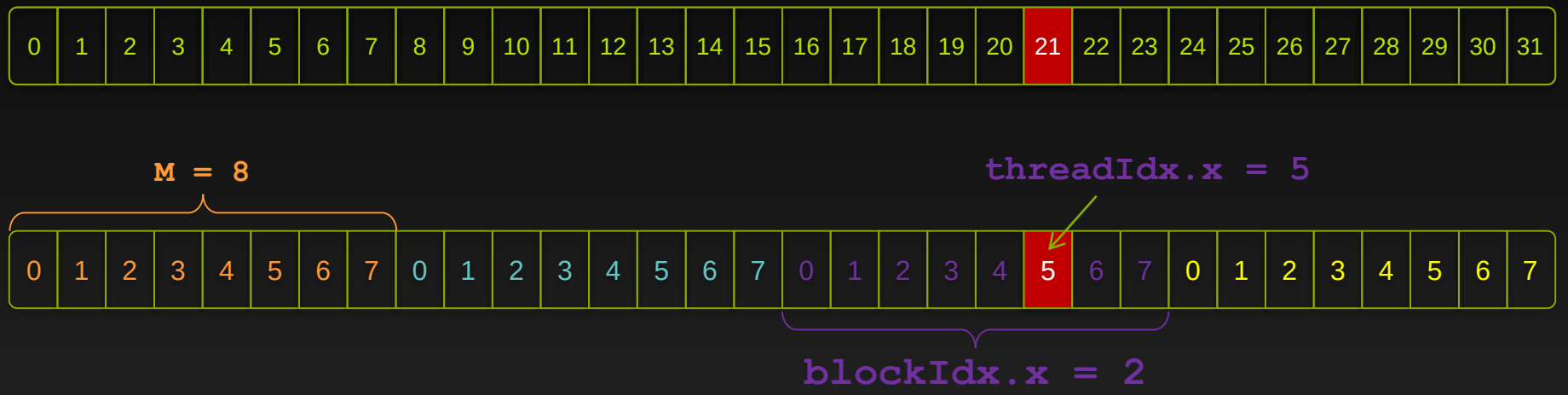


- With M threads/block a unique index for each thread is given by:

```
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()`?
- Open exercises/cuda/simple_add_blocks_threads/kernel.cu

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>>>(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

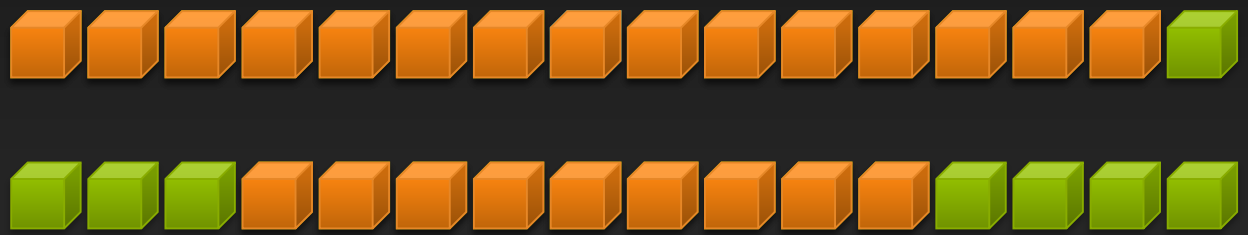
1D Stencil

- 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



dim3 datatype

- 3-dimensional vector type
- Used for setting threadblock and grid dimensions

```
dim3 threads( THREADS_PER_BLOCK, 1, 1 );
dim3 blocks( N / threads.x, 1, 1);
kernel_call<<< blocks, threads >>>( args… );
```

- Equivalent to:

```
kernel_call<<< N / THREADS_PER_BLOCK,
THREADS_PER_BLOCK >>> ( args…);
```

Simple Stencil in 1d

- Open exercises/cuda/simple_stencil/kernel.cu
- Finish the kernel and the kernel launch
  - Each thread calculates one stencil value
  - Reads 2*RADIUS + 1 values
  - dim3 type: CUDA 3 dimensional struct used for grid/block sizes
- Inserted GPU timers into code to time the execution of the kernel

- Try various sizes of N, RADIUS, BLOCK
- Time a large (2048*2048) value of N with a RADIUS of 7

Can we do better?

- Input elements are read multiple times
  - With RADIUS=3, each input element is read seven times!
  - Neighbouring threads read most of the same elements.
    - Thread 7 reads elements 4 through 10
    - Thread 8 reads elements 5 through 11

- Can we avoid redundant reading of data?

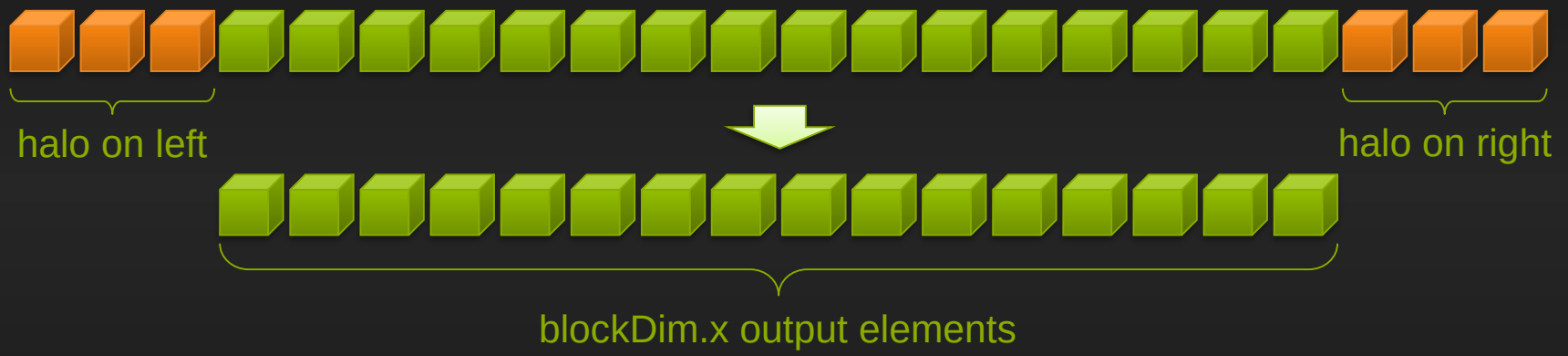
Sharing Data Between Threads

- 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

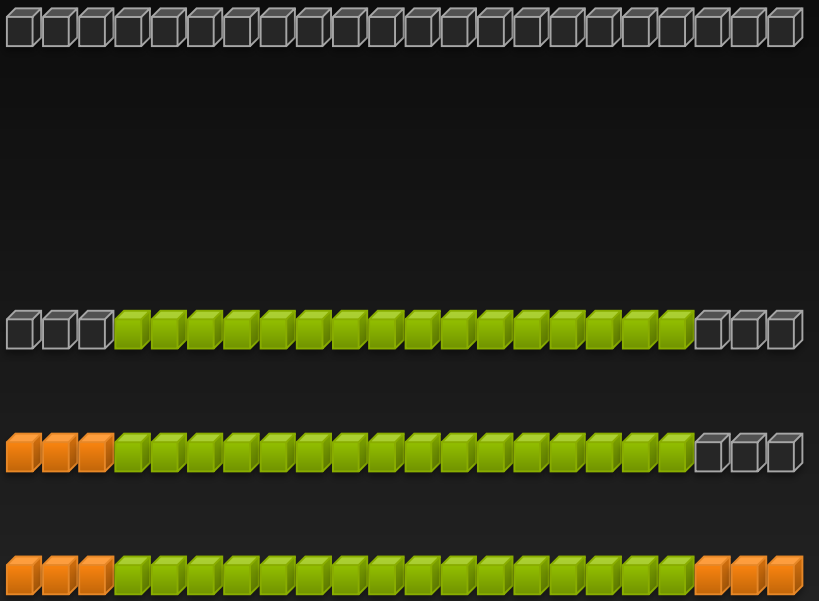
- Cache data in shared memory (user managed scratch-pad)
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory

- Each block needs a halo of `radius` elements at each boundary



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;
}
```

Simple Stencil 1d with shared memory

- cd exercises/cuda/simple_stencil_smem/

- Run the code. It should build/run without modification.
  - If Errors occur, first offending element will be printed to the screen
  - FAIL will be printed to the screen
- What is the result with N=(1024*1024) and THREADS_PER_BLOCK=32?
- What is the result with N=(1024*1024) and THREADS_PER_BLOCK=64?
  - Why?

Data Race!

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

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

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

- Insert `__syncthreads()` into the kernel in the proper location
- Compare timing of previous simple stencil with the current shared memory implementation for same (large N) and BLOCK=512

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

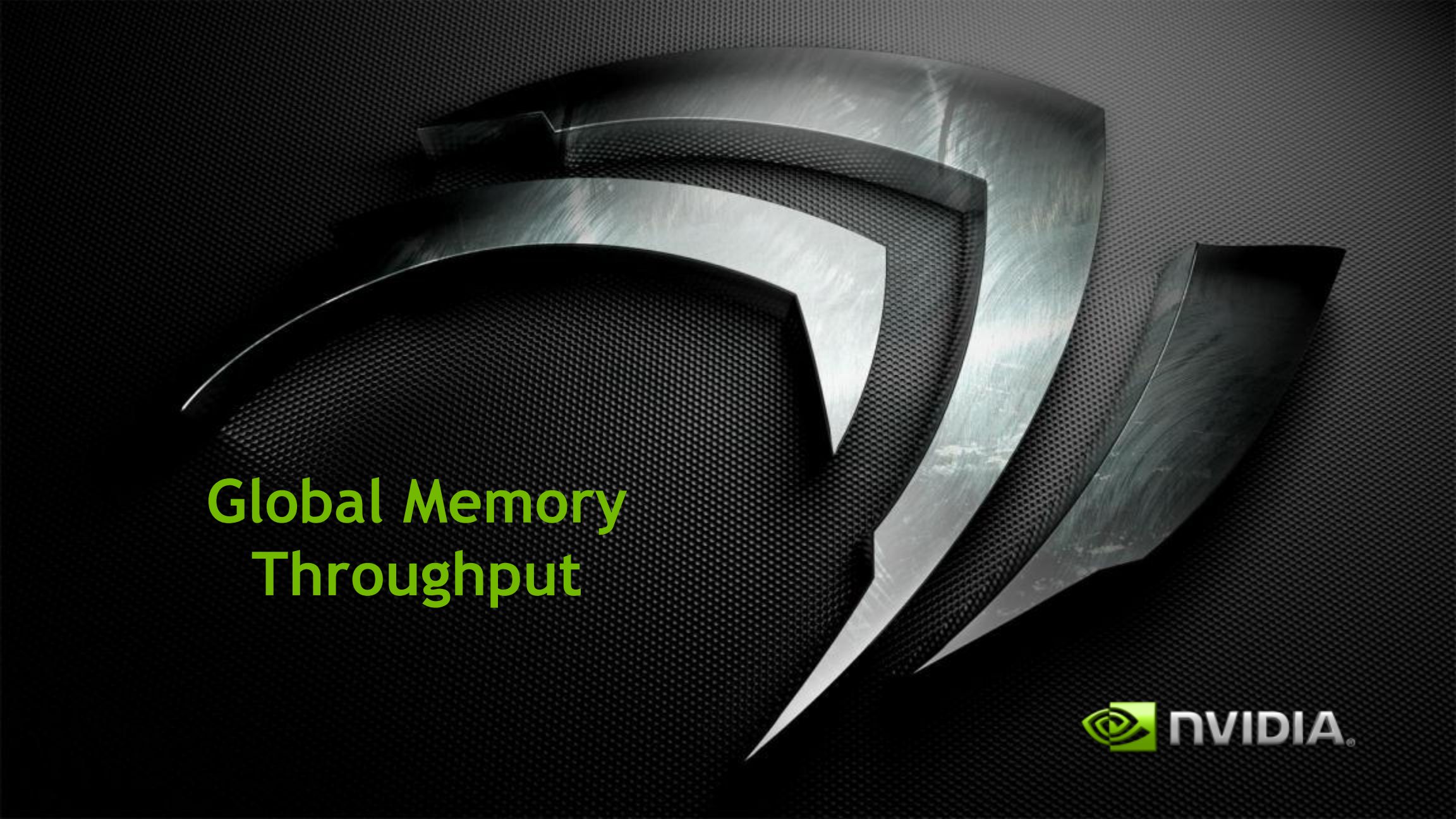
- Allocate elements to threads:

```
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
  - Using large shared mem size impacts number of blocks that can be scheduled on an SM (48K total smem size)
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

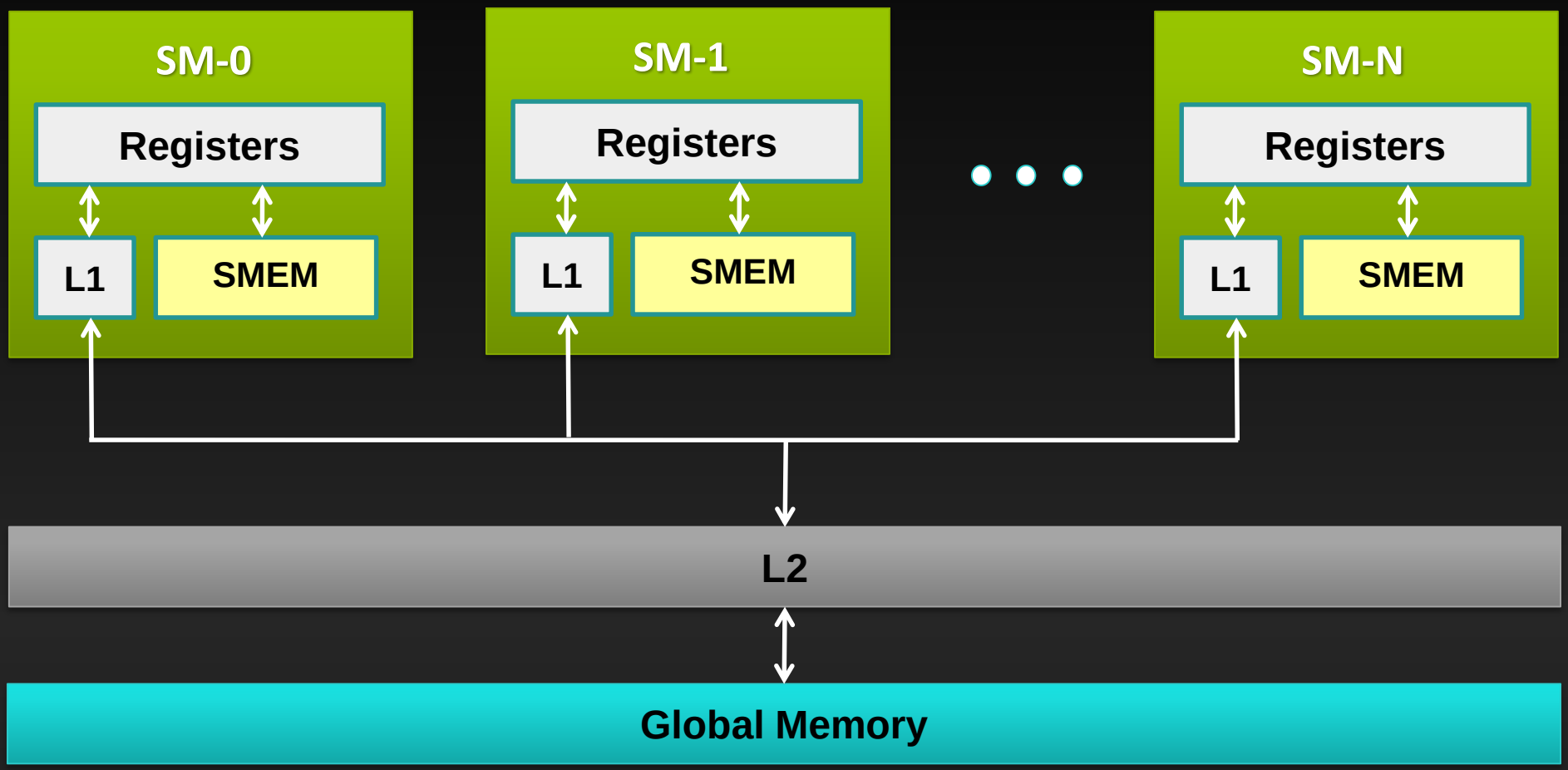
Global Memory Throughput



Memory Hierarchy Review

- Local storage
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
- Shared memory / L1
  - Program configurable: 16KB shared / 48 KB L1 OR 48KB shared / 16KB L1
  - Shared memory is accessible by the threads in the same threadblock
  - Very low latency
  - Very high throughput: 1+ TB/s aggregate
- L2
  - All accesses to global memory go through L2, including copies to/from CPU host
- Global memory
  - Accessible by all threads as well as host (CPU)
  - High latency (400-800 cycles)
  - Throughput: up to 177 GB/s

GPU Memory Hierarchy Review

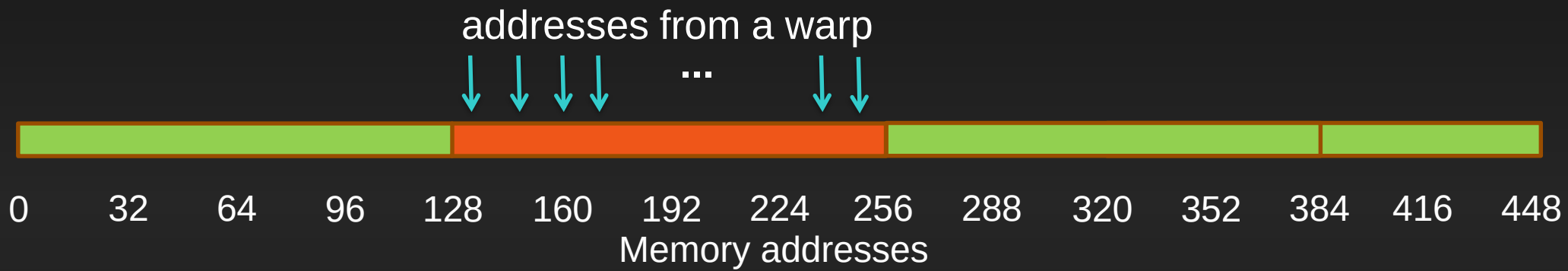


Load Operation

- Memory operations are issued per warp (32 threads)
  - Just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

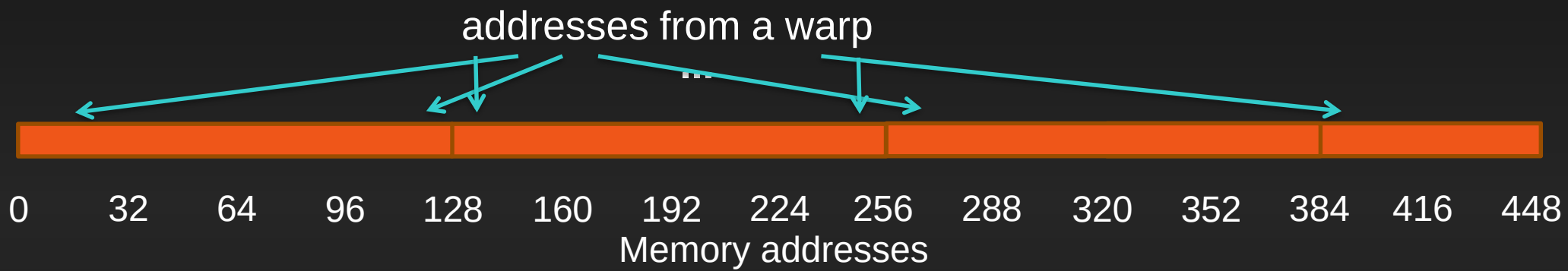
Caching Load--coalesced

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



Caching Load--uncoalesced

- Warp requests 32 scattered 4-byte words
- Addresses fall within N cache-lines
  - Warp needs 128 bytes
  - N*128 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*128)



GMEM OPTIMIZATION GUIDELINES

- Strive for perfect coalescing!!!
- Have enough concurrent accesses to saturate the bus
  - Process several elements per thread
    - Multiple loads get pipelined
    - Indexing calculations can often be reused
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)

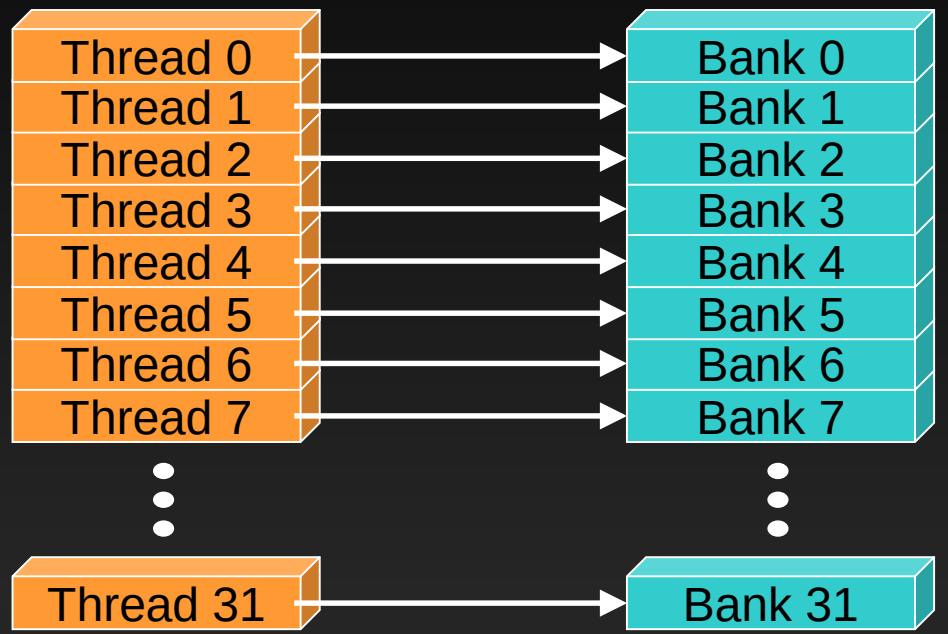
Shared Memory

SHARED MEMORY

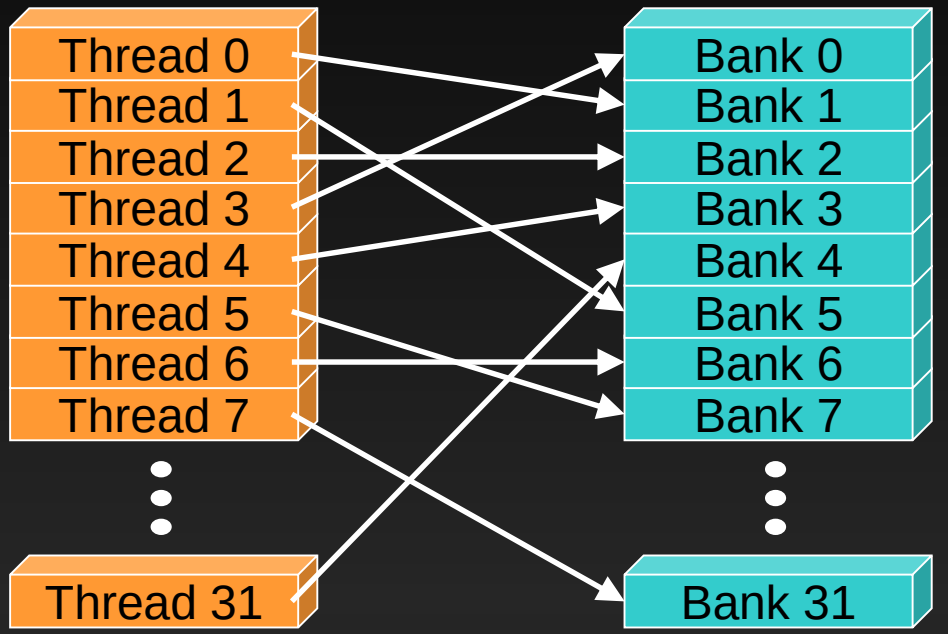
- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Organization:
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks
- If you use shared memory in a kernel, you should almost always use `__syncthreads()` to avoid race conditions!!!

Bank Addressing Examples

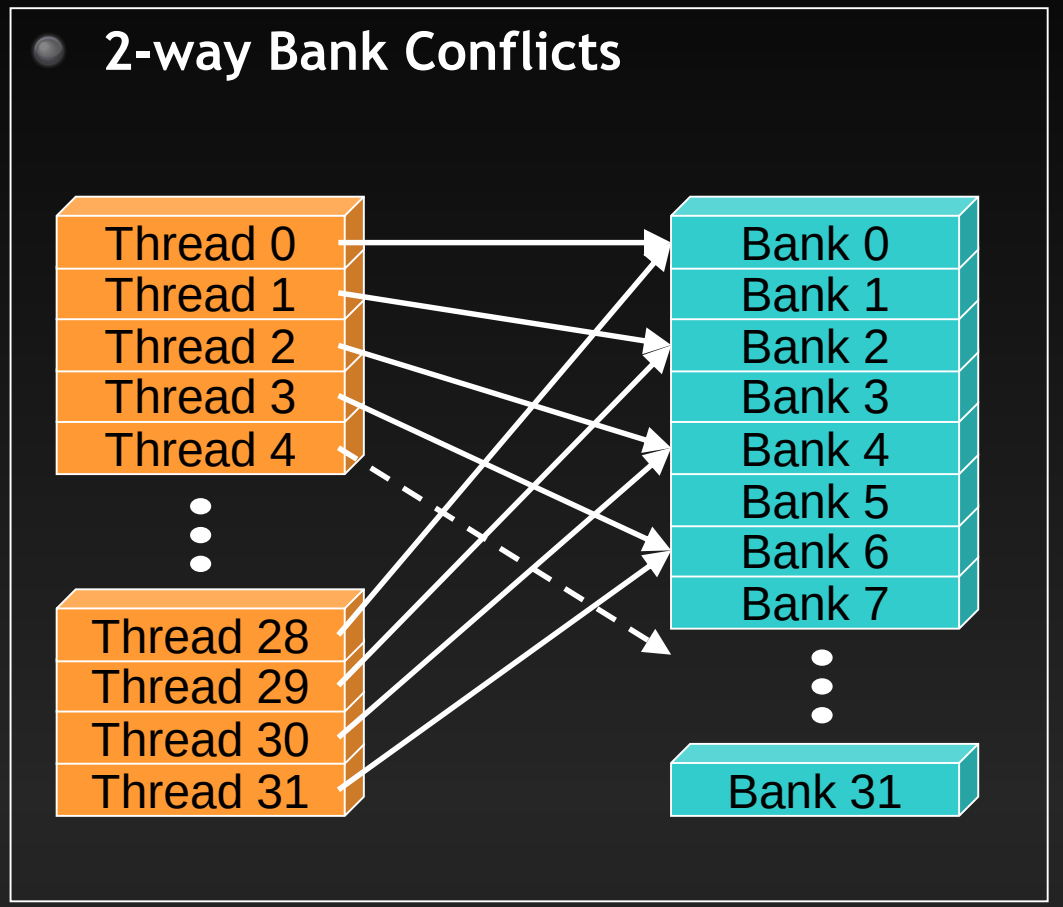
- No Bank Conflicts

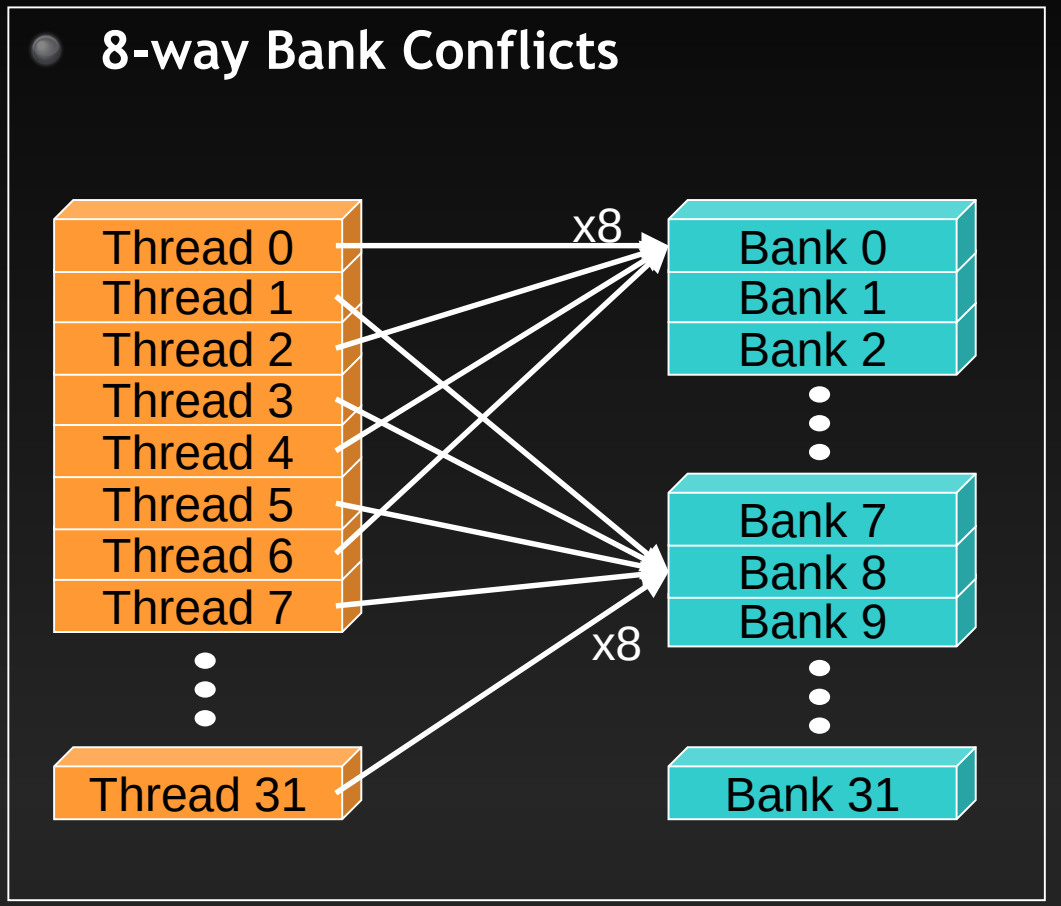


- No Bank Conflicts



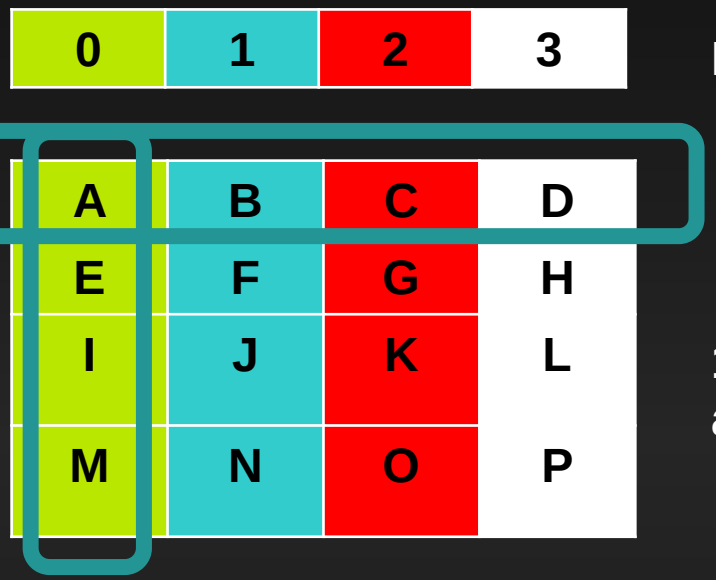
Bank Addressing Examples





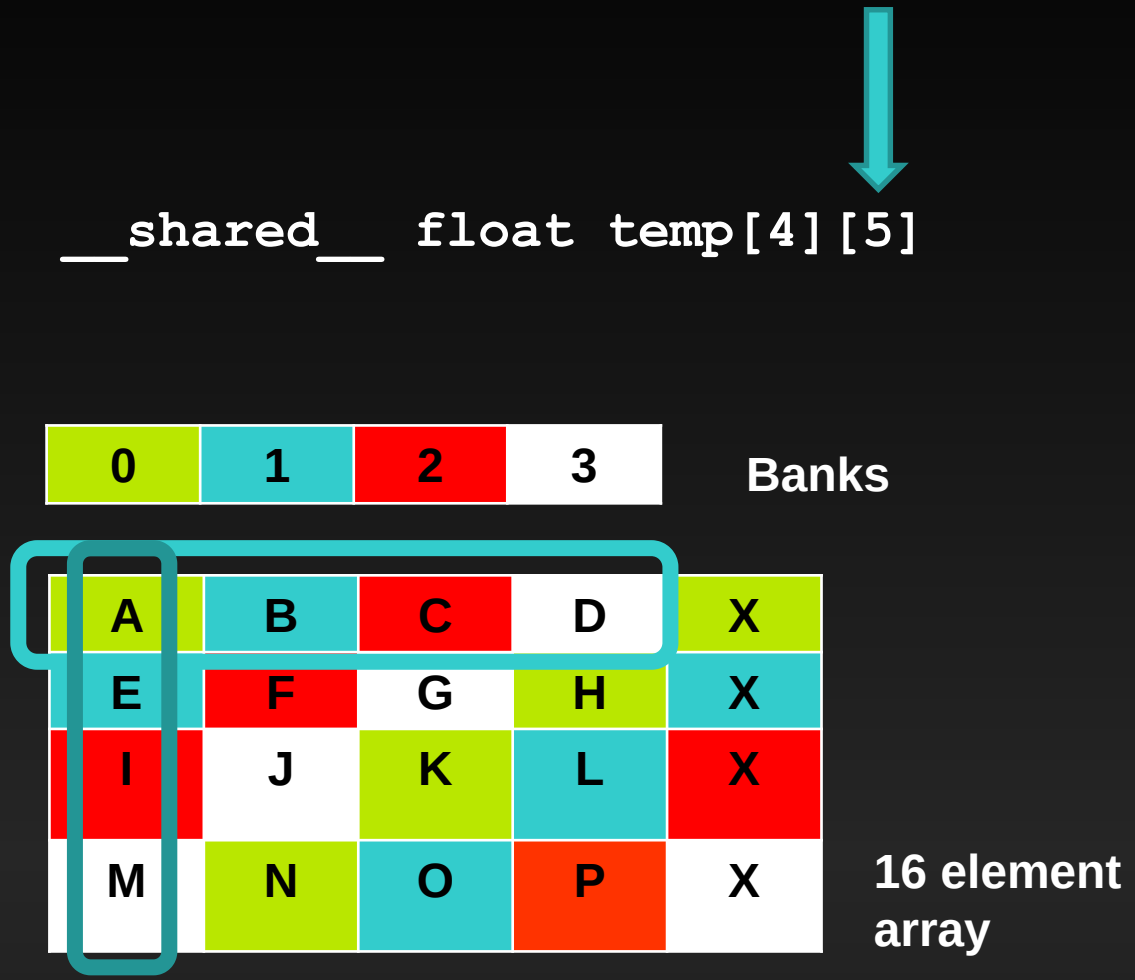
Shared memory: Banks and Conflicts

```
__shared__ float temp[4][4]
```



- Consider 4x4 SMEM array
- Consecutive threads access consecutive elements in a row
  - PERFECT ACCESS!
- Consecutive threads access consecutive elements in a column
  - 4-way bank conflict because 4 threads try to access same bank.

Shared memory: Avoiding bank conflicts

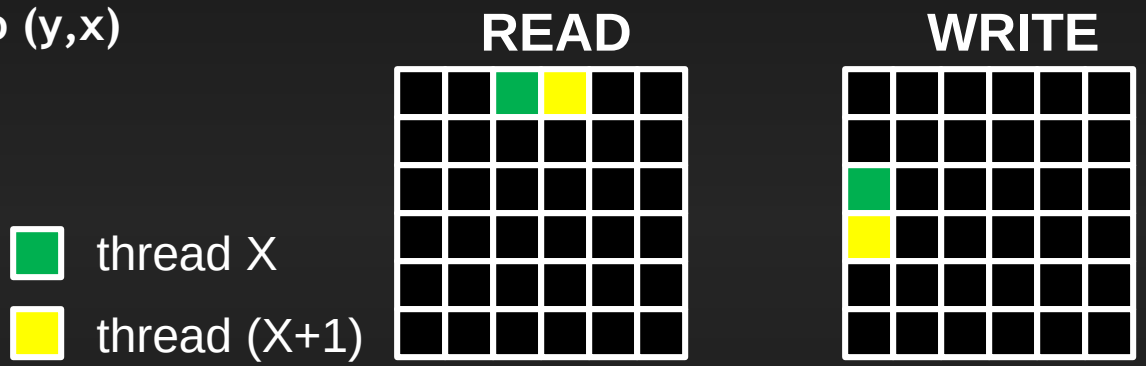


- Consider 4x5 SMEM array
  - Padded the column dimension
- Consecutive threads access consecutive elements in a row
  - PERFECT ACCESS!
- Consecutive threads access consecutive elements in a column
  - PERFECT ACCESS!

Matrix Transpose

Matrix Transpose

- Open exercises/cuda/naive_transpose/kernel.cu
  - Defined INDX(row,col,ld) to translate 2d coordinates to 1d index
- Column-major double precision elems
  - Out-of-place transpose
- Naïve implementation:
  - Square threadblocks
  - Each thread:
    - Computes its global x and y coordinates
    - Reads from (x,y), writes to (y,x)



naive_transpose/kernel.cu

- Use INDX(row,col,ld) to translate 2d coordinates to 1d index
  - Defined as MACRO at top of source code
- Finish the kernel and kernel launch parameters
  - Each thread transposes one element from A to C, out-of-place
- Kernel is compared to a CPU-side transpose
  - Both performance and answers are compared

- Once you get correct answers in your kernel:
  - Measure performance of kernel with N=1024 and N=4096

NVIDIA Visual Profiler

- `nvvp &`
- `Choose File->New Session`
- Choose exercises/cuda/naïve_transpose/x.transpose
- Select proper “naïve_transpose” for working directory
- Then click “Next”
- Choose timeout = 60 seconds.
- Uncheck “Run Analysis” then click “Finish”
- Click “Analyze All”
  - Profiler will execute the code multiple times to record all performance counters
- In the timeline window, click on the kernel of interest

Profiler results?

- What does the profiler output show us?
- Why are load and store performance so divergent?
- Ideas for potential improvement?

Cause and Remedy

- Cause:
  - Due to nature of operation, one of the accesses (read or write) will be at large strides, i.e., uncoalesced!
    - 32 doubles (256 bytes) in this case (on a warp basis)
    - Thus, bandwidth will be wasted as only a portion of a transaction is used by the application
- Remedy
  - Stage accesses through shared memory
  - A threadblock:
    - Reads a tile from GMEM to SMEM
    - Transposes the tile in SMEM
    - Write a tile, in a coalesced way, from SMEM to GMEM

smem_transpose/kernel.cu

- Finish the kernel code
  - Particularly the index calculations for the transpose.
- One thread block cooperatively operates on a block of the matrix
  - Index calculations need to know which block I am in the full matrix
- HINT: You are using shared memory. What should you include when using smem?
- Once you get correct answers in your kernel:
  - Measure performance of kernel with N=1024 and N=4096

Visual Profiler

- Run the profiler
- What are the results?
- What is happening?
- How to fix it?
  - Hint: requires adding only 2 characters to the kernel source!

SMEM bank conflicts

- Recall that smem has 32 banks of 4 bytes each
- When multiple threads IN THE SAME WARP access the same bank, a conflict occurs and performance is affected negatively
- Consider `__shared__ double s[16][16]`
  - Read/write `s[tidx][tidy]` (each read/write requires two 4byte banks)
  - tidx are consecutive threads
  - They are accessing the s[][] array with stride of 16 doubles
  - 16 doubles == 128 bytes == 32 banks * 4 bytes.
  - `s[0][tidy]` accesses banks 0,1 to grab its 8 byte double
  - `s[1][tidy]` accesses banks 0,1 to grab its 8 byte double
  - 16 threads are all accessing banks 0,1 in the same transaction!!!

SMEM bank conflicts cont'd

- tidx from 0 to 16 all access banks 0,1
  - 16 way bank conflict! VERY BAD for performance
- How to remedy?
  - Pad shared memory.
- `__shared__ double s[16][17]`
  - Now the stride between success tidx is 17 doubles,
  - i.e., 17 * 8bytes = 136 bytes
  - More importantly, 136 byte stride will be 32 + 2 banks
  - `s[0][tidy]` accesses banks 0,1 to grab its 8 byte double
  - `s[1][tidy]` accesses banks 2,3 to grab its 8 byte double
  - 16 threads are all accessing different banks!!!

Still in smem_transpose/kernel.cu

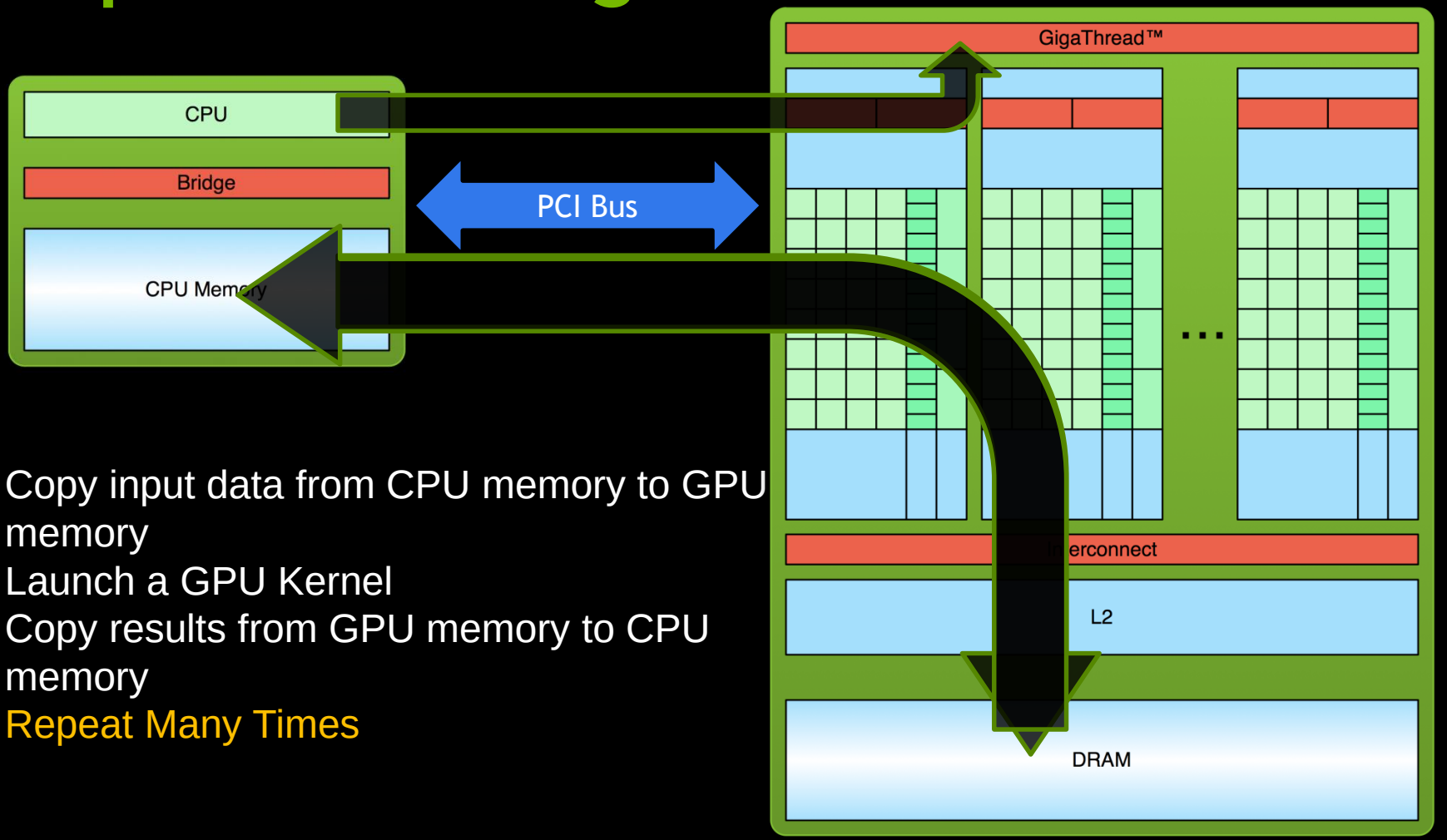
- Remedy the smem bank conflicts
- Once you get correct answers in your kernel:
  - Measure performance of kernel with N=1024 and N=4096
- Run with the profiler again.
  - Verify the bank conflicts have gone away

Review

- SMEM often used to alleviate poor GMEM accesses
  - Uncoalesced loads/stores were solved using SMEM
- SMEM almost always requires __syncthreads()
- SMEM often requires an analysis to minimize bank conflicts.
- Use NVIDIA Visual Profiler to identify performance bottlenecks

CUDA STREAMS (slides adapted from Justin Luitjens, NVIDIA)

Simple Processing Flow

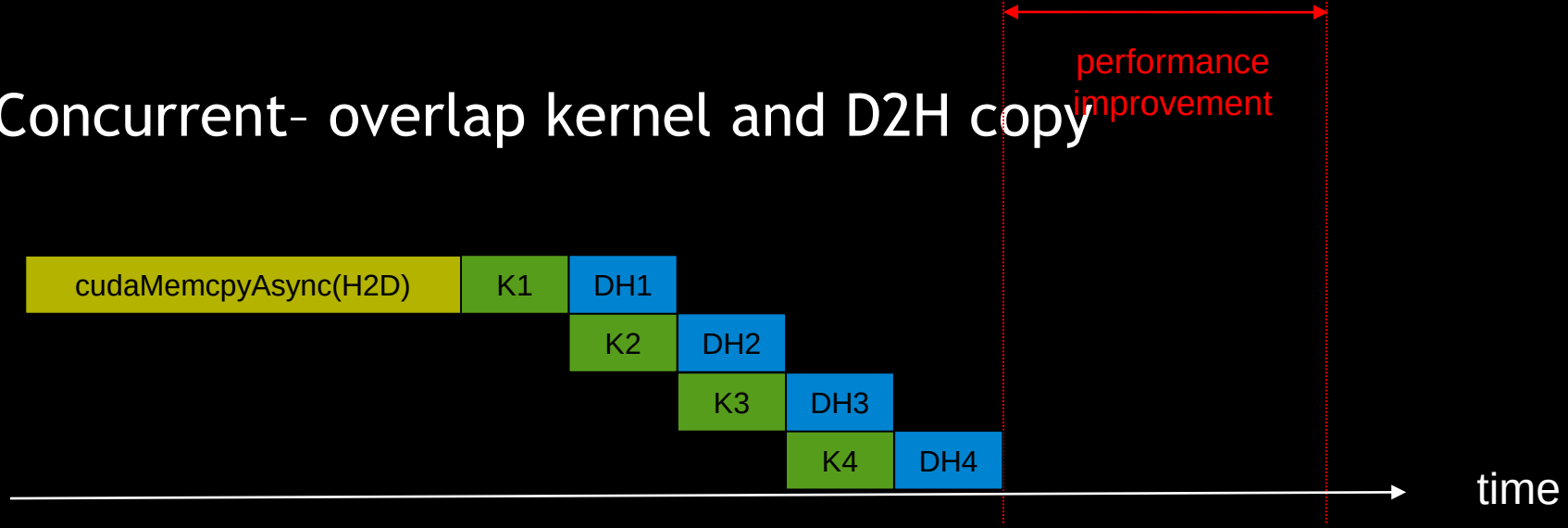


1. Copy input data from CPU memory to GPU memory
2. Launch a GPU Kernel
3. Copy results from GPU memory to CPU memory
4. Repeat Many Times

CONCURRENCY THROUGH PIPELINING

- Serial

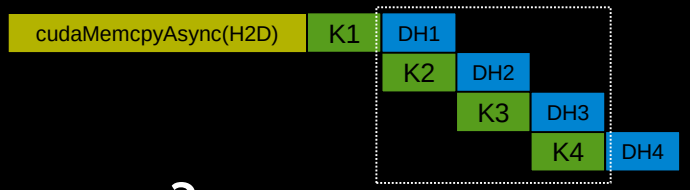
- Concurrent– overlap kernel and D2H copy



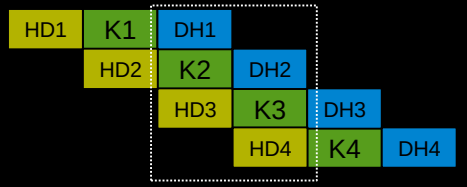
CONCURRENCY THROUGH PIPELINING

- Serial (1x)

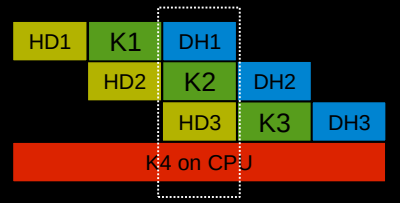
- 2-way concurrency (up to 2x)



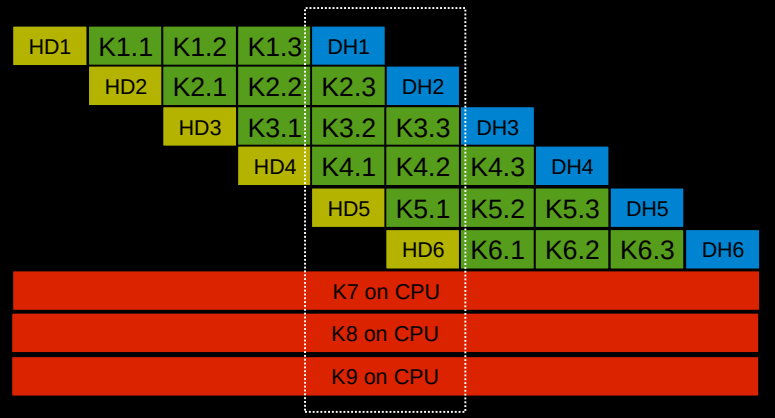
- 3-way concurrency (up to 3x)



- **4-way concurrency** (3x+)



- **4+ way concurrency**

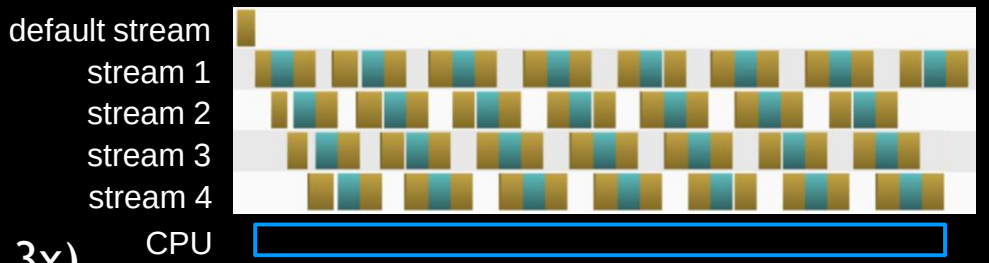


EXAMPLE – TILED DGEMM

- CPU (dual 6 core SandyBridge E5-2667 @2.9 Ghz, MKL)
  - 222 Gflop/s
- GPU (K20X)
  - Serial: 519 Gflop/s (2.3x)
  - 2-way: 663 Gflop/s (3x)
  - 3-way: 990 Gflop/s (4x)
- GPU + CPU
  - 4-way con.: 1180 Gflop/s (5.3x)

DGEMM: m=n=16384, k=1408

Nvidia Visual Profiler (nvvp)



- Obtain maximum performance by leveraging concurrency
- All PCI-E traffic is hidden
  - Effectively removes device memory size limitations!


Enabling Concurrency with Streams

SYNCHRONICITY IN CUDA

- All CUDA calls are either synchronous or asynchronous w.r.t the host
  - Synchronous: enqueue work and wait for completion
  - Asynchronous: enqueue work and return immediately
- Kernel Launches are asynchronous Automatic overlap with host

CUDA STREAMS

- A stream is a queue of device work
  - The host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- CUDA operations are placed within a stream
  - e.g. Kernel launches, memory copies
- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap

MANAGING STREAMS

- `cudaStream_t stream;`
  - Declares a stream handle
- `cudaStreamCreate(&stream);`
  - Allocates a stream
- `cudaStreamDestroy(stream);`
  - Deallocates a stream
  - Synchronizes host until work in stream has completed

PLACING WORK INTO A STREAM

- Stream is the 4th launch parameter
  - kernel<<< blocks , threads, smem, stream>>>();
- Stream is passed into some API calls
  - cudaMemcpyAsync(dst, src, size, dir, stream);

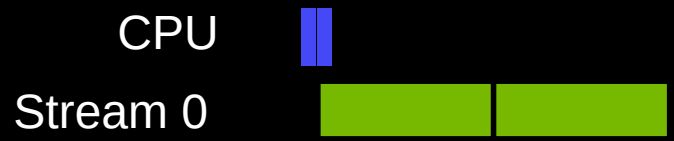
DEFAULT STREAM

- Unless otherwise specified all calls are placed into a default stream
  - Often referred to as “Stream 0”
- Stream 0 has special synchronization rules
  - Synchronous with all streams
    - Operations in stream 0 cannot overlap other streams
- Exception: Streams with non-blocking flag set
  - `cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking)`
  - Use to get concurrency with libraries out of your control (e.g. MPI)

KERNEL CONCURRENCY

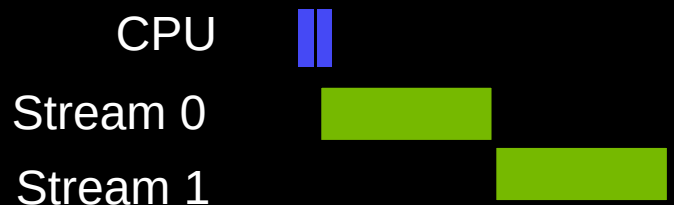
- Assume foo only utilizes 50% of the GPU
- Default stream

```
foo<<<blocks,threads>>>();
foo<<<blocks,threads>>>();
```



- Default & user streams

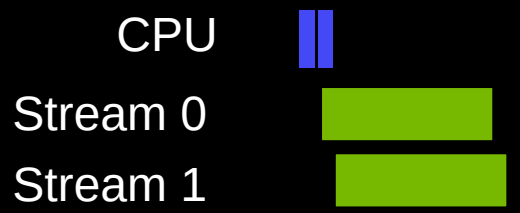
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
- Default & user streams

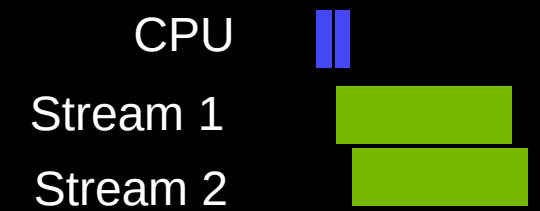
```
cudaStream_t stream1;
cudaStreamCreateWithFlags(&stream1,cudaStreamNonBlocking);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
  User streams

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
foo<<<blocks,threads,0,stream1>>>();
foo<<<blocks,threads,0,stream2>>>();
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



REVIEW

- The host is automatically asynchronous with kernel launches
- Use streams to control asynchronous behavior
  - Ordered within a stream (FIFO)
  - Unordered with other streams
  - Default stream is synchronous with all streams.


Concurrent Memory Copies

CONCURRENT MEMORY COPIES

- First we must review CUDA memory

THREE TYPES OF MEMORY

- Device Memory
  - Allocated using cudaMalloc
  - Cannot be paged
- Pageable Host Memory
  - Default allocation (e.g. malloc, calloc, new, etc)
  - Can be paged in and out by the OS
- Pinned (Page-Locked) Host Memory
  - Allocated using special allocators
  - Cannot be paged out by the OS

ALLOCATING PINNED MEMORY

- `cudaMallocHost(...) / cudaHostAlloc(...)`
  - Allocate/Free pinned memory on the host
  - Replaces malloc/free/new
- `cudaFreeHost(...)`
  - Frees memory allocated by cudaMallocHost or cudaHostAlloc
- `cudaHostRegister(...) / cudaHostUnregister(...)`
  - Pins/Unpins pagable memory (making it pinned memory)
  - Slow so don't do often
- Why pin memory?
  - Pagable memory is transferred using the host CPU
  - Pinned memory is transferred using the DMA engines
    - Frees the CPU for asynchronous execution
    - Achieves a higher percent of peak bandwidth

CONCURRENT MEMORY COPIES

- `cudaMemcpy(...)`
  - Places transfer into default stream
  - Synchronous: Must complete prior to returning
- `cudaMemcpyAsync(..., &stream)`
  - Places transfer into stream and returns immediately
- To achieve concurrency
  - Transfers must be in a non-default stream
  - Must use async copies
  - 1 transfer per direction at a time
  - Memory on the host must be pinned

PAGED MEMORY EXAMPLE

```
int *h_ptr, *d_ptr;

h_ptr=malloc(bytes);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

free(h_ptr);
cudaFree(d_ptr);
```

PINNED MEMORY: EXAMPLE 1

```
int *h_ptr, *d_ptr;

cudaMallocHost(&h_ptr,bytes);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

cudaFreeHost(h_ptr);
cudaFree(d_ptr);
```

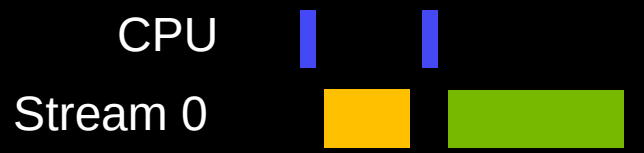
PINNED MEMORY: EXAMPLE 2

```
int *h_ptr, *d_ptr;

h_ptr=malloc(bytes);
cudaHostRegister(h_ptr,bytes,0);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

cudaHostUnregister(h_ptr);
free(h_ptr);
cudaFree(d_ptr);
```

CONCURRENCY EXAMPLES

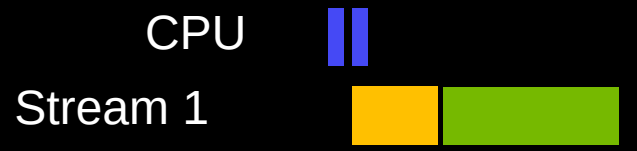
Synchronous

```
cudaMemcpy(...);
foo<<<...>>>();
```



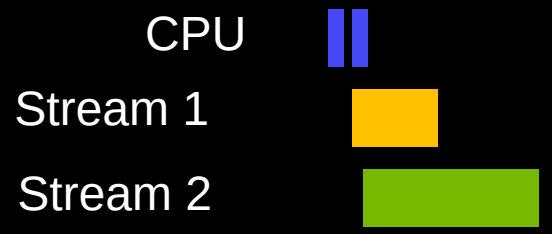
Asynchronous Same Stream

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream1>>>();
```



Asynchronous Different Streams

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream2>>>();
```



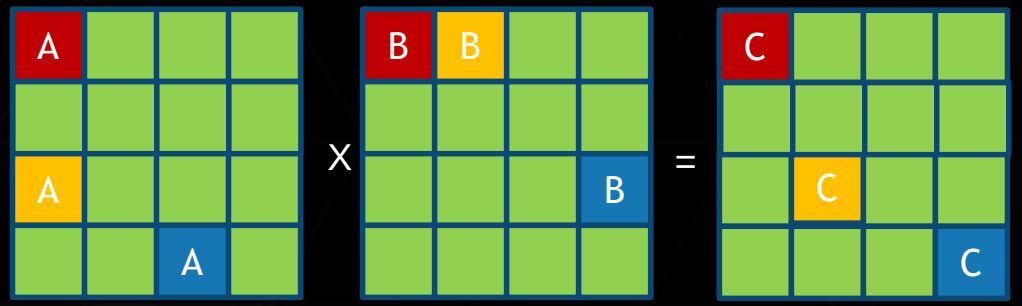
REVIEW

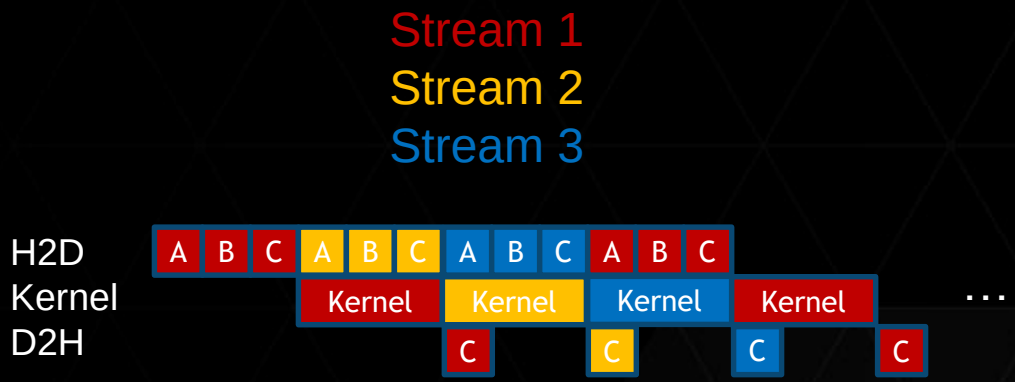
- Memory copies can execute concurrently if (and only if)
  - The memory copy is in a different non-default stream
  - The copy uses pinned memory on the host
  - The asynchronous API is called
  - There isn't another memory copy occurring in the same direction at the same time.


Tiled Matrix Multiply with Streams

TILED MATRIX MULTIPLY

- Handle arbitrarily large data
  - Full matrix never on the device





- GPU running at full capacity
  - Communication cost is hidden due to overlap

ALGORITHM OUTLINE

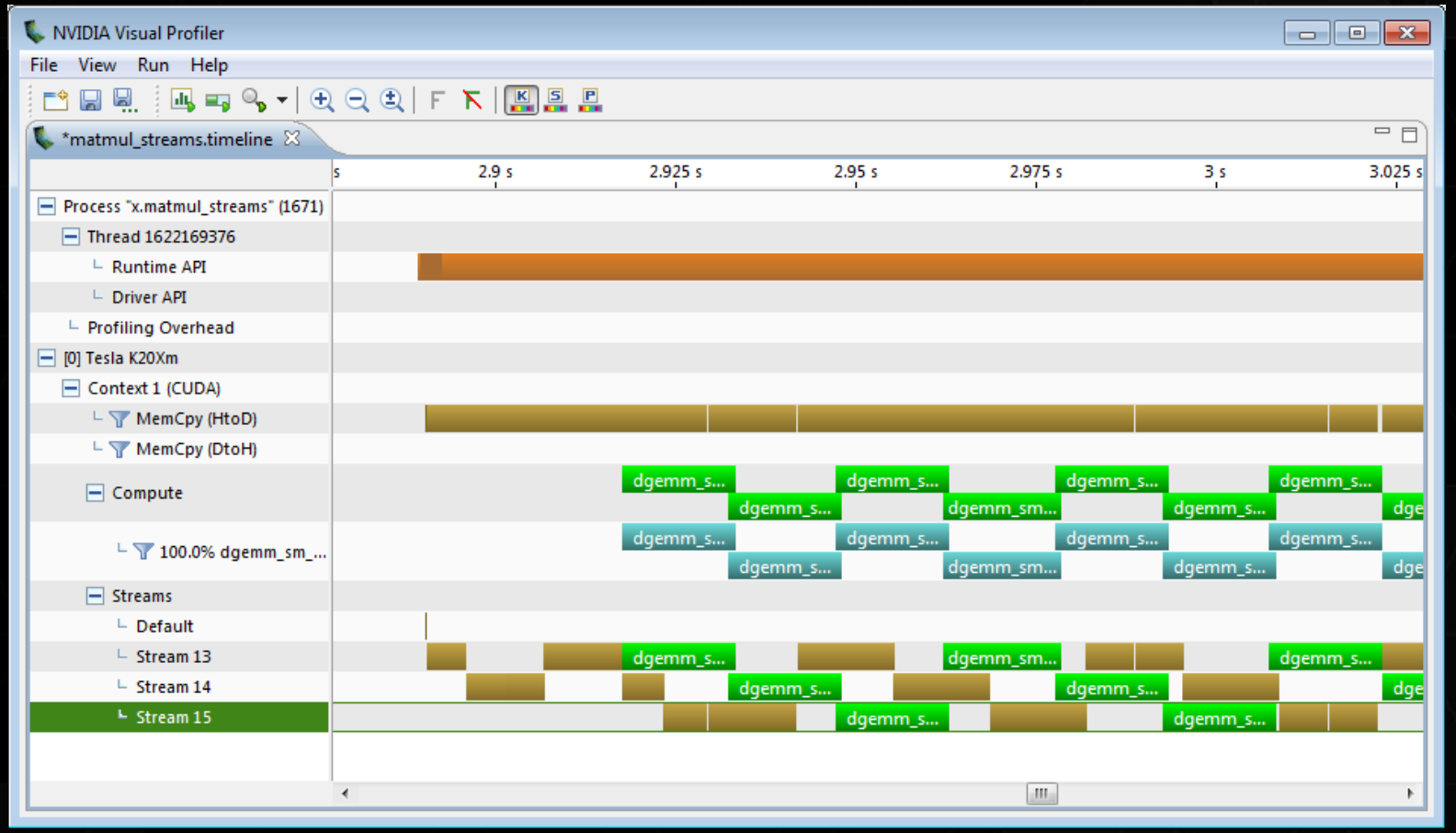
- Copy tiles of A/B/C into pinned memory buffers
- Loop over tiles of C
  - Copy tile of C to device
  - loop over K
    - Copy tiles of A and B to device
    - Call cublasDgemm and accumulate to tile of C
  - Copy C back to host

matmul_streams/kernel.cu

- Finish the kernel by inserting appropriate streams/cublas API calls
  - Use cheatsheet and/or docs.nvidia.com for syntax
- Answers (and perf) compared against single call of cublas
- Once you get correct answers try to find best combo of TILESIZE, SIZE and NUM_STREAMS
  - How much better perf is the streams code?
- If time permits, profile the app to ensure overlap
- CUDA_LAUNCH_BLOCKING=1 to enforce synchronous behavior

NVVP PROFILE

- K20X/CUDA6.5
  - N=8192
  - TILE=2048
  - Streams=3
  - 1061 GF
- Single dgemm
  - 548 GF



Synchronization