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 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

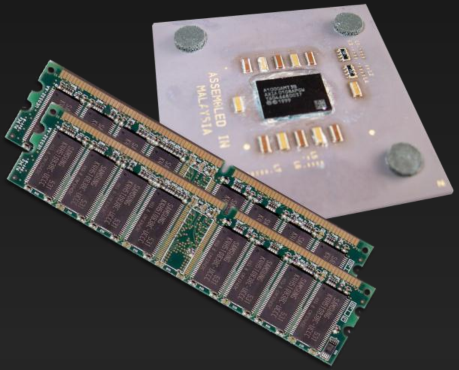
HELLO WORLD!

CONCEPTS

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

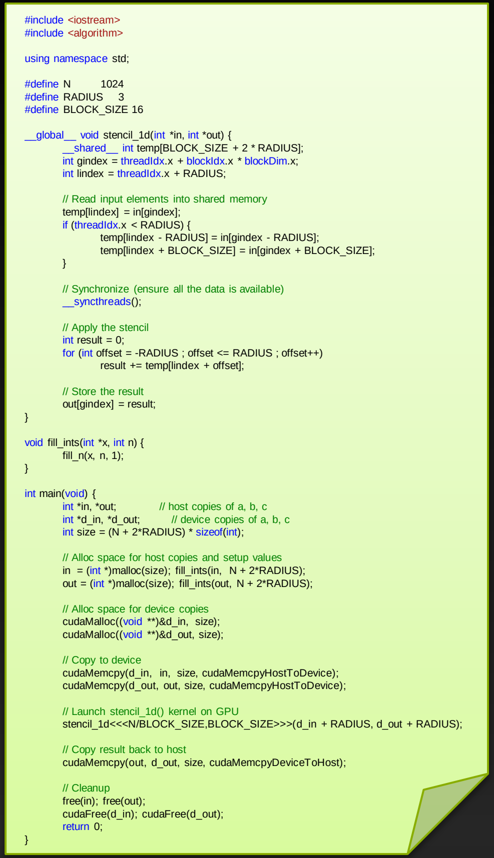
Heterogeneous Computing

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

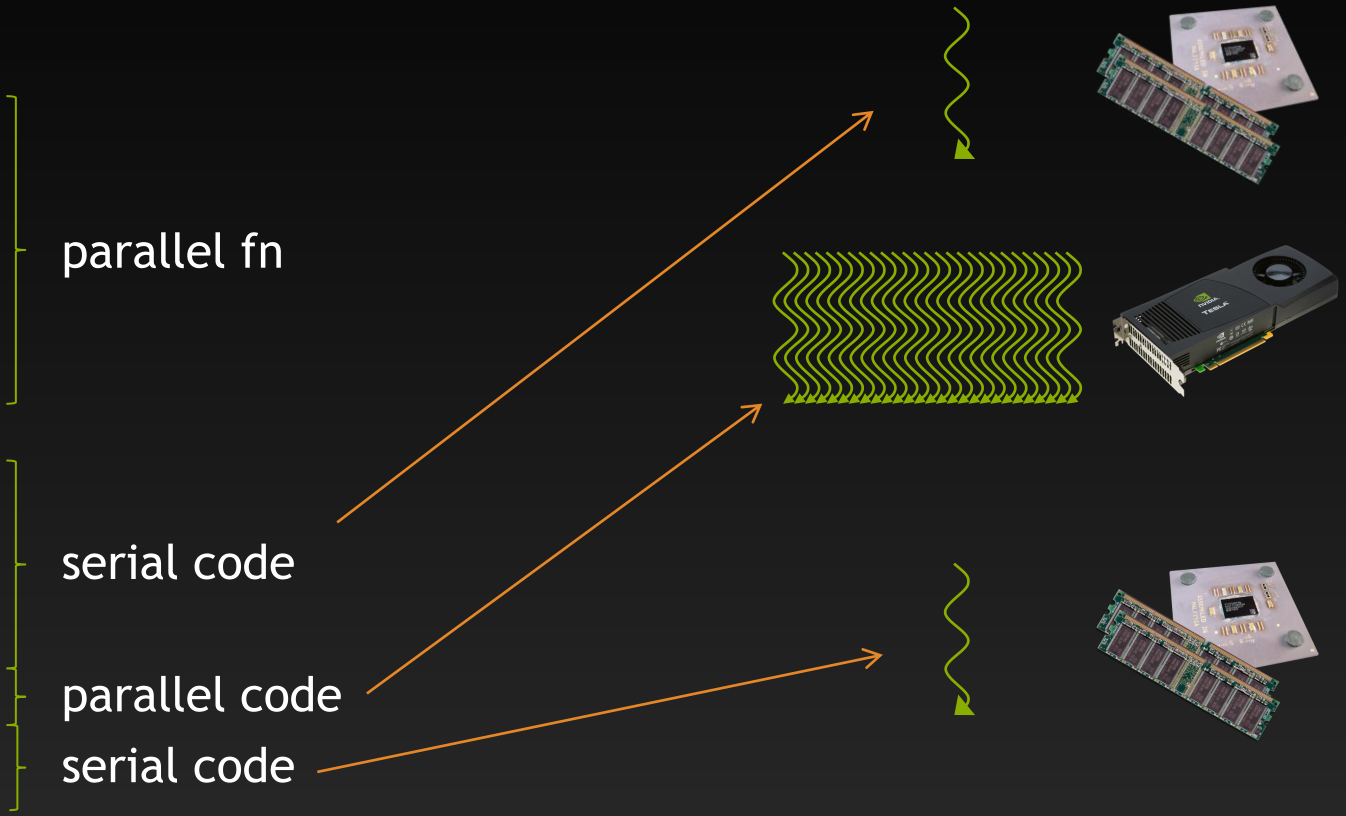

Host

Device

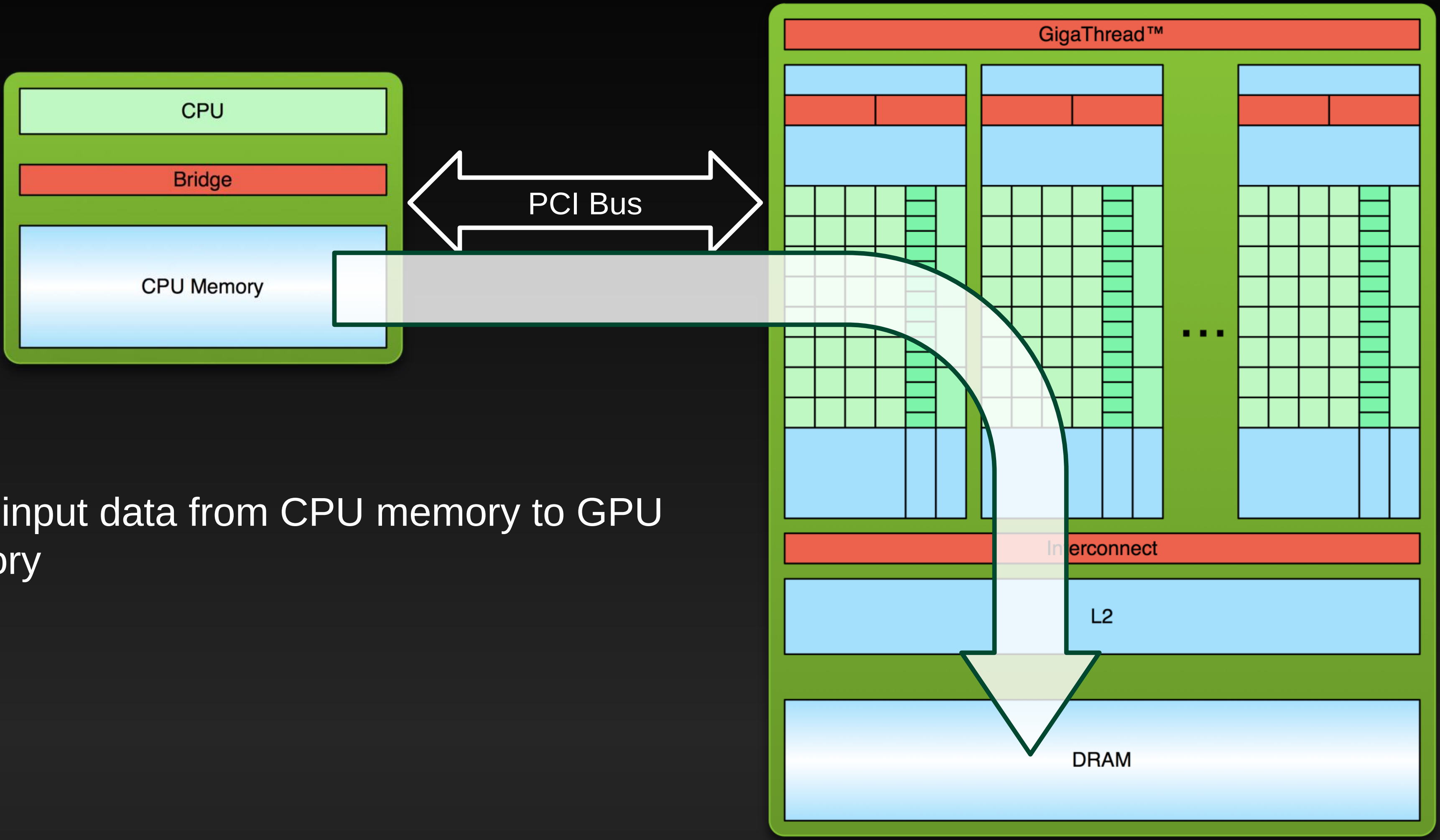
Heterogeneous Computing



```
#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;
}
```

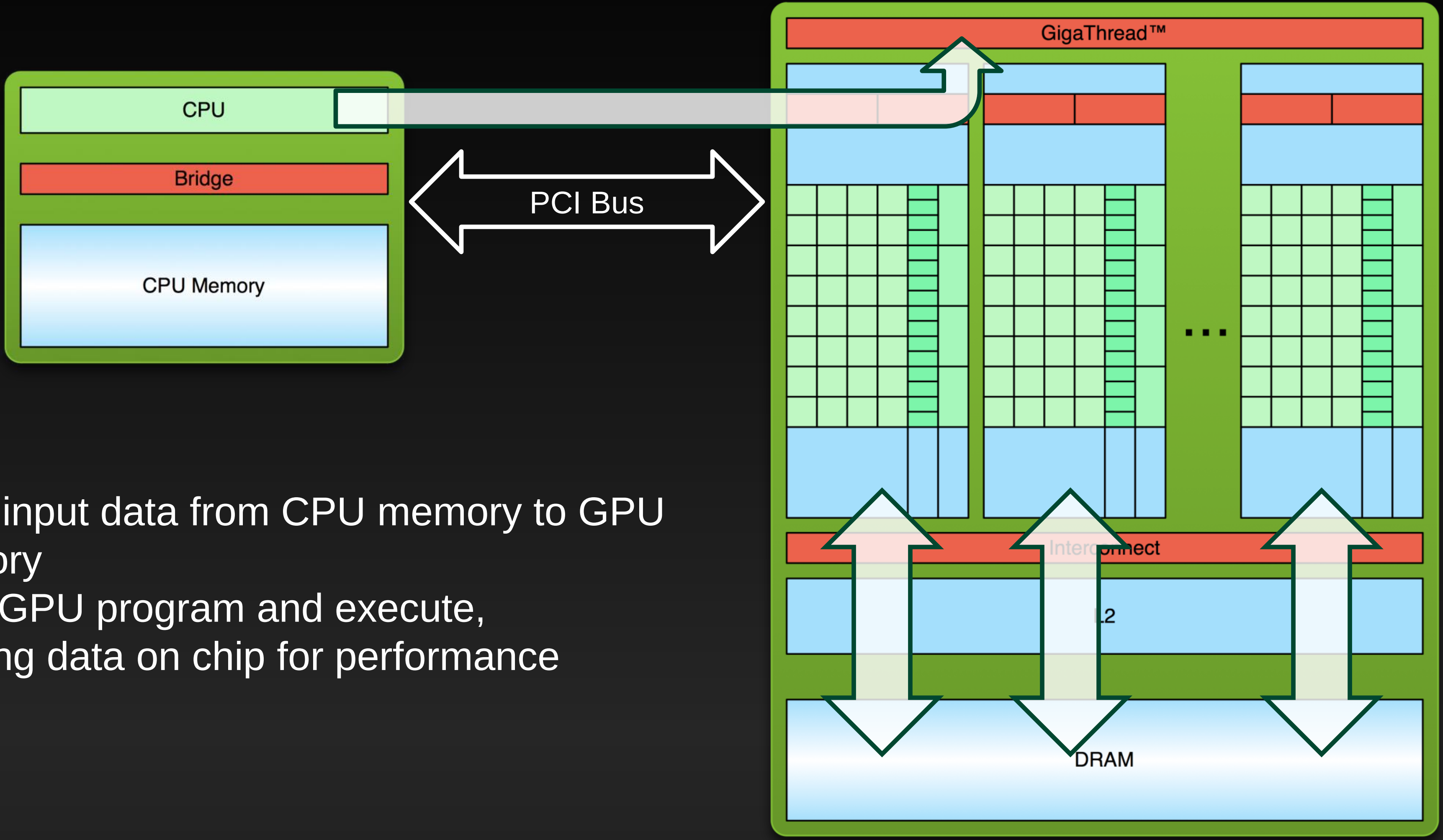


Simple Processing Flow



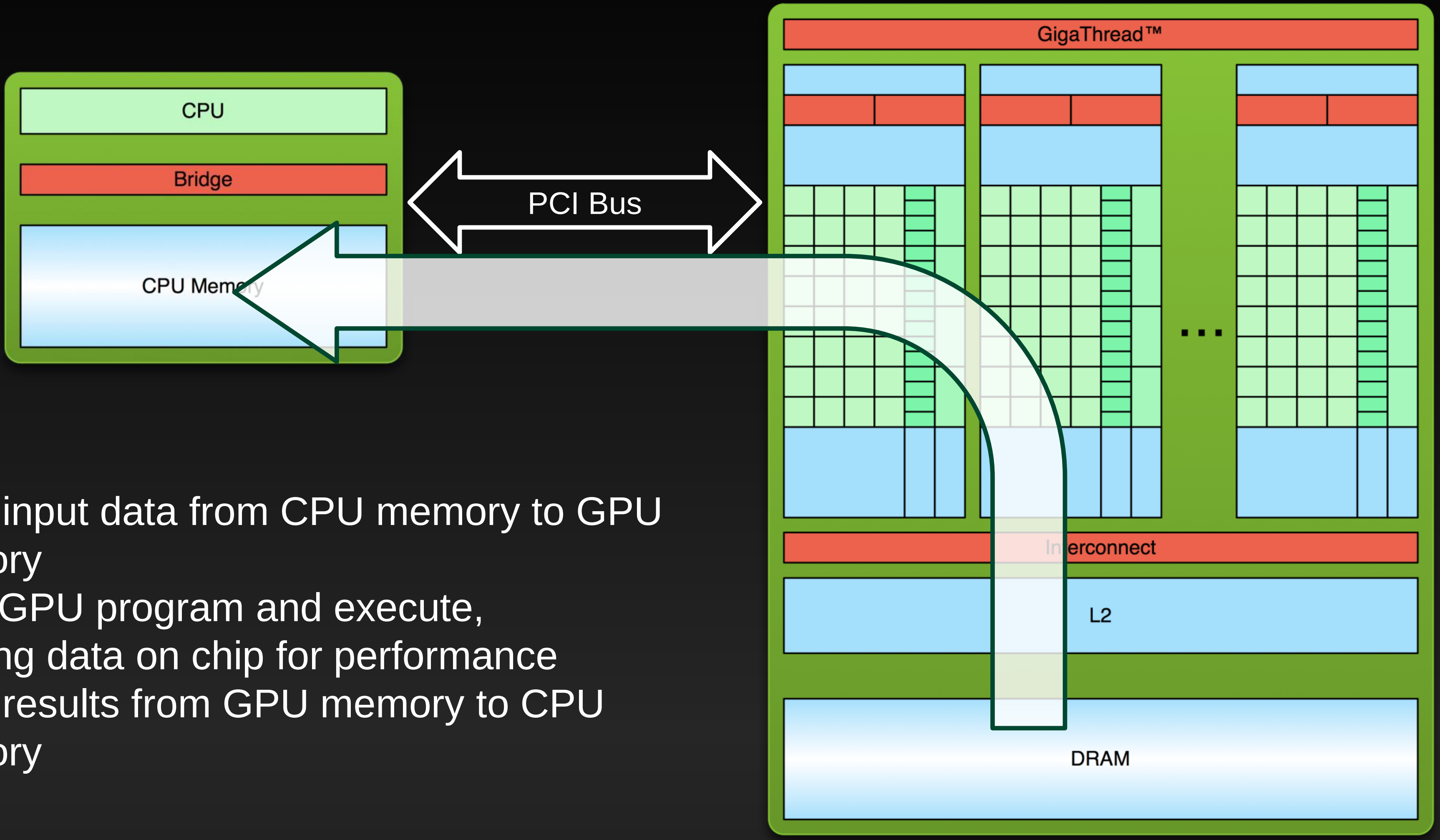
1. Copy input data from CPU memory to GPU memory

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) {
}

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>>>();
```

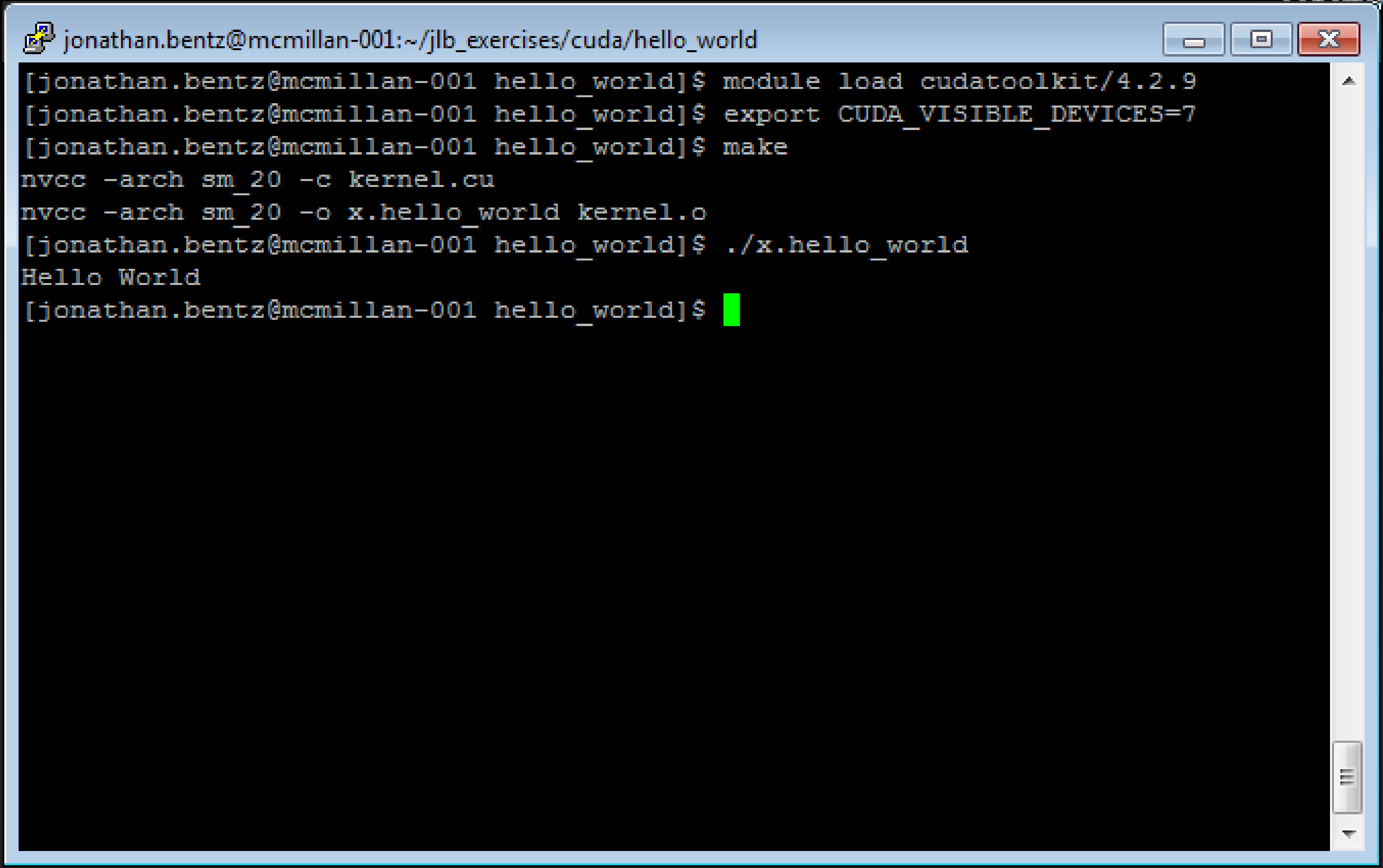
- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

Setup your environment/Hello world

- Login to mcmillan.princeton.edu with your guest account
- module load cudatoolkit/4.2.9
- export CUDA_VISIBLE_DEVICES=<number>
  - I will assign you this number so that we utilize all the GPUs evenly
- `cp -r ~jonathan.bentz/Princeton .`
- Each coding project in a separate folder
- `cd exercises/cuda/hello_world`
- “make” to build the code
- Try building/running the hello_world solution.

Screenshot


jonathan.bentz@mcmillan-001:~/jlb_exercises/cuda/hello_world

```
[jonathan.bentz@mcmillan-001 hello_world]$ module load cudatoolkit/4.2.9
[jonathan.bentz@mcmillan-001 hello_world]$ export CUDA_VISIBLE_DEVICES=7
[jonathan.bentz@mcmillan-001 hello_world]$ make
nvcc -arch sm_20 -c kernel.cu
nvcc -arch sm_20 -o x.hello_world kernel.o
[jonathan.bentz@mcmillan-001 hello_world]$ ./x.hello_world
Hello World
[jonathan.bentz@mcmillan-001 hello_world]$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

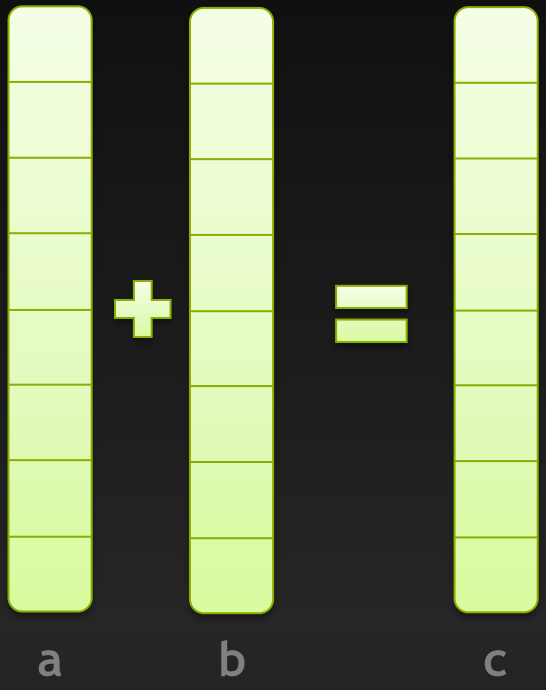
Output:

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

- `mykernel()` does nothing, 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

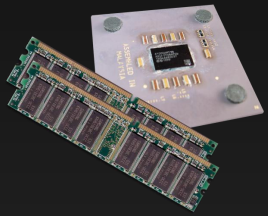
```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- `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()`

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, wherever you see “FIXME”
  - Comments tell you what to do.
  - If something isn’t clear, PLEASE ASK! ☺

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

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`

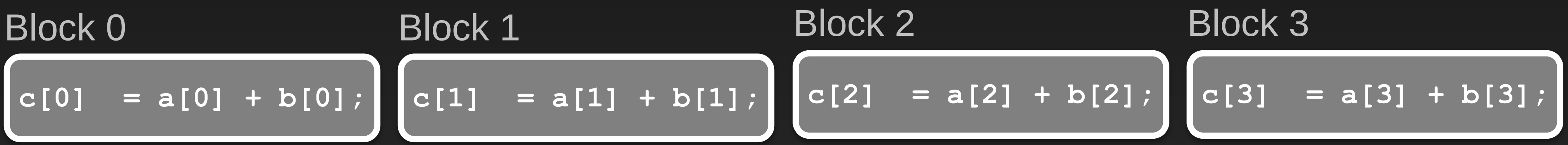
```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- 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:



```
c[0]   = a[0] + b[0];
c[1]   = a[1] + b[1];
c[2]   = a[2] + b[2];
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()…
- 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;
}
```

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

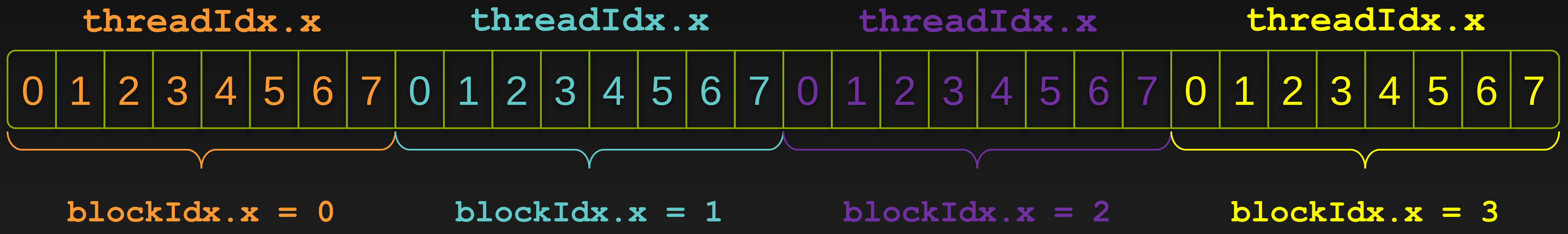
- Let's adapt vector addition to use both *blocks* and *threads*

- 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)

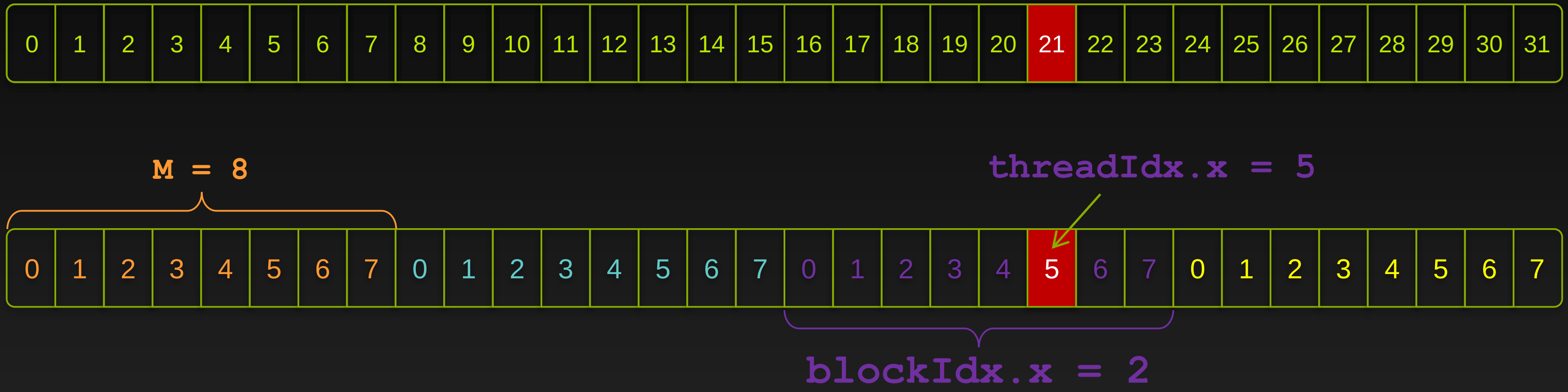


- With M threads/block a unique index for each thread is given by:

```
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 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,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;
```

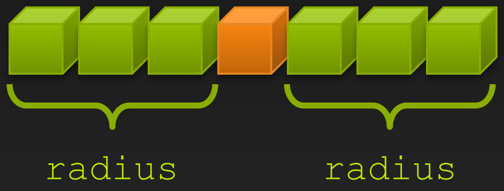
CONCEPTS

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

COOPERATING THREADS

1D Stencil

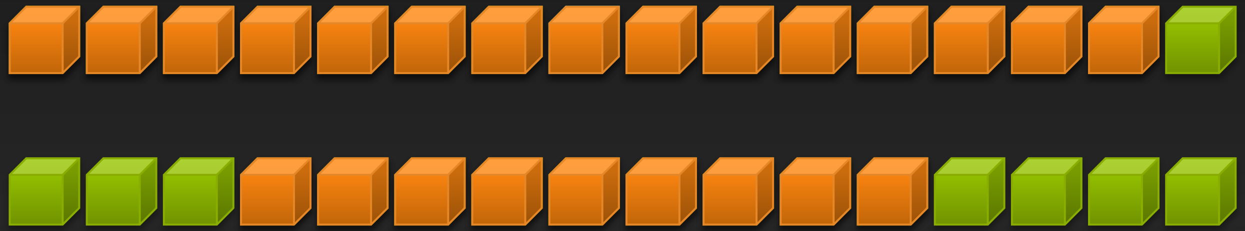
- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



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



Simple Stencil in 1d

- Open cuda/simple_stencil/kernel.cu
- Finish the kernel and the kernel launch
  - Each thread calculates one stencil value
  - Reads 2*RADIUS + 1 values

- Inserted GPU timers into code to time the execution of the kernel

- Try various sizes of N, RADIUS, BLOCK
- Time a large (over a million) 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?

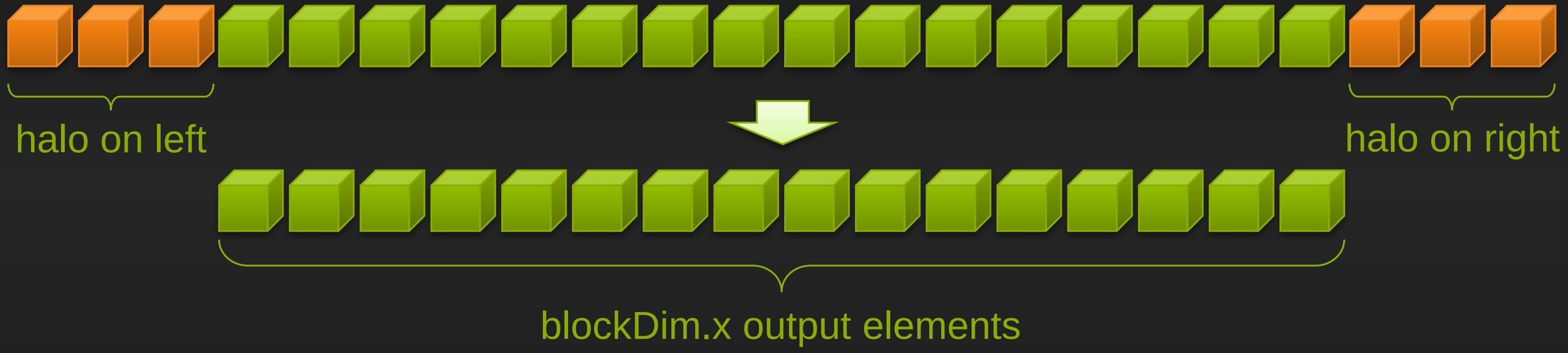
Sharing Data Between Threads

- 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

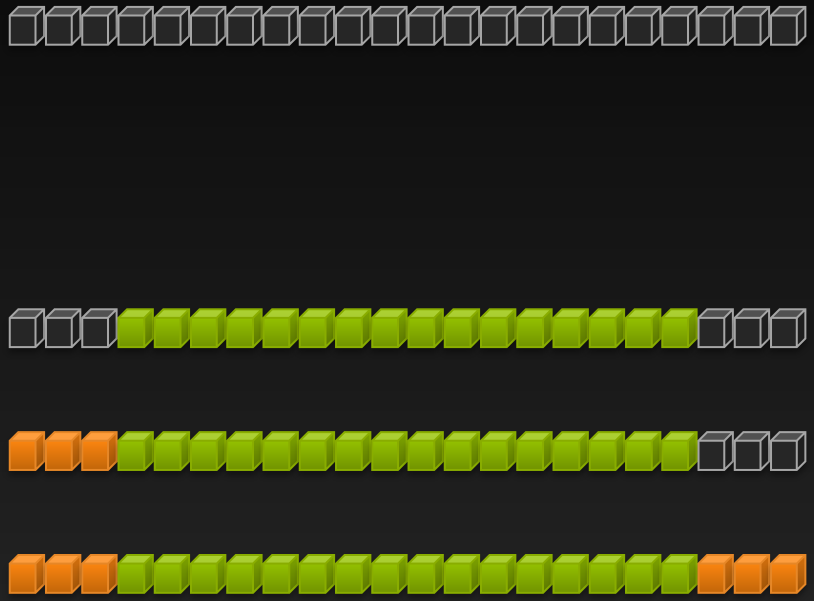
- Cache data in shared memory (user managed scratch-pad)
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory

- 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

- Open cuda/simple_stencil_smem/kernel.cu
- Run the code. It will build/run without modification.
  - If Errors occur, each offending element will be printed to the screen
- What is the result with N=10,000 and BLOCK=32?
- What is the result with N=10,000 and 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

CONCEPTS

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

MANAGING THE DEVICE

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()`

Compute Capability

- The compute capability of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

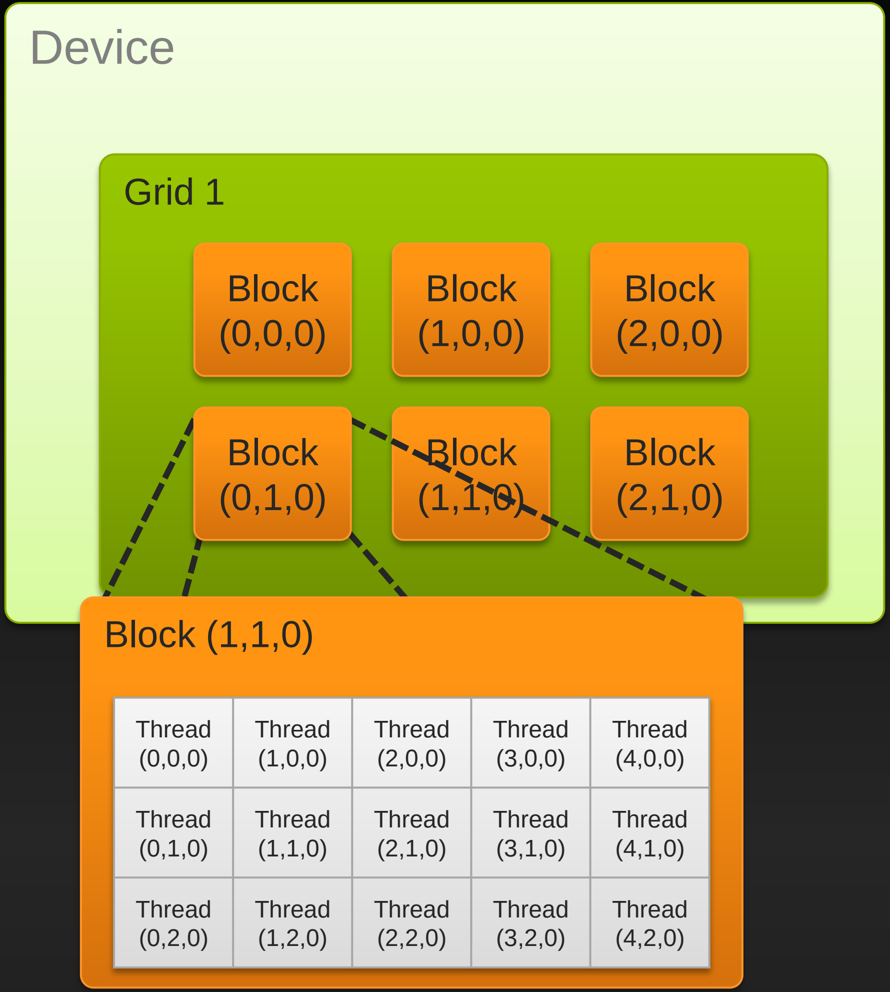
Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

- The following presentations concentrate on Fermi devices
  - Compute Capability >= 2.0

IDs and Dimensions

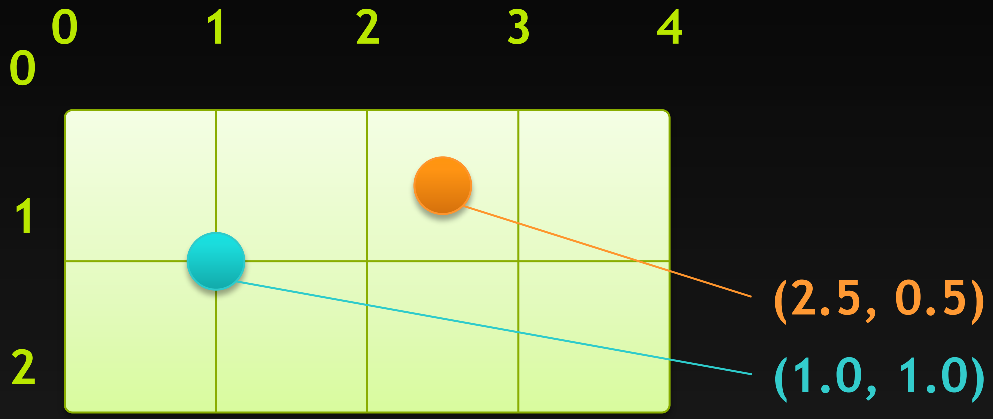
- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - We showed only one dimension (`x`)

- Built-in variables:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



Textures

- Read-only object
  - Dedicated cache
- Dedicated filtering hardware
  (Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling
  (Wrap, clamp)



Topics we skipped

- We skipped some details, you can learn more:
  - CUDA Programming Guide
  - CUDA Zone - tools, training, webinars and more
    http://developer.nvidia.com/cuda
- Need a quick primer for later:
  - Multi-dimensional indexing
  - Textures

Questions?

Global Memory Throughput

Fermi 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

GMEM Optimization Guidelines

- Strive for perfect coalescing
  - Align starting address (may require padding)
  - A warp should access within a contiguous region
- 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)
- Try L1 and caching configurations to see which one works best
  - Caching vs non-caching loads (compiler option)
  - 16KB vs 48KB L1 (CUDA call)

Shared Memory

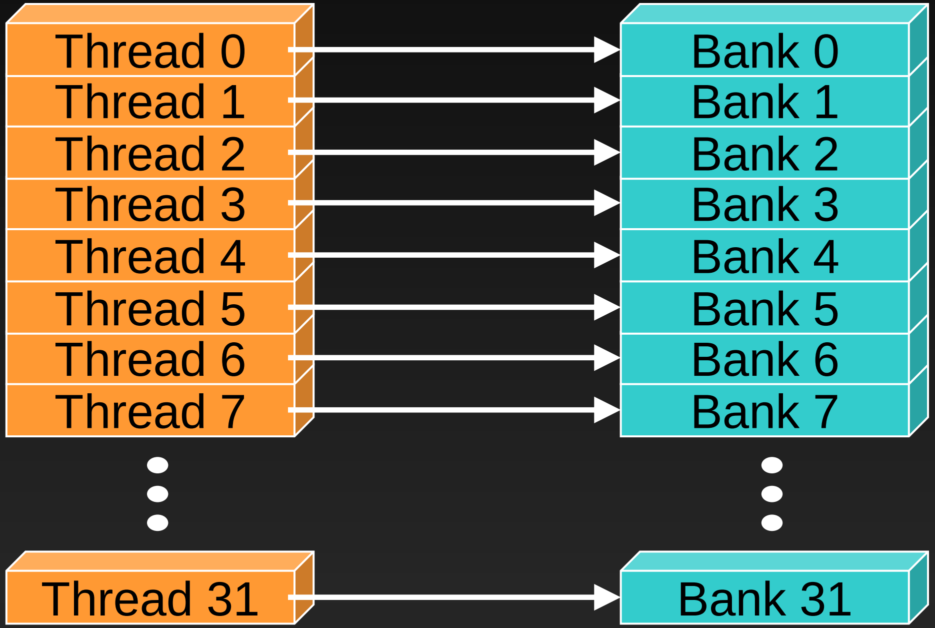
Shared Memory

- 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

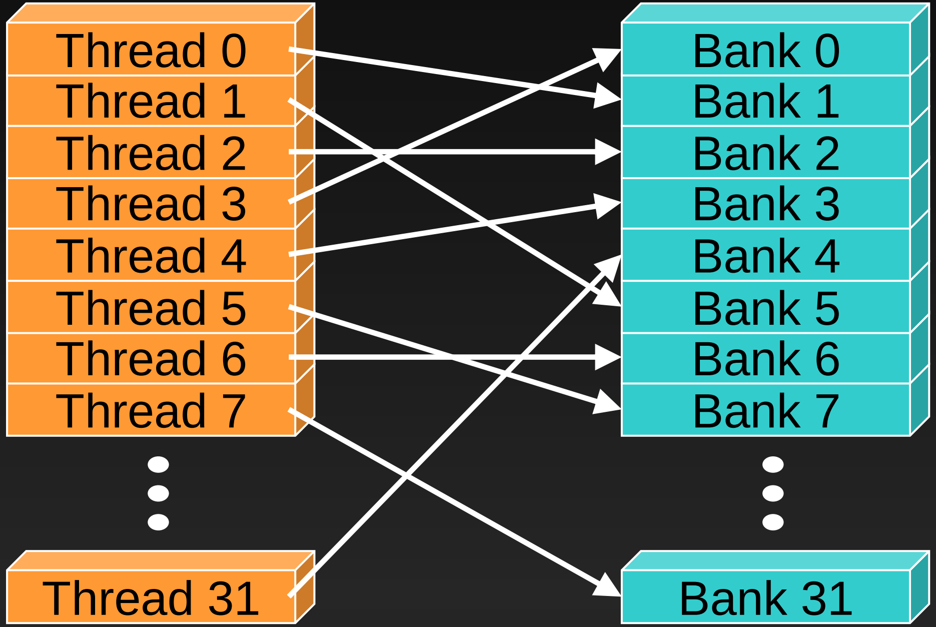
- If you use shared memory in a kernel, you should almost always use `__syncthreads()` to avoid race conditions!!!

Bank Addressing Examples

- No Bank Conflicts

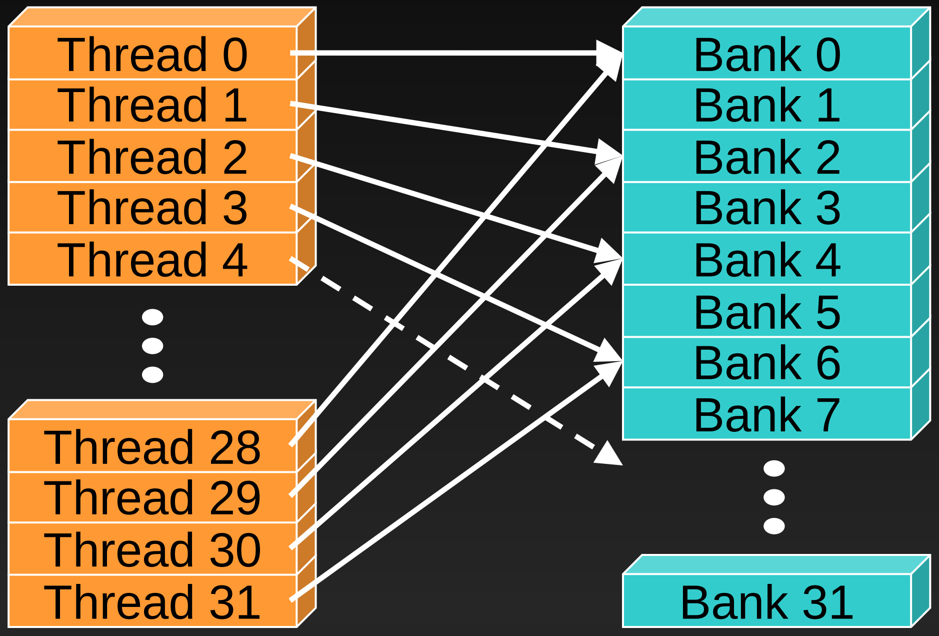


- No Bank Conflicts

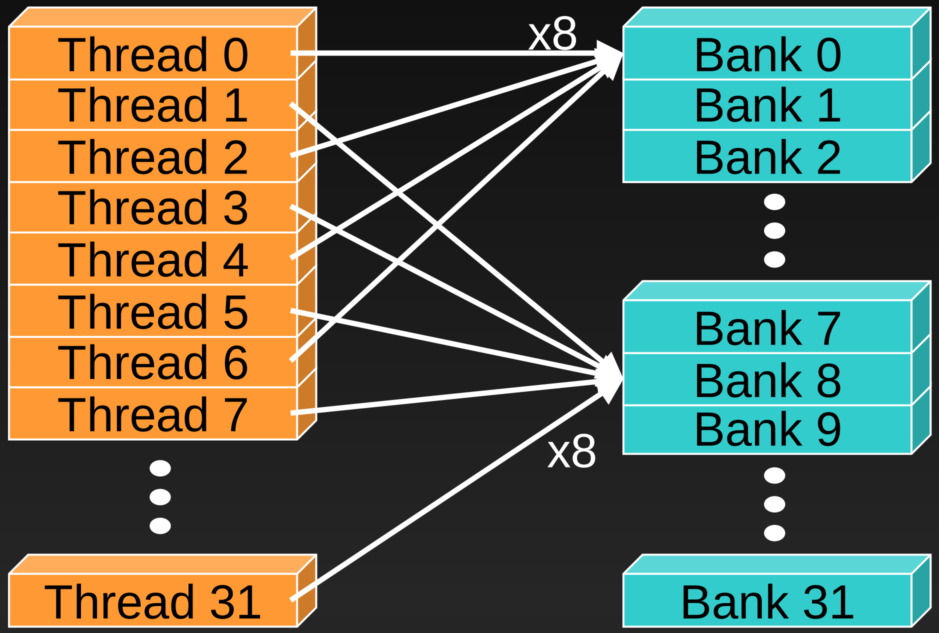


Bank Addressing Examples

- 2-way Bank Conflicts

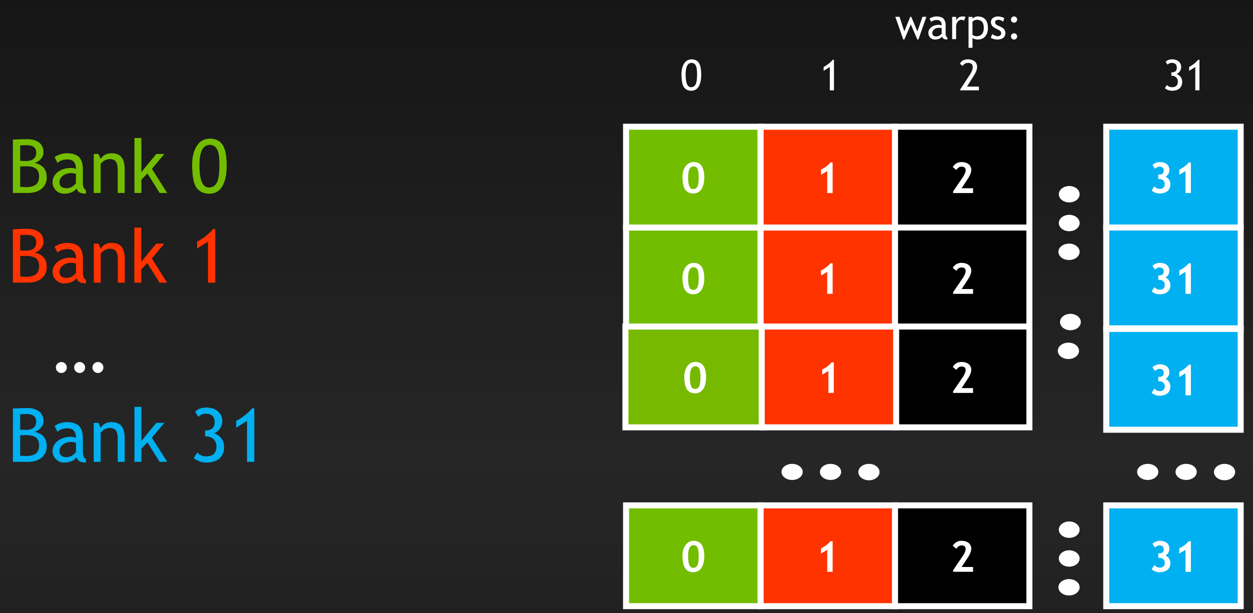


- 8-way Bank Conflicts



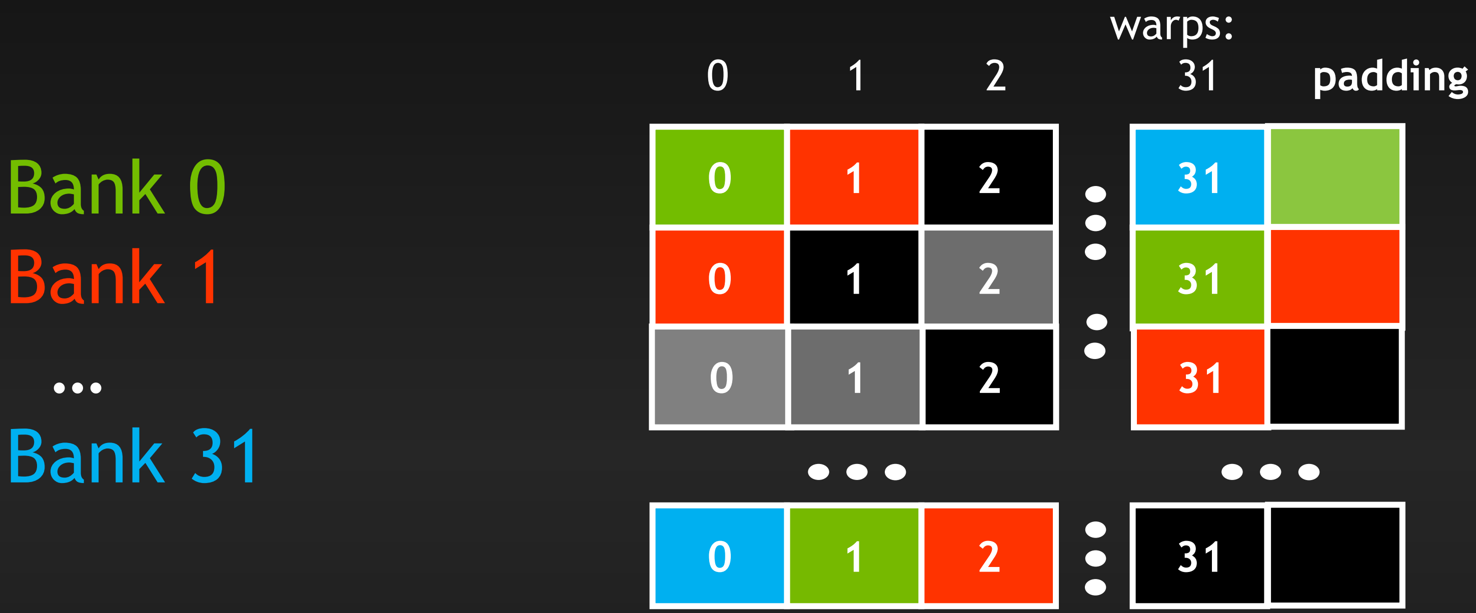
Shared Memory: Avoiding Bank Conflicts

- 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)



Shared Memory: Avoiding Bank Conflicts

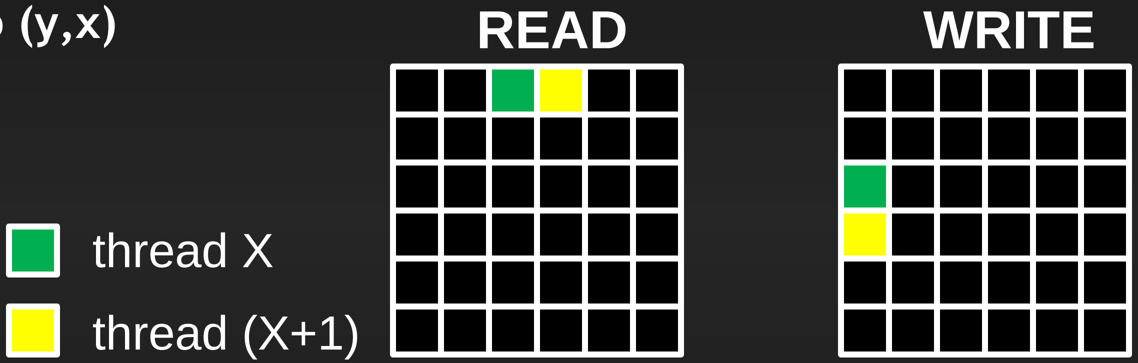
- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts



Matrix Transpose

Matrix Transpose

- Open exercises/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

- 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
- 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 windows, 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

Matrix multiply

Matrix Multiply

- The foundation of lots of linear algebra
- High compute/communication ratio
  - Access O(N^2) data and execute O(N^3) operations
- Relatively simple algorithm
  - Great teaching algorithm
- Well-written code shows off the power of CPU and/or GPU

Matrix Multiply cont'd

- Matrix A with M rows and K cols
- Matrix B with K rows and N cols
- A * B = C
  - C has M rows and N cols
  - $C_{i,j} = \sum_{k}^{K} A_{i,k} * B_{k,j}$
- The dot product of the *ith* row of A and *jth* col of B yields the *i,j* element of C

matmul_CPU

- Open matmul_CPU/kernel.cu

- CPU-only example
- Complete “host_dgemm” function
  - Finish the index calculations for the arithmetic in inner loop.

- What is the performance of your naïve CPU matrix multiply?
- DGEMM is often measured in terms of percentage of peak.
  - Intel X5690@3.47GHz has peak of 3.47GHz * 4 DP flops/clock = 13.88GF
  - What is your code’s percent of peak?

Matmul on GPU

- In reality we'd never write a matmul for GPU
  - Call NVIDIA's CUBLAS library
- Open matmul_CUBLAS/kernel.cu
  - Code should run without modification.
- Run the code with N=1024 and record performance of CUBLAS dgemm
  - How does it compare to your naïve CPU code?

Matmul on GPU

- We will write matrix multiply on GPU
- Use square matrices for simplicity
- Use powers of 2 so we can avoid writing the extra code for end cases
- Write three versions utilizing successively advanced optimization ideas, based in part on profiling results we obtain

matmul_GPU_naive

- Open matmul_GPU_naive/kernel.cu
- Finish the kernel
  - Add the appropriate index calculations to the kernel
  - Answers are compared against CUBLAS
    - You should see an error message printed to the console if your results are suspect!
- Record performance of N=1024
  - How does it compare to CUBLAS?

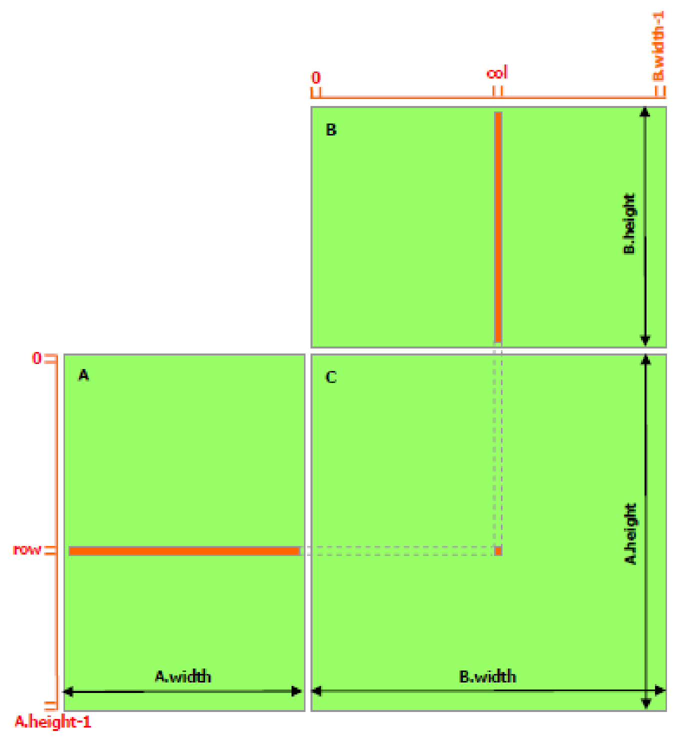


Figure 3-1. Matrix Multiplication without Shared Memory

Visual profiler

- Profile the code with Visual Profiler
- What are some performance considerations?
- Consider two successive threads
  - How is global memory accessed for matrices A, B, C?
- What is a choice we have if we wish to remedy uncoalesced GMEM accesses?

Strategies to improve

- Use shared memory to achieve better coalescing
    - Allows us to share data among threads
    - Reduces number of times data must be read because we reuse from smem rather than fetch from GMEM each time.
- Similar to the matrix transpose example:
    - Load block of A into SMEM
    - Load block of B into SMEM
    - Compute block of C from A and B in SMEM

- Open matmul_GPU_shmem/kernel.cu
    - `#pragma unroll` to unroll loops of predefined trip count

matmul_GPU_shmem

- For each block in K direction
  - Load block of A into SMEM
  - Load block of B into SMEM
  - Use these blocks to contribute to block of C
- When using SMEM, what function should you include???
- Record performance of N=1024
  - How does it compare to naïve kernel?
  - Why???

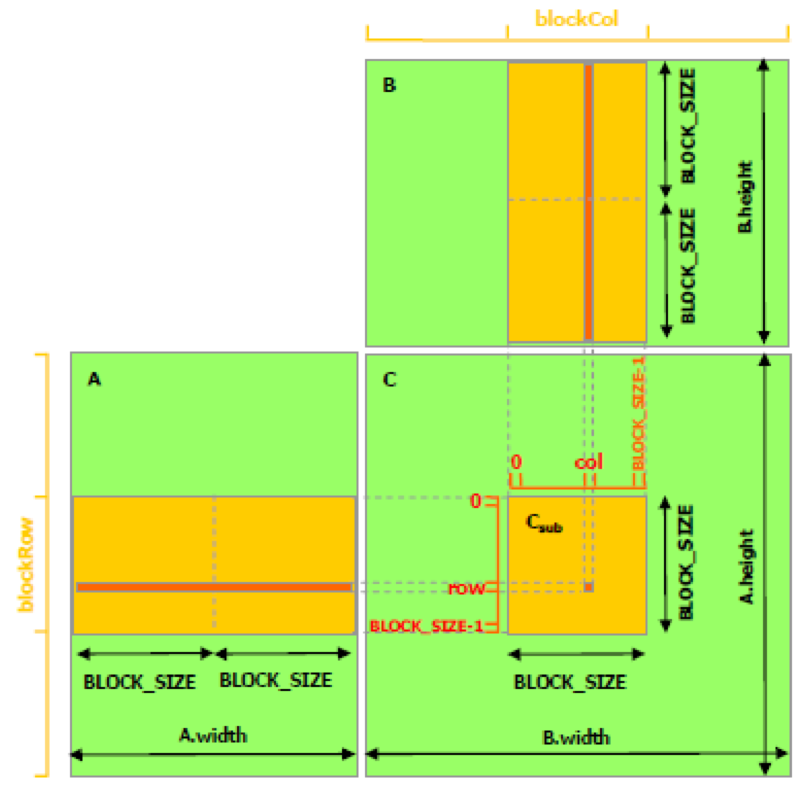


Figure 3-2. Matrix Multiplication with Shared Memory

Visual profiler

- We solved the uncoalesced memory issue.
- What does profiler show us?
- What is the fix for this?
- What is the performance with this change?

Algorithmic improvements

- Currently one thread block calculates one block of C
  - By extension one thread only calculates one element of C
  - A natural extension is to try having one thread block calculate multiple portions of C and thus one thread calculating multiple values of C
  - This would reduce the number of times A and B are fetched from GMEM and increase the computational intensity of the thread block
  - A priori not obvious how many blocks C should calculate
- Write a new kernel that does this idea in a general fashion
  - Experiment with different block sizes etc.
- MAGMA from UTK http://icl.cs.utk.edu/magma/ has done extensive work on dense linear algebra

matmul_GPU_shmem1

- Open matmul_gpu_shmem1/kernel.cu
- Lots of `#define` at the top of the source code
  - Pay special attention to the defined constants and how they interact with each other!
  - Keep TX=TY=BK=16
  - TBX=TBY=16 and NX=NY=1 is equivalent to the previous kernel.
- Complete the kernel with some important reminders
  - What function should you automatically use with SMEM???
  - How can you avoid those nasty bank conflicts in SMEM???
  - Refer to diagram in cheatsheet for pictorial view of algorithm
  - Lots of code to fill in. This one is challenging!!!

http://www.netlib.org/lapack/lawnspdf/lawn227.pdf

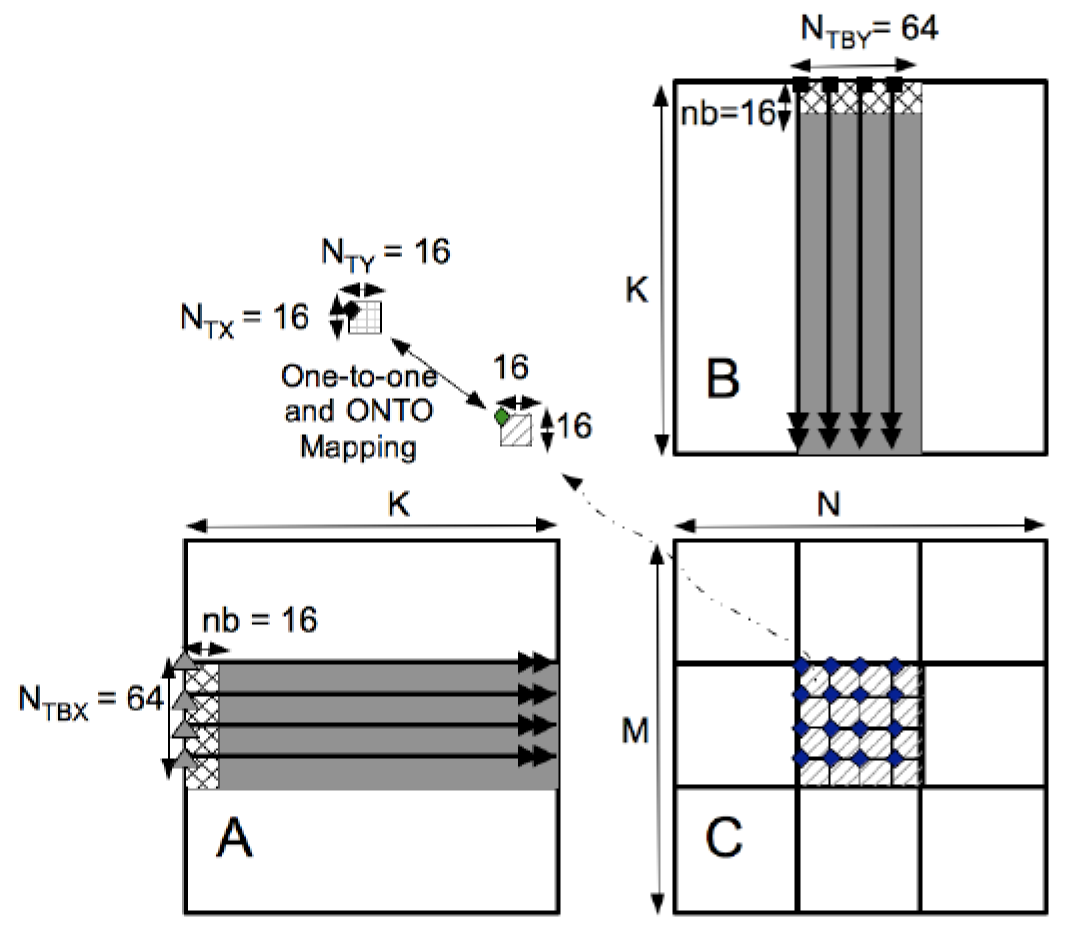


Fig. 2. The GPU GEMM ($C := \alpha AB + \beta C$) of a single TB for Fermi.

matmul_GPU_shmem1

- Keep N=1024
  - Try different values of TBX, TBY, NX, NY
  - What is the best combination?
  - Are there multiple “best” values?
- With the best combination you found, record performance for N=1024
  - How does it differ with padded or unpadded SMEM?
  - How does it compare to the CUBLAS result we saw earlier?

Review

- Compute intensity (arithmetic ops/memory reference) on a per-thread basis impacts performance
- Avoiding SMEM bank conflicts is critical
- Visual Profiler is helpful to identify performance issues

Final Wrap-up

- Optimizations we looked at
  - Coalesced global memory accesses
  - Shared memory usage
    - __syncthreads()
    - Padding to avoid bank conflicts
  - Appropriate arithmetic intensity
- Things we didn’t examine
  - Optimizations of host->device data transfer
    - Overlap of communication/computation
  - Texture memory usage
  - Multi-GPU programming

Questions?