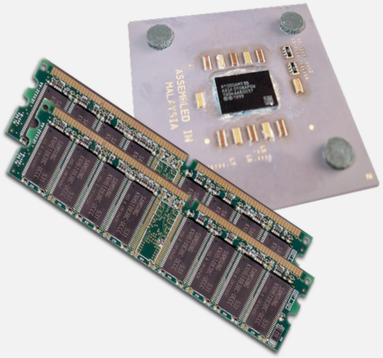
Why GPU Computing

after: https://developer.nvidia.com/cuda-education

Heterogeneous Computing

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

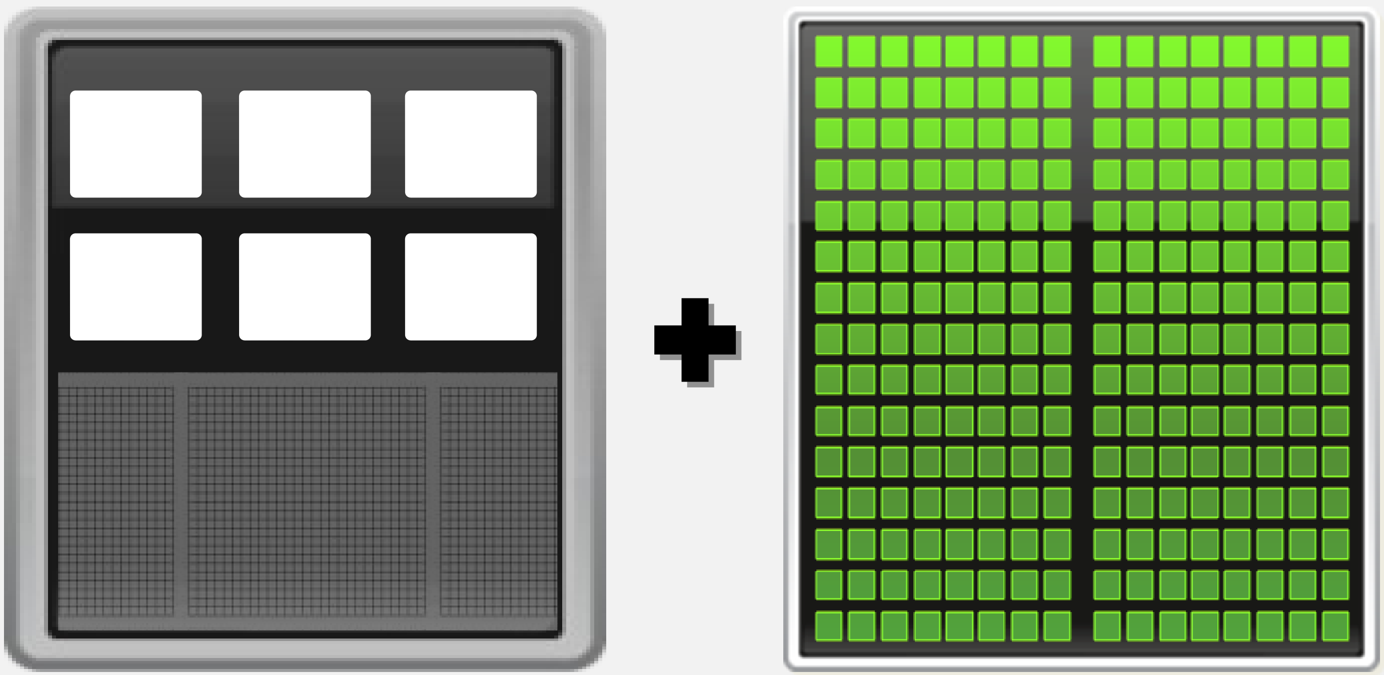


Host

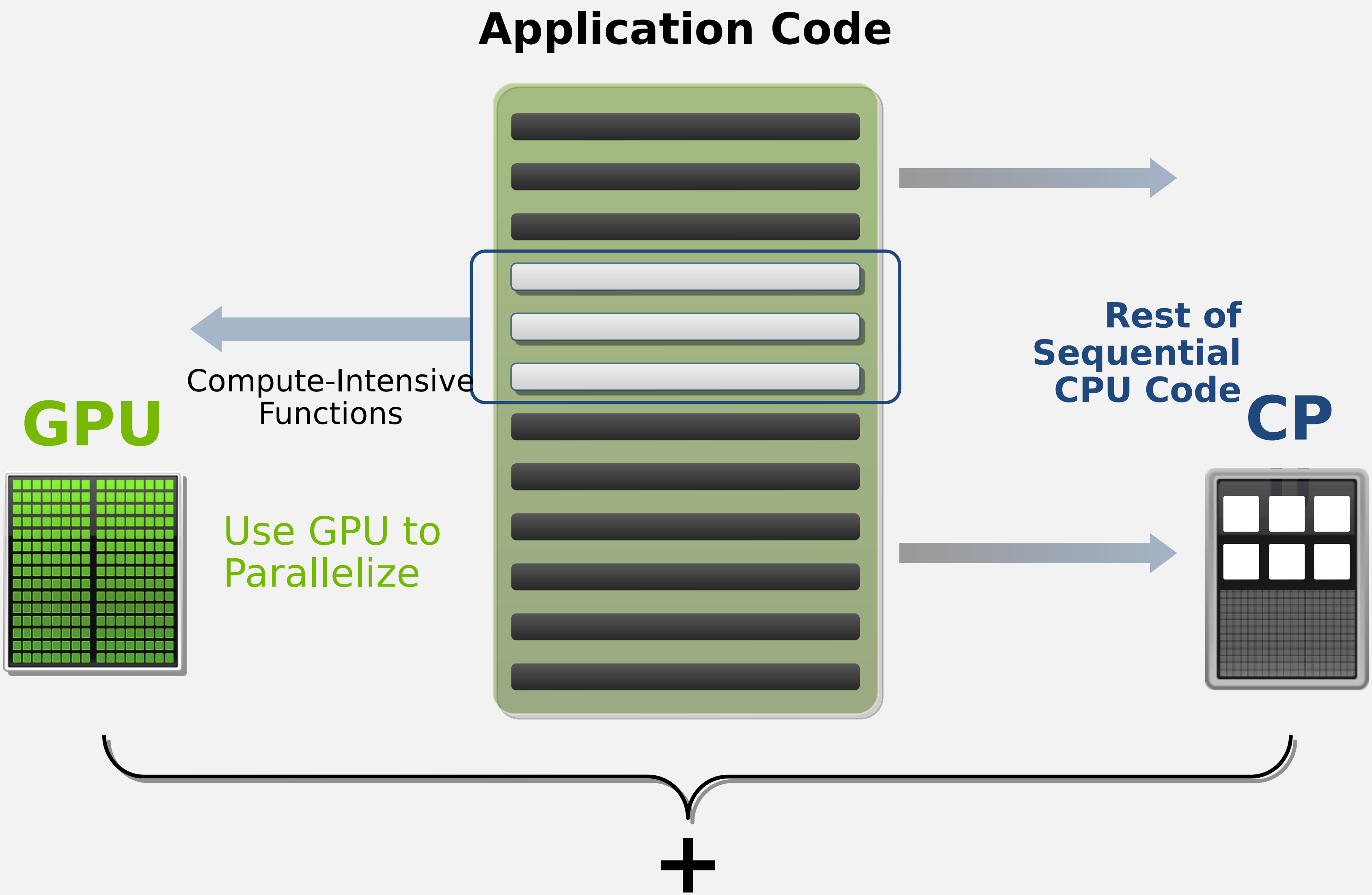
Device

Add GPUs: Accelerate Science Applications

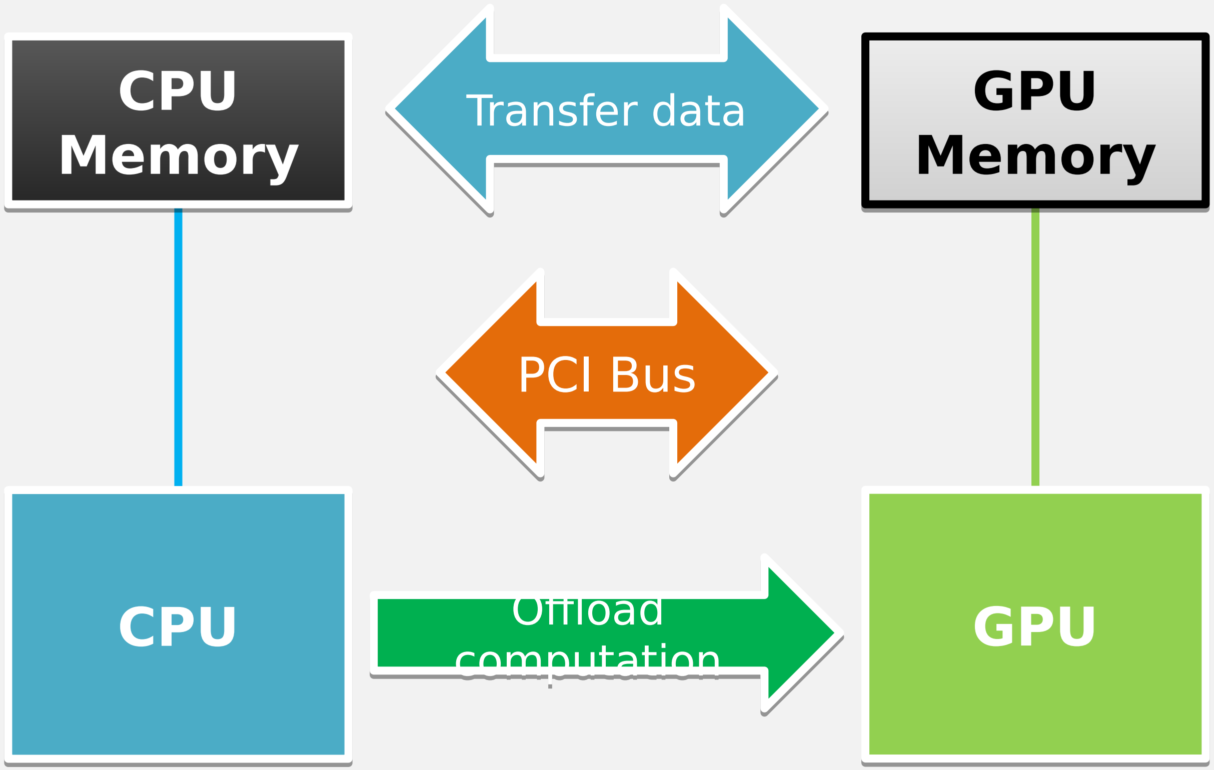
CPU **GPU**



Small Changes, Big Speed-up



Basic Concepts



For efficiency, decouple data movement and compute off-load

Introduction to the CUDA Platform

CUDA Parallel Computing Platform

www.nvidia.com/getcuda

Programming Approaches

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	**Easily Accelerate Apps**	**Maximum Flexibility**

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

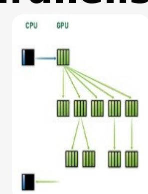
Open Compiler Tool Chain

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

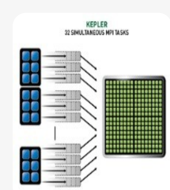
Hardware Capabilities

SMX

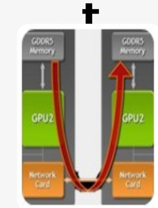
Dynamic Parallelism



HyperQ



GPUDirect



3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directive
s

Programmi
ng
Languages

"Drop-in"
Acceleration

Easily Accelerate
Applications

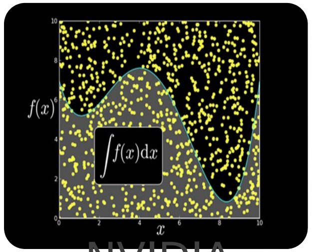
Maximum
Flexibility

Libraries: Easy, High-Quality Acceleration

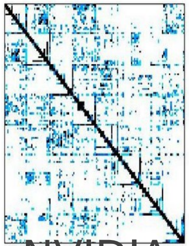
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

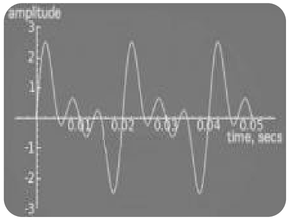
CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

open source initiative

Thrust

C++ STL Features for CUDA

open source initiative

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC Directives

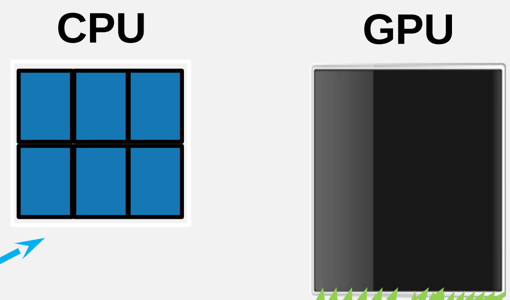
Programming Languages

"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

OpenACC Directives




```
Program myscience
  ... serial code ...
!$acc kernels
  do k = 1,n1
    do i = 1,n2
      ... parallel code ...
    enddo
  enddo
!$acc end kernels
  ...
End Program myscience
```

OpenACC compiler Hint

Your original Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC

The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications

- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

A Very Simple Exercise: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
$!acc kernels
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
$!acc end kernels
end subroutine saxpy

...
$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

`kernels`: Your first OpenACC Directive

Each loop executed as a separate *kernel* on the GPU.

```
!$acc kernels
     do i=1,n
        a(i) = 0.0
        b(i) = 1.0
        c(i) = 2.0
     end do

     do i=1,n
        a(i) = b(i) + c(i)
     end do
!$acc end kernels
```

kernel 1

kernel 2

Kernel:
A parallel function that runs on the GPU

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directive
s

Programmin
g Languages

"Drop-in"
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	Thrust, CUDA C++
Python	PyCUDA, Copperhead
F#	Alea.cuBase

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.

Introduction to CUDA C/C++

– 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 Computin
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Heterogeneous Computing

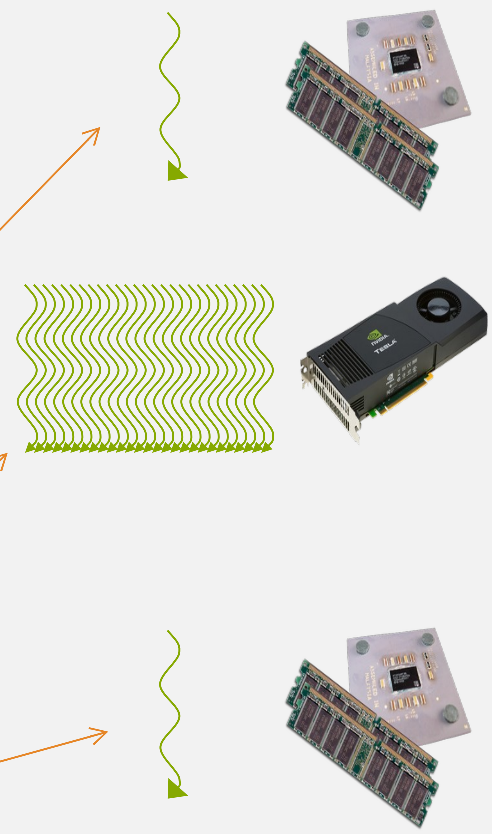
```
#include <iostream>
#include <algorithm>

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

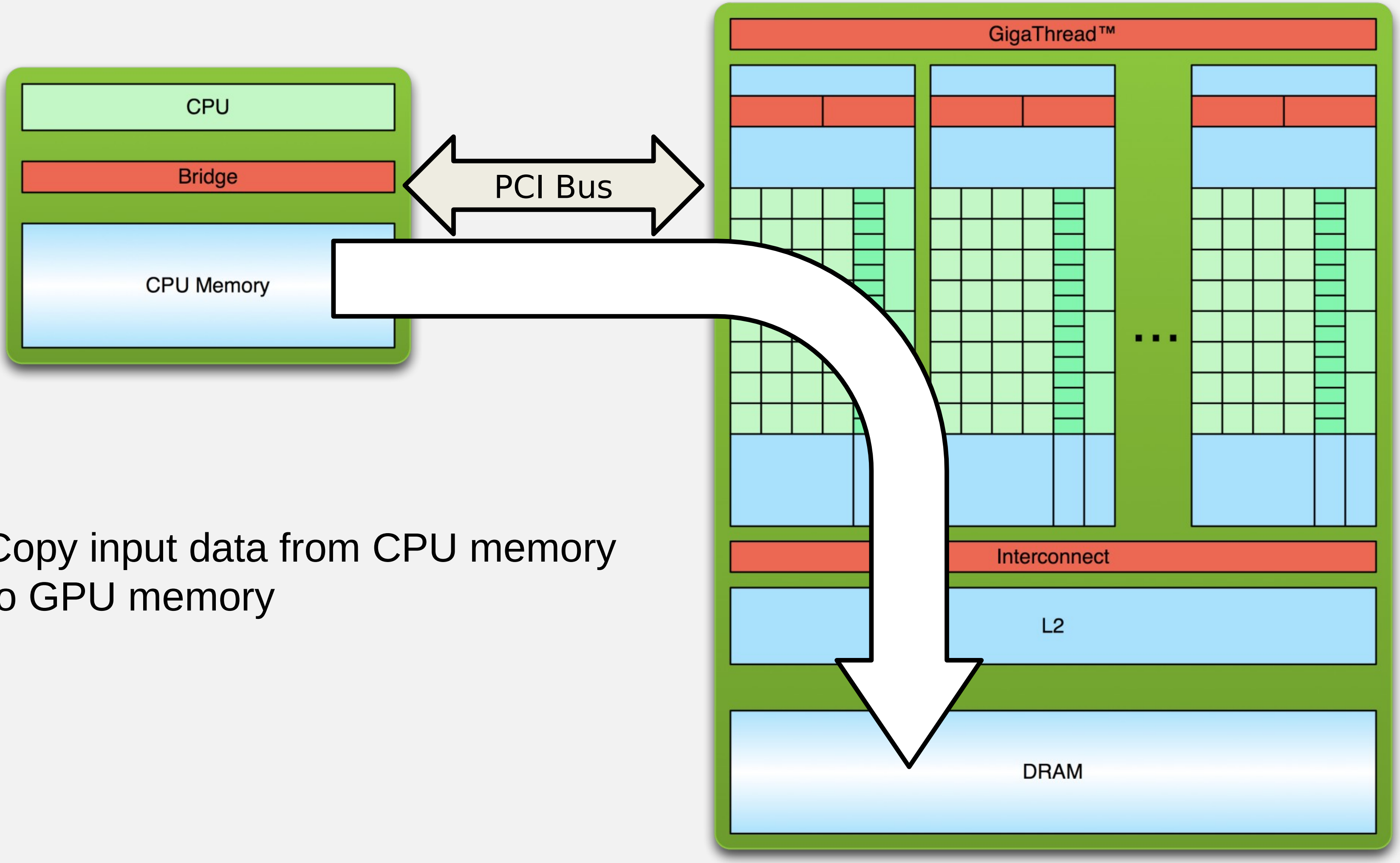
serial code

parallel code

serial code

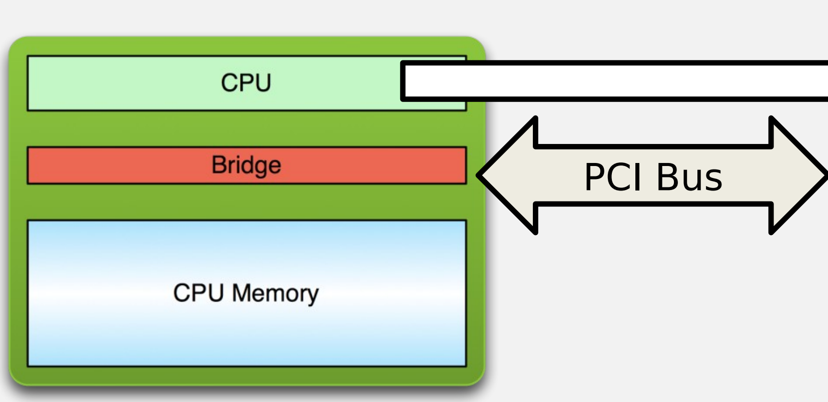


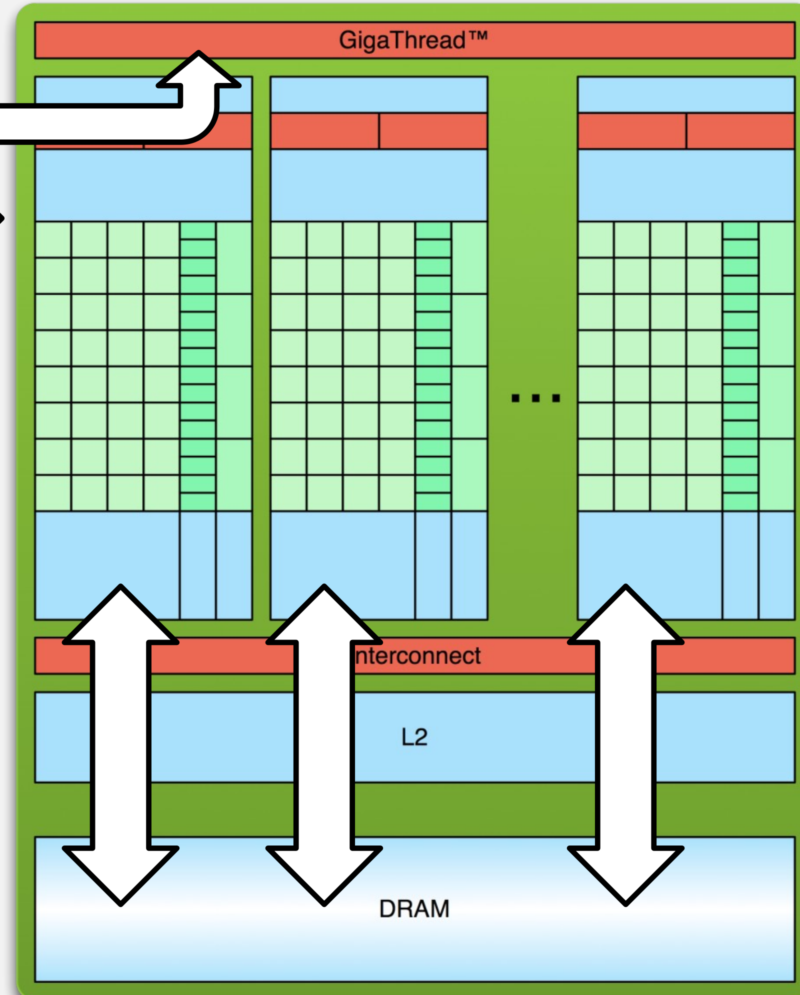
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

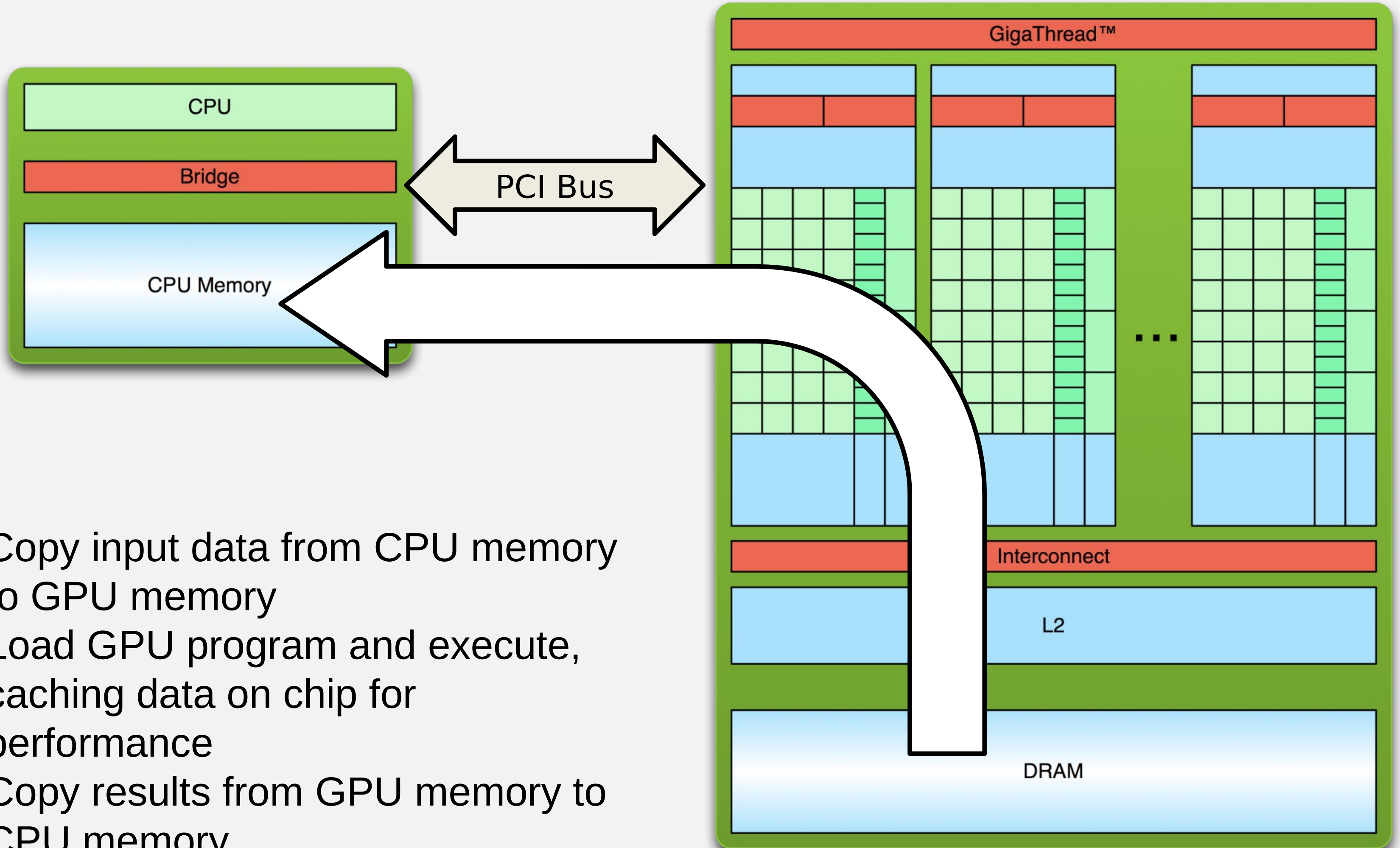
Simple Processing Flow





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! with Device Code

```
__global__ void mykernel(void) {
}

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

- Two new syntactic elements…

Hello World! with Device Code

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

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- nvcc separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - **`gcc`, `cl.exe`**

Hello World! with Device COde

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

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

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

Hello World! with Device Code

```
__global__ void mykernel(void){
}

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

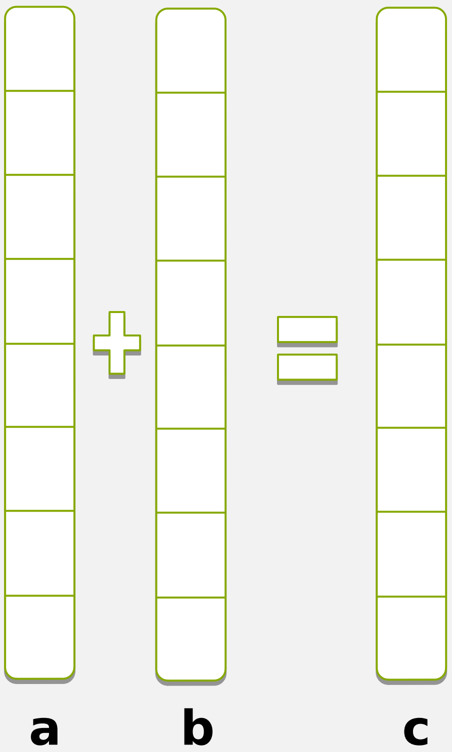
Output:

```
$ 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;
}
```

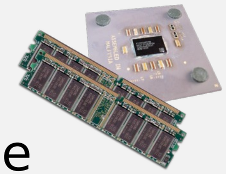
- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- 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: `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

Vector Addition on the Device

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

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

Block 0

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

Block 1

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

Block 2

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

Block 3

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

CONCEPTS

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

INTRODUCING THREADS

CUDA Threads

- Terminology: a block can be split into parallel threads

- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

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

- We use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`...

Vector Addition Using Threads: main()

```
 // 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 Computin
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

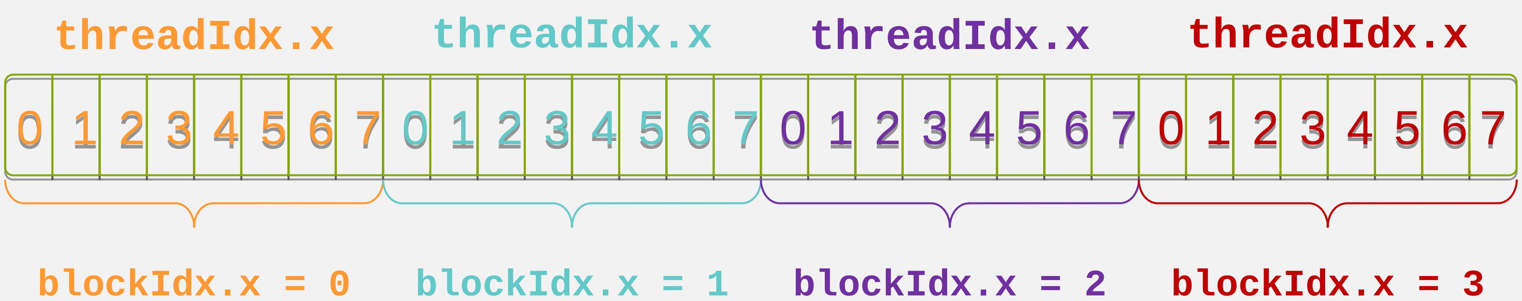
COMBINING THREADS AND BLOCKS

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)

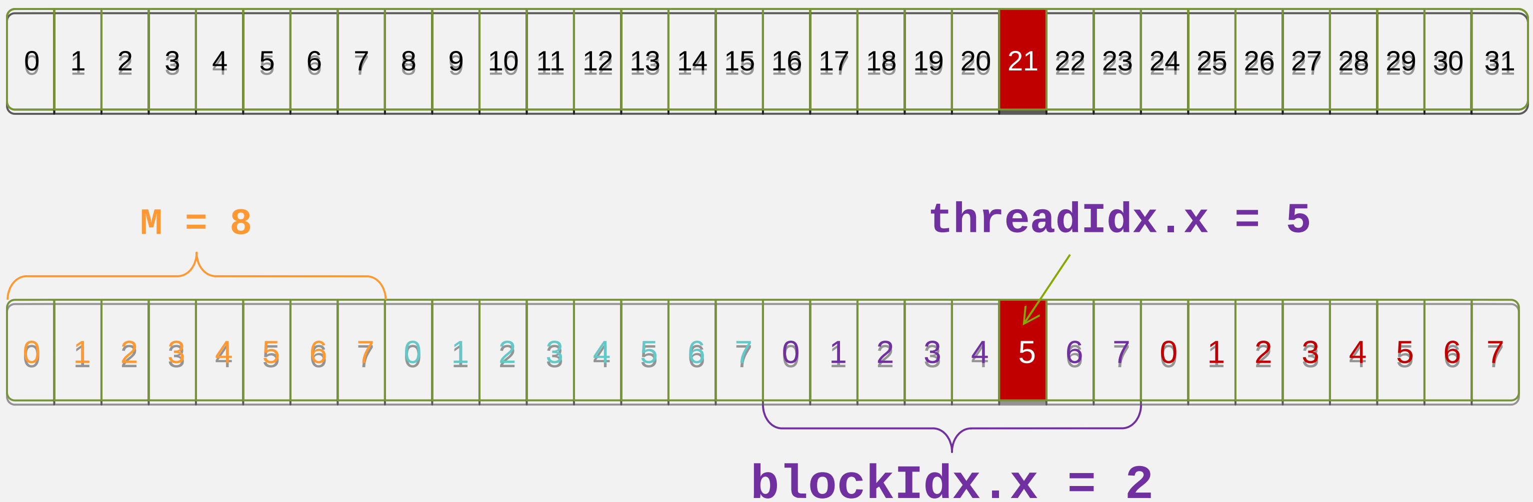


- With M threads/block a unique index for each thread is given by:

```
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;
```

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…

COOPERATING THREADS

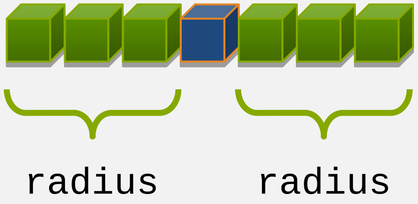
CONCEPTS

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

1D Stencil

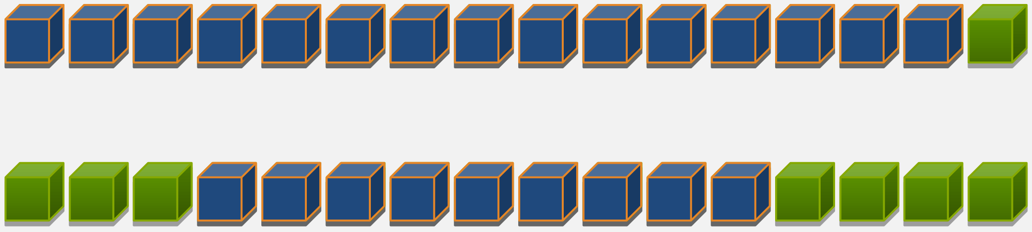
- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

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

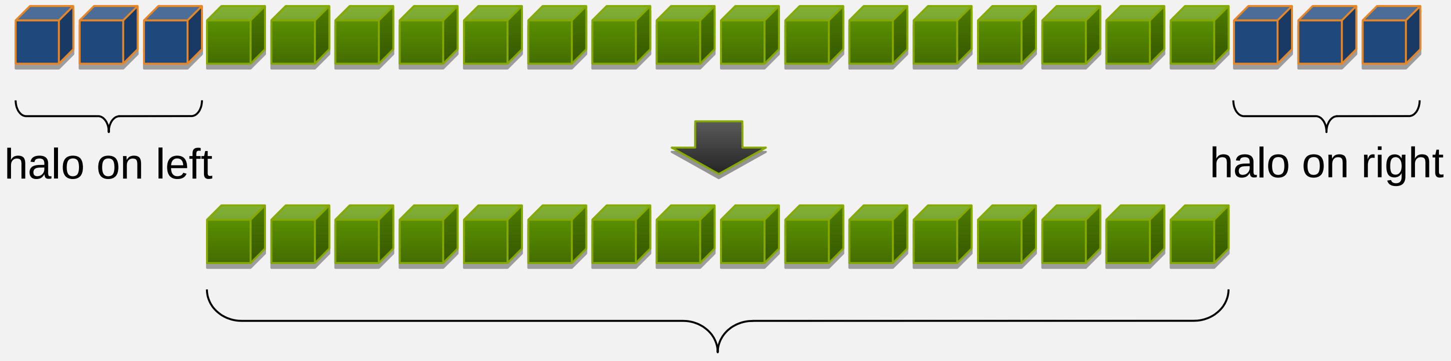


Sharing Data Between Threads

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

Implementing With Shared Memory

- Cache data in shared memory
  - Read (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];
  }
```

Data Race!

- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched it…

Store at temp[18] 

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

Skipped, threadIdx > RADIUS

```
int result = 0;
result += temp[lindex + 1];
```

Load from temp[19] 

__syncthreads()

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

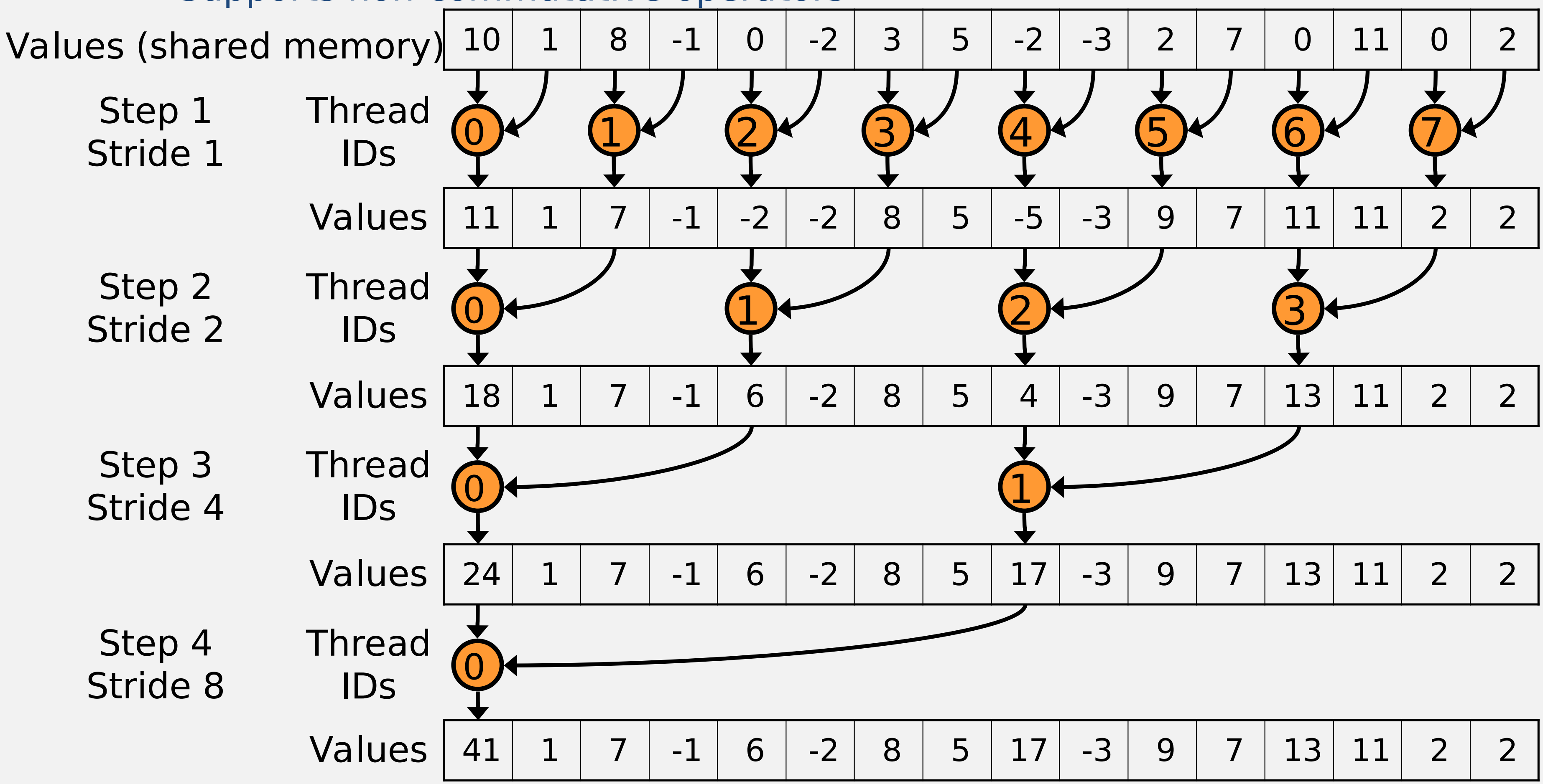
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

CONCEPTS that we missed

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

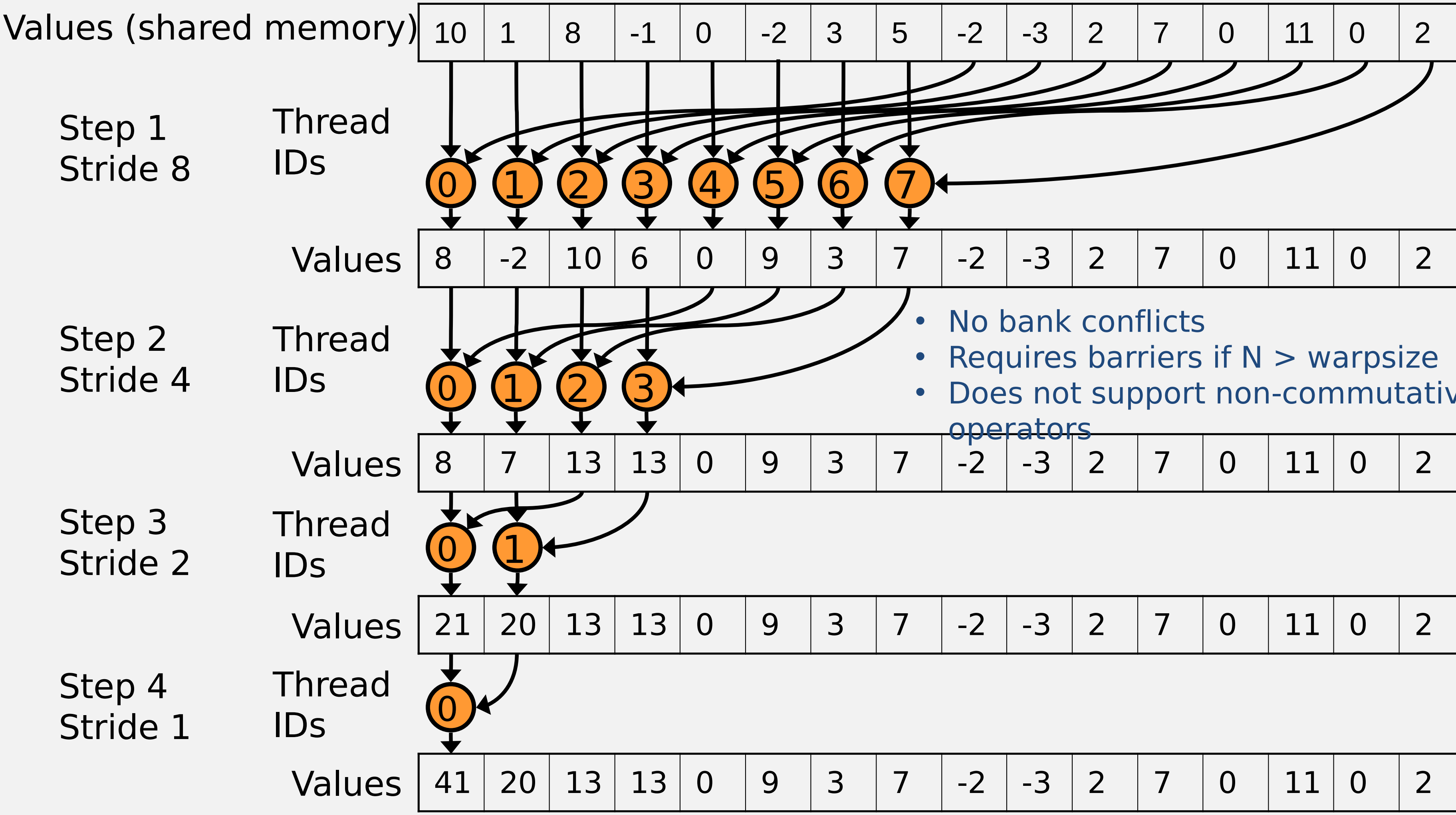
Parallel Reduction: Interleaved Addressing

- Arbitrarily bad bank conflicts
- Requires barriers if N > warpsize
- Supports non-commutative operators



Interleaved addressing results in bank conflicts

Parallel Reduction: Sequential Addressing



- No bank conflicts
- Requires barriers if N > warpsize
- Does not support non-commutative operators

Sequential addressing is conflict free