



Introduction to GPU and GPU programming

- Stefano Cozzini
- CNR-IOM and eXact lab srl

Agenda

- Why GPU ? (a little bit of history)
- GPU architecture
- How to use/programming GPUs ?

A little bit of history 1: the rising (90s..)

- The CPU has always been slow for Graphics Processing
 - Visualization
 - Games
- Graphics processing is inherently parallel and there is a lot of parallelism – $O(\text{pixels})$
- GPUs were built to do graphics processing only
- Initially, hardwired logic replicated to provide parallelism
 - Little to no programmability

A little bit of history 2: the 00's

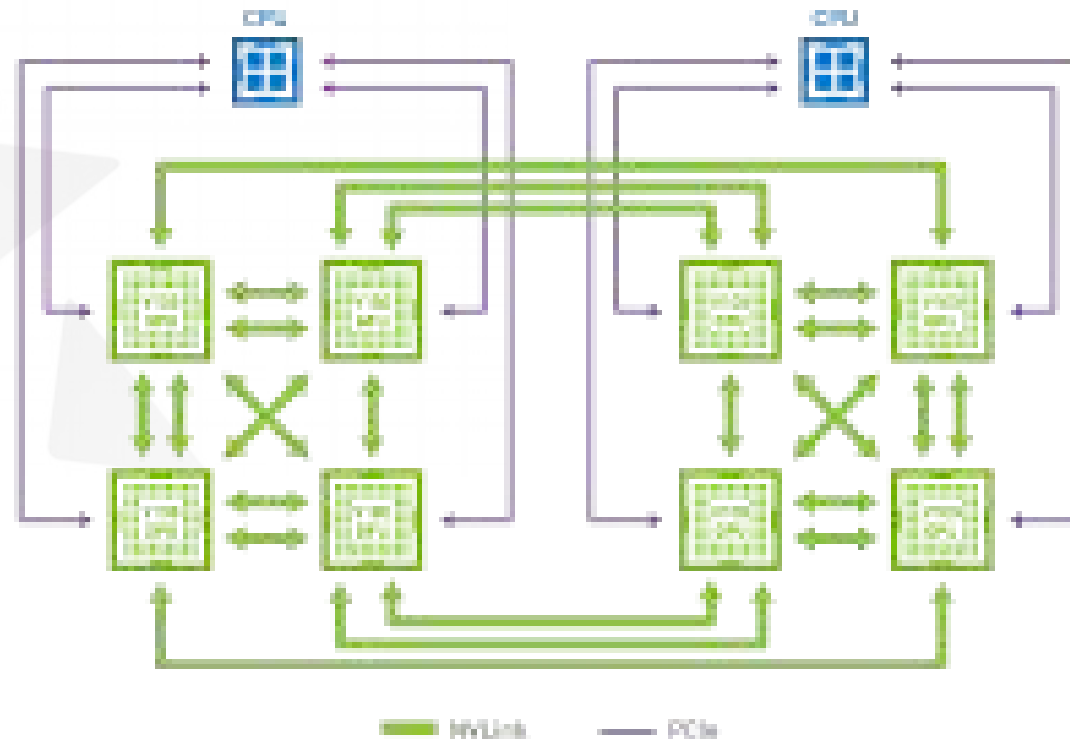
- Like CPUs, GPUs benefited from Moore's Law
 - Evolved from fixed-function hardwired logic to flexible, programmable ALUs
- Around 2004, GPUs were programmable “enough” to do some non-graphics computations
 - Severely limited by graphics programming model (shader programming)
- In 2006, GPUs became “fully” programmable
 - NVIDIA releases “CUDA” language to write non-graphics programs that will run on GPUs

A little bit of history 3: the 10's (the present)

- GPUs are widely deployed as accelerators
- GPUs so successful that other CPU alternatives are dead
 - Sony/IBM Cell BE
 - Clearspeed RSX
 - Intel MIC
- GPU enabled the ML/DL/AI revolution and started the HPC/AI convergence
 - Tensorcore
- There is ONE winner: NVIDIA

Nvlink : avoid PCI- bus contention

- Direct communication link between two GPUs
- Improves accuracy and convergence of high-performance computing (HPC) and AI
- Achieves speeds over an order of magnitude faster than PCIe.



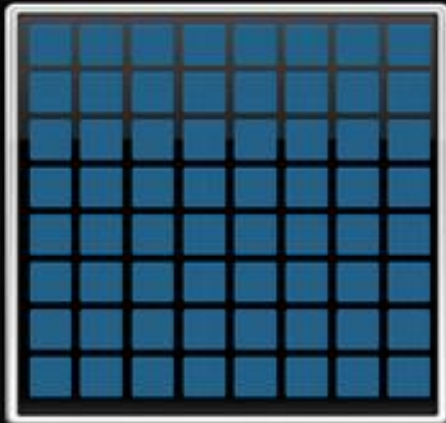
Nvidia Cards

	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"	"Volta"
Tesla GPU	GF100	GF104	GK104	GK110	GM200	GP100	GV100
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0	7.0
Streaming Multiprocessors (SMs)	16	16	8	15	24	56	84
FP32 CUDA Cores / SM	32	32	192	192	128	64	64
FP32 CUDA Cores	512	512	1536	2880	3072	3584	5376
FP64 Units	-	-	512	960	96	1792	2688
Tensor Core Units							672
Threads / Warp	32	32	32	32	32	32	32
Max Warps / SM	48	48	64	64	64	64	64
Max Threads / SM	1536	1536	2048	2048	2048	2048	2048
Max Thread Blocks / SM	8	8	16	16	32	32	32
32-bit Registers / SM	32768	32768	65536	65536	65536	65536	65536
Max Registers / Thread	63	63	63	255	255	255	255
Max Threads / Thread Block	1024	1024	1024	1024	1024	1024	1024
Shared Memory Size Configs	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB	Config
	48 KB	48 KB	32 KB	32 KB			Up To
			48 KB	48 KB			96 KB
Hyper-Q	No	No	No	Yes	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes	Yes
Unified Memory	No	No	No	No	No	Yes	Yes
Pre-Emption	No	No	No	No	No	Yes	Yes

Nvidia slides: computing mode for accelerator

Many-Weak-Cores (MWC) Model
Single CPU Core for Both Serial & Parallel Work

Xeon Phi (And Others)
Many Weak Serial Cores

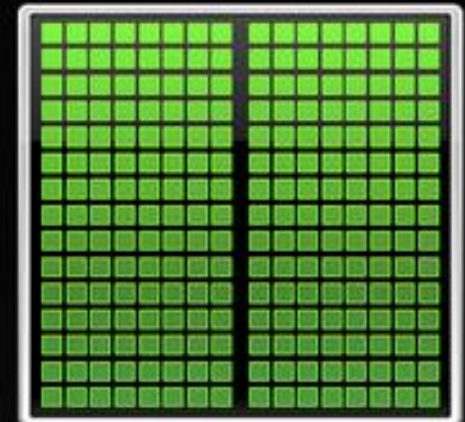


Heterogeneous Computing Model
Complementary Processors Work Together

CPU
Optimized for
Serial Tasks

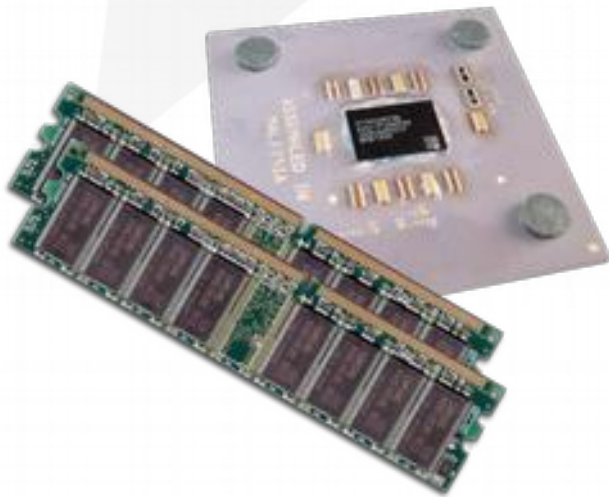


GPU Accelerator
Optimized for
Parallel Tasks



Heterogeneous Computing

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

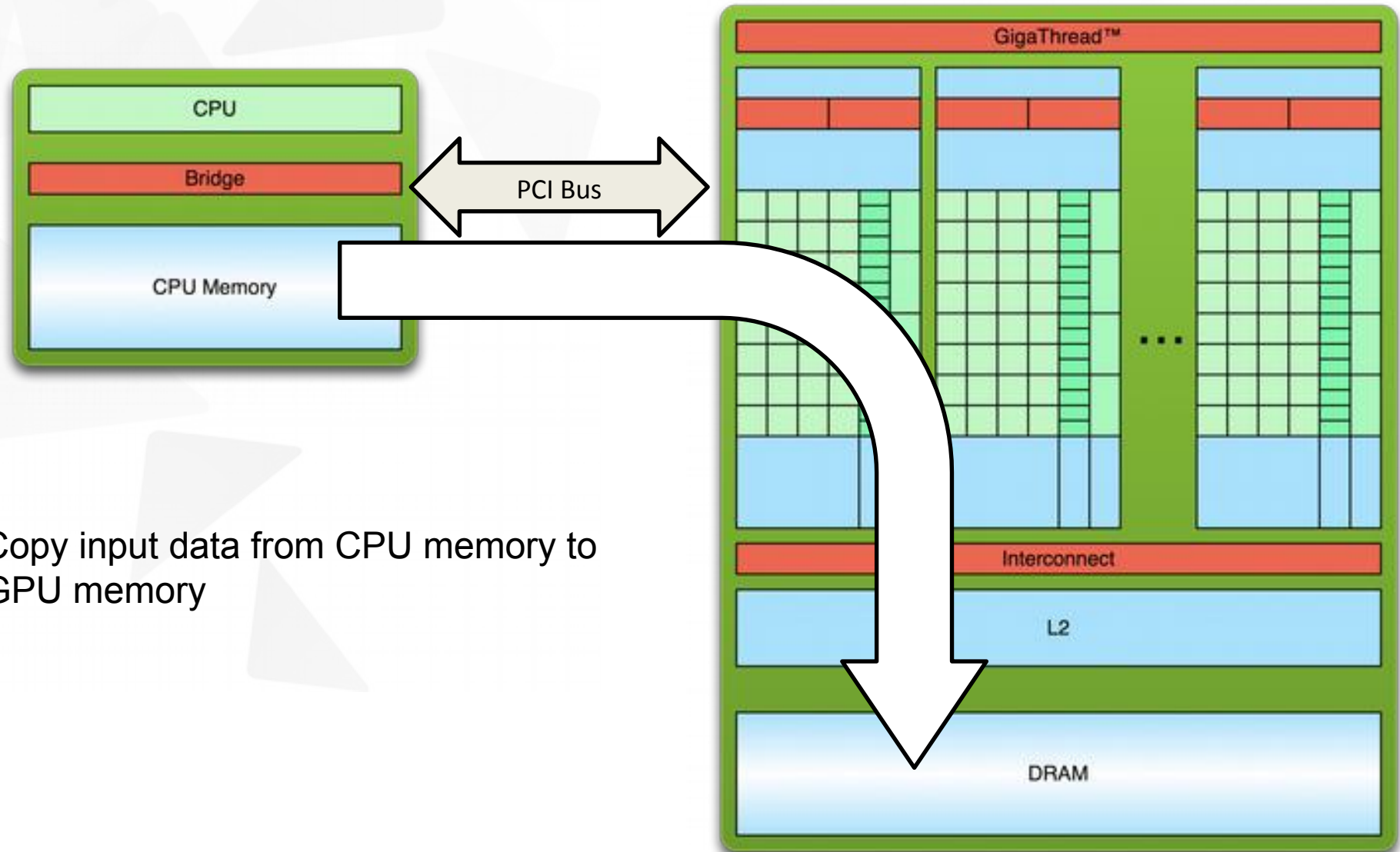


Host



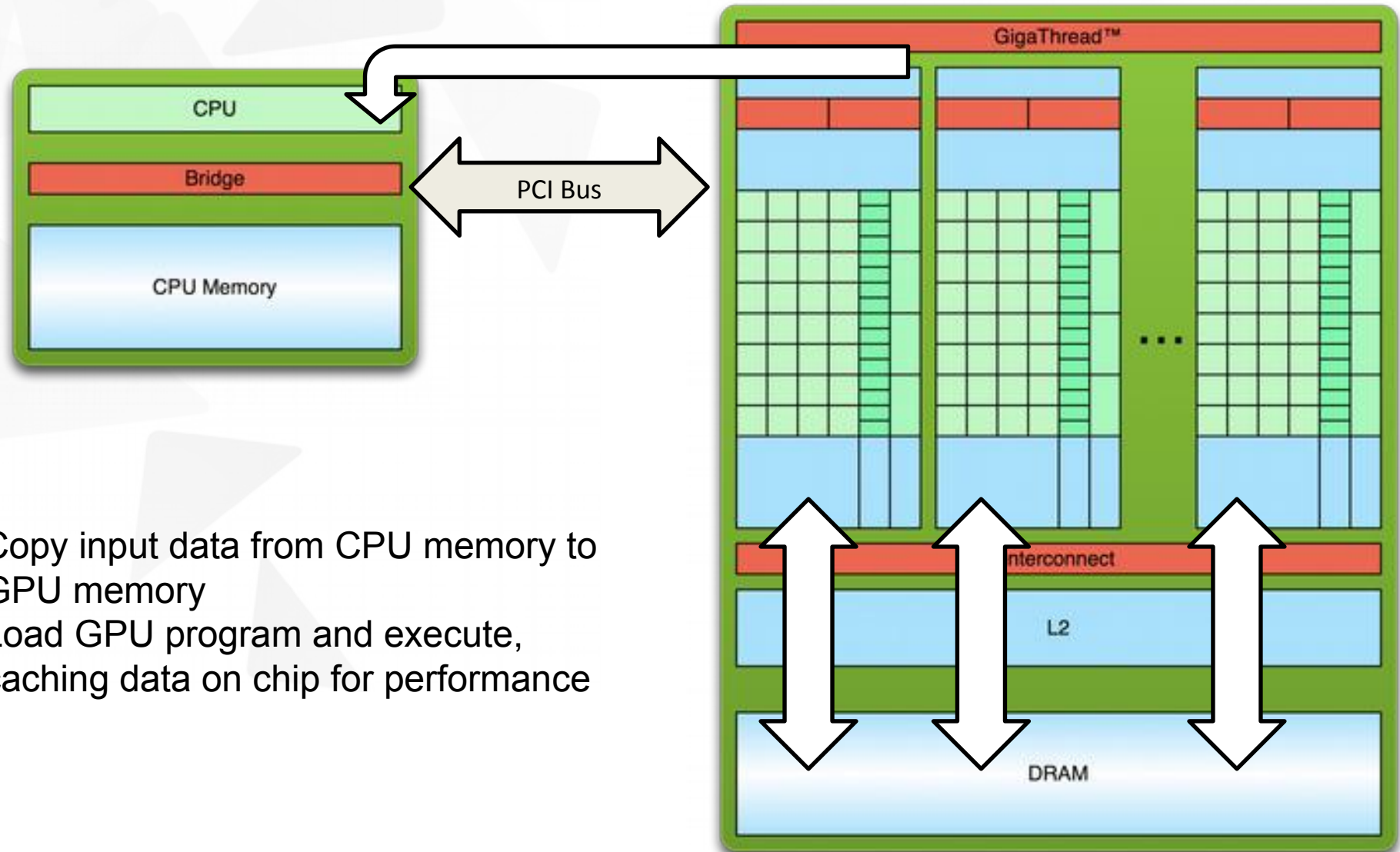
Device

Simple Processing Flow



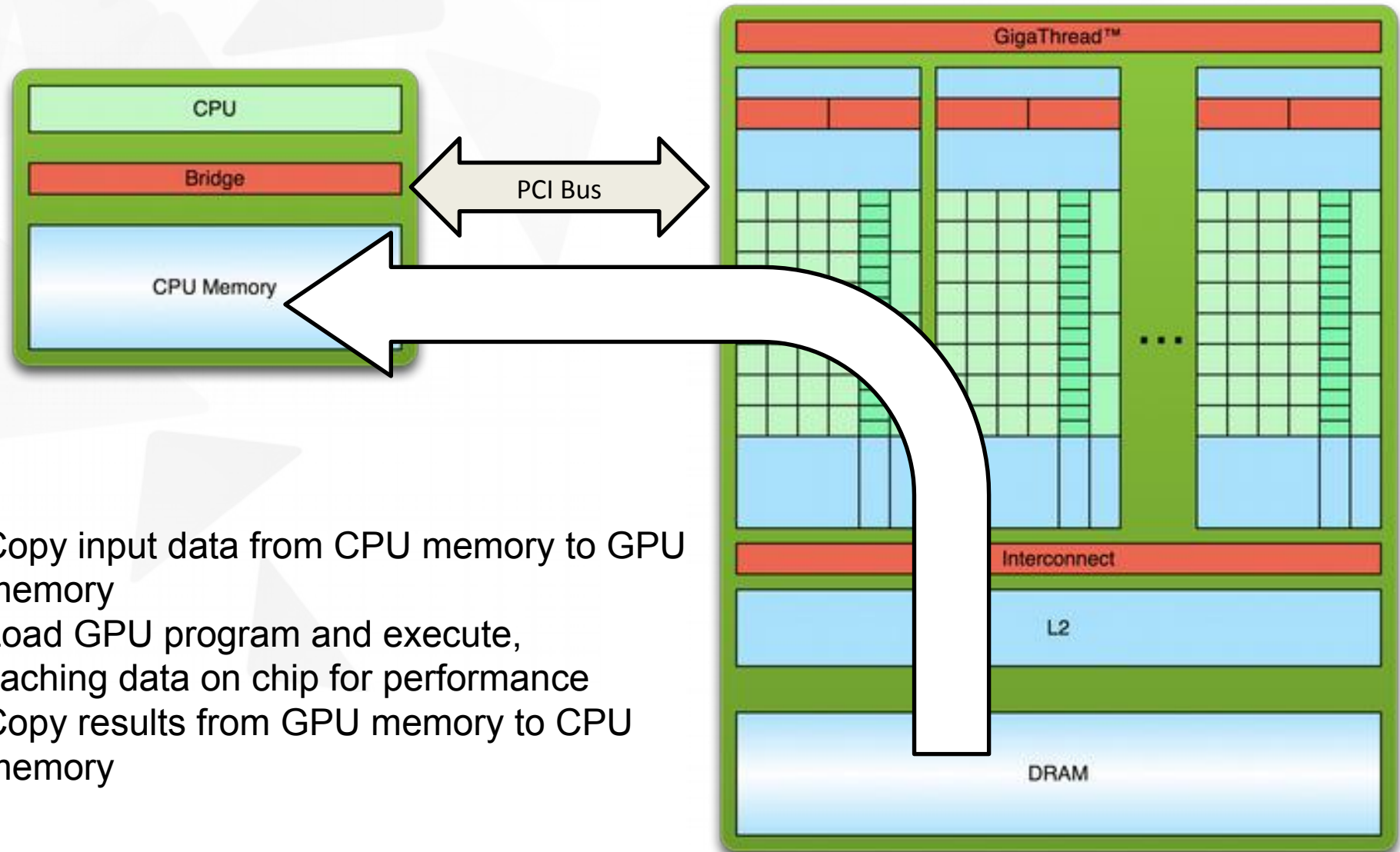
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

CUDA Parallel Computing Platform

Programming Approaches

Libraries

“Drop-in” Acceleration

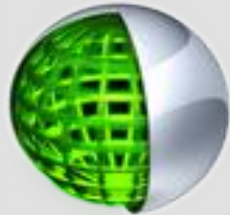
OpenACC Directives

Easily Accelerate Apps

Programming Languages

Maximum Flexibility

Development Environment



Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

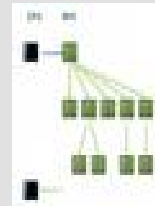
CUDA-GDB debugger
NVIDIA Visual Profiler

Hardware Capabilities

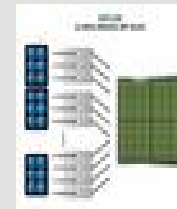
SMX



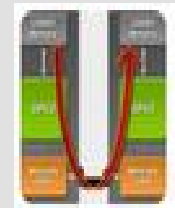
Dynamic Parallelism



HyperQ



GPUDirect



From: www.nvidia.com/getcuda

3 ways to accelerate applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

3 ways to accelerate applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

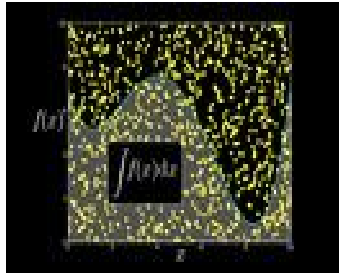
Libraries: Easy, High-Quality Acceleration

- 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

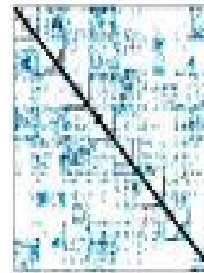
Some GPU-accelerated Libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



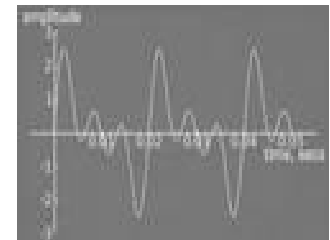
Vector Signal
Image Processing



GPU Accelerated
Linear Algebra



Matrix Algebra on GPU
and Multicore



NVIDIA cuFFT

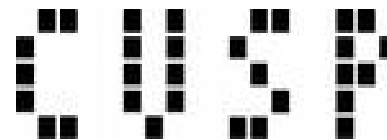


IMSL Library



ArrayFire Matrix
Computations

Building-block
Algorithms for CUDA



Sparse Linear Algebra



C++ STL Features for
CUDA



3 Steps to CUDA-accelerated application

- Step 1: Substitute library calls with equivalent CUDA library calls
`saxpy (...)` ----> `cublasSaxpy (...)`
- Step 2: Manage data locality
 - with CUDA: `cudaMalloc()`, `cudaMemcpy()`, etc.
 - with CUBLAS: `cublasAlloc()`, `cublasSetVector()`, etc.
- Step 3: Rebuild and link the CUDA-accelerated library
`nvcc myobj.o -l cublas`

Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:

developer.nvidia.com/cuda-tools-ecosystem



Tutorial: performing DGEMM on CPU and GPU

- See github repo...

3 ways to accelerate applications

Applications

Libraries

“Drop-in”
Acceleration

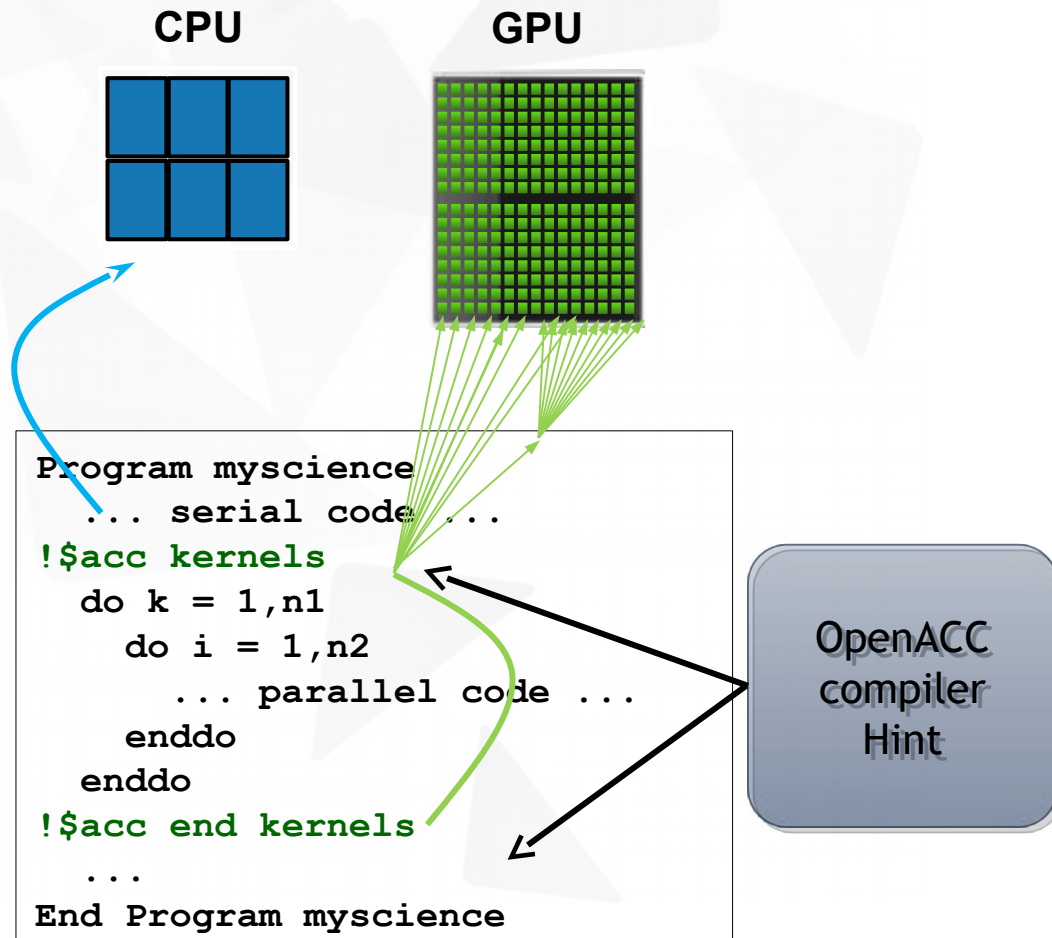
OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

OpenACC Directives



- Simple Compiler hints
- Compiler Parallelizes code
- Works on many-core GPUs & multicore CPUs

Your original Fortran or C
code

- **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

Is OpenACC available ?

- OpenACC is at version 2.7
- PGI compiler fully implements it
- GCC 9 includes initial support for OpenAcc 2.6

OpenACC friendly disclaimer..

OpenACC does not make GPU programming easy. (...)

GPU programming and parallel programming is not easy. It cannot be made easy.

However, GPU programming need not be difficult, and certainly can be made straightforward, once you know how to program and know enough about the GPU architecture to optimize your algorithms and data structures to make effective use of the GPU for computing.

OpenACC is designed to fill that role.

(Michael Wolfe, The Portland Group)

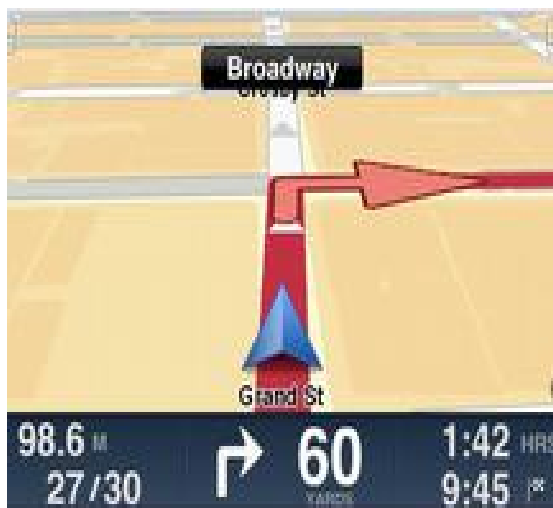
OpenACC –Directive Based Approach

- Directives are added to serial source code
 - Manage loop parallelization
 - Manage data transfer between CPU and GPU memory
- Works with C, C++, or Fortran
 - Can be combined with explicit CUDA C/Fortran usage
- Directives are formatted as comments
 - They don't interfere with serial execution
- Maintains portability of original code

Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems



5x in 40 Hours

Valuation of Stock Portfolios using Monte Carlo

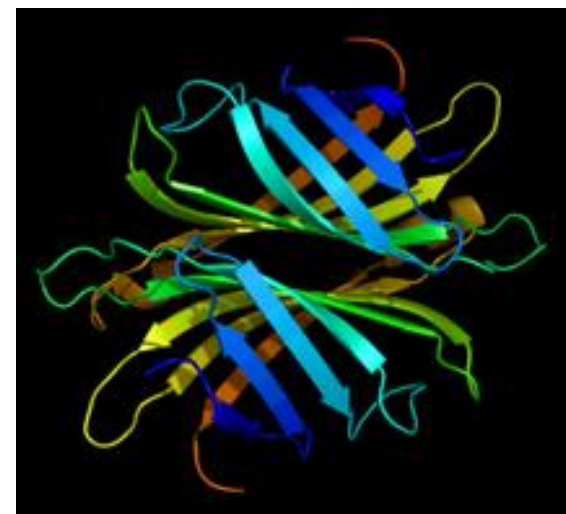
Global Technology Consulting Company



2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio

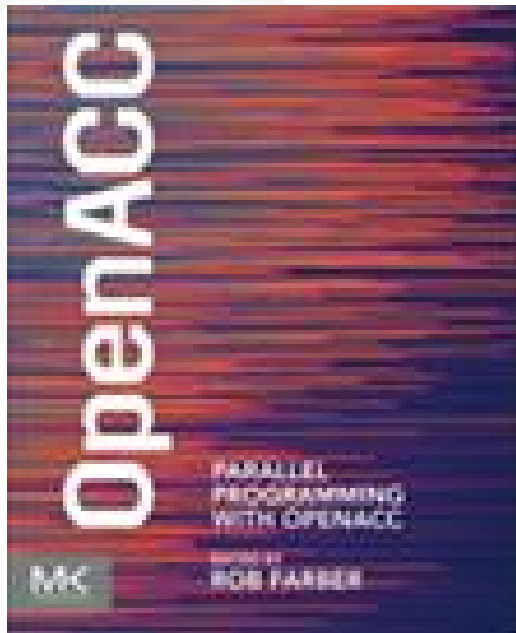


5x in 8 Hours

www.nvidia.com/gpudirectives

OpenACC tutorial

- A section of chapter 1 of this book:



<https://www.amazon.com/Parallel-Programming-OpenACC-Rob-Farber/dp/0124103979>

<https://github.com/rmfarber/ParallelProgrammingWithOpenACC>

3 ways to accelerate applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

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

CUDA programming

- CUDA = **Compute Unified Device Architecture**
 - Expose general-purpose GPU computing as first-class capability
 - Retain traditional DirectX/OpenGL graphics performance
- CUDA C
 - Based on industry-standard C
 - A handful of language extensions to allow heterogeneous programs
 - Straightforward APIs to manage devices, memory, etc.

CUDA basic concepts

- The GPU is viewed as a compute device that:
 - has its own RAM (device memory)
 - runs data-parallel portions of an application as kernels by using many threads
- GPU vs. CPU threads
 - GPU threads are extremely lightweight
 - Very little creation overhead
 - GPU needs 1000s of threads for full efficiency
 - A multi-core CPU needs only a few (basically one thread per core)

Hello World!

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

```
>nvcc hello_world.cu
```

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

Hello World! with Device Code

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

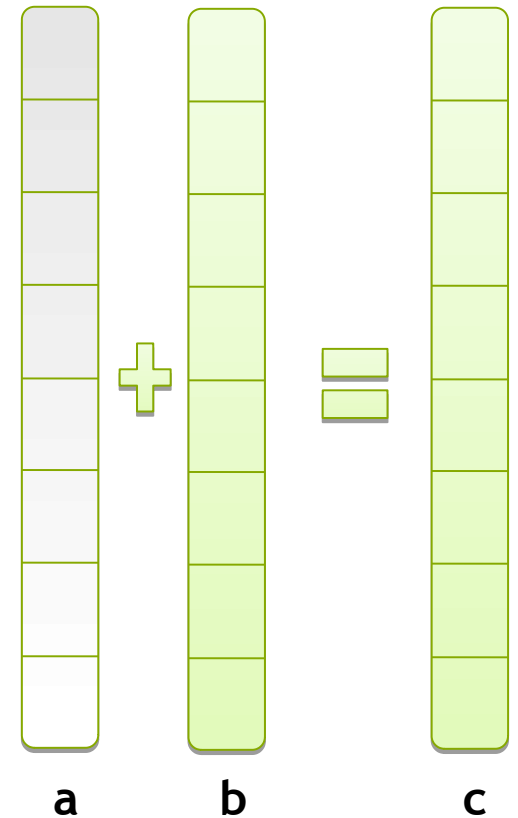
Hello World! with Device Code

```
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

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

Memory Management

- Host and device memory are separate entities
 - *Device* pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - *Host* pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



Addition on the Device: add()

- Returning to our `add()` kernel

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

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

Addition on the Device: main()

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


Moving to Parallel

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

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



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

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

Vector Addition on the Device

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

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

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

Vector Addition on the Device

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

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

Block 0

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

Block 1

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

Block 2

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

Block 3

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

Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

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

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

Vector Addition on the Device: main() (1)

```
#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() (2)

// 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

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() (1)

```
#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;
```

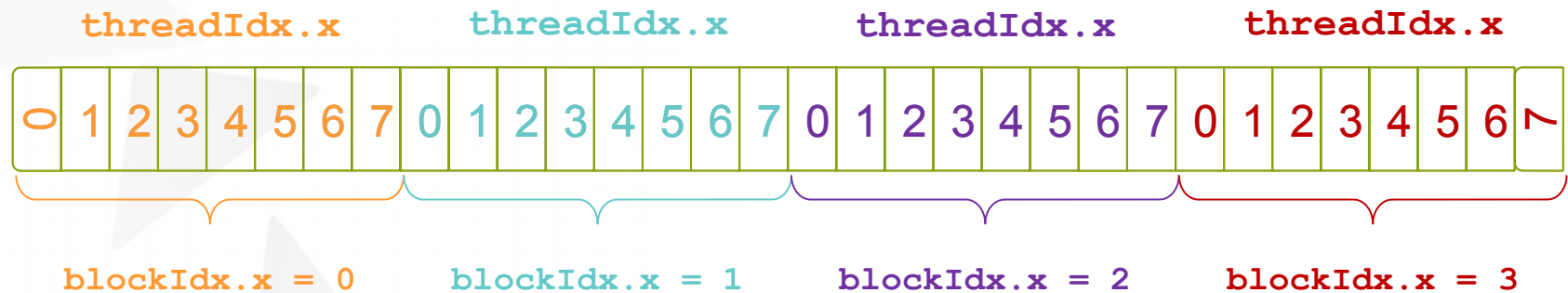
```
}
```

Combining 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

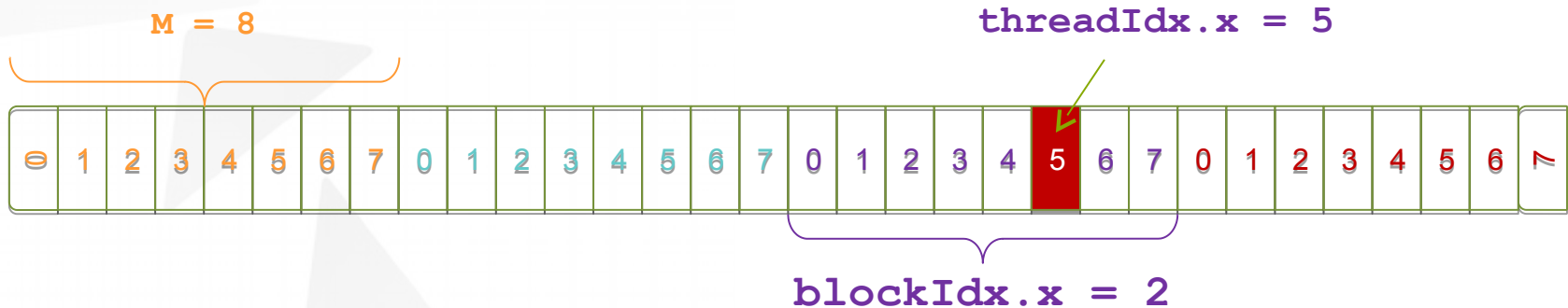
- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/block)



- 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;
```

Addition with Blocks and Threads

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

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

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

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

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

Addition with Blocks and Threads: main() (1)

```
#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() (2)

```
// 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);
```

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:

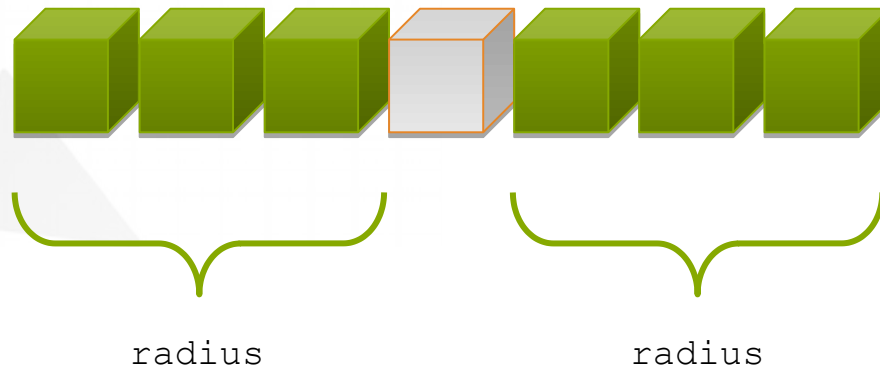
```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

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...

1D Stencil

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



Implementing Within a Block

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

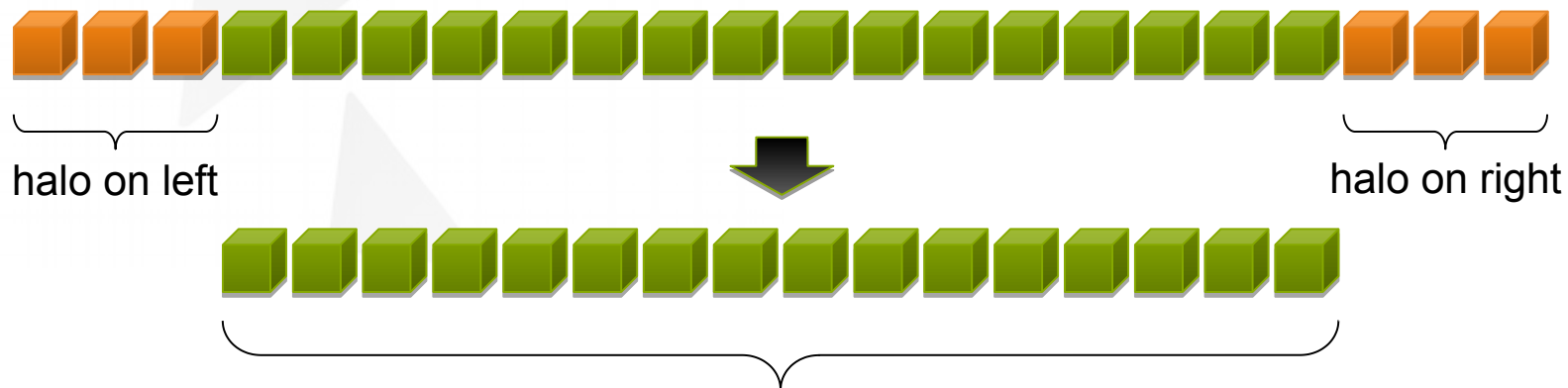


Sharing Data Between Threads

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

Implementing With Shared Memory

- Cache data in shared memory
 - Read ($\text{blockDim.x} + 2 * \text{radius}$) input elements from global memory to shared memory
 - Compute blockDim.x output elements
 - Write blockDim.x output elements to global memory



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

Stencil kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
            in[gindex + BLOCK_SIZE];
    }
}
```




Stencil kernel

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


Data Race!

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

```
temp[lindex] = in[gindex];  
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

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;  
}
```

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

End...



All text and image content in this document is licensed under the [Creative Commons Attribution-Share Alike 3.0 License](#) (unless otherwise specified). "LibreOffice" and "The Document Foundation" are registered trademarks. Their respective logos and icons are subject to international copyright laws. The use of these therefore is subject to the [trademark policy](#).