# GPU Programming

Duy-Phuc Pham

Sep. 2021

## Whoami

- 3rd year PhD candidate at IRISA/CNRS
- Machine/Deep-Learning, Malware analysis, Side-channel.

## Outline

1. Introduction to GPU
2. Introduction to CUDA C/C++
3. Rainbow table/hash generator on GPU

## 1. Introduction to GPU

- Graphics processing unit
  - A electronic circuit designed to rapidly use memory to accelerate the creation of images for output to a display device.
  - Gaming software companies, movie companies and medical research

## CPU vs. GPU

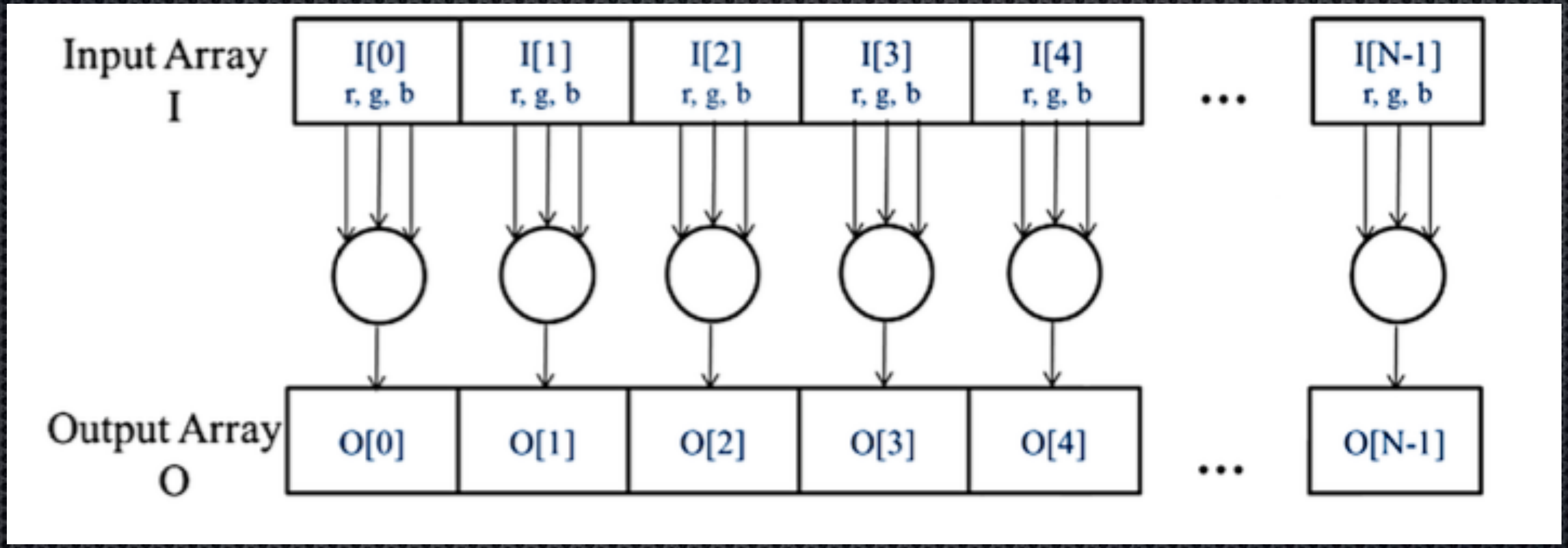
**CPU**

- General-purpose capabilities, mostly sequential operations
- Established technology
- Usually equipped with 8 or fewer, powerful cores
- Optimal for some types of concurrent processes but not large scale parallel computations

**GPU/DSP/FPGA**:

- Initially created specifically for graphics
- Became more capable of general computations
- Very fast and powerful, computationally
- Uses lots of electrical power
- This allows us to run many threads simultaneously with virtually no context switches

# Data parallelism



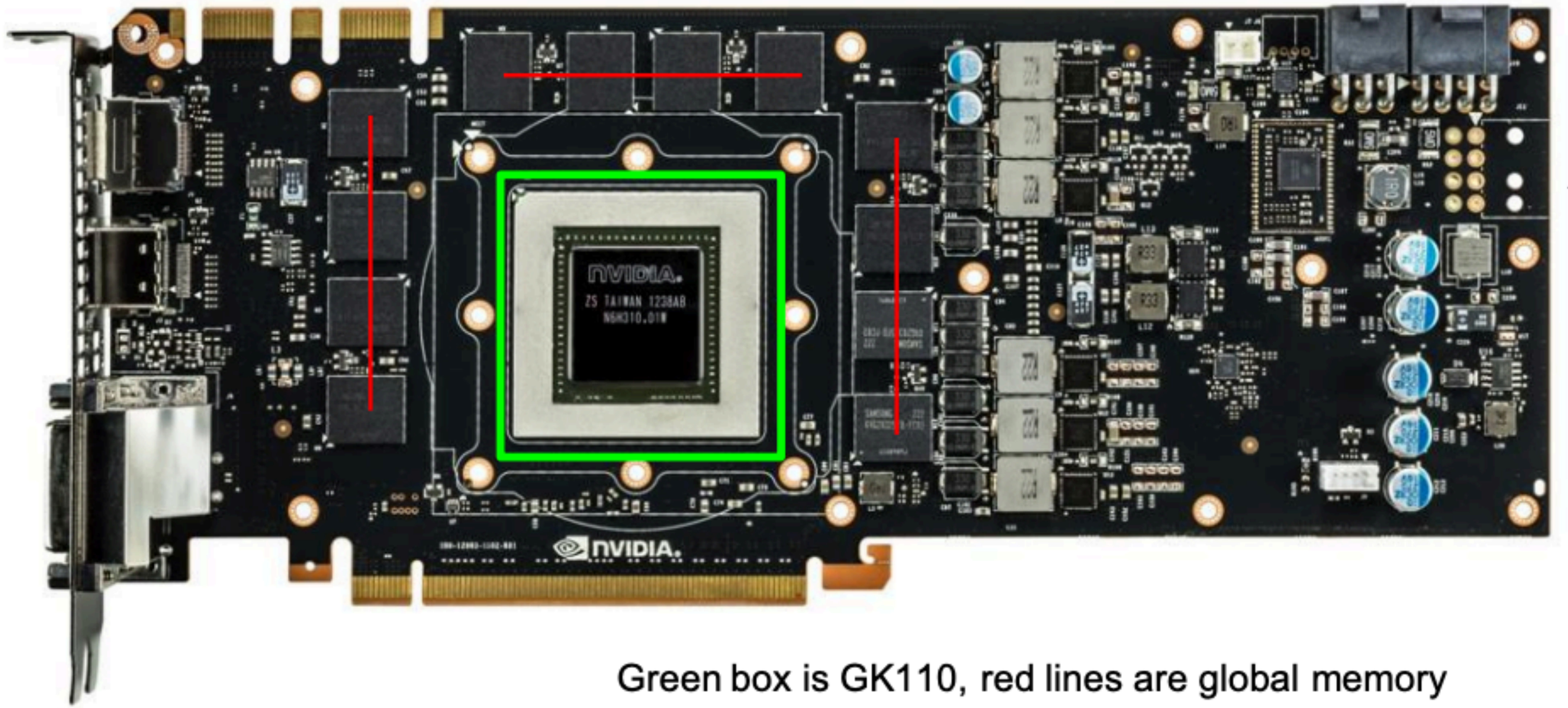
$$L = r * 0.21 + g * 0.72 + b * 0.07$$

Ref: Programming Massively Parallel Processors A Hands-On Approach by David B. Kirk et al

## GPU considerations

- Not all problems are parallelizable
- GPU performance drops dramatically if the code branches
- CPU and GPU code can overlap execution
- GPUs do not make function calls efficiently, and cannot recurse
- Certain operators are very fast on GPUs, and other functions are not. (e.g Integer division, modulus, 64bit integer division etc.)
- GPUs have a limited number of registers available to use.
- Global memory accesses are very slow
- Local shared memory is very fast

# GPU & Global memory



Green box is GK110, red lines are global memory

# CUDA vs OpenCL

Ref: incredibuild.com

| Comparison | CUDA | OpenCL |
|---|---|---|
| **Performance** | - | - |
| **Vendor Implementation** | Implemented by only NVIDIA devices | Implemented by TONS of vendors including AMD, NVIDIA, Intel, Apple, Radeon etc. |
| **OSS vs Commercial** | Proprietary framework of NVIDIA | Open Source standard |
| **OS Support** | Supported on the leading Operating systems with the only distinction of NVIDIA hardware must be used | Supported on various Operating Systems |
| **Libraries** | Has extensive high performance libraries | Has a good number of libraries which can be used on all OpenCL compliant hardware but not as extensive as CUDA |
| **Community** | Has a larger community | Has a growing community not as large as CUDA |
| **Technicalities** | Not a language but a platform and programming model that achieves parallelization using CUDA keywords | Does not enable for writing code in C++ but works in a C programming language resembling environment |

## 2. CUDA

- CUDA platform
  - Export GPU APIs for general purpose
- CUDA C/C++
  - Based on C/C++ standard
  - APIs to manage GPU devices, memory etc.

## Goals

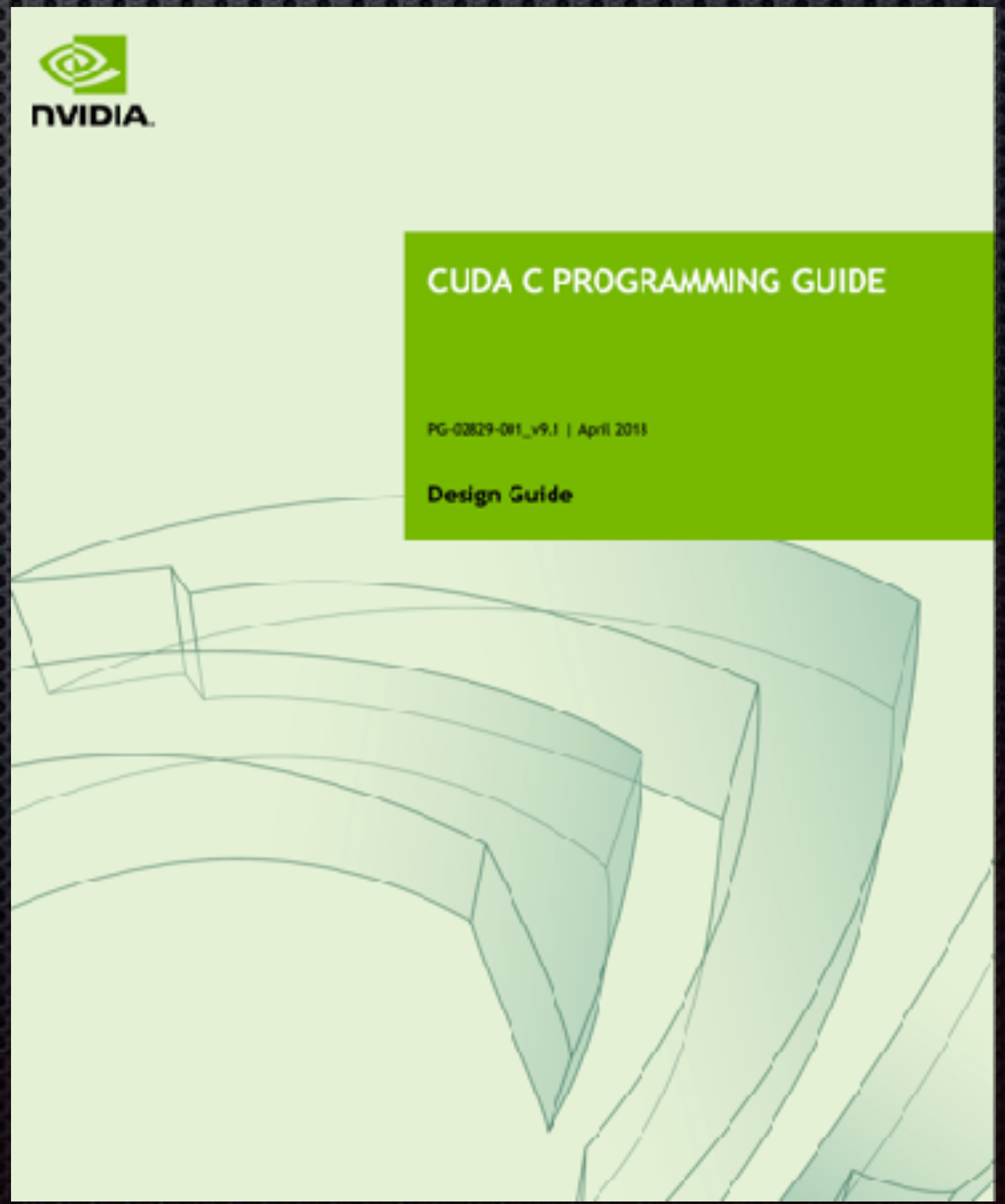
- Write and execute starter C code on GPU
- Manage GPU memory
- Communication and synchronization

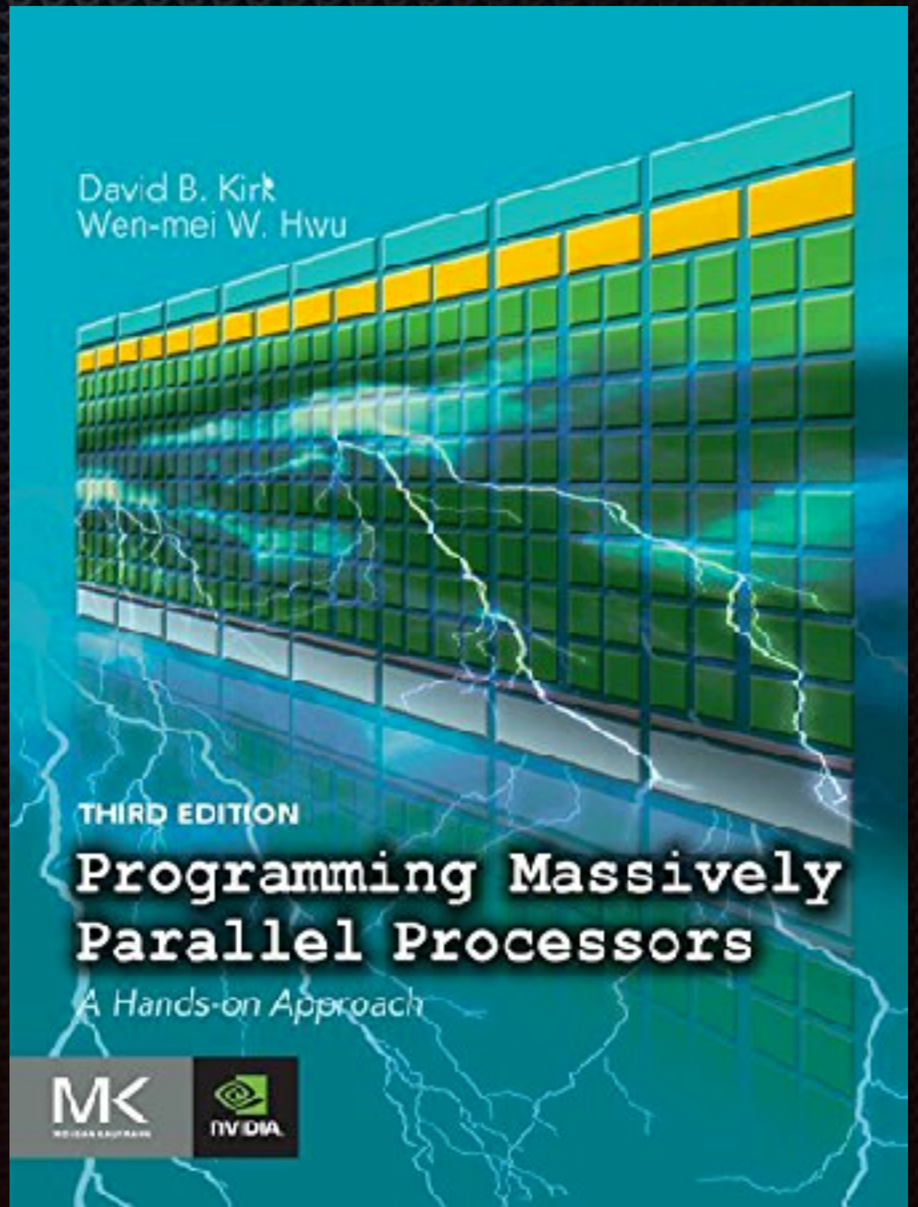
## Requirements

- Virtual (remote) / Physical access to GPU devices
- C/C++ experience
- NO graphic rendering, GPU/parallel computing experience needed

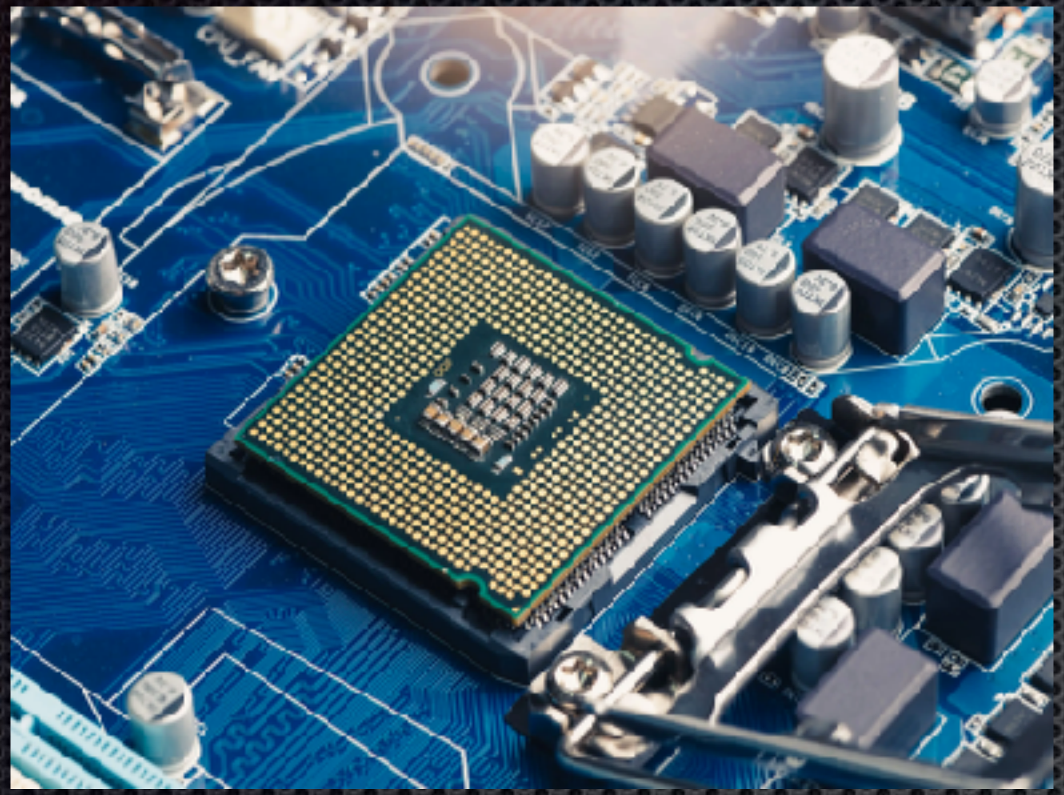
# Recommendation

- CUDA C Programming Guide v9.1 | April 2018
- Programming Massively Parallel Processors A Hands-On Approach by David B. Kirk, Wen-Mei W Hwu
- CUDA C/C++ Basics (nVidia Corp.)
- GPU Programming CS179 Caltech
- CUDA Thread Indexing Cheatsheet





# Heterogeneous computing



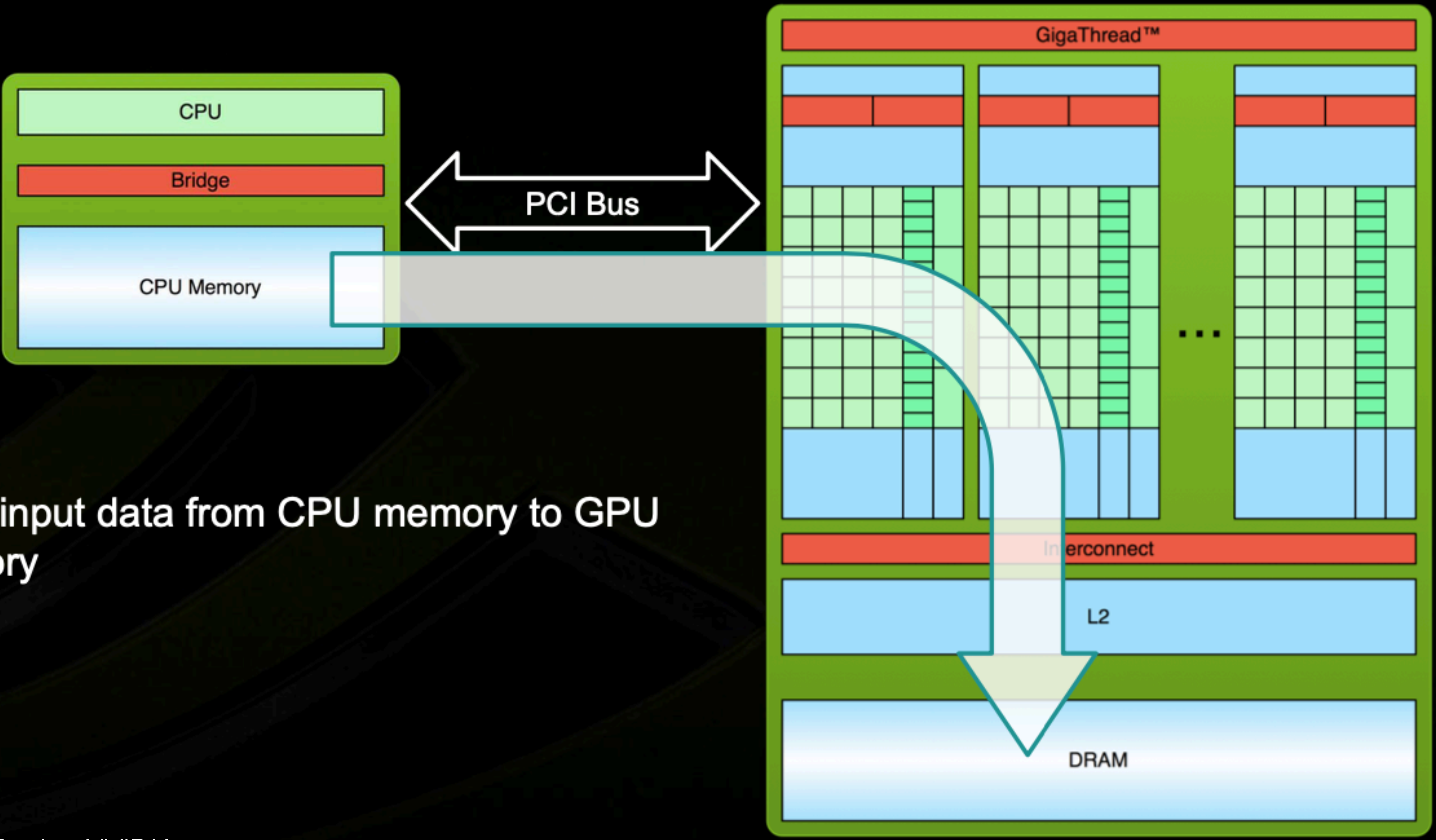
- Host : CPU and host memory (RAM)
- Device: GPU and device memory (VRAM)

*Ref: business insider*

## Typical GPU computing workflow

- Setup inputs on the host (CPU-accessible memory)
- Allocate memory for outputs on the host CPU
- Allocate memory for inputs on the GPU
- Allocate memory for outputs on the GPU
- Copy inputs from host to GPU (*slow*)
- Start GPU **kernel** (function that executes on gpu – *fast*!)
- Copy output from GPU to host (*slow*)

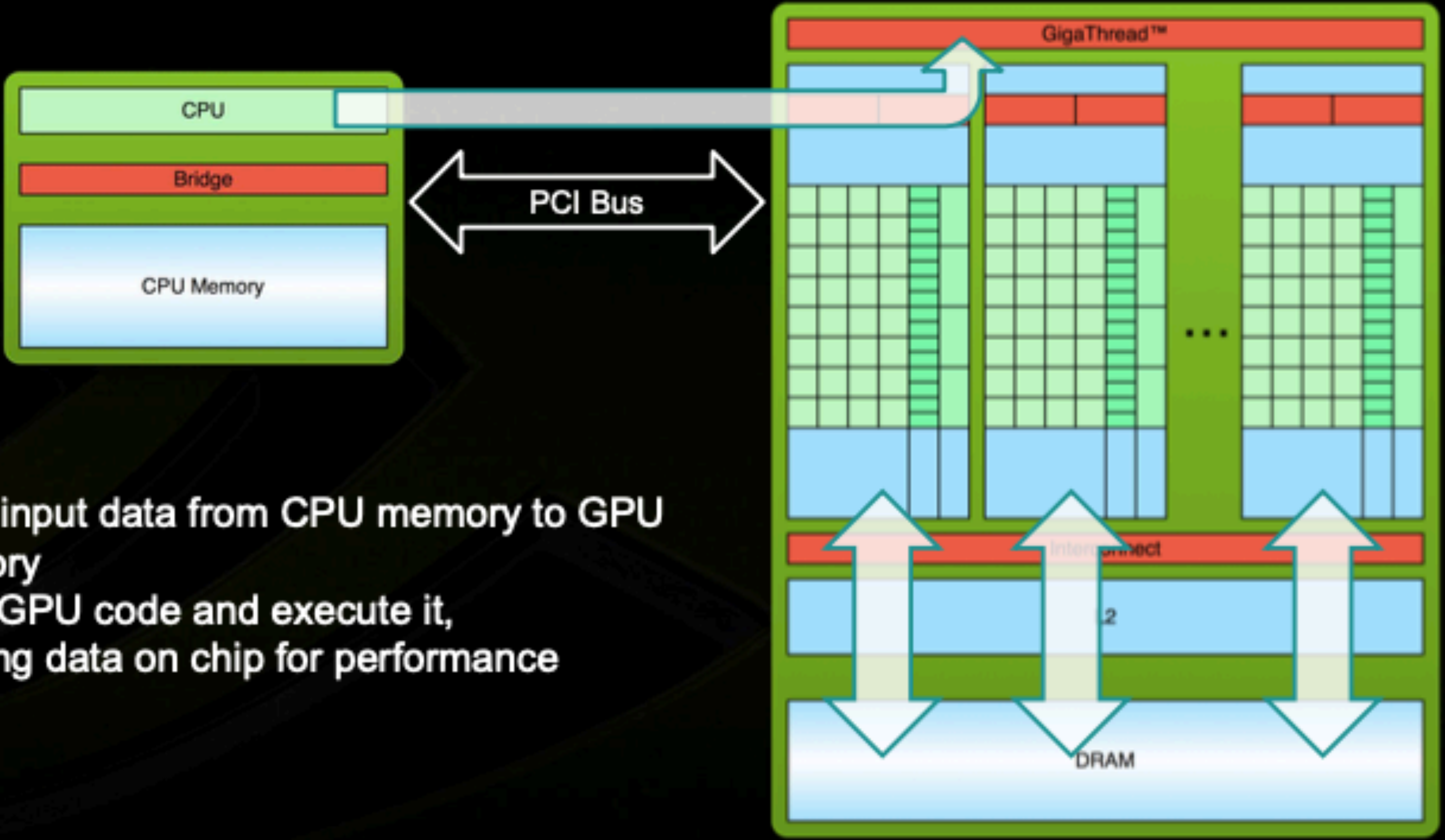
# Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

*Ref: CUDA C/C++ Basics NVIDIA*

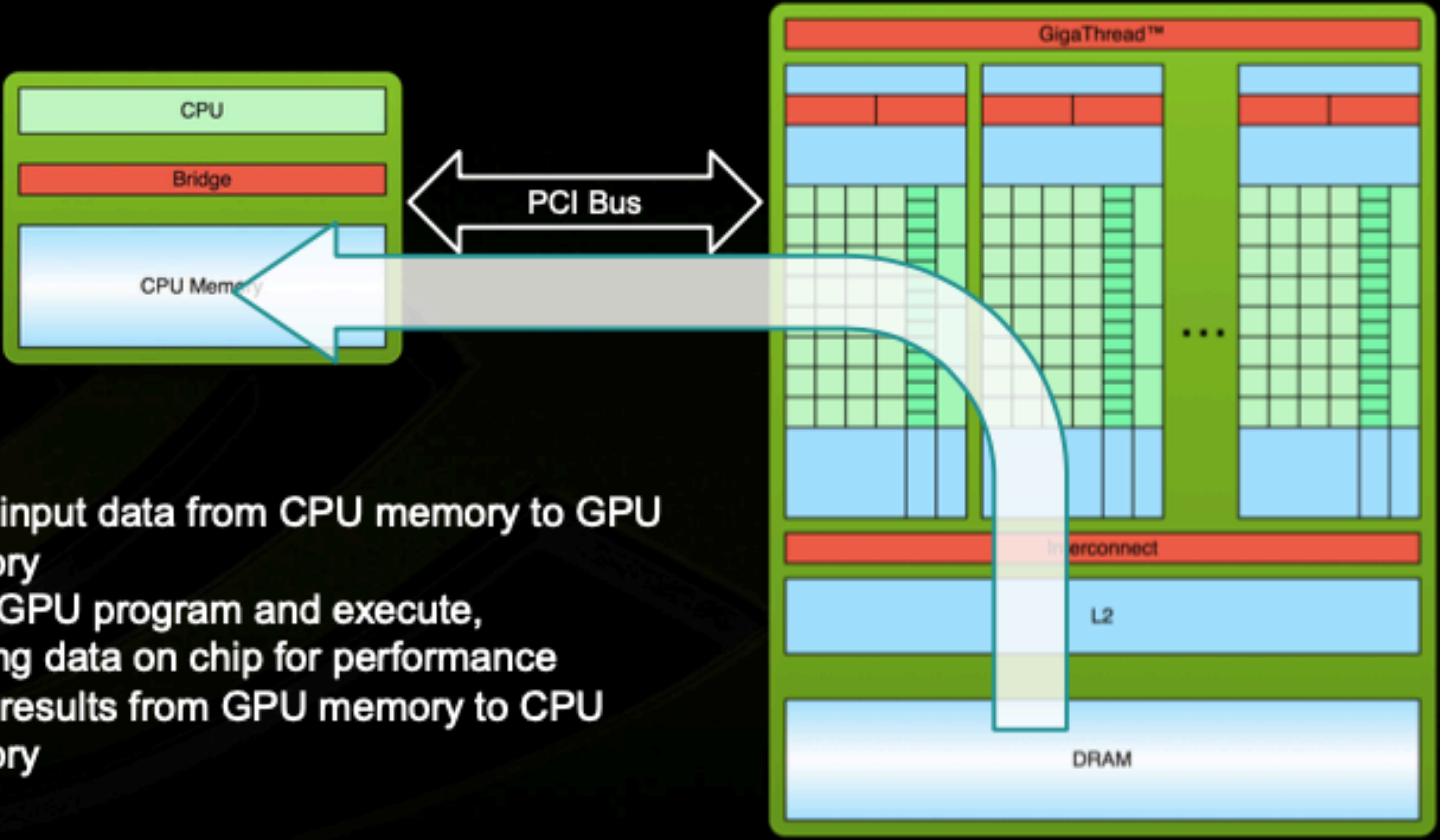
# Simple Processing Flow



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

*Ref: CUDA C/C++ Basics NVIDIA*

# 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

*Ref: CUDA C/C++ Basics NVIDIA*

# Helloworld

nvcc hello.cu -o helloC

```
#include <stdio.h>
int main(void) {
printf("Hello World!\n");
return 0;
}
/**
0.00s user 0.00s system 77%
cpu 0.001 total
**/
```

```
#include <stdio.h>
__global__ void mykernel(void) { }
int main(void){
mykernel<<<1,1>>>();
printf("Hello, World\n");
return 0;
}
/**
0.02s user 0.10s system 55% cpu
0.223 total
**/
```

## Syntax

__global__ :

1. A qualifier added to standard C. This alerts the compiler that a function should be compiled to run on a device (GPU) instead of host (CPU).

2. Function mykernel() is called from host code.

Compile: nvcc hello.cu
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 like gcc

## Simple addition (1)

nvcc add.cu -o add

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

***What live in memory rest in memory***

- Device (/Host) pointer point to Device(/Host) memory
- Device (/Host) pointer maybe passed from to Host(/Device) memory

## Simple addition (2)

```
nvcc add.cu -o add
```

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

*c = a+b; }

***What live in memory rest in memory***

- a,b,c point to device memory -> need to allocate memory on GPU
- cudaMalloc(), cudaFree(), cudaMemcpy()
- Similar to C: malloc(), free(), memcpy()

# Simple addition (3)

nvcc add.cu -o add

```
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 of a, b, c

cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
a = 2; b = 7;               // Setup input values
```

# Simple addition (4)

nvcc add.cu -o add

```
// 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);
printf("%d+%d = %d\n", a, b, c);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c); return 0;
}
```

## Simple Parallelization (1)

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

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

$\downarrow$

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

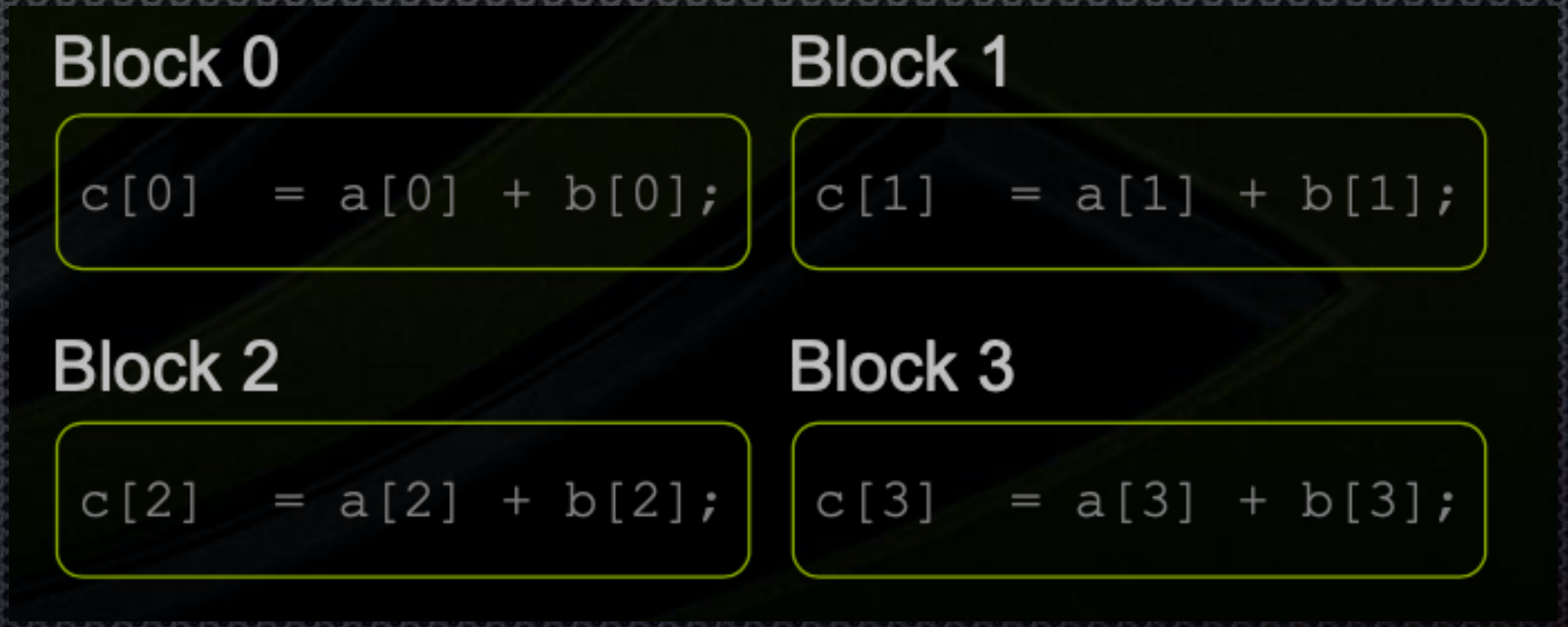
- Each parallel invocation of `add()` is referred to as a block
- The set of blocks is referred to as a grid
- Each invocation can refer to its block index using `blockIdx.x`

## Simple Parallelization (2)

```
__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 element of the array

.



```
c[0]  = a[0] + b[0];
c[1]  = a[1] + b[1];
c[2]  = a[2] + b[2];
c[3]  = a[3] + b[3];
```

## Simple Parallelization (3)

```
#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);// Allocate space for device of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
a = (int *)malloc(size);
b = (int *)malloc(size);
c = (int *)malloc(size);
for(int i=0; i<N; i++)
{
a[i] = -i;
b[i] = i*i*i;
}
```

## Simple Parallelization (4)

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
add<<<N,1>>>(d_a, d_b, d_c);
// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
for(int i=0;i<N;i++) printf("%d+%d = %d\n", a[i], b[i], c[i]);
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c); return 0;
}
```

## GPU Grids, blocks, threads (1)

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

## GPU Grids, blocks, threads (2)

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

- Terminology: a block can be split into parallel threads
- We use threadIdx.x instead of blockIdx.x

# GPU Grids, blocks, threads (3)

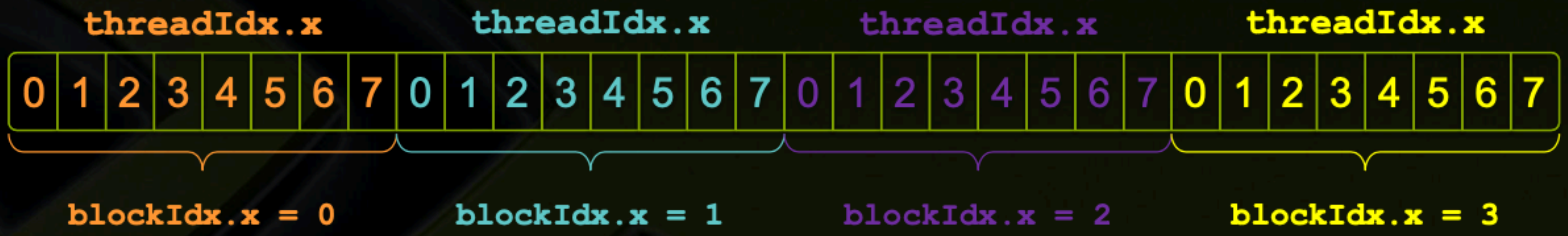
```
__global__ void add(int *a, int *b, int *c) {
 c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

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

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

## Combining GPU Blocks & Threads (1)

- Limit of the number of blocks in a single launch: 65,535.
- Limit of the number of threads per block.
  – For many GPUs, maxThreadsPerBlock = 512 (or 1024, ..).
- Blocks and threads are often combined.
- kernel access: Array[*block_index*+*thread_index*]

# Combining GPU Blocks & Threads (2)

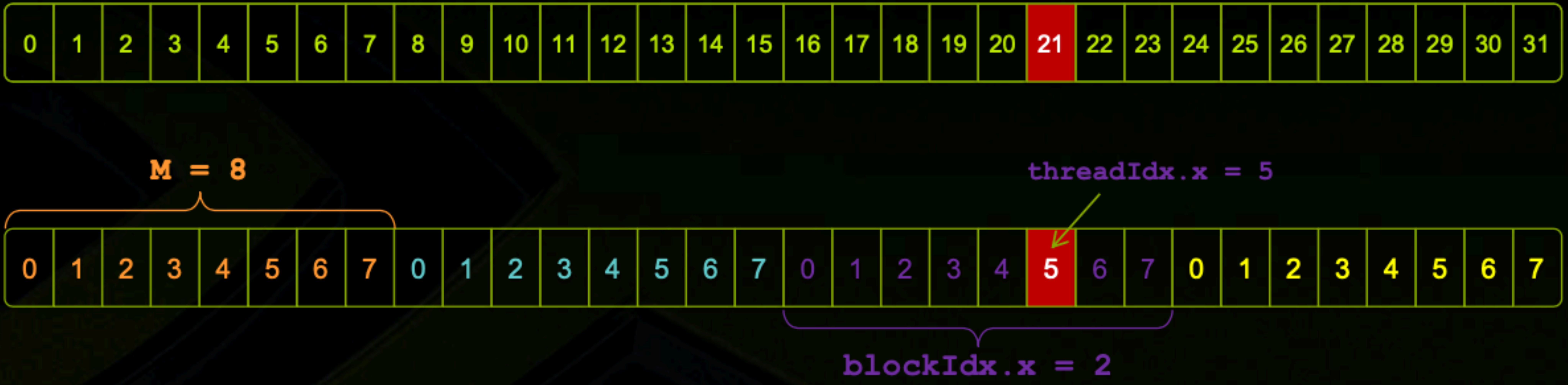
- Consider indexing an array with one element per thread (8 threads/block)



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

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

## Combining GPU Blocks & Threads (3)



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

## Combining GPU Blocks & Threads (4)

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

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

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

#define N (2048*2048) // 2**22
#define THREADS_PER_BLOCK 512
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

## Combining GPU Blocks & Threads (5)

Avoid access beyond the array dimension.

```
__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];
}

#define N (2048*2048)
#define TPB 512
add<<<(N+TPB-1)/TPB,TPB>>>(d_a, d_b, d_c, N);
```

# Reasons of GPU Blocks & Threads

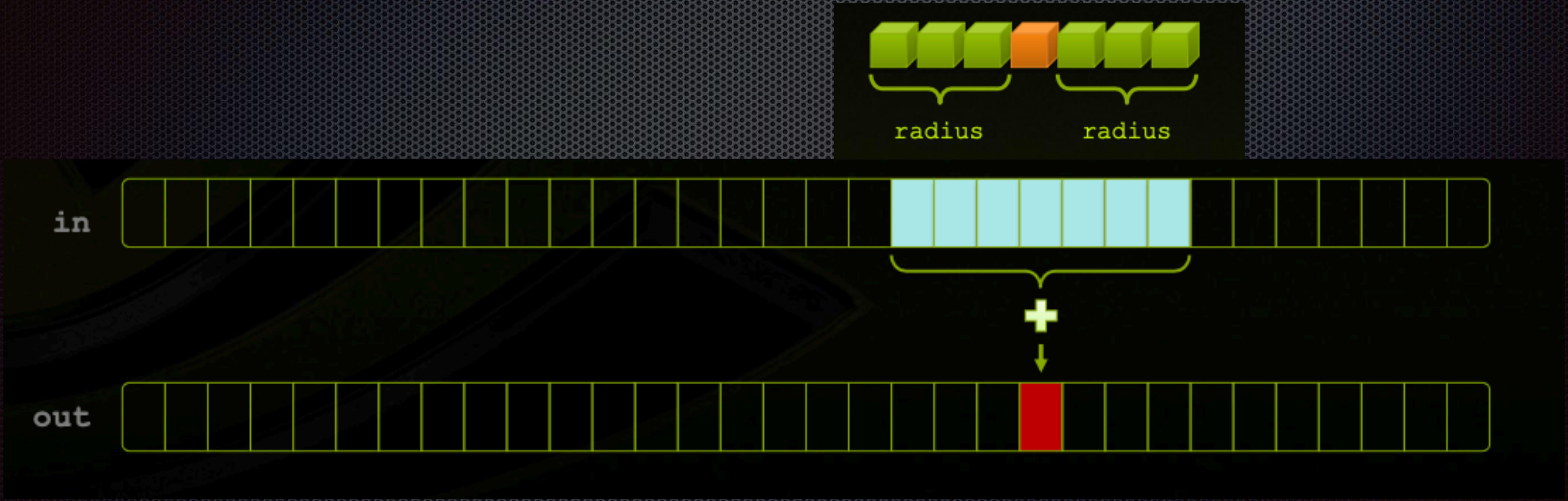
Unlike parallel blocks, threads have mechanisms to efficiently:

- Communicate
- Synchronize

$\longrightarrow$ Sharing between threads

# Sharing between threads (1)

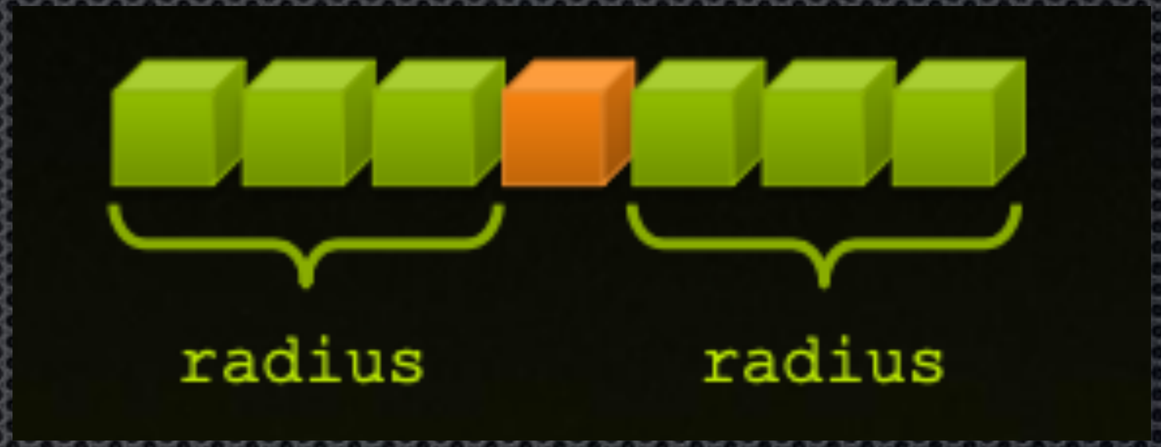
- Example: 1D stencil calculation

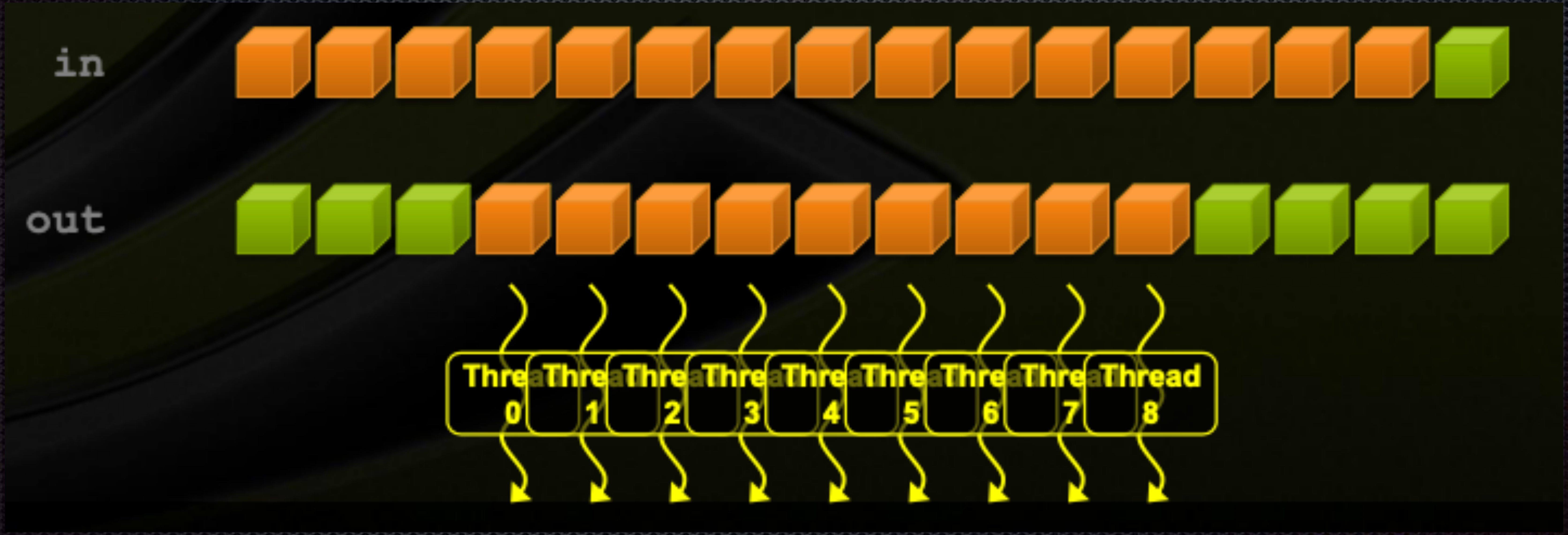


Each output element is the sum of input elements within a radius

# Sharing between threads (2)

- Example: 1D stencil calculation

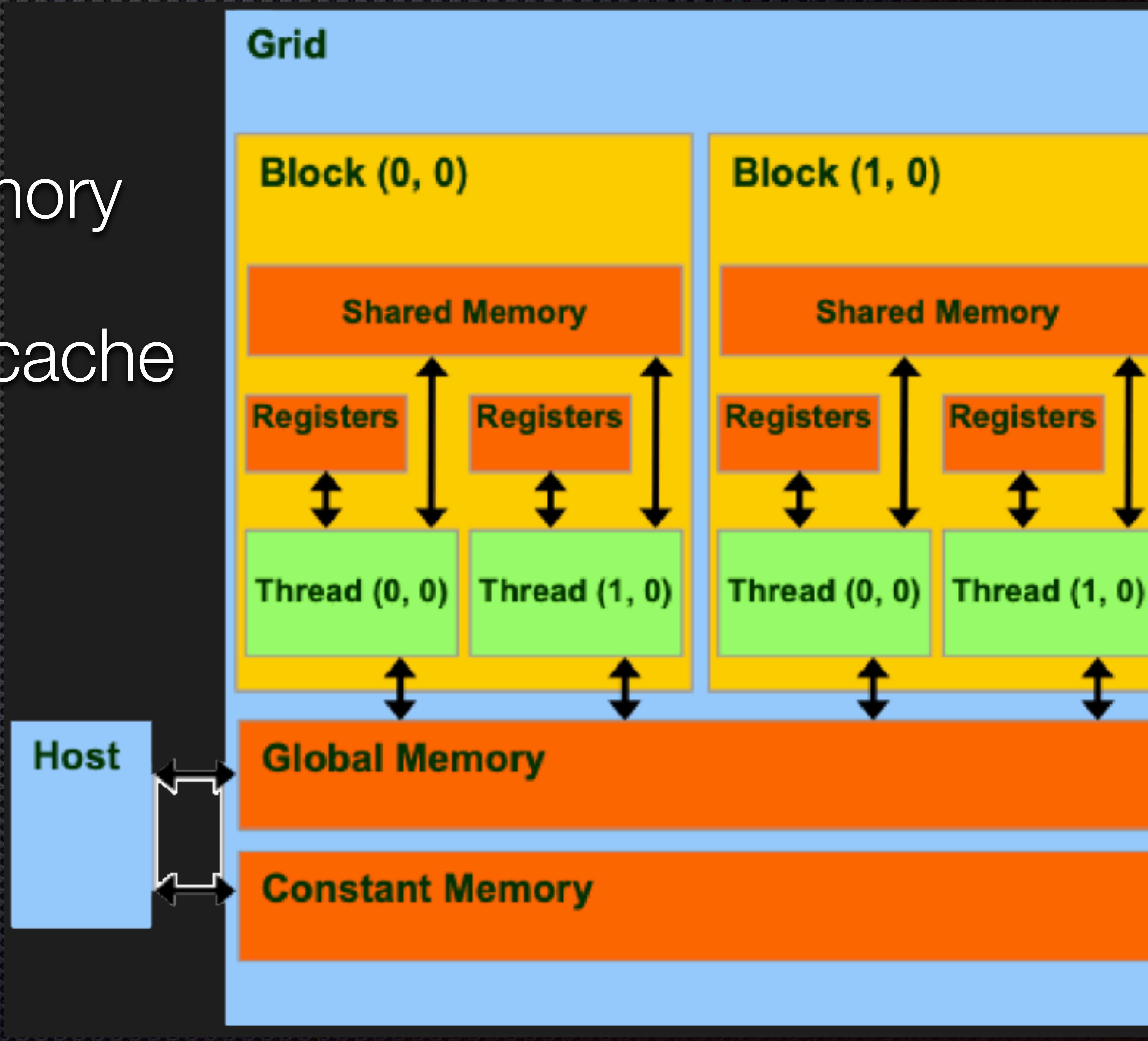




With radius 3, each input element is read seven times

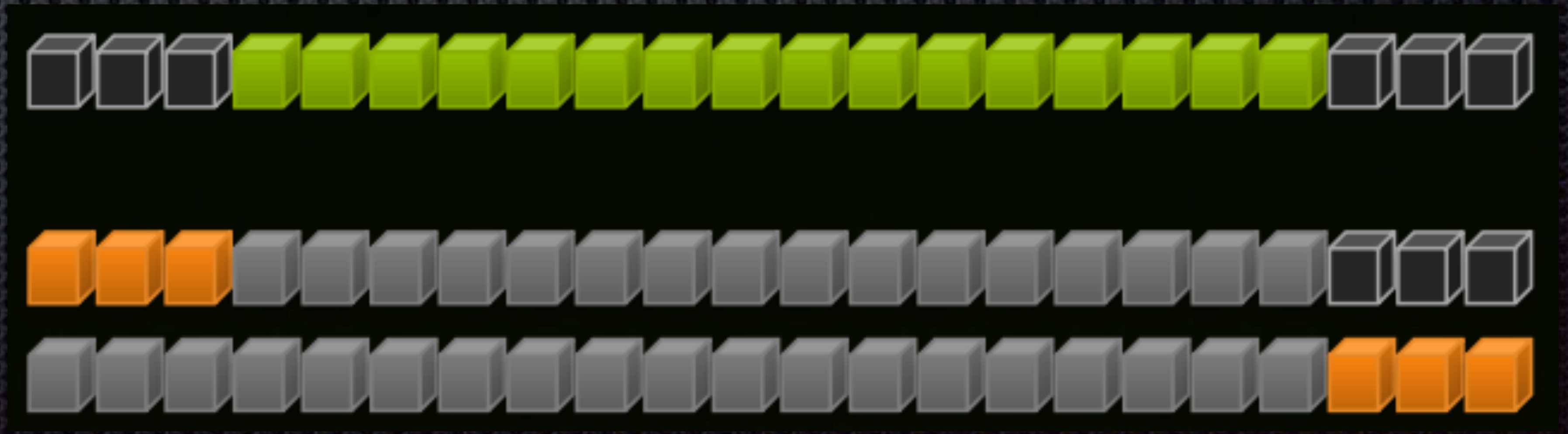
# Sharing between threads (3)

- Within a block, threads share data via shared memory
- Extremely fast on-chip memory: a user-managed cache
- Declare using __shared__, allocated per block
- Not visible to threads in other blocks



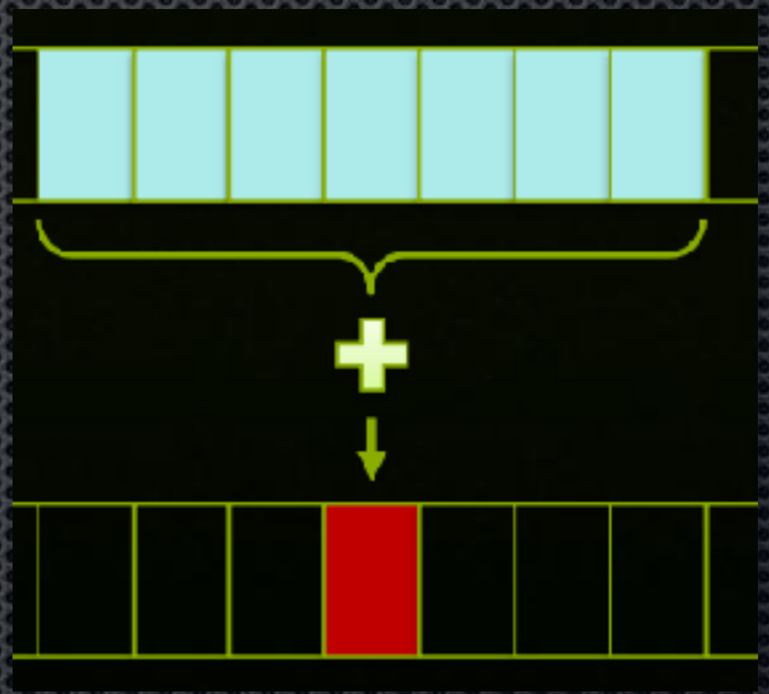
## Sharing between threads (4)

```
__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];
}
```



# Sharing between threads (5)

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

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



# Sharing between threads (6)

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

BUGS

DATA RACE

DATA ACCESS VIOLATION

# Sharing between threads (7)

- `void __syncthreads();`
- Similar to barrier() in C/C++: https://en.wikipedia.org/wiki/Barrier_(computer_science)
- 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

## Sharing between threads (6)

```
// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
temp[lindex - RADIUS] = (gindex >= RADIUS)?in[gindex - RADIUS]:0;
temp[lindex + BLOCK_SIZE] = ((gindex + BLOCK_SIZE)<N)?in[gindex +
BLOCK_SIZE]:0;
}
__syncthreads( );
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
result += temp[lindex + offset];
```

## Performance

- Example is used for its simplicity.
- the overhead of allocating device memory, input data transfer from host to device, output data transfer from device to host, and de-allocating device memory will likely make the resulting code slower than the original sequential code
- This is because the amount of calculation done by the kernel is small relative to the amount of data processed.

## Takeaways

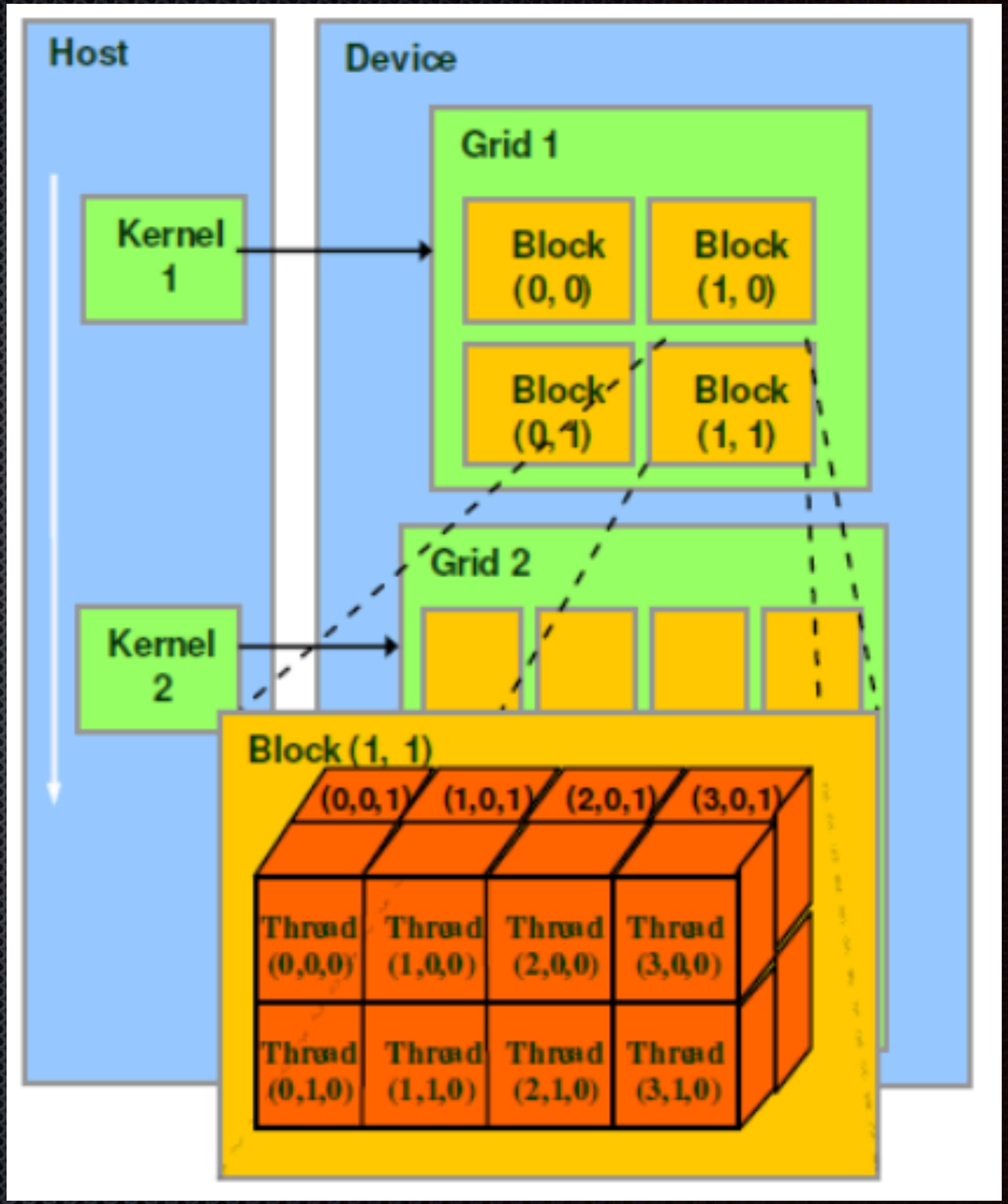
- Launching parallel threads
- Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`

- Allocate elements to threads:

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

- Use `__shared__` to declare a variable/array in shared memory: Data is shared between threads in a block & Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier: Use to prevent data hazards

# Skipped topics



- Multi-dimensional indexing :
  - A kernel is launched as a grid of blocks of threads
    - blockIdx and threadIdx are 3D
    - We showed only one dimension (x)
- Compute capability
- Optimization, reduction operator https://en.wikipedia.org/wiki/Reduction_operator

# CUDA ERROR HANDLING

```
cudaError_t err=cudaMalloc((void **) &d_A, size);
if (error !=cudaSuccess) {
printf(“%s in %s at line %d\n”,
cudaGetErrorString(err),__FILE__,__LINE__);
exit(EXIT_FAILURE);
}
```

Consider to use MACRO version:

```
static void HandleError( cudaError_t err,const char
*file,int line ) {…}

#define HANDLE_ERROR(err) (HandleError( err, __FILE__,
__LINE__ ))
```

## CUDA DEBUG

```
nvcc -g -G add.cu -o add
cuda-gdb ./add
> p blockDim.x
> info cuda threads
> cuda thread 1
> cuda block 1

cuda-memcheck

nvprof

Event Timers
```

Ref: Introduction to CUDA Utilities
https://cis.gvsu.edu/~wolffe/courses/cs677/projects/tutorial_CUDA-utilities.html

## CUDA Best Practices

- Find ways to parallelize sequential code,
- Minimize data transfers between the host and the device,
- Adjust kernel launch configuration to maximize device utilization,
- Ensure global memory accesses are coalesced,
- Minimize redundant accesses to global memory whenever possible,
- Avoid long sequences of diverged execution by threads within the same warp.
- Ref: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

# Rainbow table/hash generator on GPU(1)

| Name | Hash | OSS | Notes | Functionalities |
|---|---|---|---|---|
| Advanced RT Gen | md5 | Open | GPU is only beta tested | RT generator |
| Cryptohaze GPU Rainbow Cracker | MD4,MD5,NTLM,SHA1 | Closed | Full GPU acceleration for table generation (incl. reduction function) and cracking | RT generator, merge, indexing, lookup |
| Rainbow Crackalack | NTLM | Open | Well documented | RT generator, lookup |
| md5-rainbow-table-gen-opencl | md5 | Open | Only PoC | RT generator |
| RainbowCrack | LM, NTLM, MD5, SHA1, SHA256 .. | Closed | GPU supported in rainbow table lookup but not generation | RT generator, merge, indexing, lookup |

# Rainbow table/hash generator on GPU(2)

Recommended Reading

- Steven Meyer EPFL, Breaking 53 bits passwords with Rainbow tables using GPUs https://docplayer.net/50461249-Breaking-53-bits-passwords-with-rainbow-tables-using-gpus.html

- Russell Edward Graves, Iowa State University, High performance password cracking by implementing rainbow tables https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=2860&context=etd

## Rainbow table/hash generator on GPU(2)

Recommended references

- Rainbow2 Rainbow tables using Nvidia GPU: https://github.com/voyager23/Rainbow2
- NTHashTickler_CUDA. NT hash bruteforcer on CUDA. https://github.com/ryanries/NTHashTickler_CUDA

## Assignments

- Installation of CUDA Toolkit
- Parallelise hash computing

## References

- CUDA C/C++ Basics NVIDIA
- ACMS 40212/60212: Advanced Scientific Computing, U.ND.