

An Introduction to GPU and CUDA C/C++

Slides adapted by Dr Sparsh Mittal

Courtesy for slides: NVIDIA, Mutlu/Kirk/Hwu, Michael
Boyer and others

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.
- We discuss CUDA C/C++ and GPU architecture (briefly)

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

Architectural parameters of recent NVIDIA GPUs

CC = compute capability

RF = register file

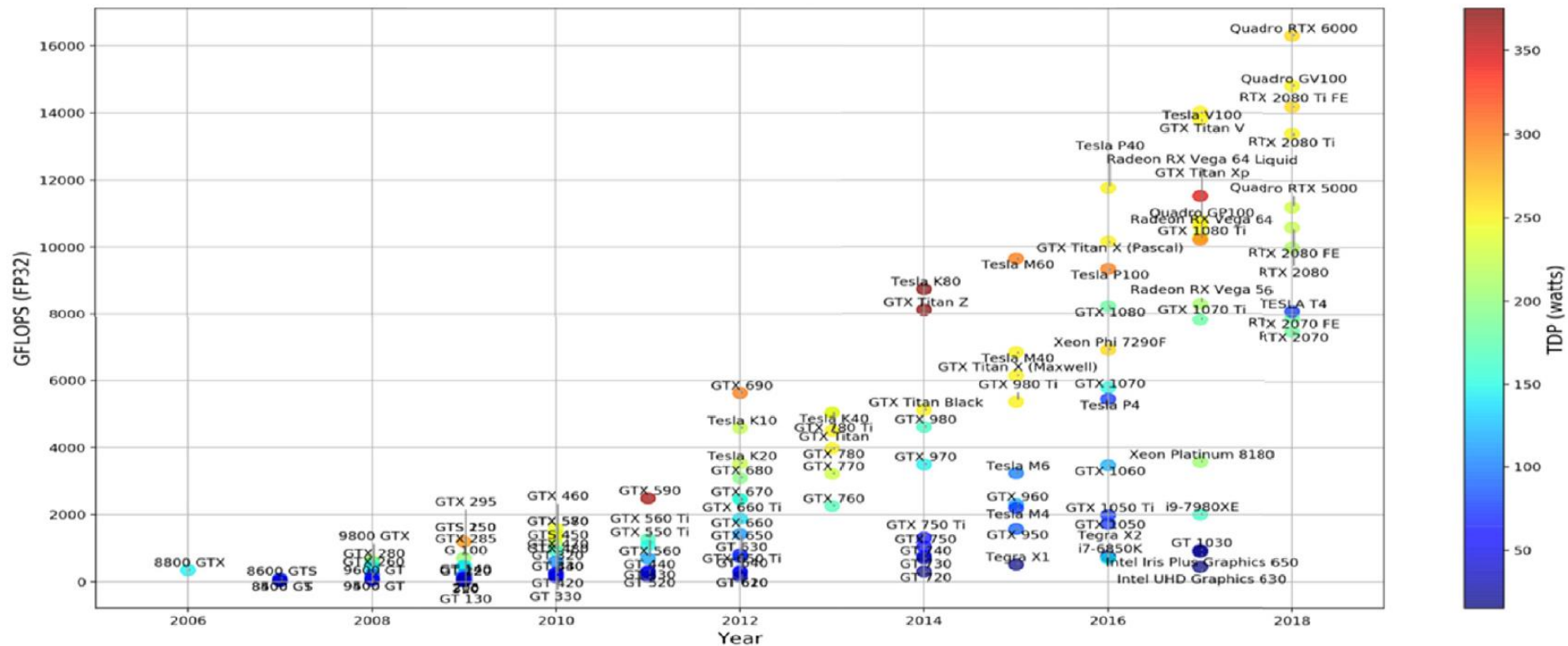
	Architecture	#Tran.	Node	CC	Per-SM L1 size (KB)	Shared L2 size (KB)	Per-SM RF size (KB)	# of SMs	Total RF size (KB)
G80	Tesla	0.68B	90nm	1.0	None	None	32	16	512
GT200	Tesla	1.4B	65nm	1.3	None	None	64	30	1920
GF100	Fermi	3B	40nm	2.0	48	768	128	16	2048
GK110	Kepler	~7B	28nm	3.5	48	1536	256	15	3840
GK210	Kepler	~7B	28nm	3.7	48	1536	512	15	7680
GM204	Maxwell	8B	28nm	5.2	48	2048	256	16	4096
GP100	Pascal	15.3B	16nm	6.0	48	4096	256	56	14336
GV100	Volta	21B	12nm	7.0	128	6144	256	80	20480
TU102	Turing	18.6B	12nm	7.5	64	6144	256	72	18432

This PPT applies to devices with capability ≥ 2.0

Year	GPU Name	Cores	GFLOPS	Memory	Bandwidth	CPU Name	Cores	GFLOPS	Bandwidth
2006	Tesla G80	128	346	768 MB	86 GB/second	Core	6	60	8 GB/second
2008	Tesla GT200	240	708	1024 MB	159 GB/second	Nehalem	8	90	10 GB/second
2010	Fermi	512	1581	1536 MB	192 GB/second	Westmere	10	112	24 GB/second
2012	Kepler	2304	3976	3072 MB	288 GB/second	Sandy Bridge	12	480	50 GB/second
2014	Maxwell	3072	6144	12,288 MB	336 GB/second	Ivy Bridge	15	510	58 GB/second
2016	Pascal	3584	11,366	16,384 MB	547 GB/second	Broadwell	24	1340	64 GB/second
2018	Volta	5120	15,154	16,384 MB	900 GB/second	Coffee Lake	32	2010	96 GB/second

Cano, “A survey on graphic processing unit computing for large-scale data mining”, WIDM 2017

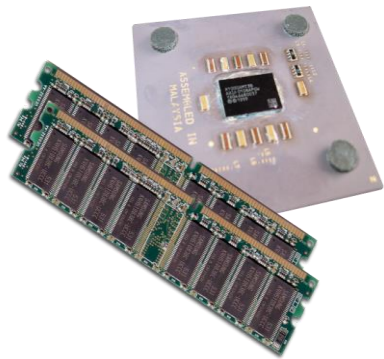
Power and Performance of GPUs



HETEROGENEOUS COMPUTING

Heterogeneous Computing

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

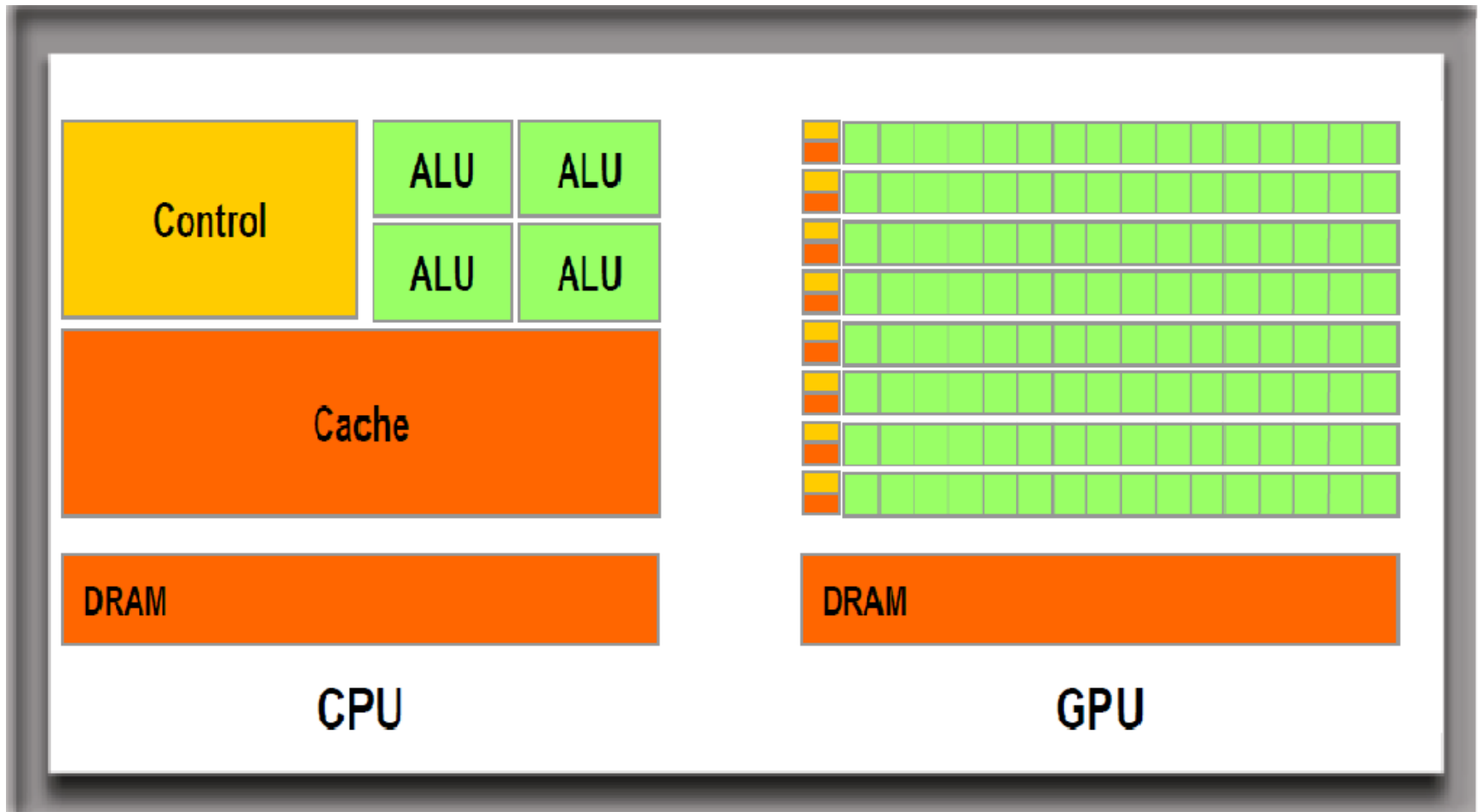


Host



Device

GPU vs. CPU



“The Tradeoff”

Optimizes
LATENCY



CPU

Optimizes
THROUGHPUT

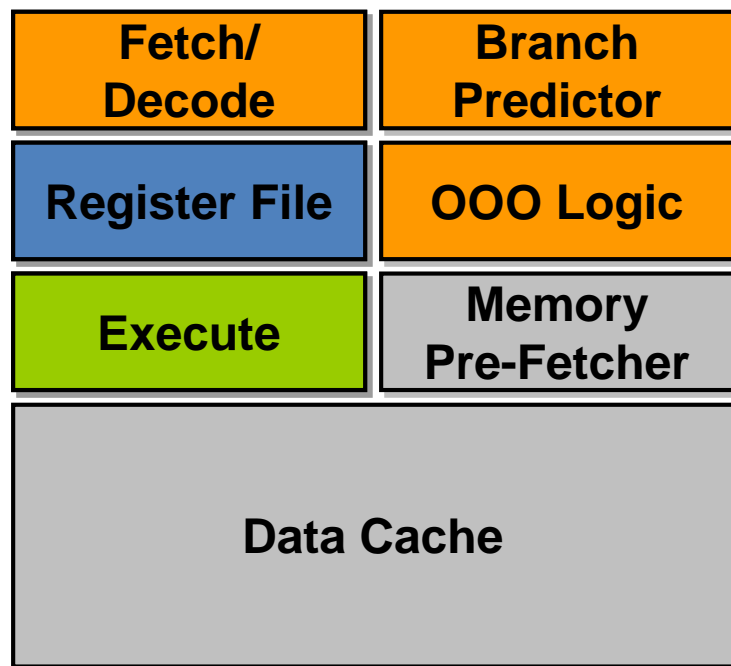


GPU

CPU vs. GPU: Architectural Difference 1

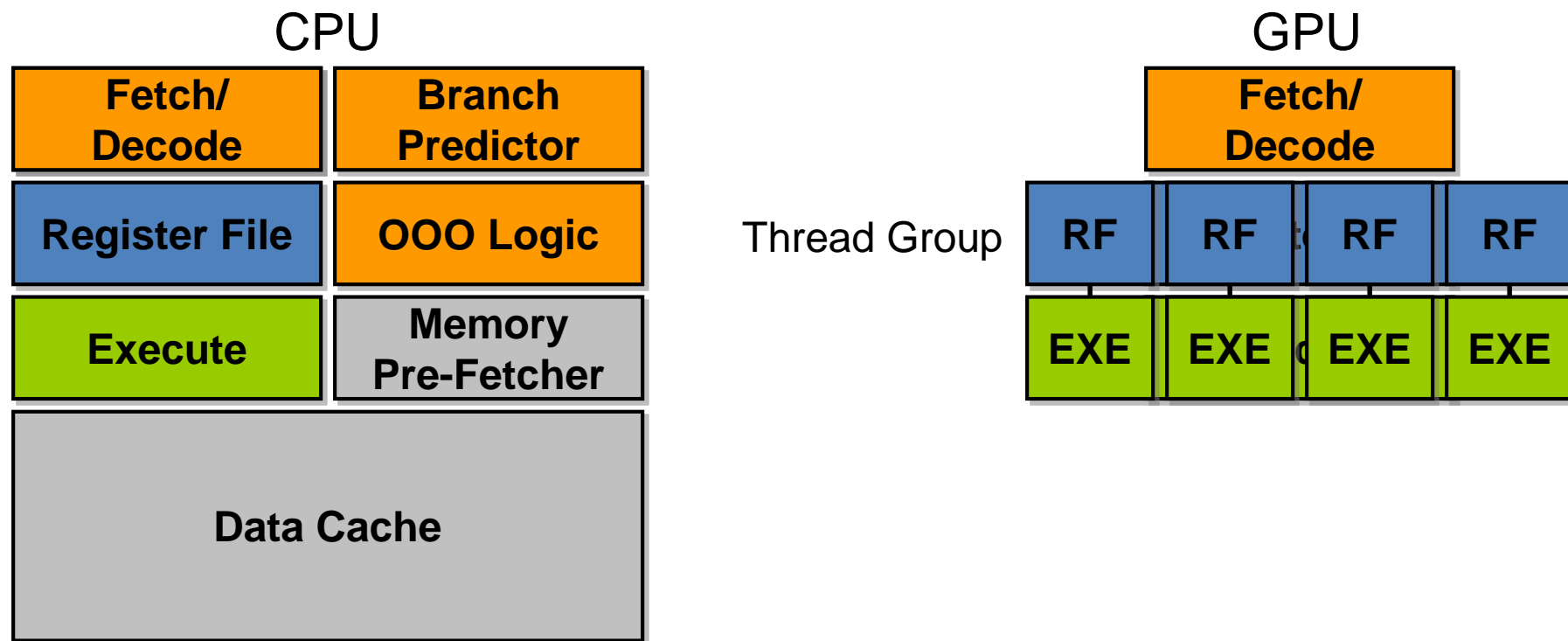
CPU

GPU



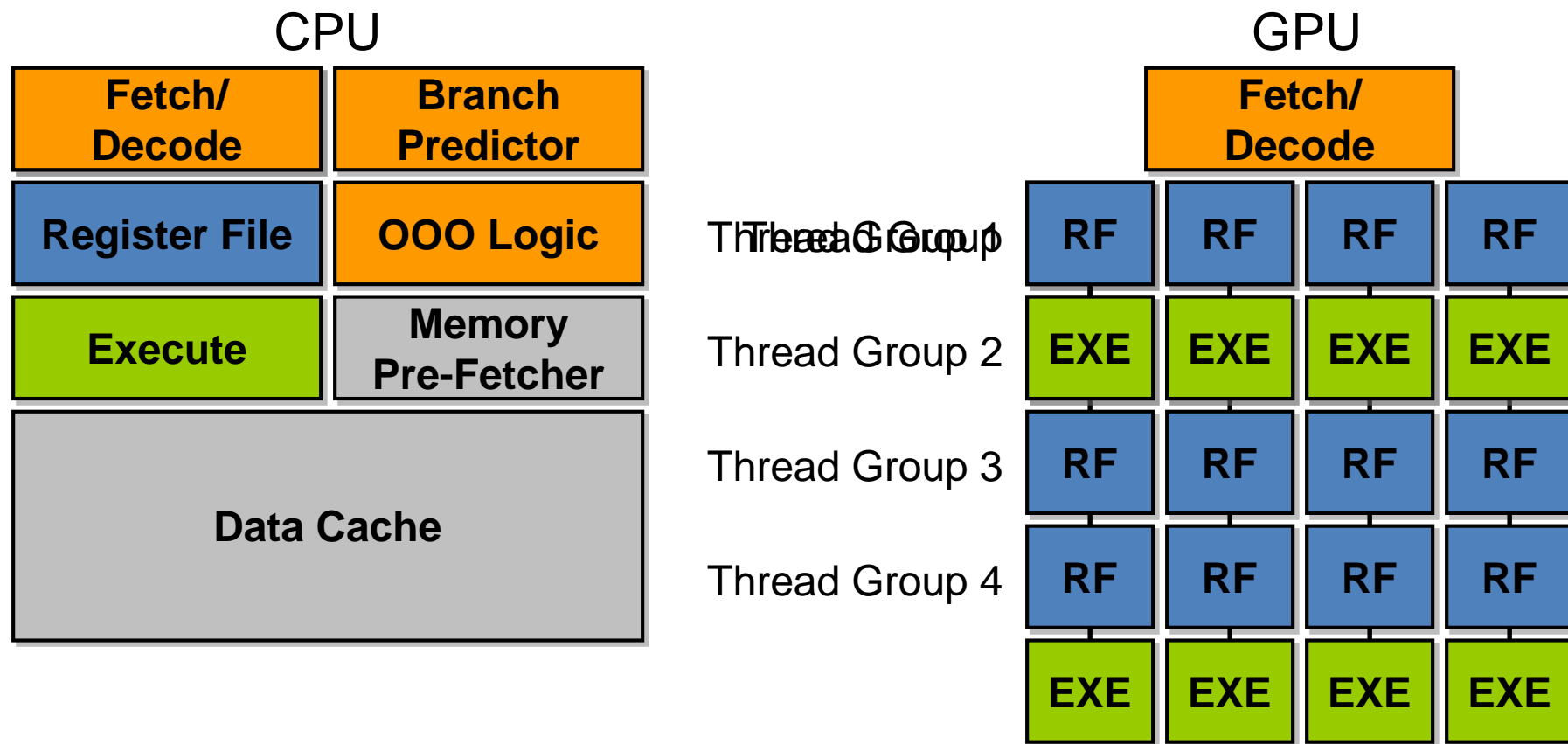
Avoid structures that only improve single-thread performance

CPU vs. GPU: Architectural Difference 2



Amortize the overhead of control logic across multiple execution units (SIMD processing)

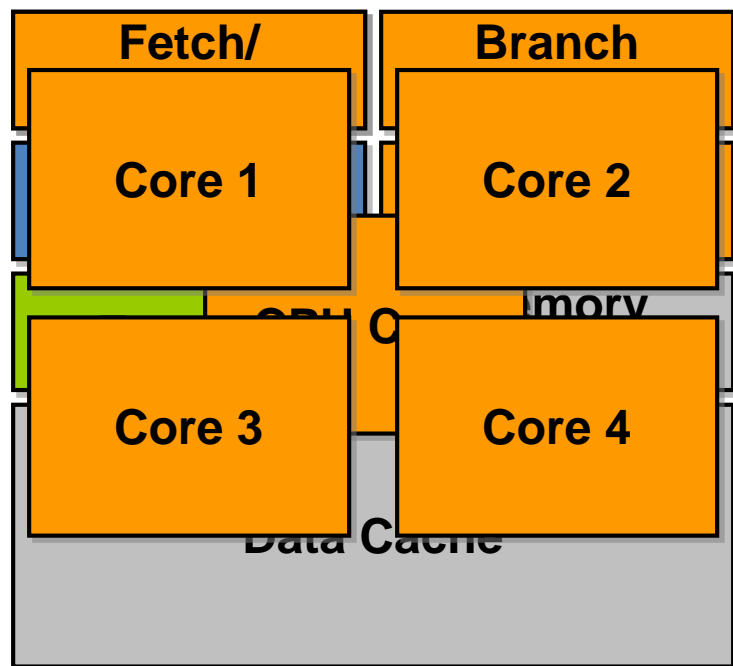
CPU vs. GPU: Architectural Difference 3



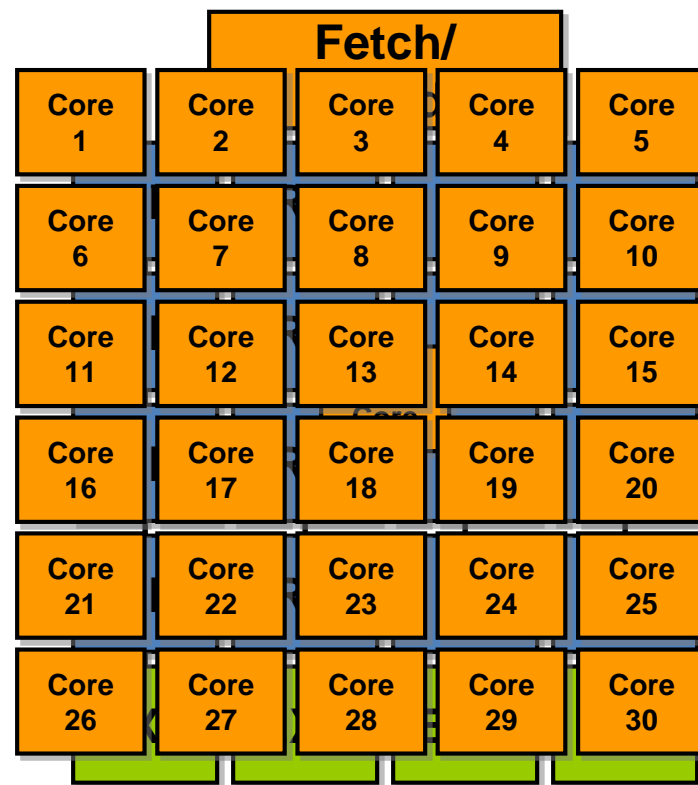
Use multiple groups of threads to keep execution units busy and hide memory latency

CPU vs. GPU: Architectural Difference 4

CPU



GPU



Replicate cores to leverage more parallelism

CPU vs. GPU: Architectural Differences

- Summary: take advantage of abundant parallelism
 - Lots of threads, so focus on aggregate performance
 - Parallelism in space:
 - SIMD processing in each core
 - Many independent SIMD cores across the chip
 - Parallelism in time:
 - Multiple SIMD groups in each core

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);

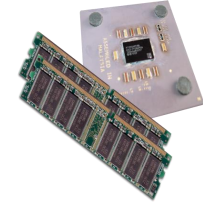
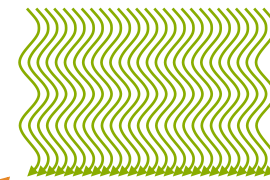
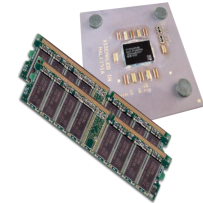
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

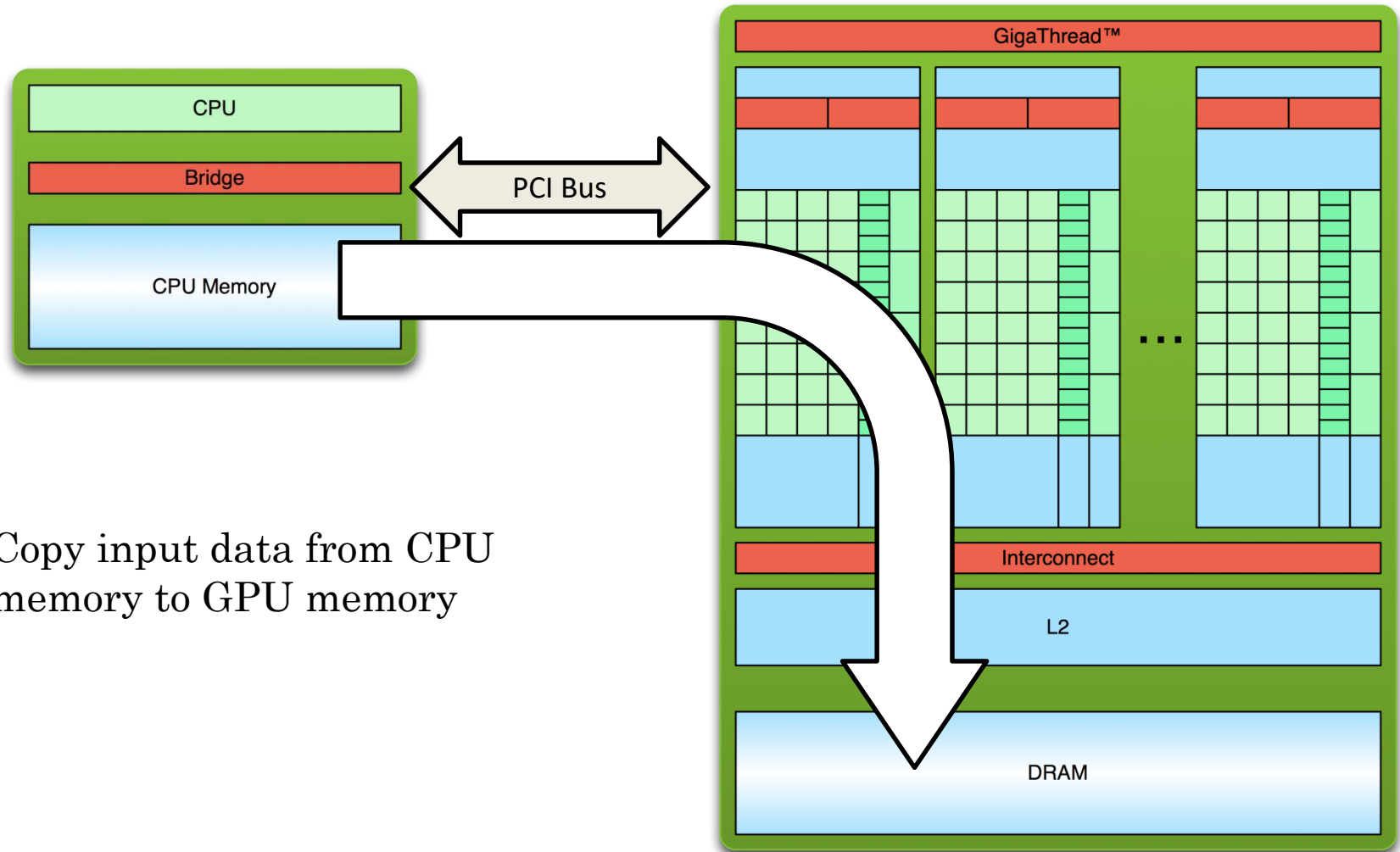
serial code

parallel code

serial code

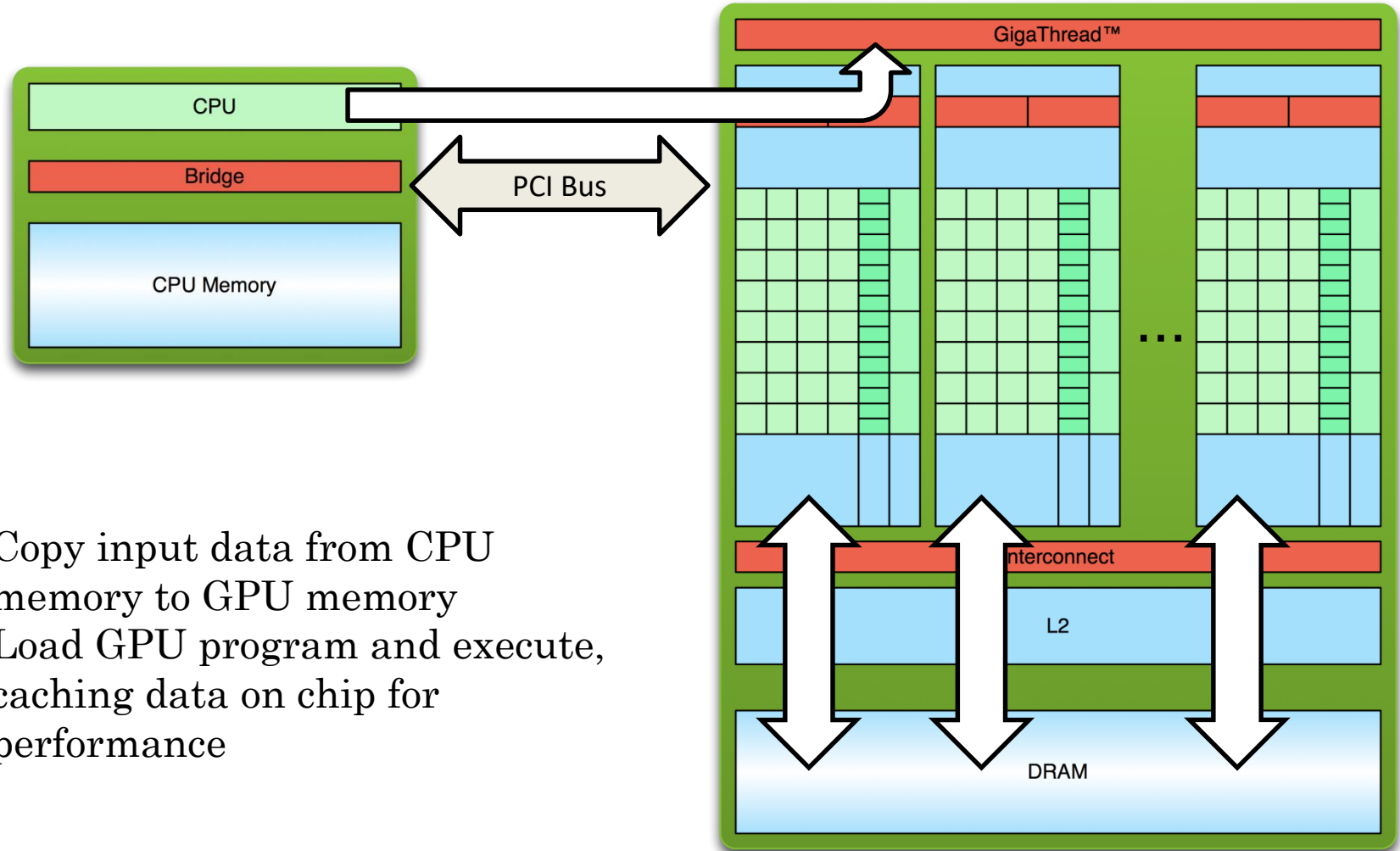


Simple Processing Flow

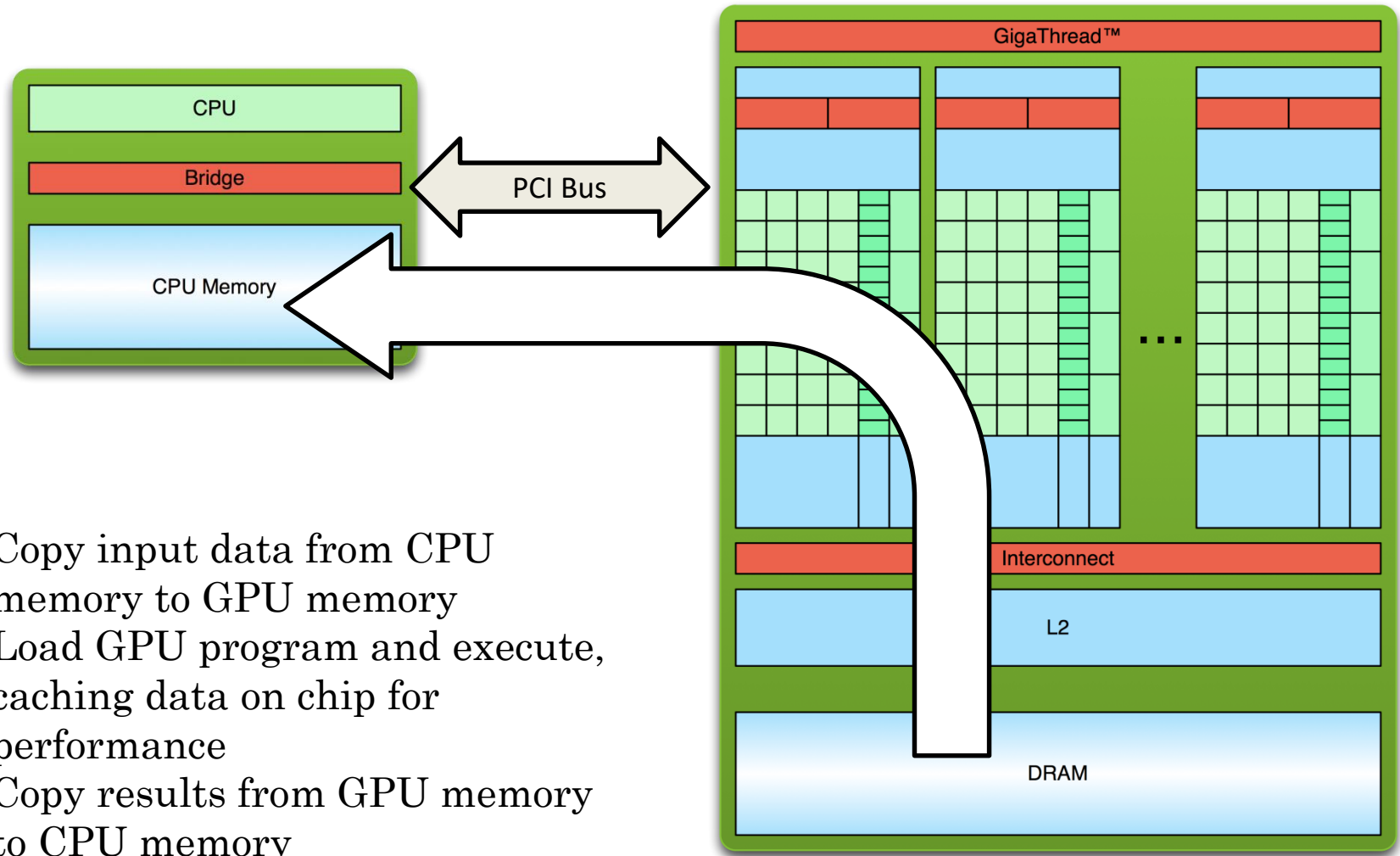


1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



Simple Processing Flow



CUDA extension to declare functions

__global__ called only from host
executes only on device

__device__ called only from device
executes only on device

__host__ called only from host
executes only on host

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 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!  
$
```

- In this example, `mykernel()` does nothing

Printing Hello World from Device

```
//filename helloPrintFromDevice.c
#include <stdio.h>
__device__ const char *STR = "HELLO WORLD!";
const char STR_LENGTH = 12;

__global__ void hello()
{
    printf("%d %c\n", threadIdx.x, STR[threadIdx.x %
STR_LENGTH]);
}
int main(void){
    int num_threads = STR_LENGTH;
    int num_blocks = 1;
    hello<<<num_blocks,num_threads>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Output

```
$ nvcc helloPrintFromDevice.cu
```

```
$ ./a.out
```

```
0 H
```

```
1 E
```

```
2 L
```

```
3 L
```

```
4 O
```

```
5
```

```
6 W
```

```
7 O
```

```
8 R
```

```
9 L
```

```
10 D
```

```
11 !
```

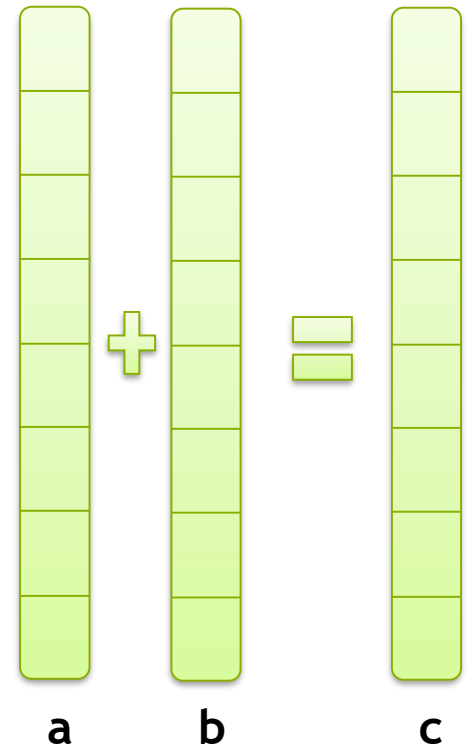
```
$
```



Each thread prints one character

Parallel Programming in CUDA C/C++

- GPU computing is about massive parallelism!
- We will discuss 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()`...

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  
    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);

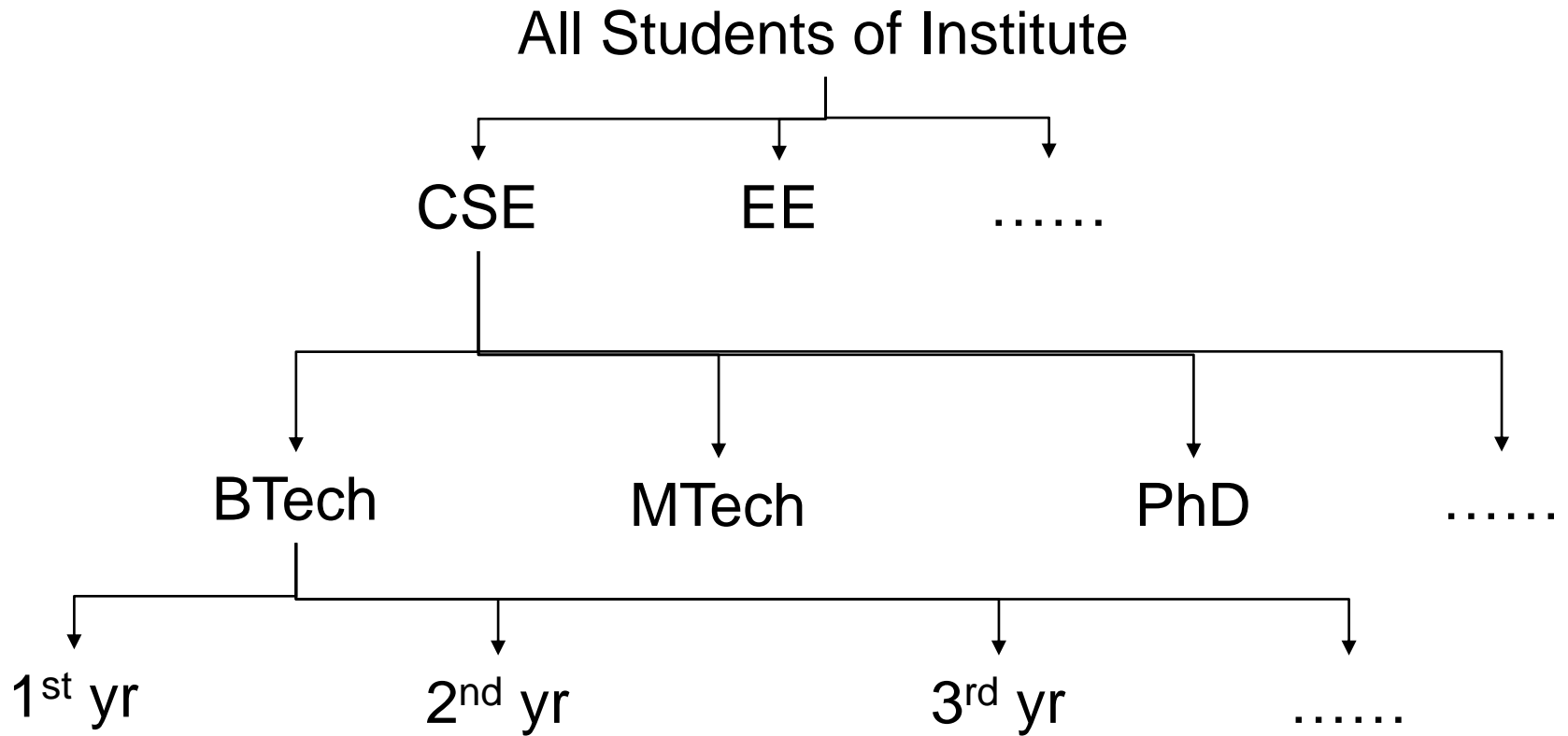
// Launch add() kernel on GPU
    add<<<1, 1>>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

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

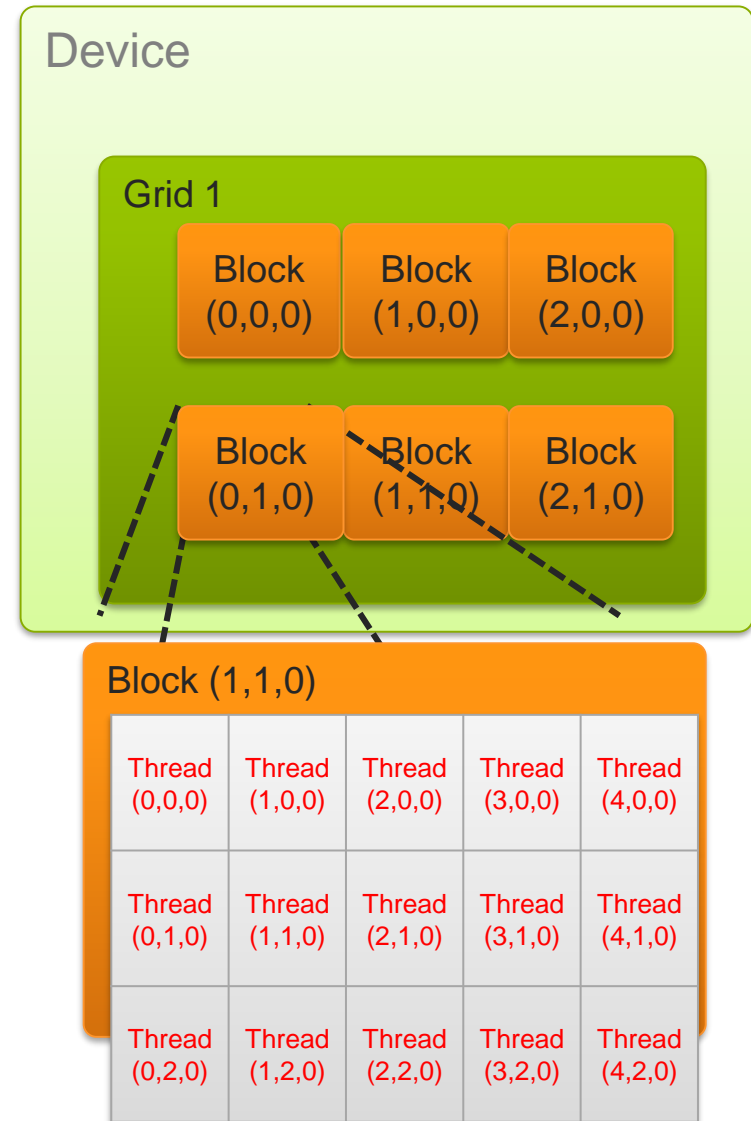
UNDERSTANDING THREAD ORGANIZATION

Understanding thread organization using example of student groups



Similarly, threads are organized

- 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`




Parallel computing using

BLOCKS

Moving from Scalar 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()`

```
#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
    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 and initialize
    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);
}
```

Parallel computing using

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()`

```
#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 and initialize
    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);
```

```
}
```

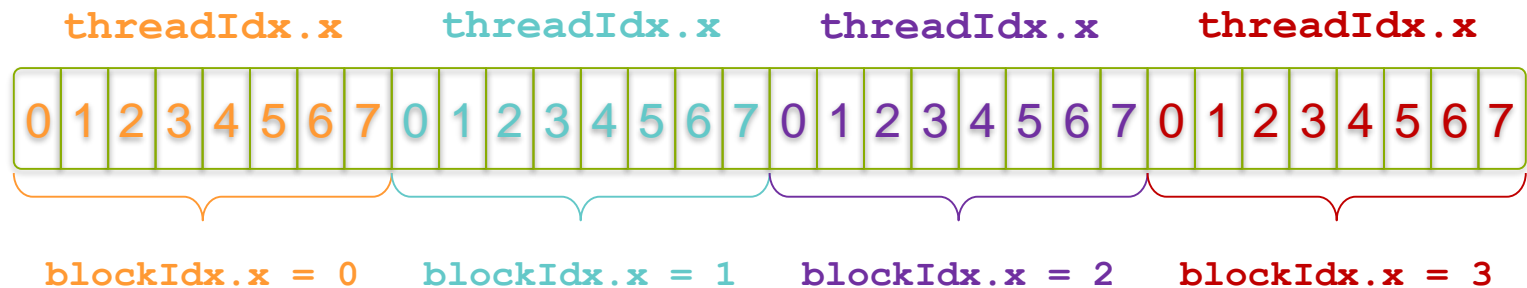

COMBINING BLOCKS AND THREADS

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 using Blocks & Threads

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

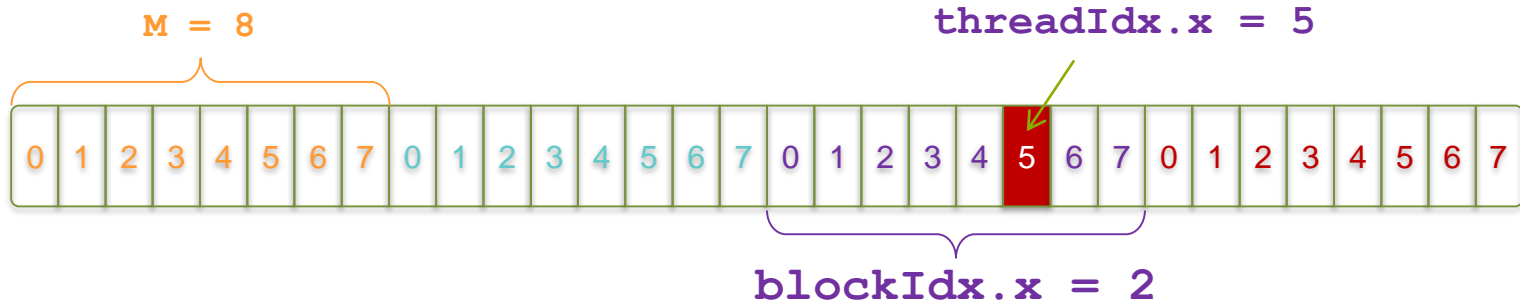


- With M threads/block a unique index for each thread is given by:

```
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]; }
```

Addition with Blocks and Threads

```
#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
    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);
    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

// 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);
```

```
}
```

Handling Arbitrary Vector Sizes

- Typical problems: non-multiples of `blockDim.x`
- Avoid accessing beyond 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

- Device number: 0, Device name: Quadro P1000, Compute capability: 6.1
Clock Rate: 1480500 kHz, Total SMs: 5
Shared Memory Per SM: 98304 bytes
Registers Per SM: 65536 32-bit
Max threads per SM: 2048
L2 Cache Size: 1048576 bytes
Total Global Memory: 4227858432 bytes
Memory Clock Rate: 2505000 kHz

Max threads per block: 1024

Max threads in X-dimension of block: 1024

Max threads in Y-dimension of block: 1024

Max threads in Z-dimension of block: 64

Max blocks in X-dimension of grid: 2147483647

Max blocks in Y-dimension of grid: 65535

Max blocks in Z-dimension of grid: 65535

Shared Memory Per Block: 49152 bytes

Registers Per Block: 65536 32-bit

Warp size: 32

Hardware limits on GPU (for an old GPU)

- Grid and block dimension restrictions
 - Grid: 64k x 64k
 - Block: 512x512x64
 - Max threads/block = 512
- A block maps onto an SM
 - Up to 8 blocks per SM
- Every thread uses registers
 - Up to 16K registers in an SM
 - There is also limit on max register per thread
- Every block uses shared memory
 - Up to 16KB shared memory

Example

Assume blocks of 16x16 threads using 20 registers each

- Each block uses 4K of shared memory
- Find the limit on the maximum

- 5120 registers / block \rightarrow **3.2 blocks/SM**
- 4K shared memory/block \rightarrow 4 blocks/SM

Lets first discuss

GPU MEMORY ADDRESS SPACES

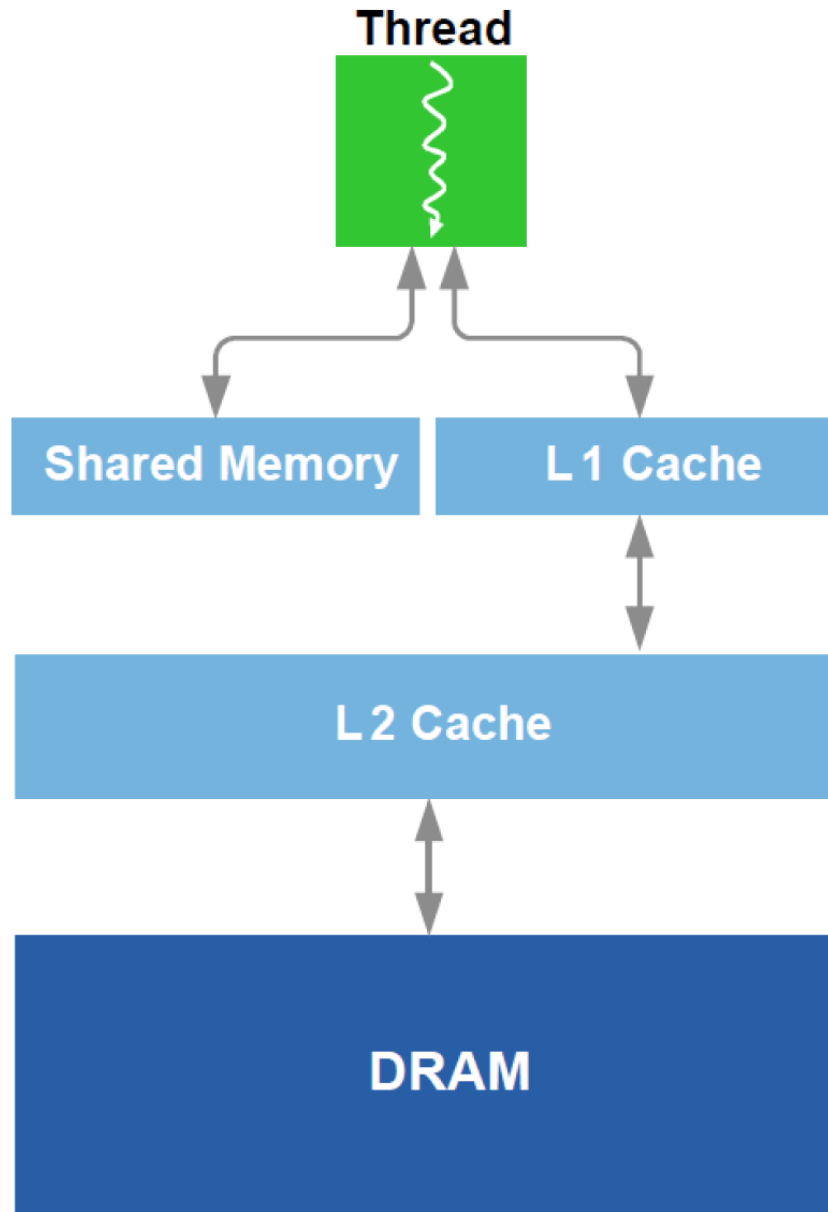
GPU Memory Address Spaces

1. Local
 2. Shared
 3. Global
- 
- Increasing visibility of data between threads

- In addition there are two more (read-only) address spaces:

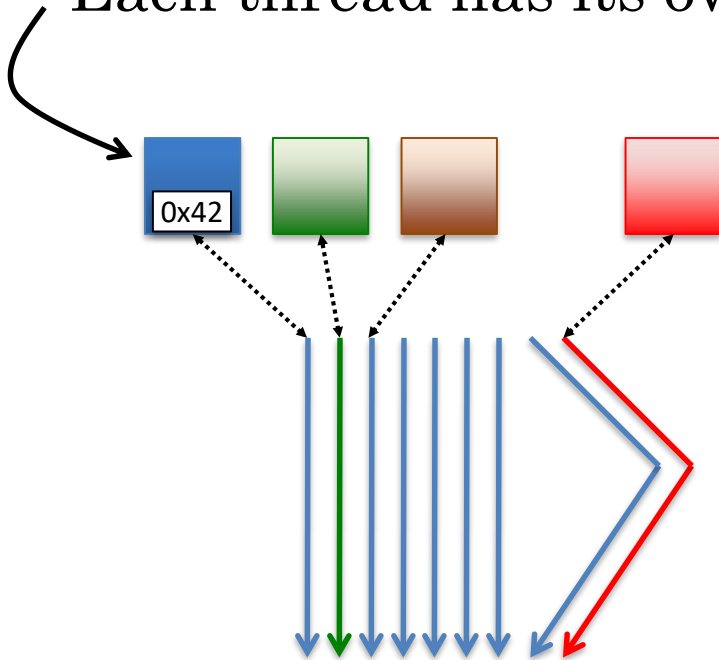
1. Constant
2. Texture.

Fermi Memory Hierarchy



Local (Private) Address Space

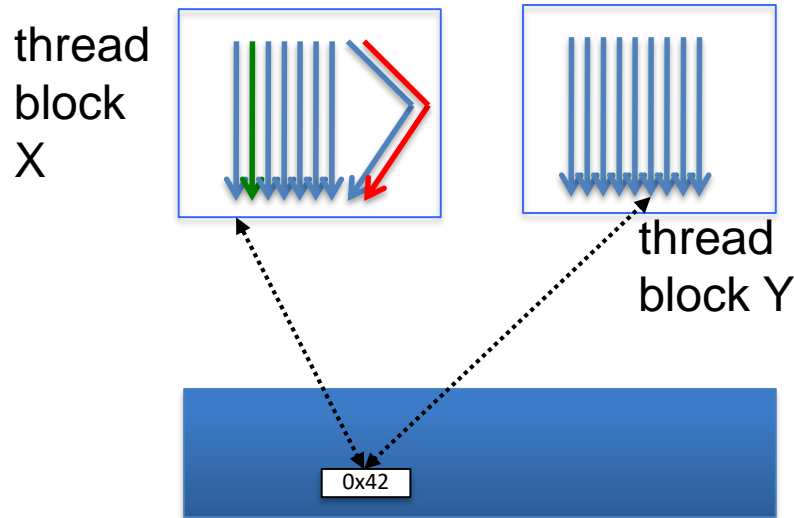
Each thread has its own “local memory”



Note: Location at address 100 for thread 0 is different from location at address 100 for thread 1.

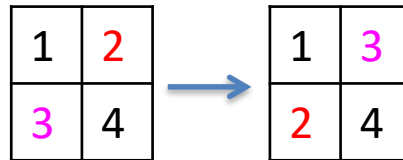
Contains local variables private to a thread.

Global Address Spaces



- Each thread in the different thread blocks (even from different kernels) can access “global memory”
- **cudaMalloc** allocates global memory
- Threads write their own portion of global memory
- Slow

Lets take example of Matrix Transpose

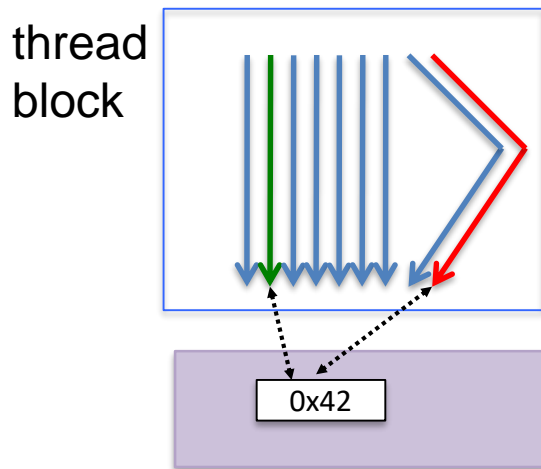


Matrix Transpose

```
__global__ void transpose(float *odata, float* idata, int width, int height){  
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;  
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;  
  
    int index_in  = xIndex + width * yIndex;  
    int index_out = yIndex + height * xIndex;  
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {  
        odata[index_out+i] = idata[index_in+i*width];  
    }  
}
```

- “xIndex”, “yIndex”, “index_in”, “index_out”, and “i” are in local memory (local variables are register allocated, stack is allocated in local memory)
- “odata” and “idata” are pointers to global memory (both allocated using calls to cudaMalloc -- not shown above)

Shared Address Space



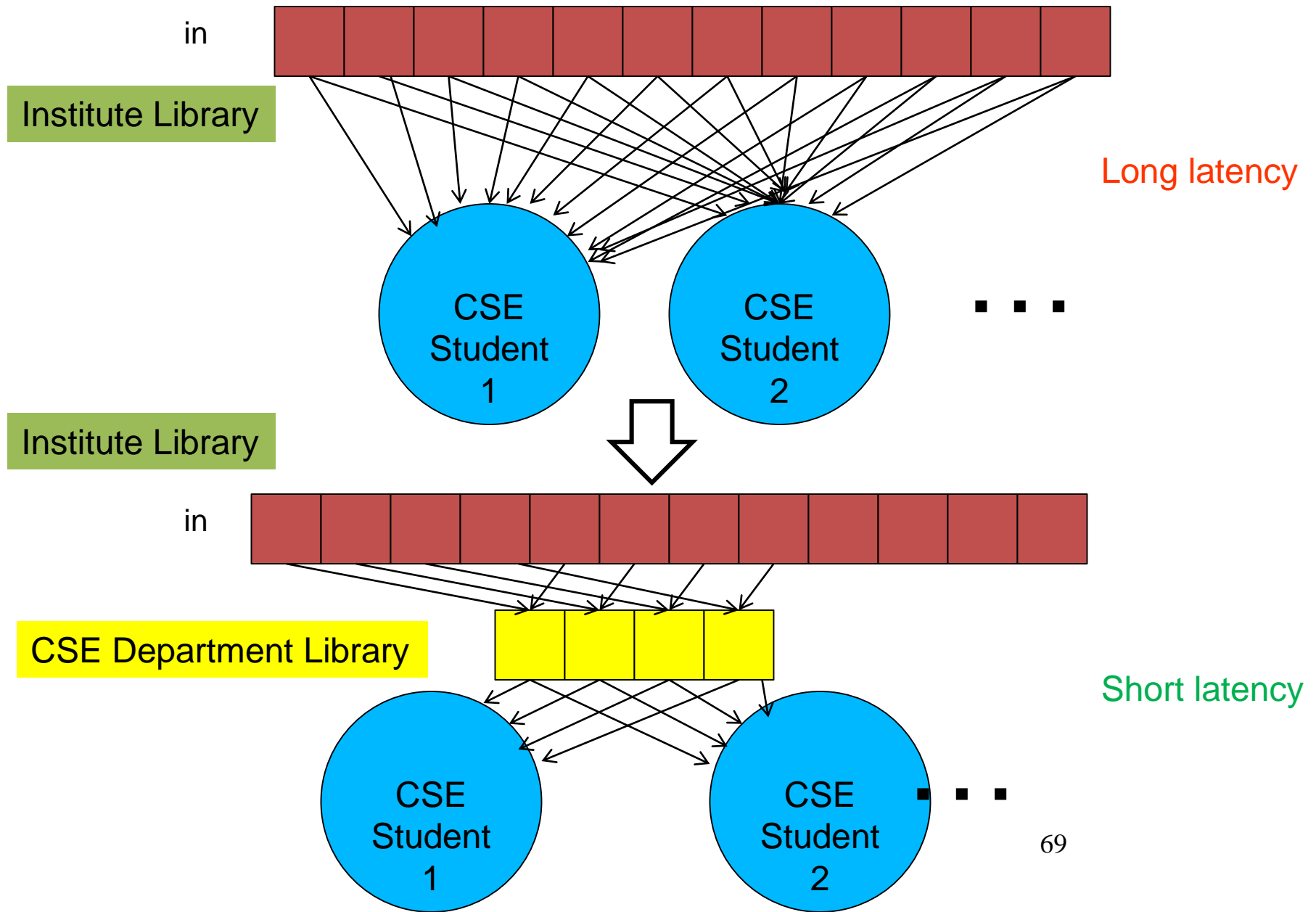
Each thread in the same block can access a memory region called “shared memory”

Limited size (16 to 48 KB).

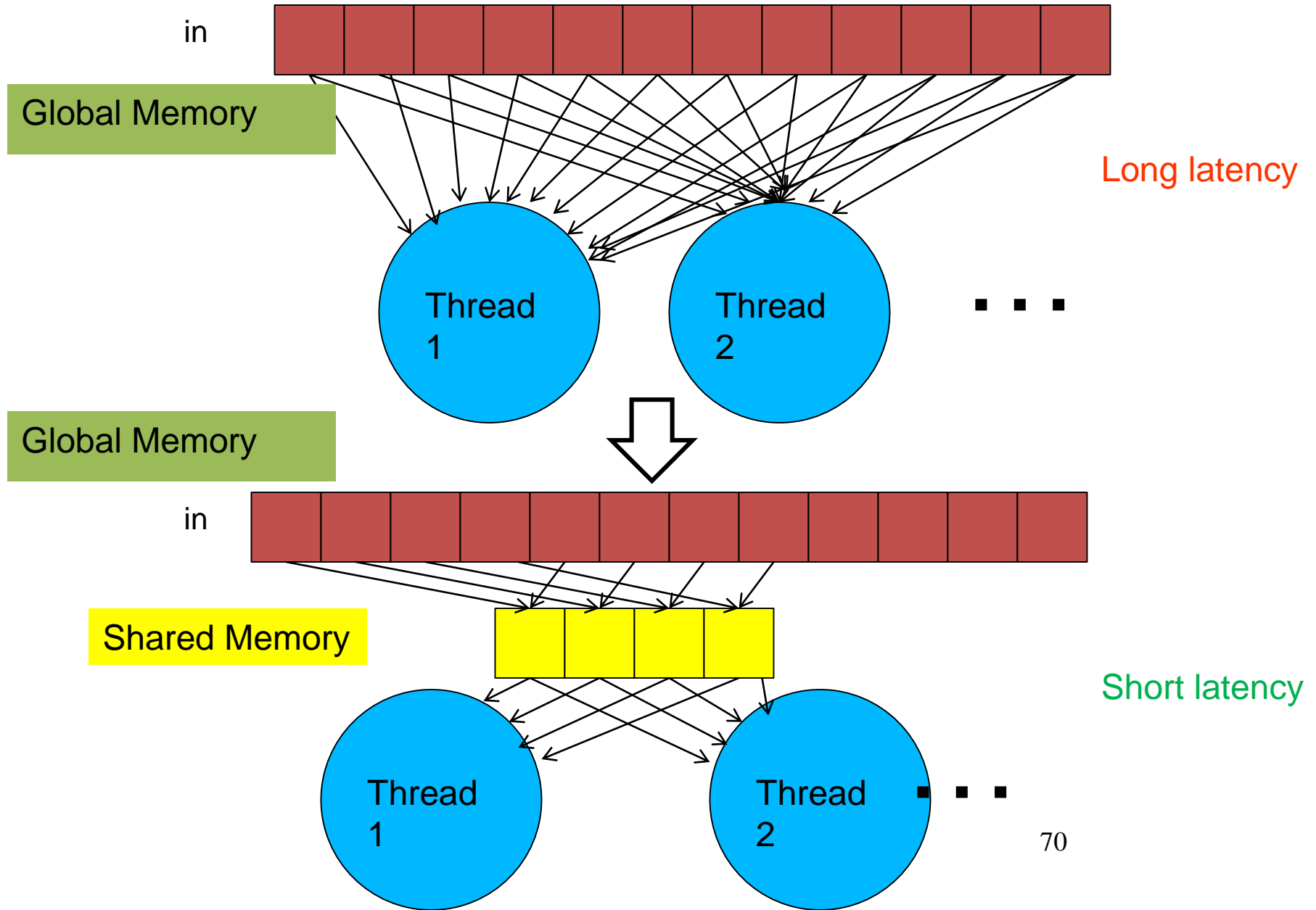
Used as a software managed “cache” to avoid off-chip memory accesses.

Synchronize threads in a thread block using `__syncthreads()`;

Analogy: Institute and Dept. Library



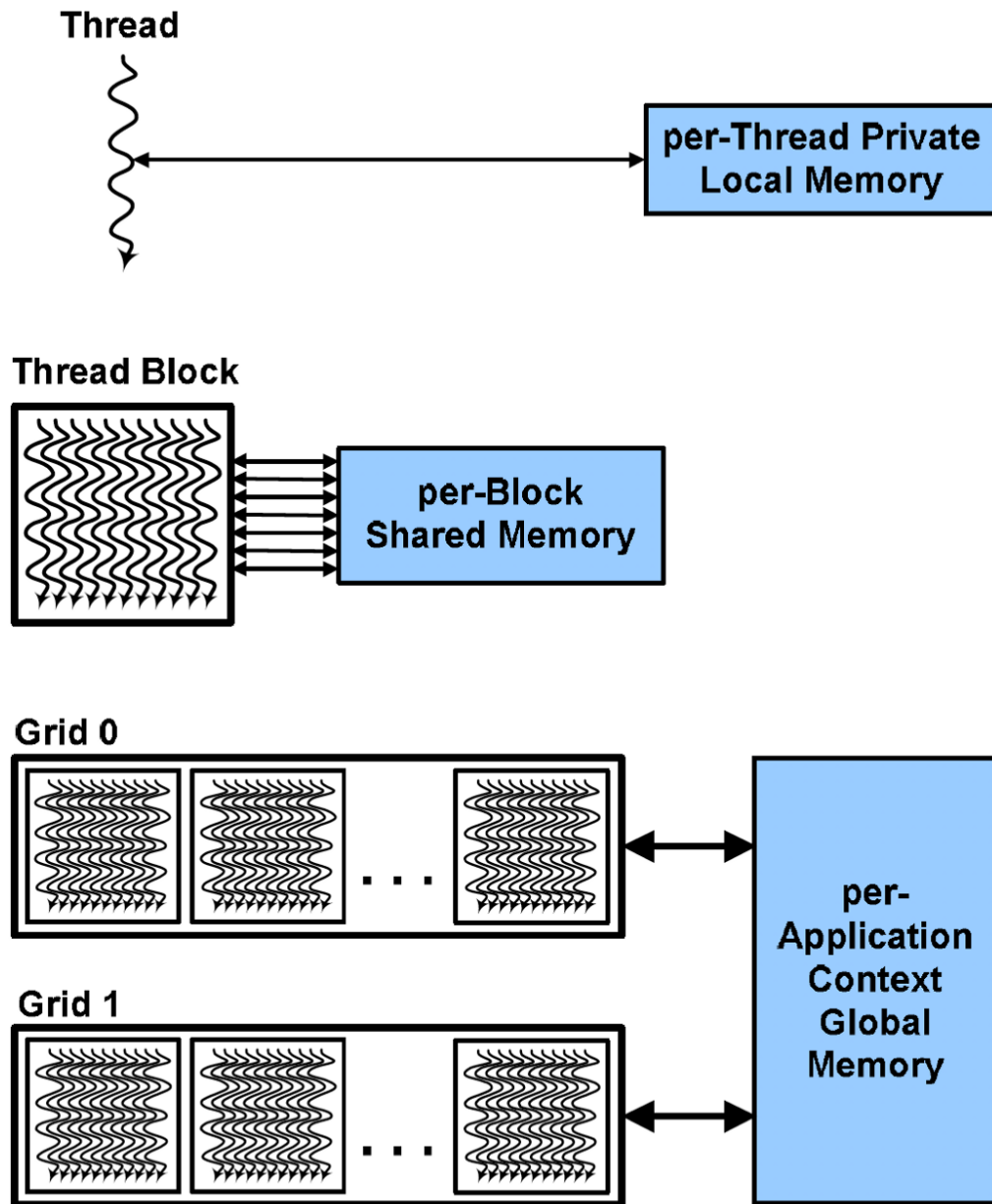
Similarly: Global and Shared memory



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime	Latency
<code>int LocalVar;</code>	register	thread	thread	1x
<code>int localArray[10];</code>	local	thread	thread	100x
<code>__shared__ int SharedVar;</code>	shared	block	block	1x
<code>__device__ int GlobalVar;</code>	global	grid	application	100x
<code>__constant__ int ConstVar;</code>	constant	grid	application	1x

- Automatic variables without any qualifier reside in a register
 - Except per-thread arrays that reside in local memory
 - Or if there are not enough registers



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

Programming scenario 1

Task:

Load data from global memory

Do **thread-local** computations

Store results to global memory

Solution:

Load data from global memory

```
float a = d_ptr[blockIdx.x * blockDim.x + threadIdx.x];
```

- Do computation with registers

```
float res = func(a)
```

- Store result

```
d_ptr[blockIdx.x*blockDim.x + threadIdx.x] =res;
```

Programming scenario 2

Task: 1. Load data from global memory 2. Do **block-local** computations 3. Store results to global memory

Solution:

Load data from global memory to shared memory

```
__shared__ float a_sh [ BLOCK_SIZE ];
```

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

```
a_sh [ threadIdx .x] = d_ptr [ idx ];
```

```
__syncthreads ();
```

- Do computation

```
float res = func(a_sh[threadIdx.x])
```

- Store result

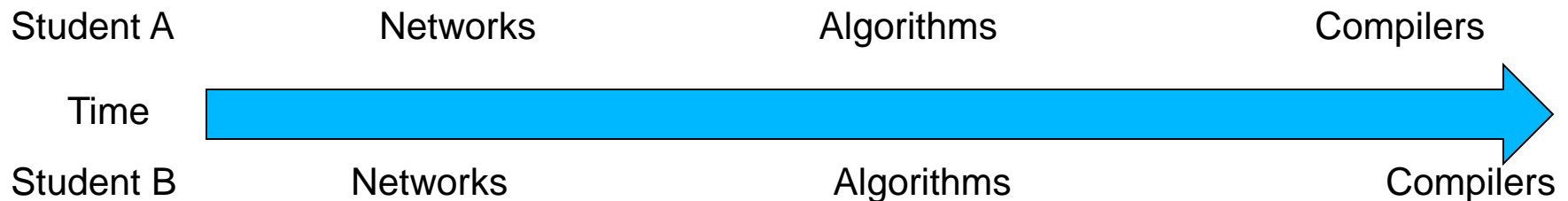
```
d_ptr[index] =res;
```

Because it's tricky, lets discuss in more detail:

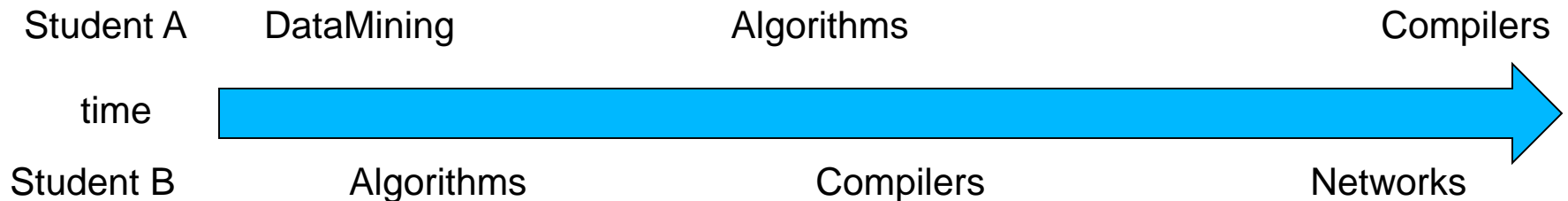
SHARED MEMORY

Dept library need synchronization

- Good – when students have similar choices

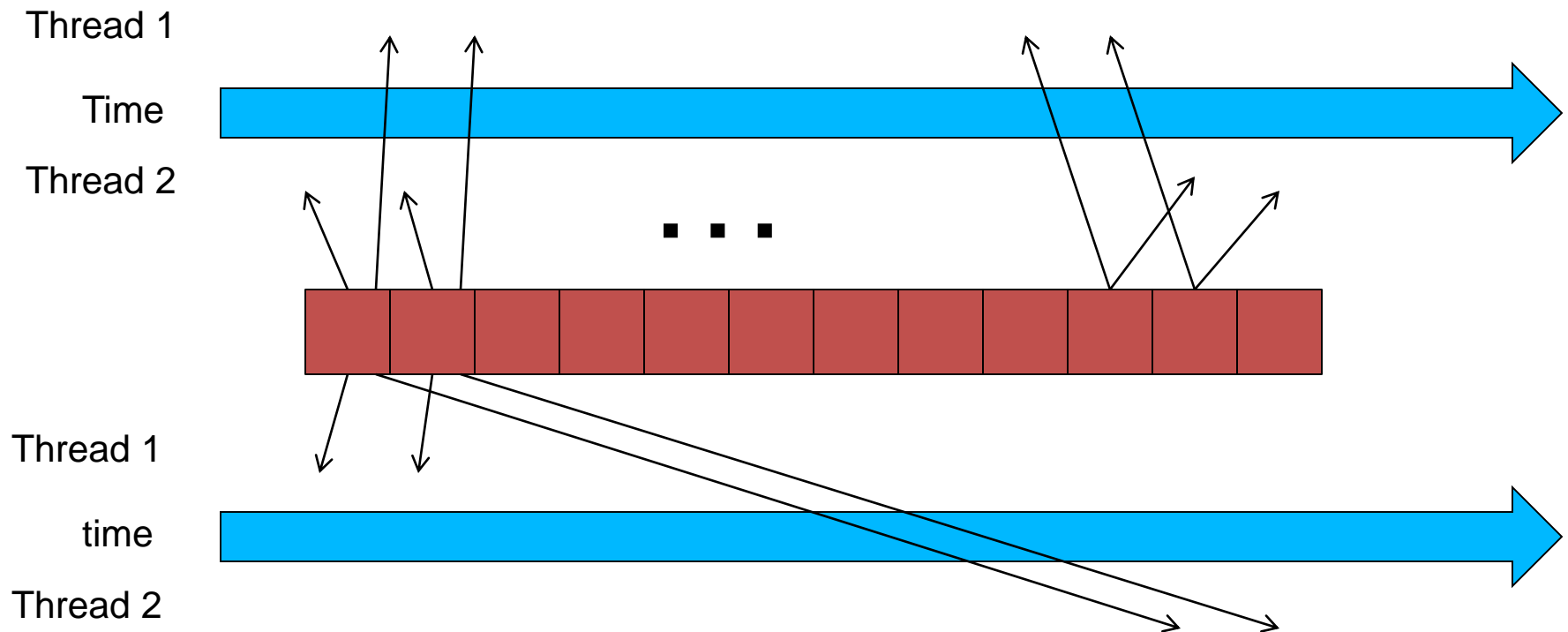


- Bad – when students have different choices



Same with Blocking/Tiling

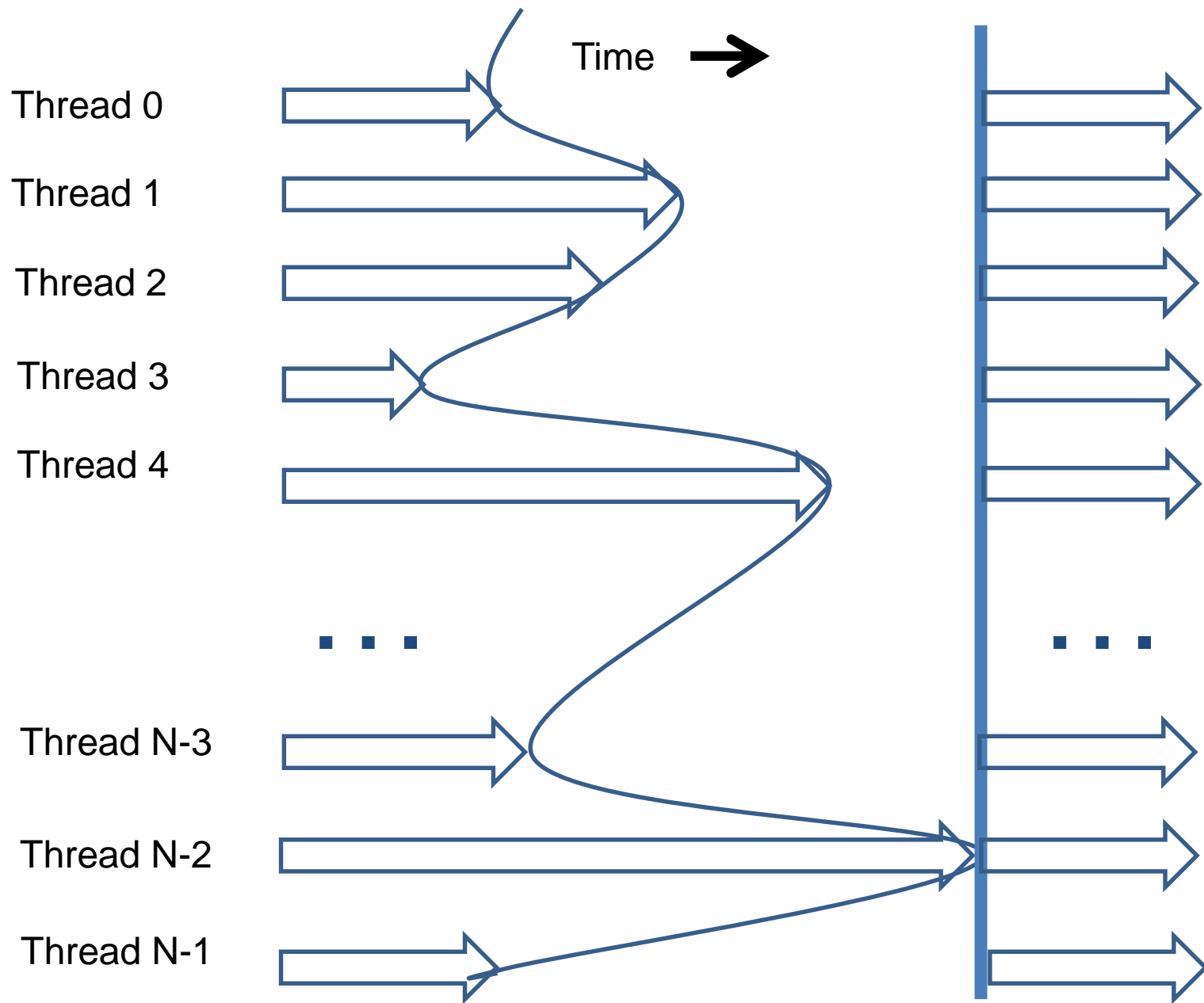
- Good –when threads have similar access timing



- Bad – when threads have very different timing

Barrier Synchronization

- A function call in CUDA
 - `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any can move on
- Best used to coordinate tiled algorithms
 - To ensure that all elements of a tile are loaded
 - To ensure that all elements of a tile are consumed

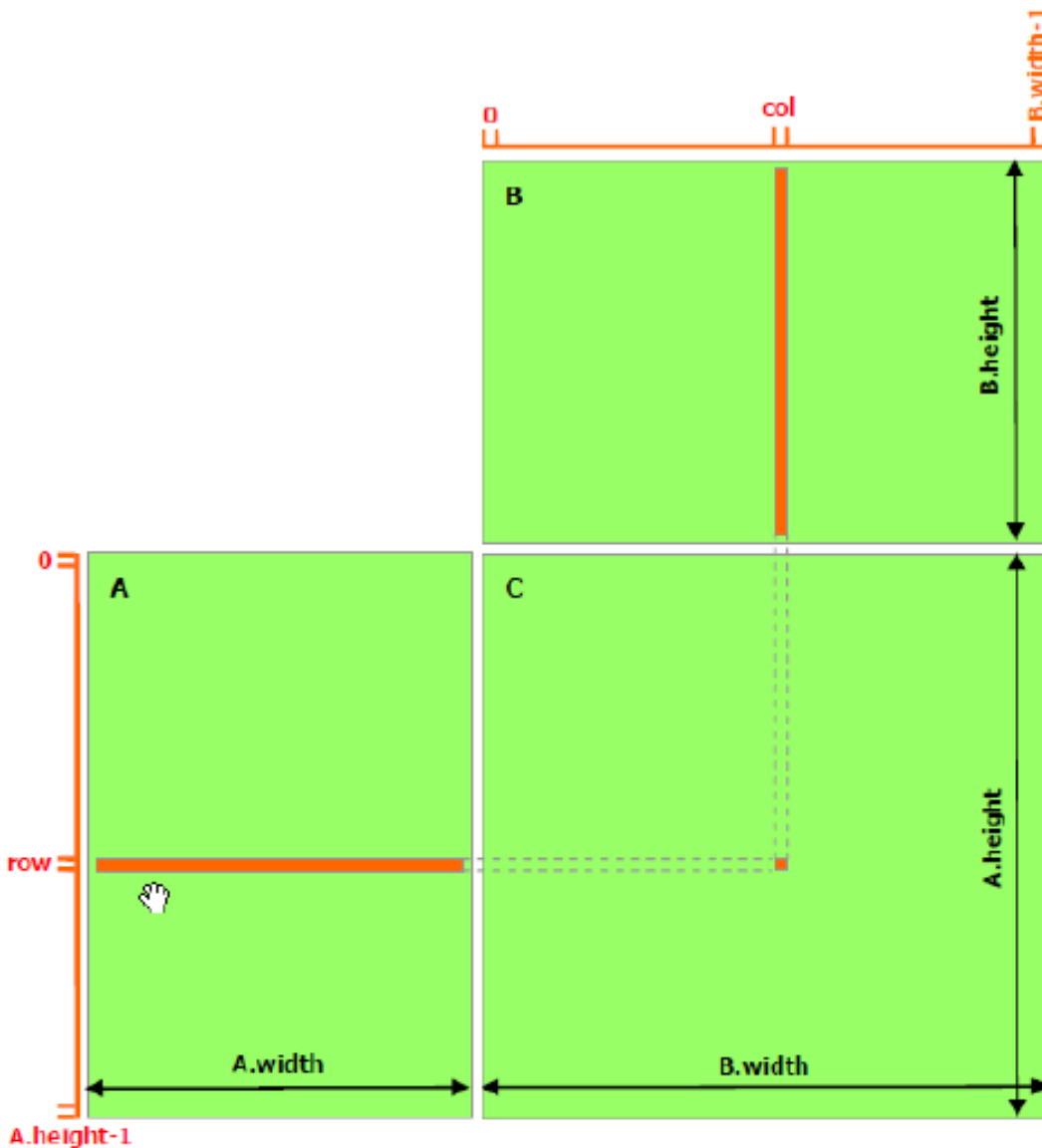


An example execution timing of barrier synchronization.

Consider matrix multiplication $C = AB$

```
for (i = 0; i < A.height; i++) {  
  for(j=0; j< B.width; j++) {  
    c[i][j] = 0;  
    for (k=0; k< A.width ; k++)  
      c[i][j] += a[i][k] * b[k][j];  
  }  
}
```

- # times each element of A accessed: **B.width**
- # times each element of B accessed: **A.height**
- # times each element of C accessed? **A.width**



Consider an element $c[\text{row}][\text{col}]$. There are $B.\text{width}$ elements on a row of C and $A.\text{height}$ elements in a column of C

To compute each of these elements, we access a row of A and a column of B

=> We access each row of A $B.\text{width}$ times and each column of B $A.\text{height}$ times

We'll see the code using both global and shared memory

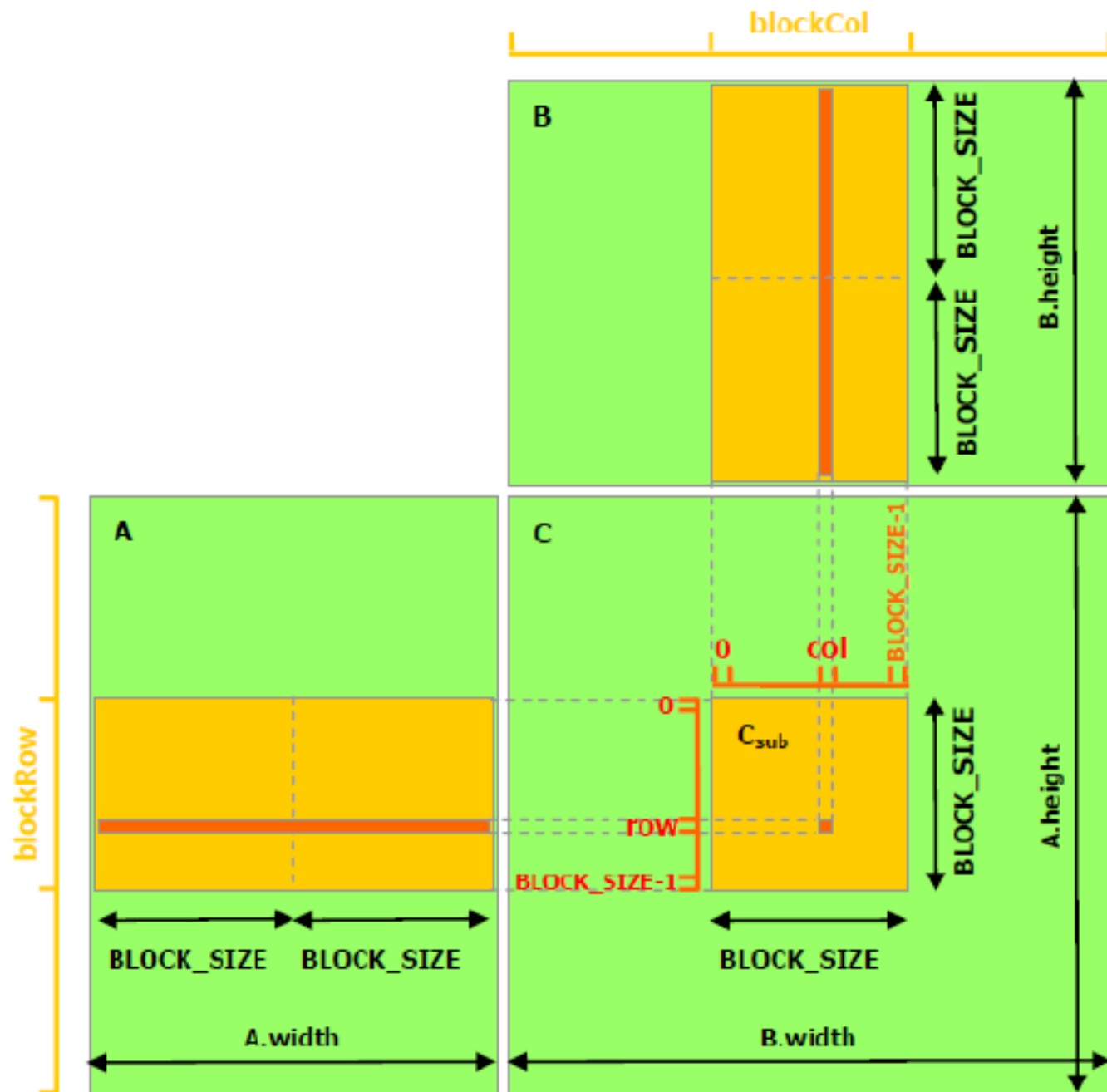
- Assume our matrices are square $N \times N$ and are stored using linear arrays
- Access to the $(i ; j)$ element is facilitated via the macro

```
# define IDX(i,j,n) ((i)*(n)+j)
```

MatMul using global memory

```
__global__ void matmulGlobal ( float * c, float * a, float * b, int N )
{
    // compute row and column for our matrix element
    int col = blockIdx .x * blockDim .x + threadIdx .x;
    int row = blockIdx .y * blockDim .y + threadIdx .y;
    if ( col < N && row < N )
    {
        float sum = 0.0;
        for ( int k = 0; k < N; k++ )
        {
            sum += a[IDX (row ,k,N)] * b[IDX (k,col ,N)];
        }
        c[IDX(row ,col ,N)] = sum;
    }
}
```

To use shared memory, we use the idea of blocking/tiling



MatMul using shared memory (1/3)

```
__global__ void matmulShared ( float * c, float * a, float * b,  
int N )  
{  
    // compute row and column for our matrix element  
    int col = blockIdx .x * blockDim .x + threadIdx .x;  
    int row = blockIdx .y * blockDim .y + threadIdx .y;  
  
    // compute the number of blocks we need  
    int M = ( N + BlockSize - 1 ) / BlockSize ;  
    float sum = 0.0;
```

MatMul using shared memory (2/3)

```
//Go through each block
```

```
for ( int m = 0; m < M; m++ ) {
```

```
// all threads in block copy their element from
```

```
// matrix a and matrix b to shared memory
```

```
__shared__ float a_s[ BlockSize ][ BlockSize ];
```

```
__shared__ float b_s[ BlockSize ][ BlockSize ];
```

```
int c = m * BlockSize + threadIdx .x;
```

```
int r = m * BlockSize + threadIdx .y;
```

```
a_s [ threadIdx .y][ threadIdx .x] = a[IDX (row ,c,N)];
```

```
b_s [ threadIdx .y][ threadIdx .x] = b[IDX (r,col ,N)];
```

```
// make sure all threads are finished & matrix is loaded
```

```
__syncthreads ();
```

MatMul using shared memory (3/3)

```
// compute partial sum using shared memory block
// K is block size except at right or bottom since we
// may not have a full block of data there
    int K = (m == M - 1 ? N - m * BlockSize : BlockSize );
    for ( int k = 0; k < K; k++ )
    {
        sum += a_s [ threadIdx.y][k] * b_s [k][ threadIdx.x];
    }
//Synchronize to make sure that the computation is done before
loading two new sub-matrices of A and B in the next iteration
    __syncthreads ();
}
if ( col < N && row < N ) c[ IDX(row ,col ,N)] = sum;
}
```

References

- CUDA language:
 - CUDA by Example, by Jason Sanders and Edward Kandrot, NVIDIA
 - “Programming Massively Parallel Processors: A Hands-on Approach” by David B. Kirk and Wen-mei W. Hwu
- GPU architecture
 - “A Survey of CPU-GPU heterogeneous computing”, S. Mittal et al., CSUR 2015
 - <https://cvw.cac.cornell.edu/gpu/coalesced>
 - <https://medium.com/@smallfishbigsea/basic-concepts-in-gpu-computing-3388710e9239>

Thanks!



Sparsh Mittal sparsh@iith.ac.in