L10

Introduction to CUDA

Hardware Accelerators

In HPC, an accelerator is hardware component whose role is to speed up some aspect of the computing workload.

• In the olden days (1980s), supercomputers sometimes had ***array processors***, which did vector operations on arrays.

• PCs sometimes had ***floating point accelerators***: little chips that did the floating point calculations in hardware rather than software.

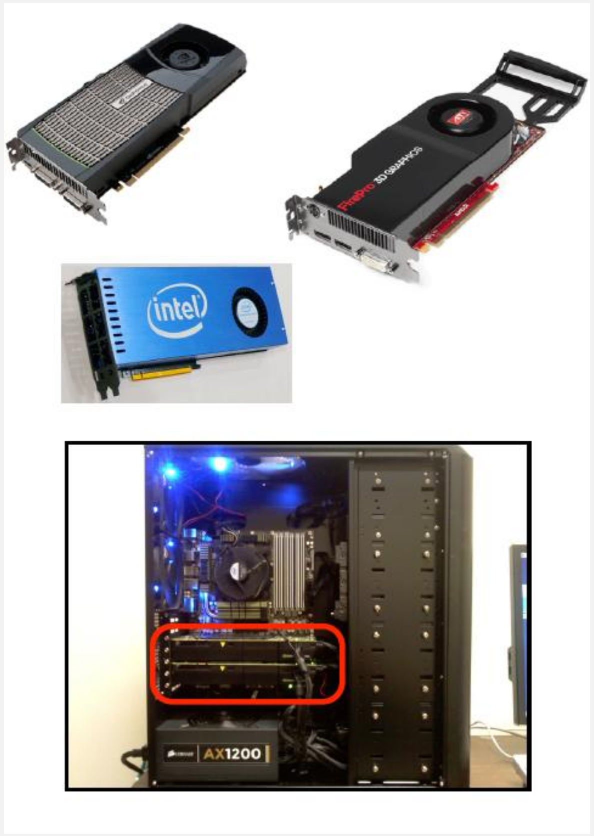
GPUs

Graphics Processing Units (GPUs) were originally designed to accelerate graphics tasks like image rendering.

• They became very popular with video gamers, because they've produced better and better images, and lightning fast, with low prices.

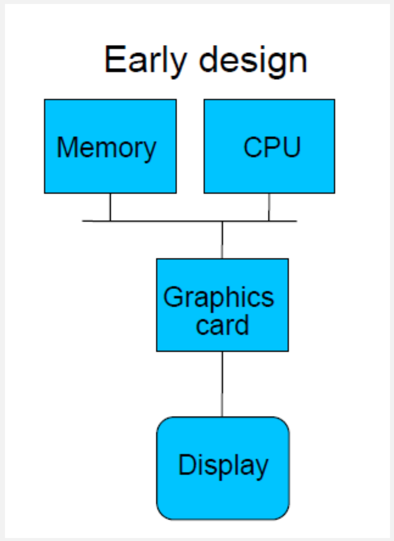
• Chips are expensive to design (hundreds of millions of $), expensive to build the factory for (billions of $), but cheap to produce.

• GPUs mostly did stuff like rendering images. This is done through mostly floating point arithmetic



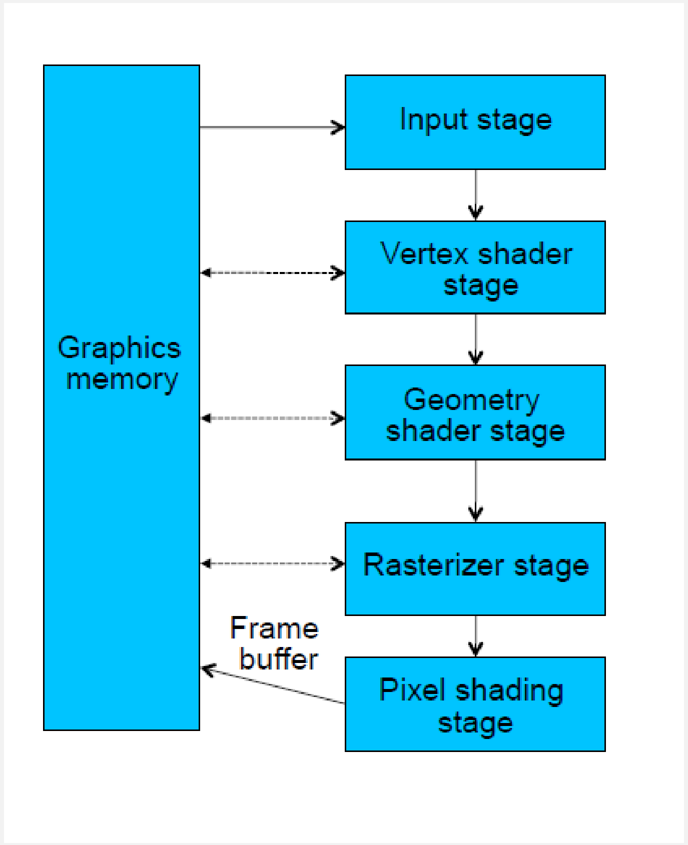
GPUs for HPC

- GPUs have developed from graphics cards into a platform for high performance computing (HPC).
- Co-processors -- very old idea that appeared in 1970s and 1980s with floating point coprocessors attached to microprocessors that did not then have floating point capability.
- These coprocessors simply executed floating point instructions that were fetched from memory.
- Around the same time, interest to provide hardware support for displays, especially with increasing use of graphics and PC games.
- Led to graphics processing units (GPUs) attached to CPU to create video display.



Modern GPU Design

- By late 1990's, graphics chips needed to support 3-D graphics, especially for games and graphics APIs such as DirectX and OpenGL.
- Graphics chips generally had a pipeline structure with individual stages performing specialized operations, finally leading to loading frame buffer for display.
- Individual stages may have access to graphics memory for storing intermediate computed data.



General Purpose GPU (GPGPU)

- High performance pipelines call for high-speed (IEEE) floating point operations.
- Known as GPGPU (General-purpose computing on graphics processing units) – Difficult to do with specialized graphics pipelines, but possible.)
- By mid 2000's, recognized that individual stages of graphics pipeline could be implemented by a more general purpose processor core (although with a data-parallel paradigm)
- 2006 -- First GPU for general high performance computing as well as graphics processing, NVIDIA GT 80 chip/GeForce 8800 card.
- Unified processors that could perform vertex, geometry, pixel, and **general computing operations**
- Could now write programs in C rather than graphics APIs.
- Single-instruction multiple thread **(SIMT) programming model**

NVIDIA Tesla Platform

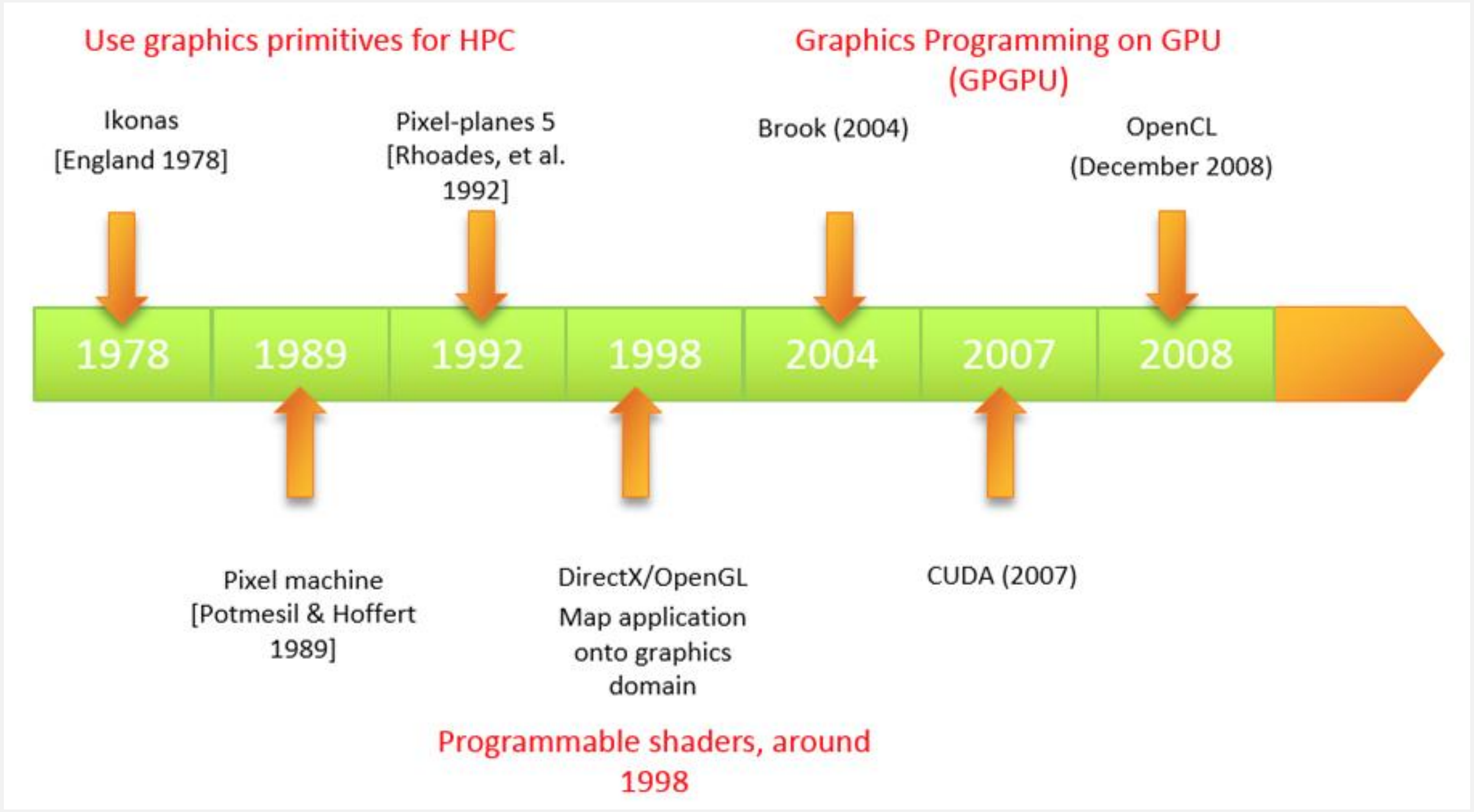
NVIDIA Tesla series was their first platform for the high performance computing market.

NVIDIA GTX 480 Specs

- 3 billion transistors
- 480 compute cores
- 1.401 GHz
- Single precision floating point performance: 1.35 TFLOPs (2 single precision flops per clock per core)
- Double precision floating point performance: 168 GFLOPs (1 double precision flop per clock per core)
- Internal RAM: 1.5 GB DDR5 VRAM
- Internal RAM speed: 177.4 GB/sec (compared 21-25 GB/sec for regular RAM)
- PCIe slot (at most 8 GB/sec per GPU card)
- 250 W thermal power

Series:
-Kepler
-Maxwell
-Pascal
-Turing
-Volta

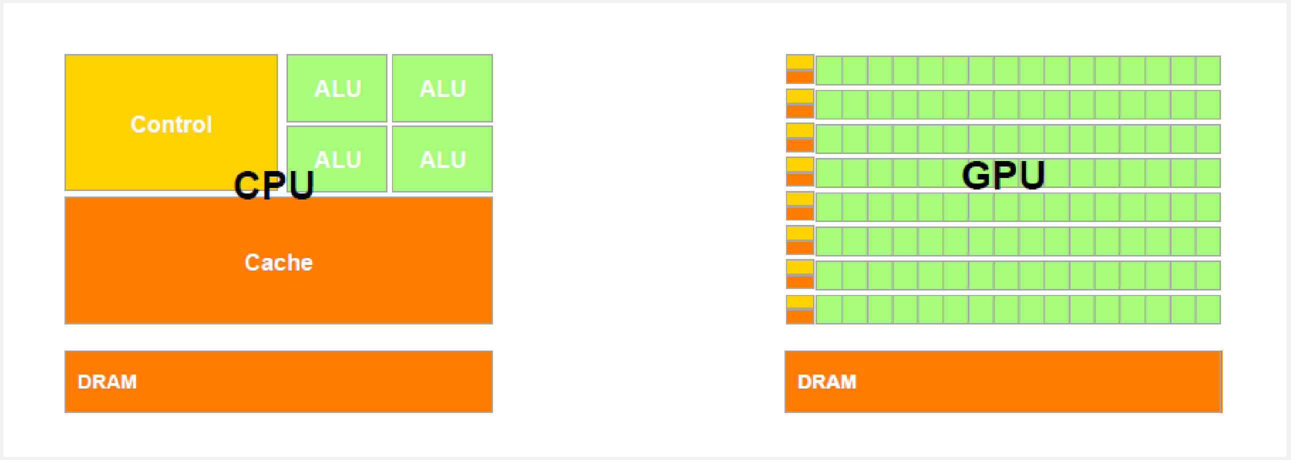
History


Use graphics primitives for HPC
Graphics Programming on GPU (GPGPU)
Ikonas [England 1978]
Pixel-planes 5 [Rhoades, et al. 1992]
Brook (2004)
OpenCL (December 2008)
1978
1989
1992
1998
2004
2007
2008
Pixel machine [Potmesil & Hoffert 1989]
DirectX/OpenGL Map application onto graphics domain
CUDA (2007)
Programmable shaders, around 1998

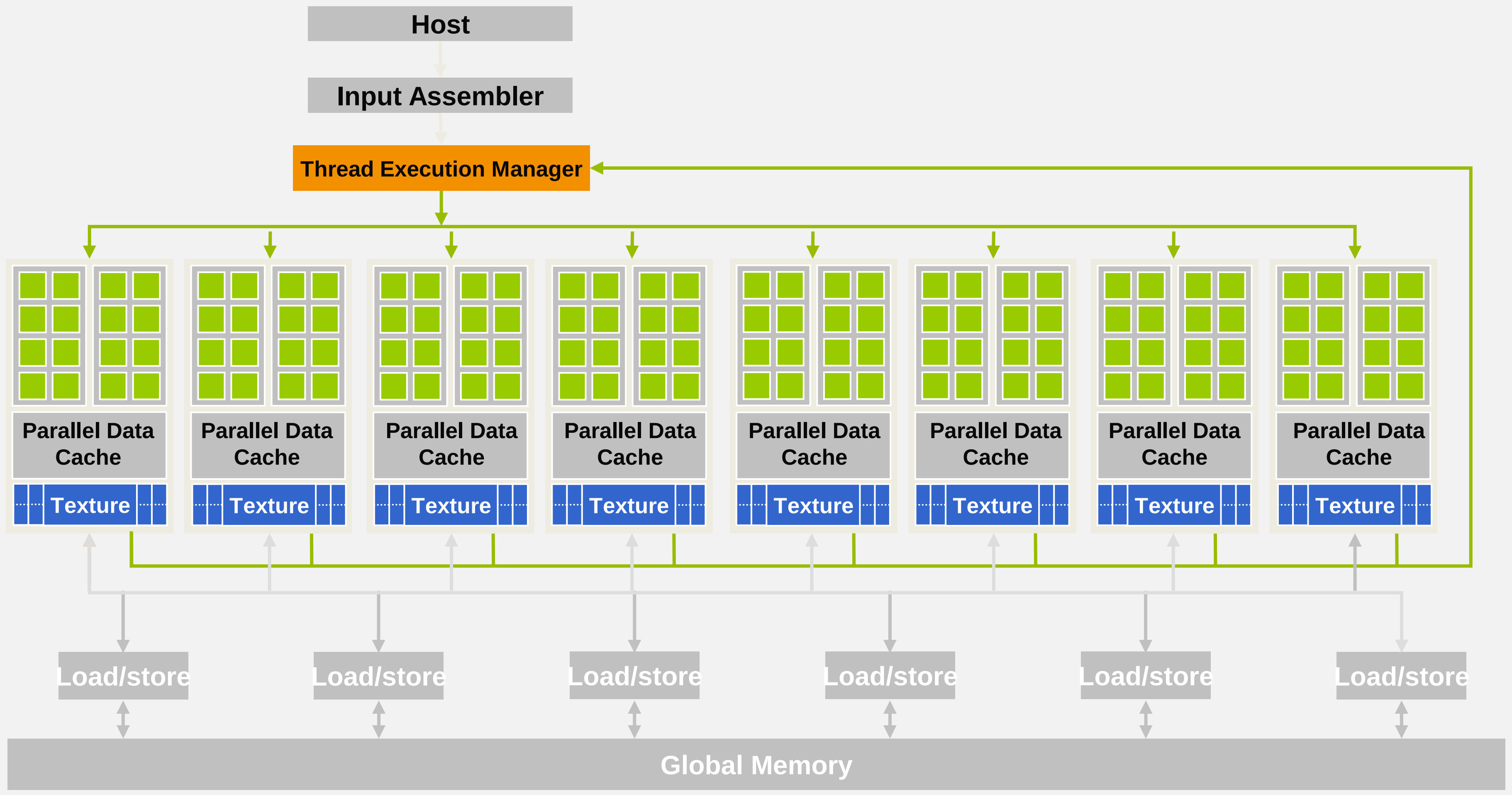
Compute Unified Device Architecture

  - 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
- CUDA-capable GPU
  - https://developer.nvidia.com/cuda-gpus
- Documentation
    - http://docs.nvidia.com/cuda/

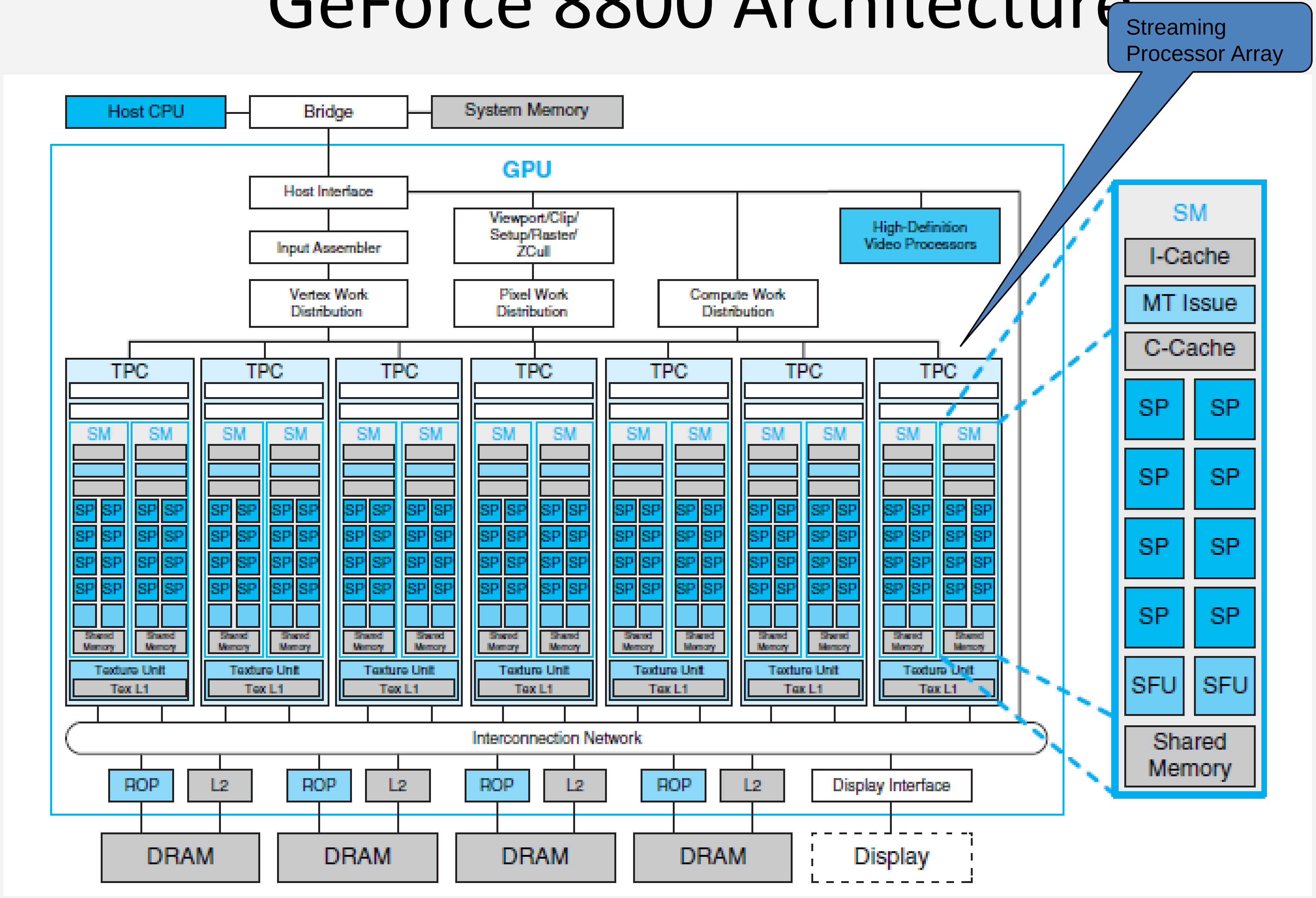
CPU vs. GPU Hardware Design

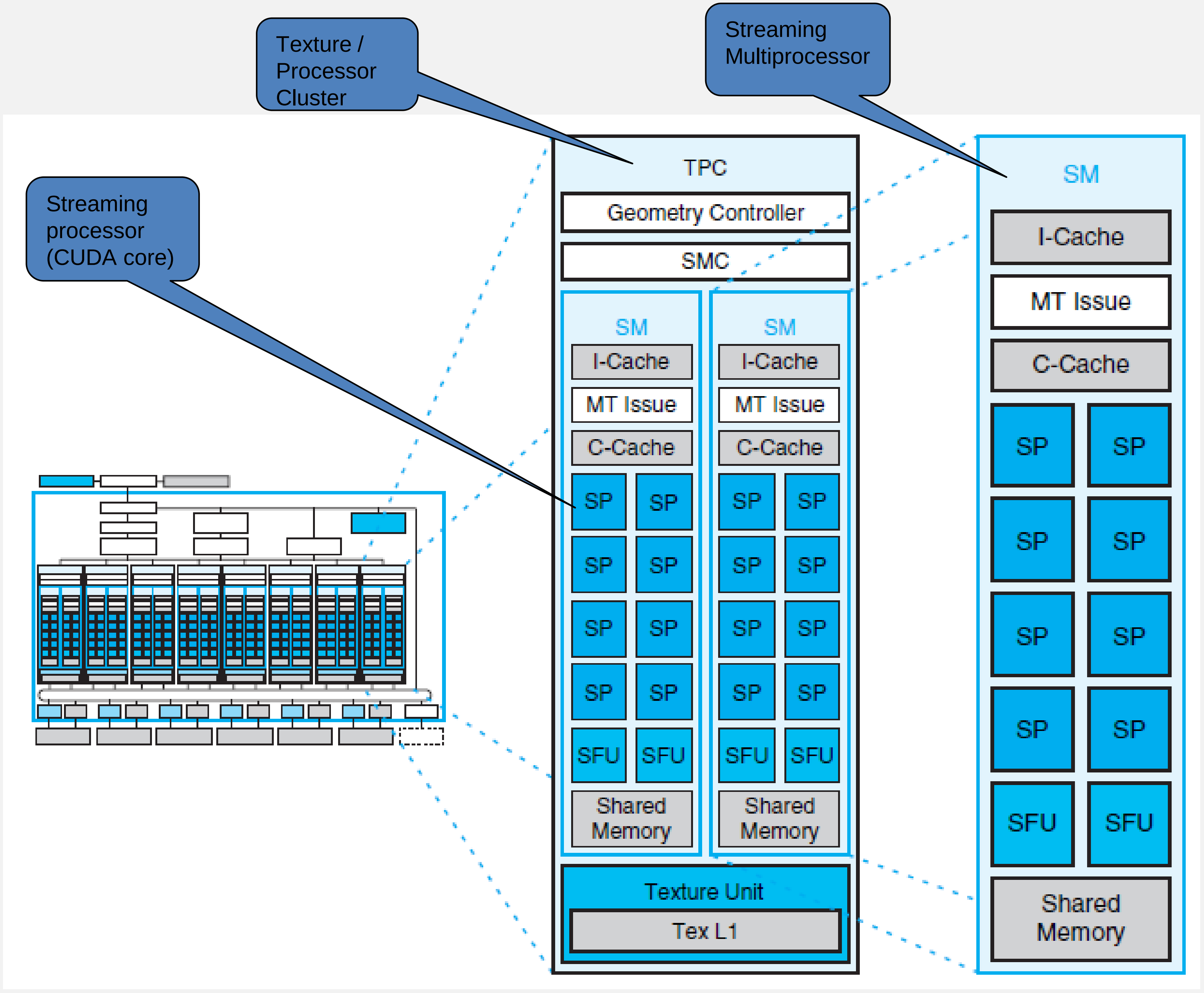


Architecture of a CUDA-capable GPU



GeForce 8800 Architecture




Texture / Processor Cluster
Streaming Multiprocessor
Streaming processor (CUDA core)
TPC
Geometry Controller
SMC
SM
I-Cache
MT Issue
C-Cache
SP
SFU
Shared Memory
Texture Unit
Tex L1

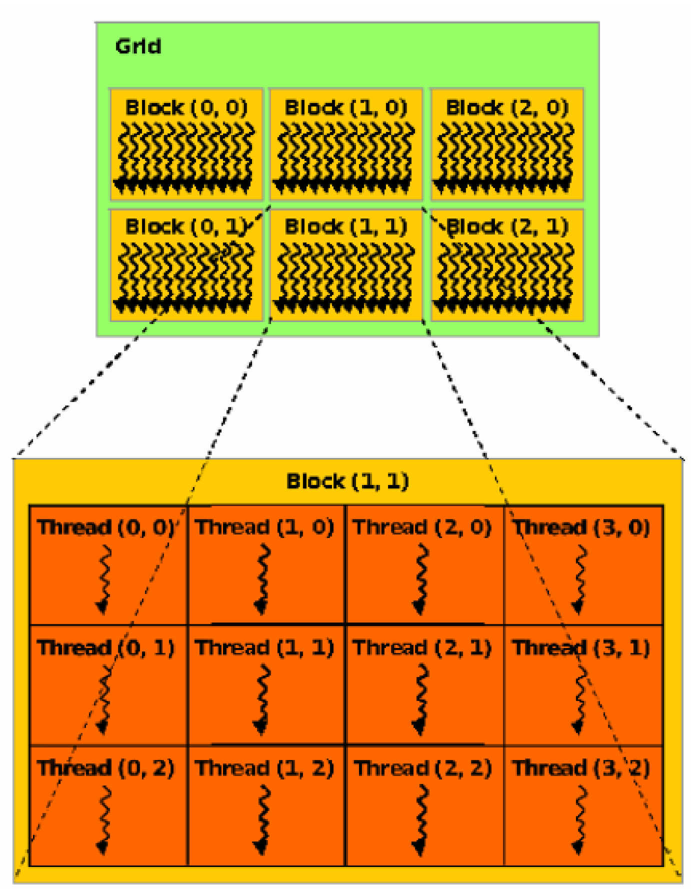
Single Instruction, Multiple Threads (SIMT)

- A version of SIMD used in GPUs.

- GPUs use a thread model to achieve a high parallel performance and hide memory latency.

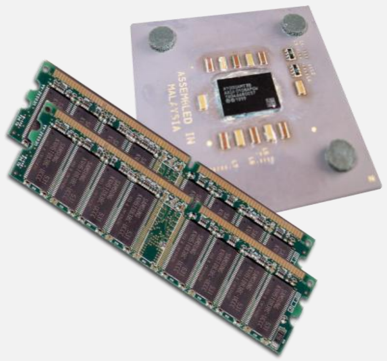
- On a GPU, 10,000s of threads are mapped on to available processors that all execute the same set of instructions (on different data addresses).

- Grids map to GPUs
- Blocks map to the MultiProcessors (MP)
- Threads map to Stream Processors (SP)
- Warps are groups of (32) threads that execute simultaneously



Heterogeneous Computing

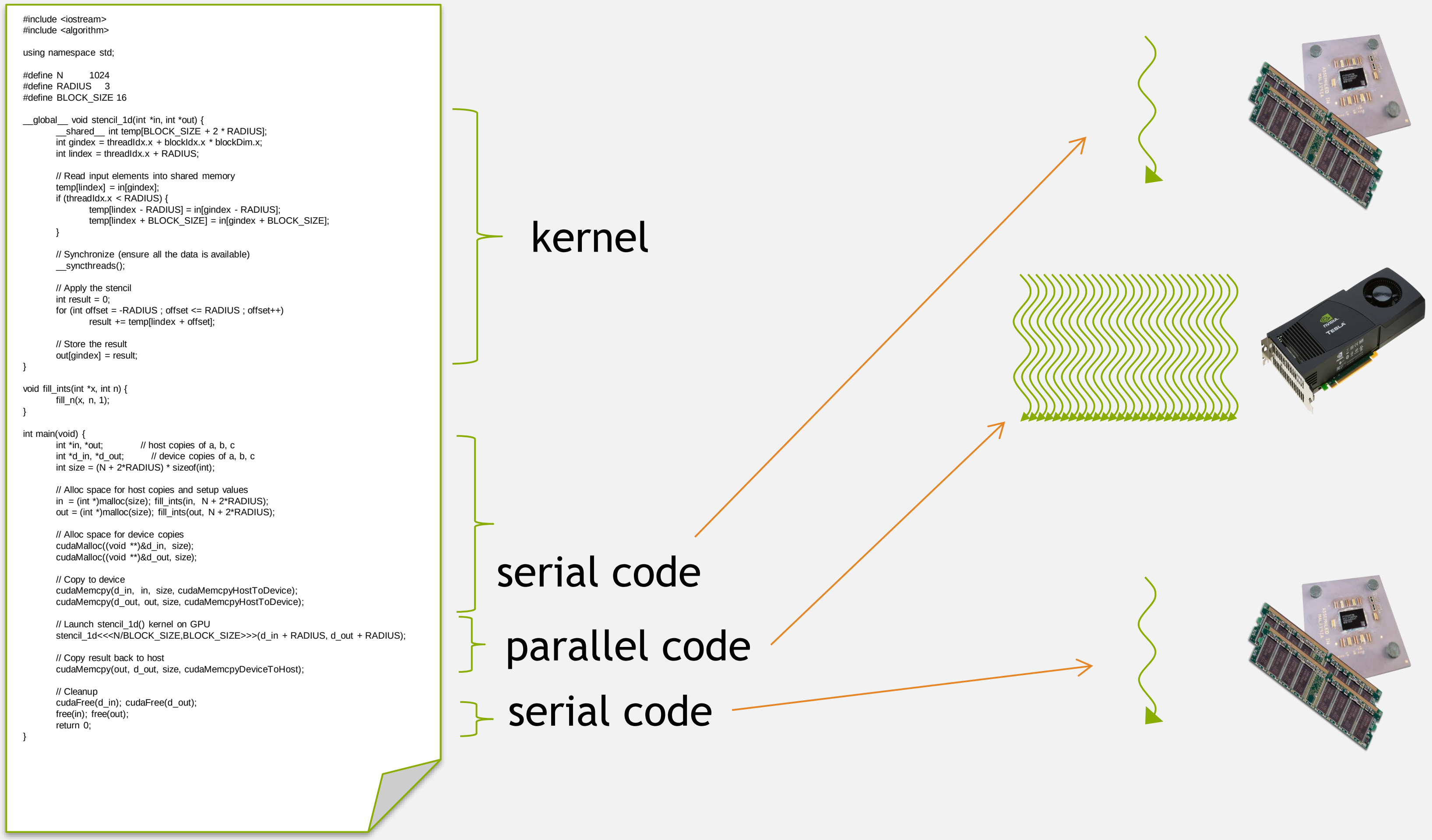
- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

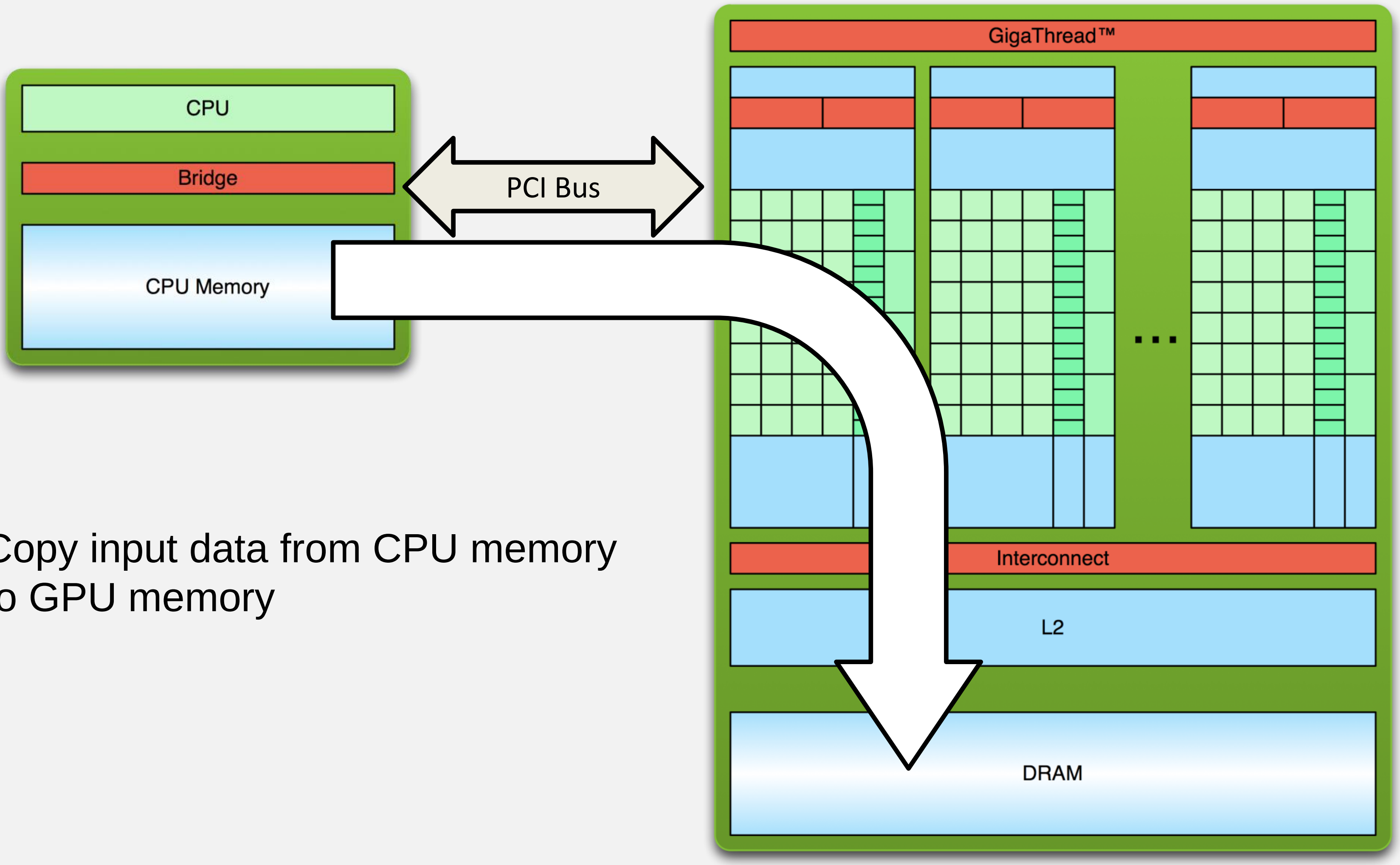
Device

Heterogeneous Computing



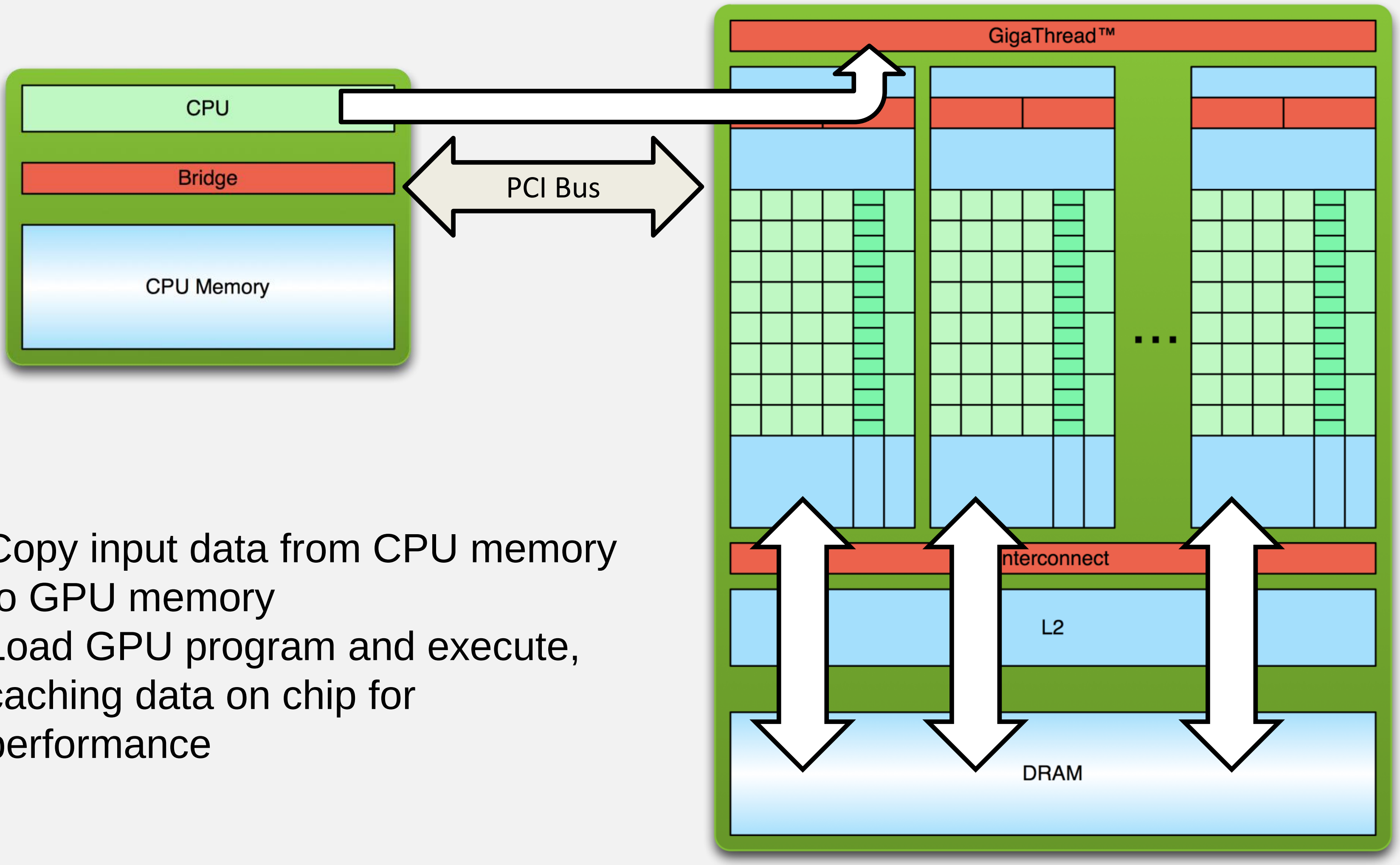
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N         1024
#define RADIUS     3
#define BLOCK_SIZE 16

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

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

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

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

Simple Processing Flow



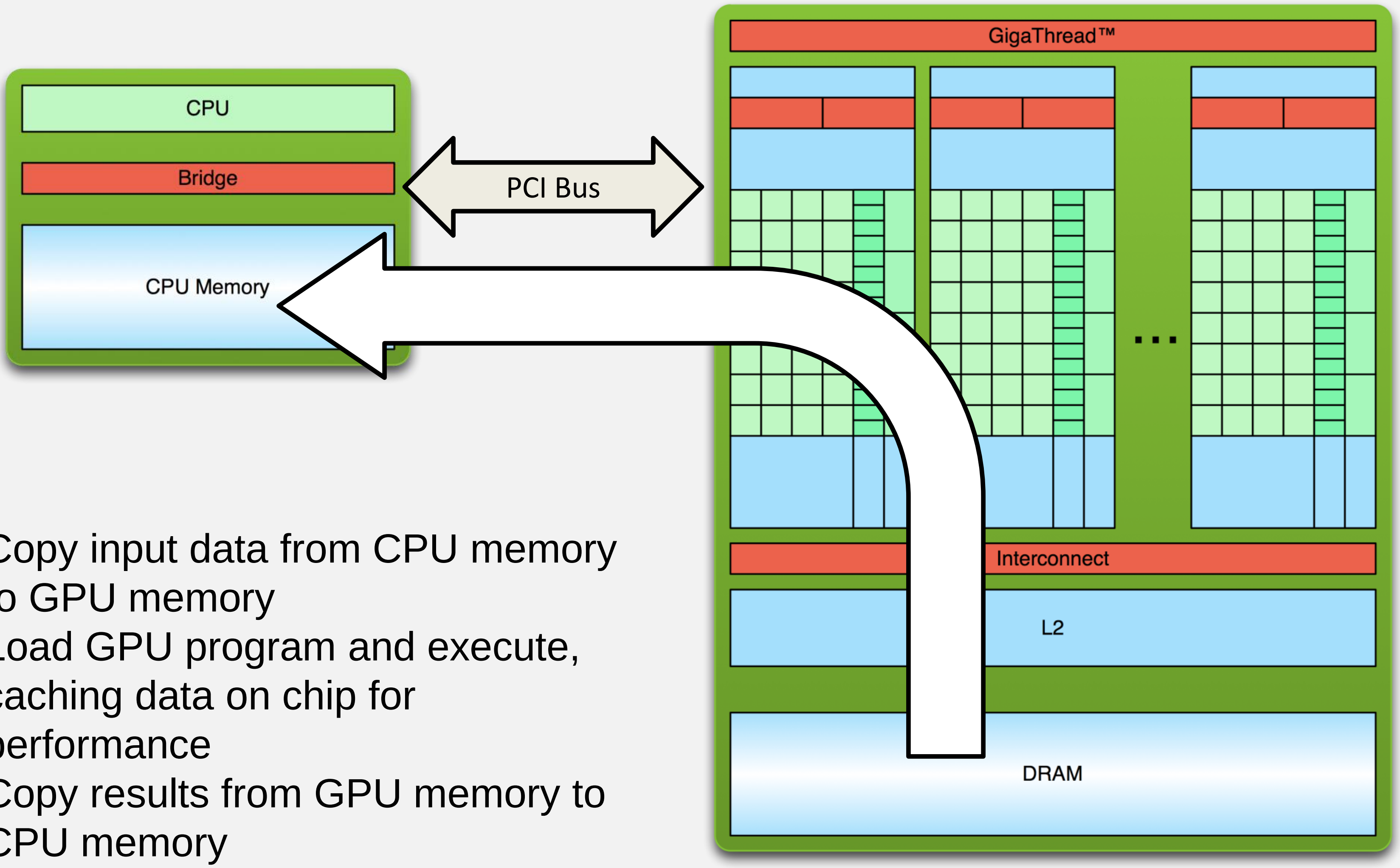
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

Simple Processing Flow



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

Hello World!

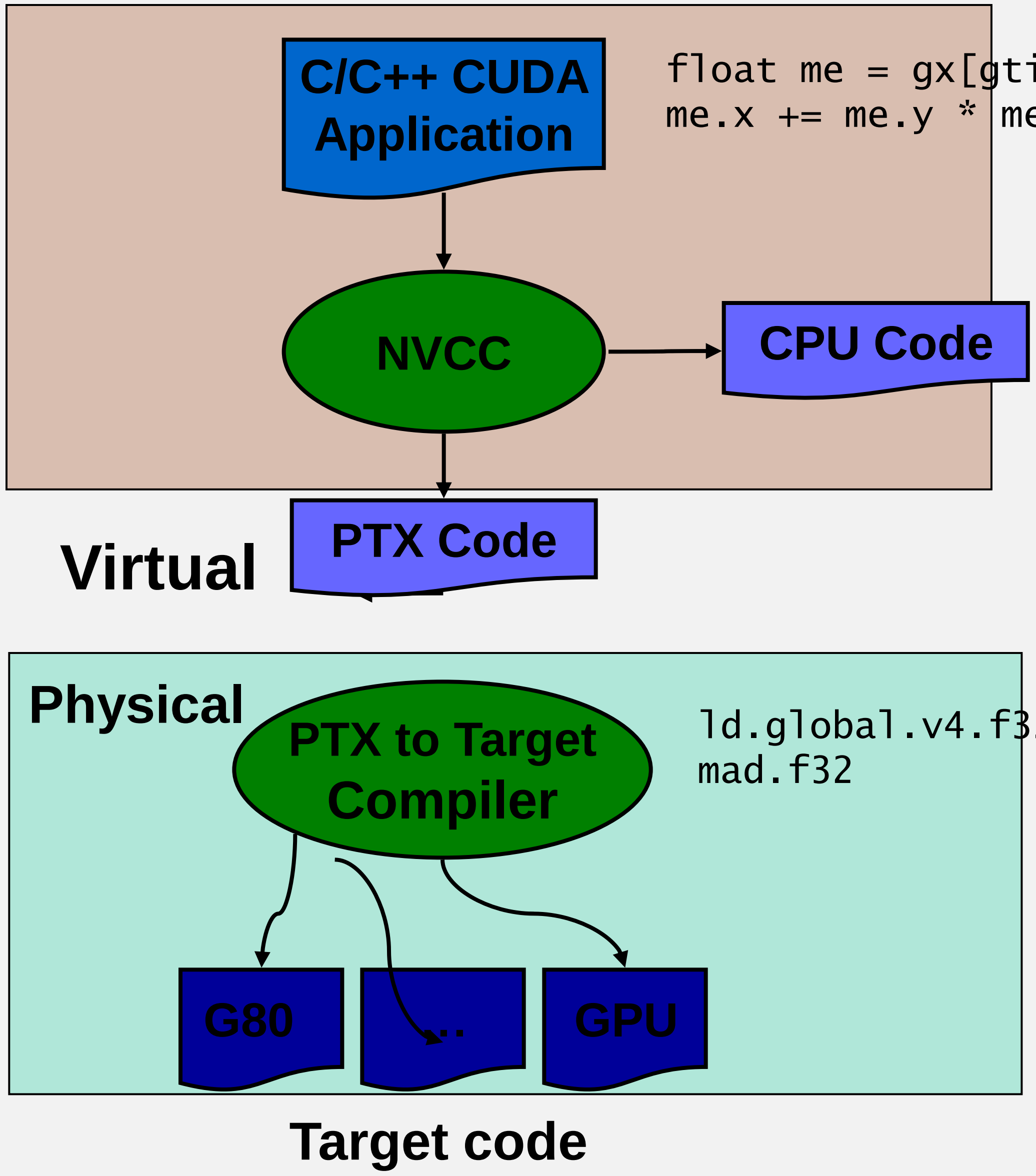
```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

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

Output:

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

Compiling a CUDA Program



```
float me = gx[gtid];
me.x += me.y * me.z;
```

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

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

```
__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”
- 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;
}
```

- `mykernel()` does nothing -

Output:

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

Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

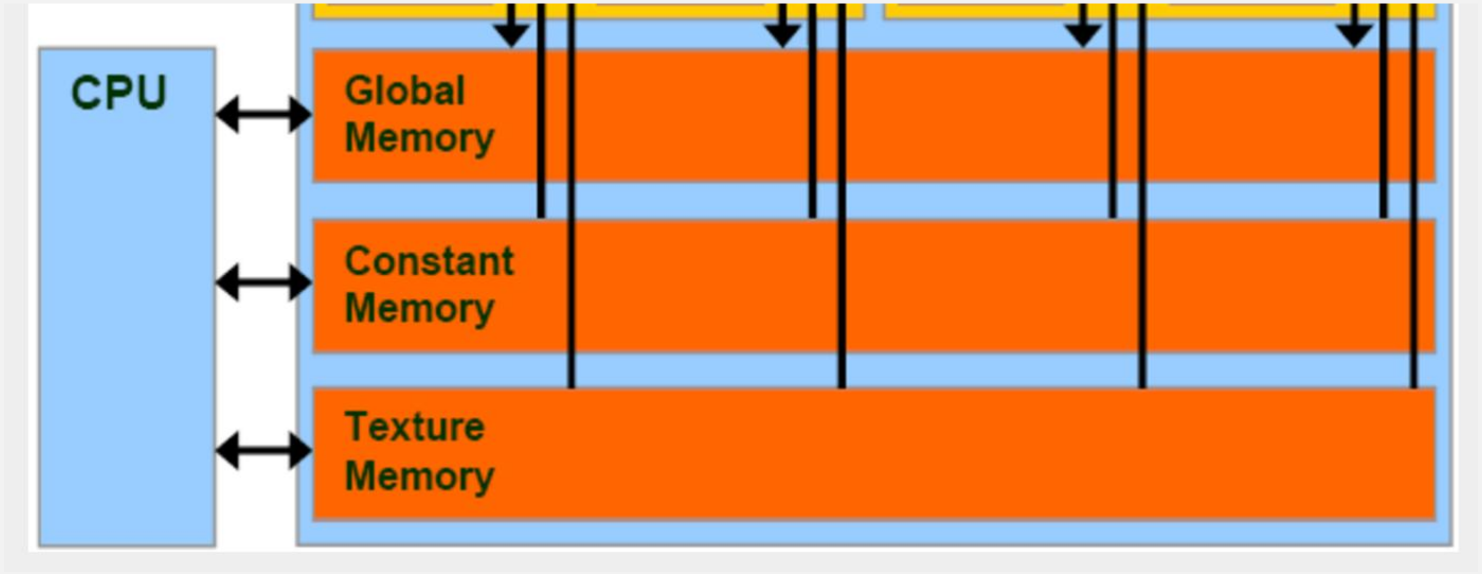
- `__global__` defines a kernel function
  - **Must return** `void`

CUDA Function Declarations

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - ***No recursion***
  - ***No static variable declarations inside the function***
  - ***No variable number of arguments***

Memory Types

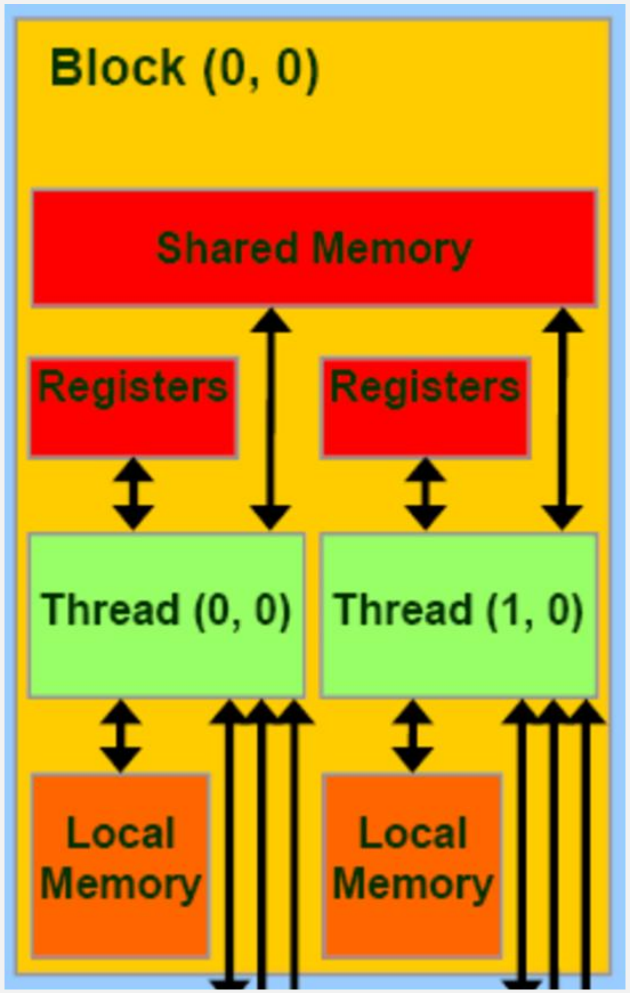
- Global memory: cudaMalloc memory, the size is large, but slow (has cache)
- Texture memory: read only, cache optimized for 2D access pattern
- Constant memory: slow but with cache (8KB)



Memory Types

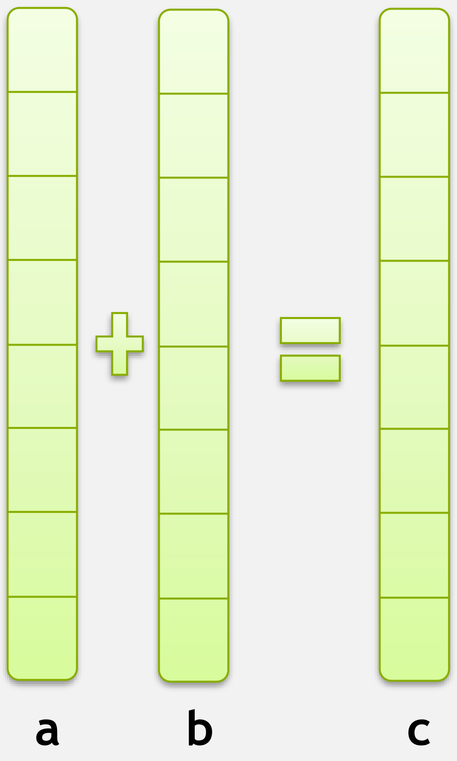
- Local memory:
  local to thread, but it is as slow as global memory

- Shared memory:
  100x fast to global memory, it is accessible to all threads in one block



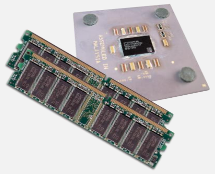
Simple Examples

- adding two integers
  - use the device for seq computation
- vector addition
  - real parallel computation



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;
}
```

Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.
- Only supported on the SCC when using the K40m or P100 GPUs

- With Unified Memory the CUDA driver will manage memory transfers using the **`cudaMallocManaged()`** function.

- Managed memory is still freed using **`cudaFree()`**

- The P100 will offer the best performance when using this feature.

- Unified Memory simplifies memory management in a CUDA code.

- For more details see:

  https://devblogs.nvidia.com/unified-memory-cuda-beginners/

```
#include <stdio.h>

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

int main(void) {
                 int *a, *b, *c;  // host AND device
                 int size = sizeof(int);

                 // Allocate space for device copies of a, b, c
                 cudaMallocManaged(&a, size);
                 cudaMallocManaged(&b, size);
                 cudaMallocManaged(&c, size);

                 // Setup input values
                 *a = 2;
                 *b = 7;

                 // Launch add() kernel on GPU. Data values are
                 // sent to the host when accessed in the kernel
                 add<<<1,1>>>(a,b,c);
                 // Wait for GPU to finish before accessing on host
                  cudaDeviceSynchronize();
                 // access will auto-transfer data back to the host
                 printf("%d %d %d\n",*a, *b, *c);

                 // Cleanup
                 cudaFree(a); cudaFree(b); cudaFree(c);
                 return 0;
}
```

Vector Addition on the Device

– block

– grid

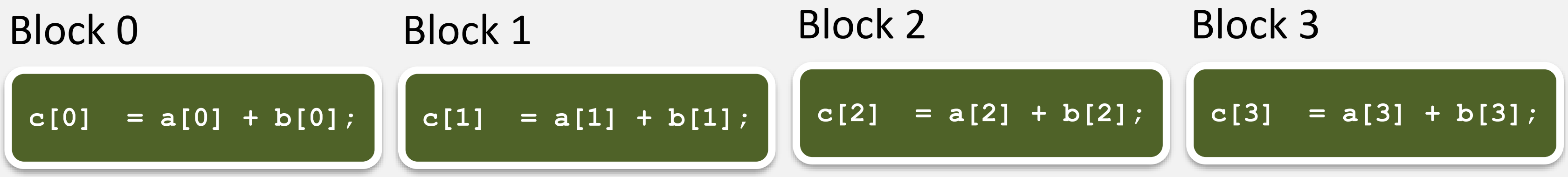
– `blockIdx.x`

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

Vector Addition on the Device

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

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



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

Vector Addition on the Device: `add()`

- Using `add()` kernel

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

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a, *b, *c;                   // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

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

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA Threads

- A block can be split into parallel threads

- It is possible to 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`

Vector Addition Using Threads: `main()`

```
#define N 512
int main(void) {
    int *a, *b, *c;                              // host copies of a, b, c
    int *d_a, *d_b, *d_c;                  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

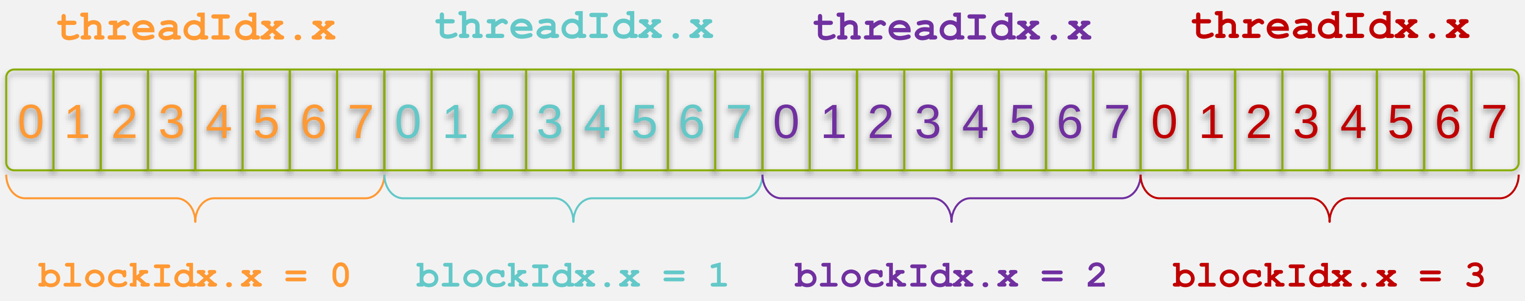
Vector Addition Using Threads: `main()`

```
     // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

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

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Combining Blocks and Threads

Indexing Arrays with Blocks and Threads

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

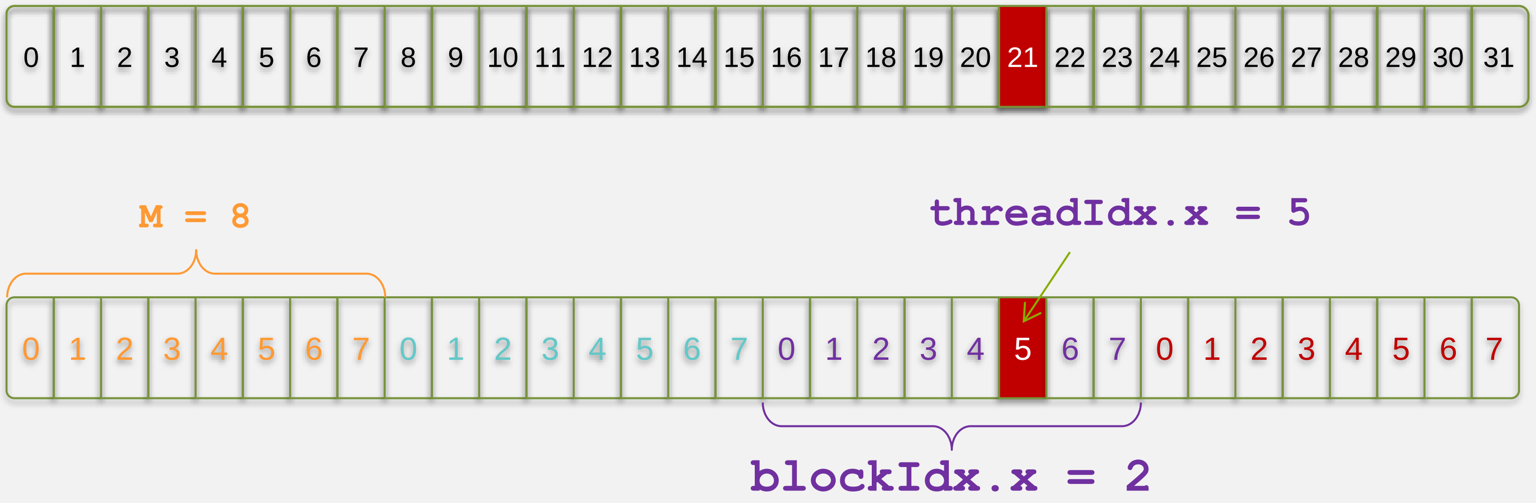


- With M threads/block a unique index for each thread is given by:

```
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: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_a, *d_b, *d_c;            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

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

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

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

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

Why Bother with Threads?

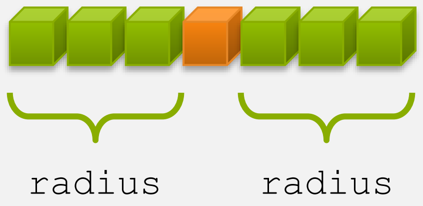
- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

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

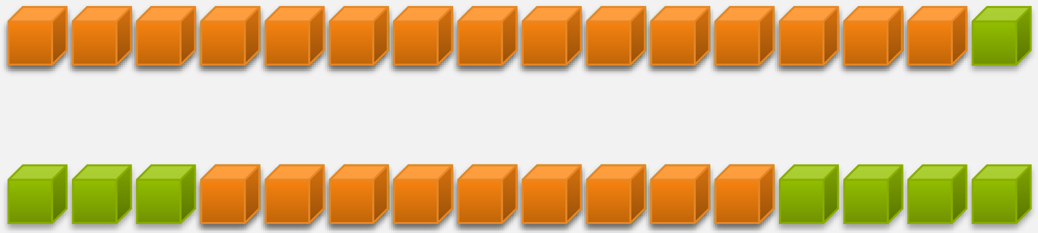
- If radius is 3, then each output element is the sum of 7 input elements:



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

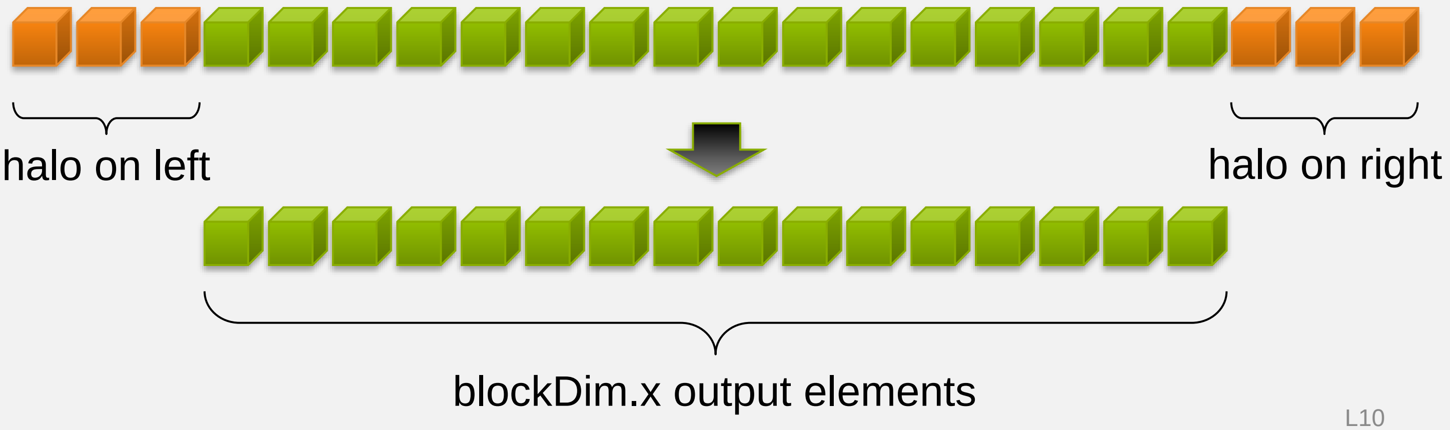
- Declare using `__shared__`, allocated per block

- Data is not visible to threads in other blocks

Implementing With Shared Memory

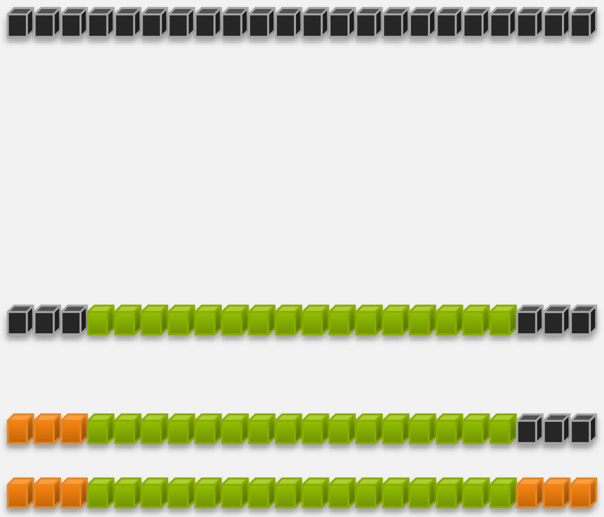
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

  - 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];
  }
```



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;
}
```

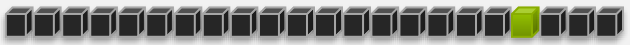
Data Race!

- The stencil example will not work…

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

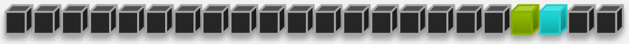
```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

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();
// 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

CUDA Built-In Variables for Grid/Block Indices

- uint3 blockIdx -- block index within grid:
    - blockIdx.x, blockIdx.y (z not used)
- uint3 threadIdx -- thread index within block:
    - threadIdx.x, threadIdx.y, threadIdx.z
- Full global thread ID in x and y dimensions can be computed by:
    - x = blockIdx.x * blockDim.x + threadIdx.x;
    - y = blockIdx.y * blockDim.y + threadIdx.y;

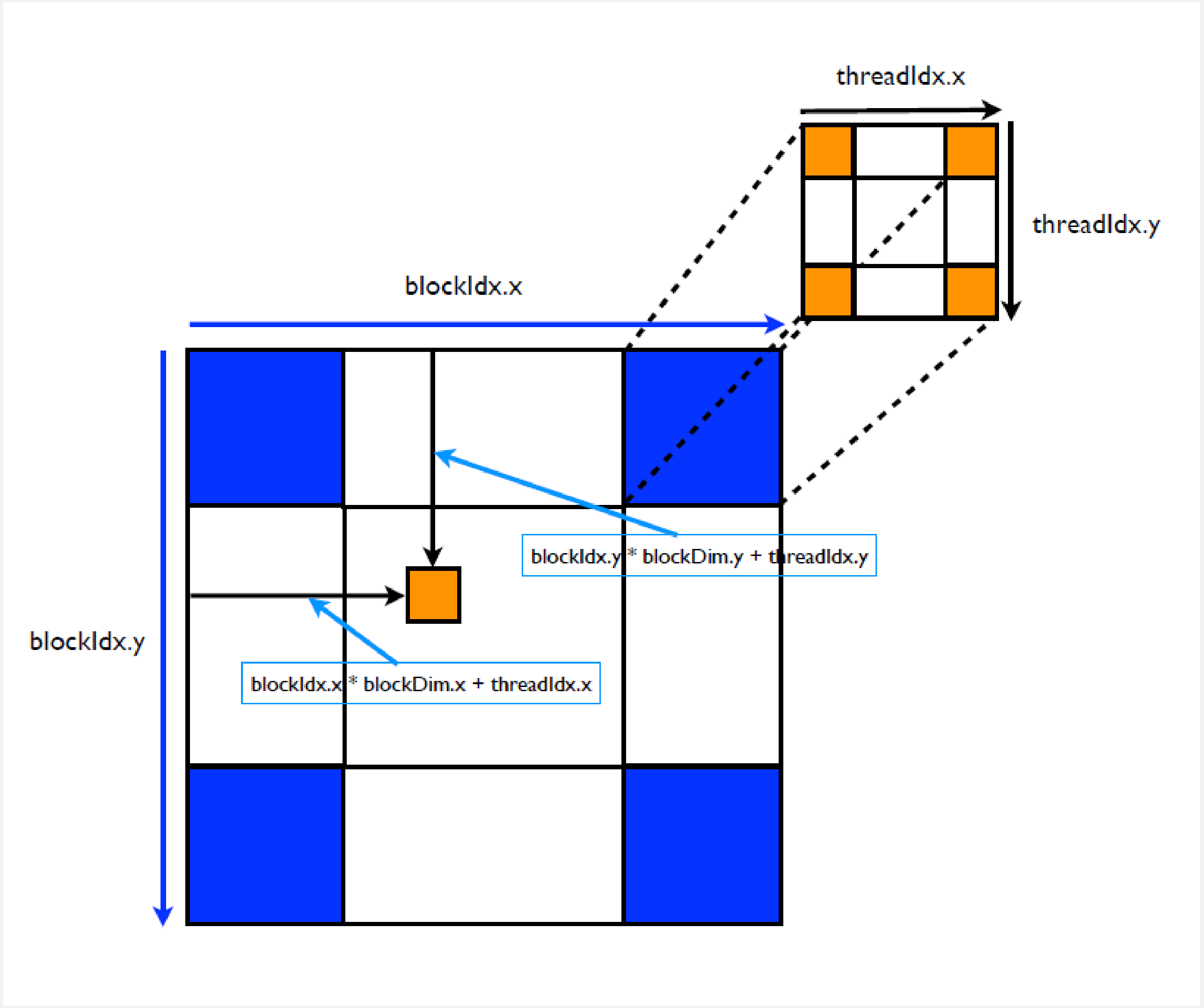
dim3 struct

```
struct __device_builtin__ dim3
{
   unsigned int x, y, z;
#if defined(__cplusplus)
   __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
   __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
   __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

Ex.

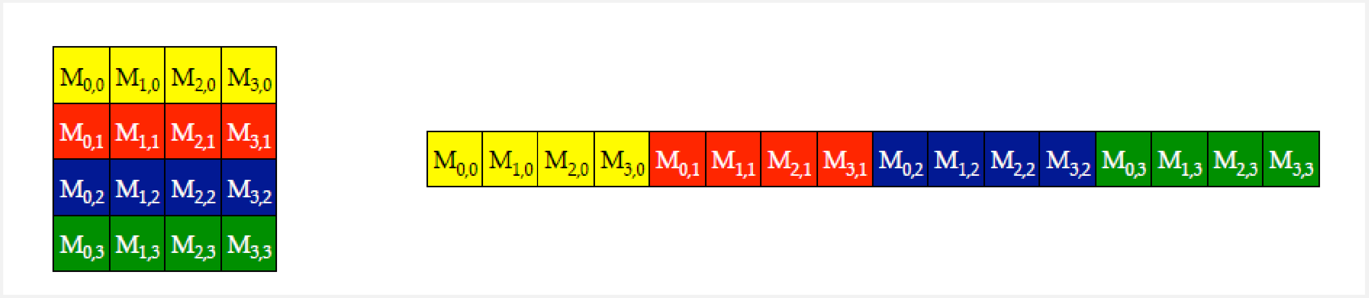
```
 dim3 grid(256);              // defines a grid of 256 x 1 x 1 blocks
 dim3 block(512,512);         // defines a block of 512 x 512 x 1 threads


foo<<<grid,block>>>(...);
```

2D Grids and 2D Blocks


threadIdx.x
threadIdx.y
blockIdx.x
blockIdx.y * blockDim.y + threadIdx.y
blockIdx.y
blockIdx.x * blockDim.x + threadIdx.x

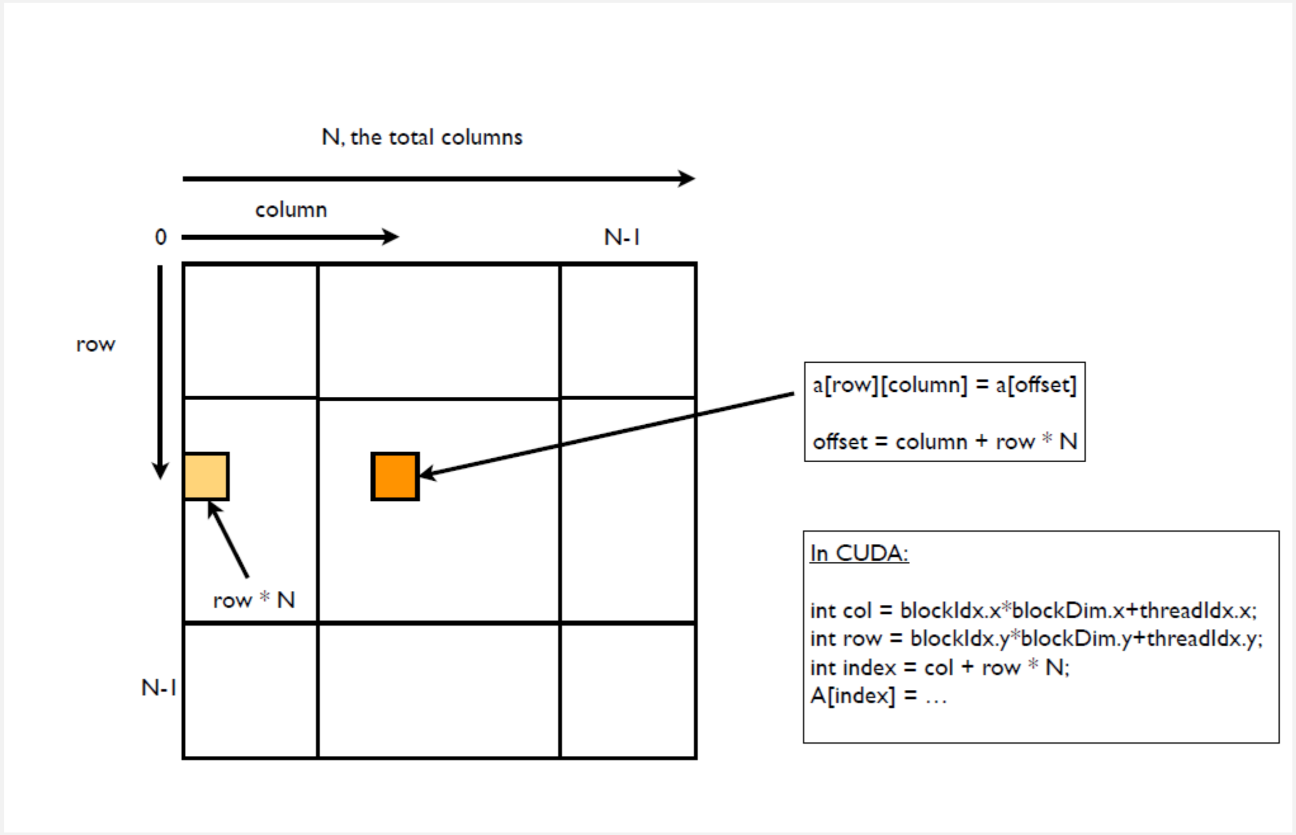
Flatten Matrices into Linear Memory

- Generally, the memory is allocated dynamically on device (GPU) and we cannot not use two-dimensional indices (e.g. A[row][column]) to access matrices.
- We will need to know how the matrix is laid out in memory and then compute the distance from the beginning of the matrix.
- C/C++ uses **row-major** order --- rows are stored one after the other in memory, i.e. row 0 then row 1 etc.



$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

Accessing Matrices in Linear Memory



a[row][column] = a[offset]

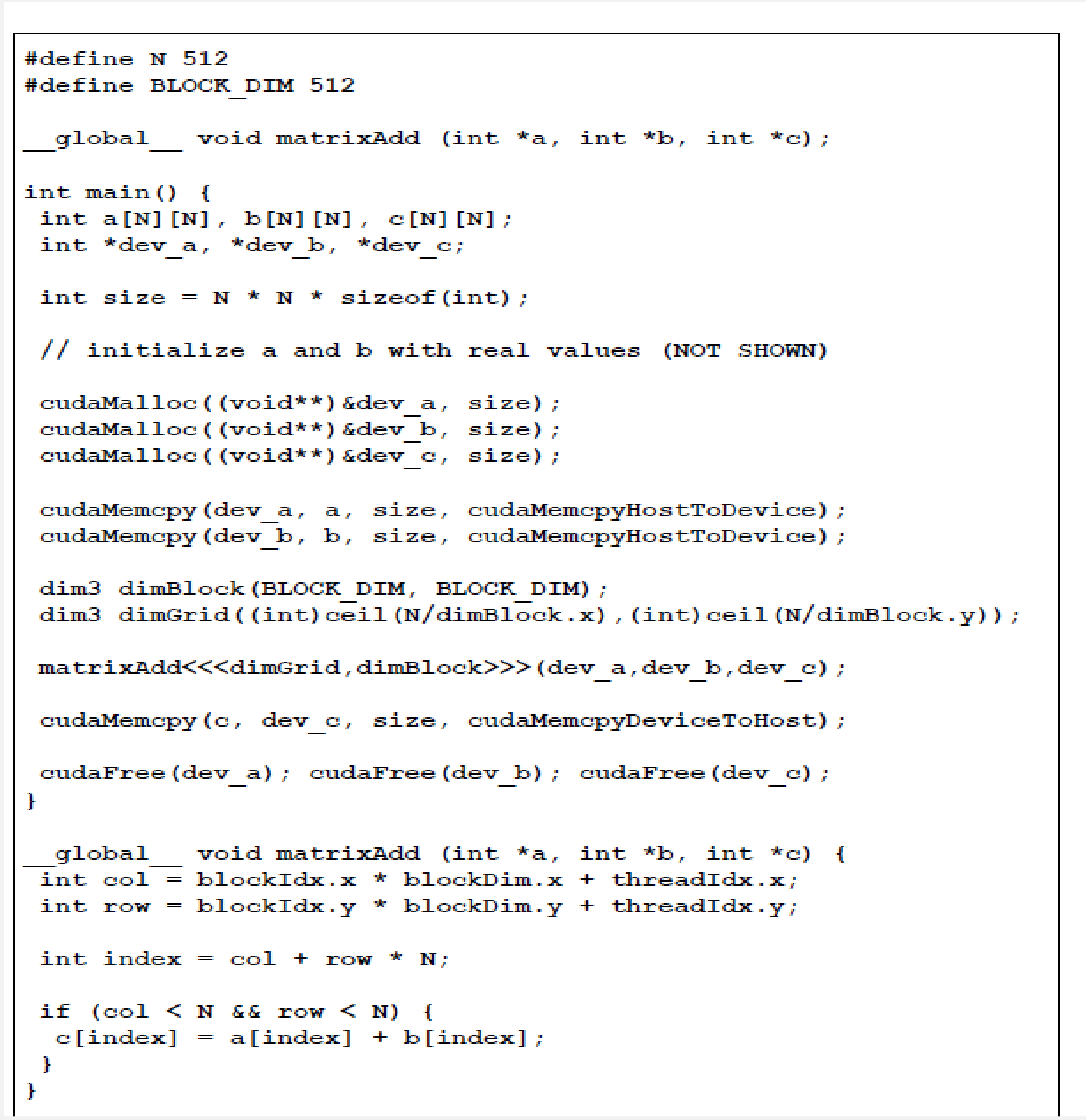
offset = column + row * N

In CUDA:

```
int col = blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int index = col + row * N;
A[index] = …
```

Matrix addition

- dim3 variables are used to set the Grid and Block dimensions.
- A global thread ID is calculated to index the column and row of the matrix.
- The linear index of the matrix is calculated .

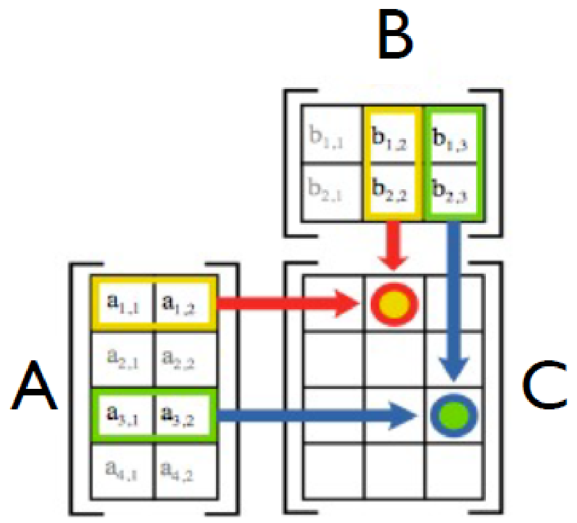


```
#define N 512
#define BLOCK_DIM 512

__global__ void matrixAdd (int *a, int *b, int *c);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 int size = N * N * sizeof(int);

 // initialize a and b with real values (NOT SHOWN)

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
 dim3 dimGrid((int)ceil(N/dimBlock.x),(int)ceil(N/dimBlock.y));

 matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

__global__ void matrixAdd (int *a, int *b, int *c) {
 int col = blockIdx.x * blockDim.x + threadIdx.x;
 int row = blockIdx.y * blockDim.y + threadIdx.y;

 int index = col + row * N;

 if (col < N && row < N) {
  c[index] = a[index] + b[index];
 }
}
```

Matrix Multiplication

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

- To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.
- Then place the sum in the appropriate position in the matrix C.



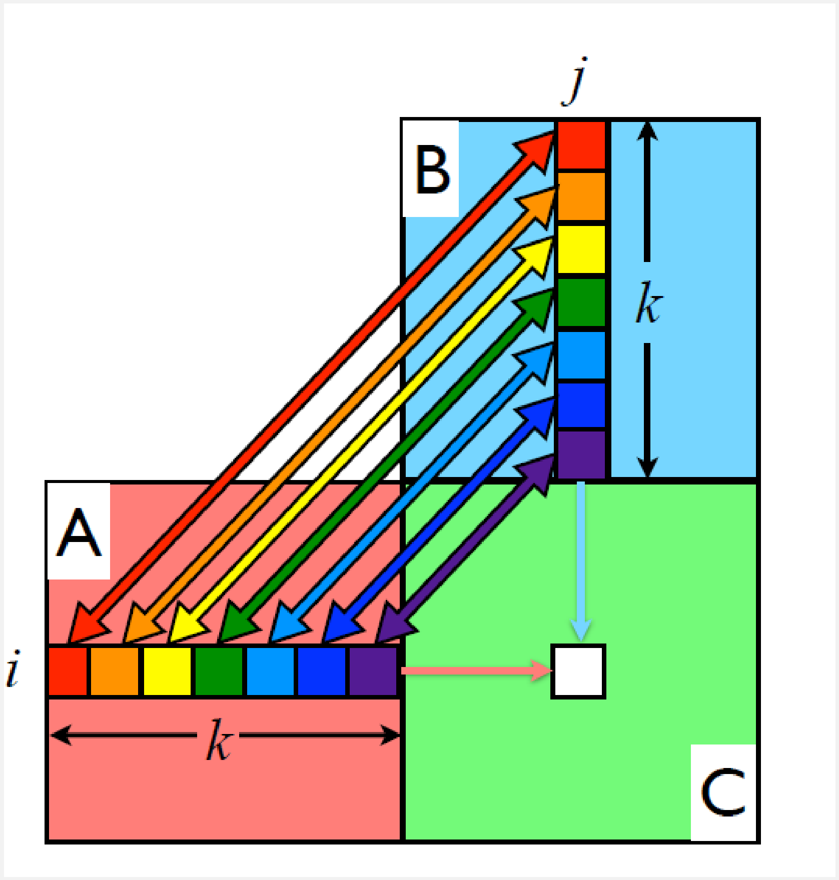
$$AB = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} (1*0)+(0*-2)+(-2*0) & (1*3)+(0*-1)+(-2*4) \\ (0*0)+(3*-2)+(-1*0) & (0*3)+(3*-1)+(-1*4) \end{bmatrix}$$

$$= \begin{bmatrix} 0+0+0 & 3+0+-8 \\ 0+-6+0 & 0+-3+-4 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix} = C$$

Parallel Matrix Multiplication



- To compute a single value of C(i,j), only a single thread is necessary to traverse the ith row of A and the jth column of B.

- Therefore, the number of threads needed to compute a square matrix multiply is O(N^2).

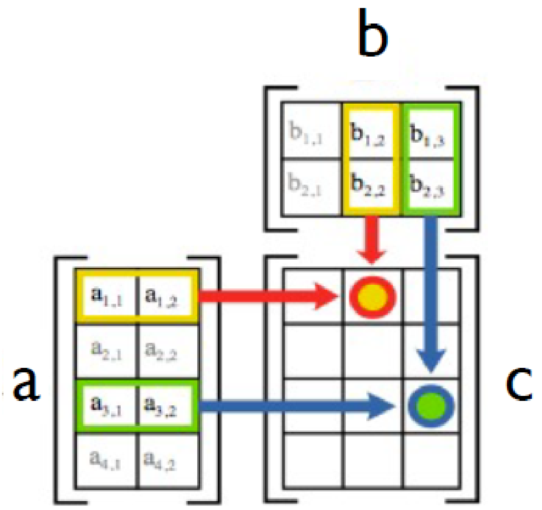
Sequential function => Kernel

C Function

```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < width; j++) {
      int sum = 0;
      for (int k = 0; k < width; k++) {
        int m = a[i][k];
        int n = b[k][j];
        sum += m * n;
      }
      c[i][j] = sum;
    }
  }
}
```

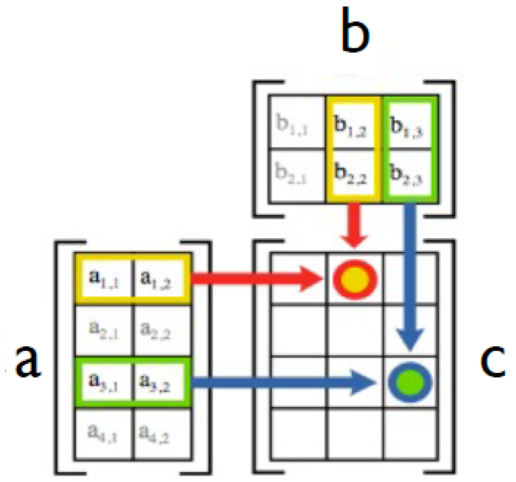
CUDA Kernel

```
__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```



…the program

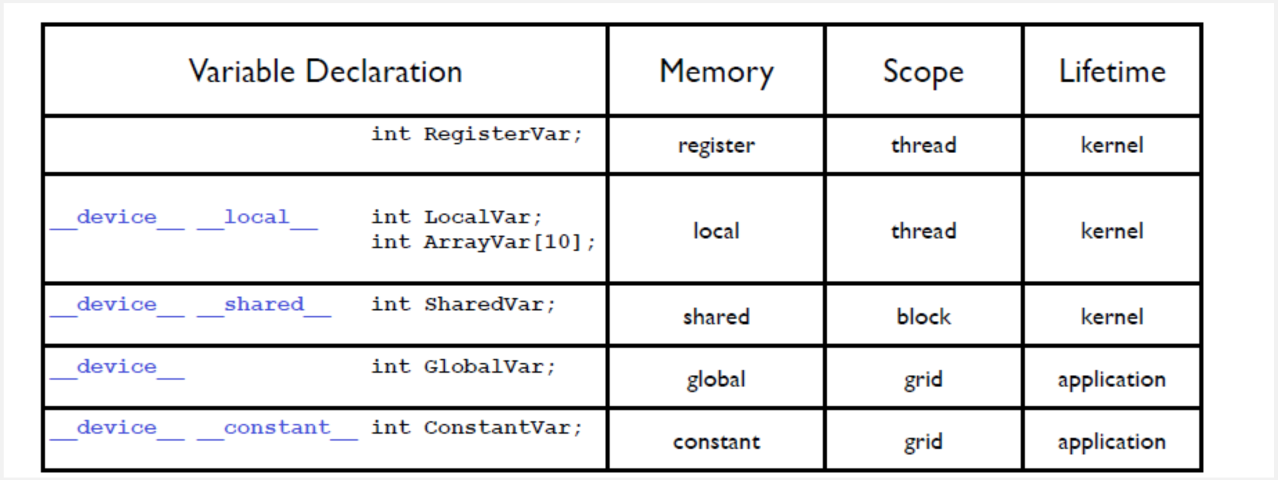
```
#define N 16
#include <stdio.h>

__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 // initialize matrices a and b with appropriate values

 int size = N * N * sizeof(int);
 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```



Floating Point Operations

- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

CUDA Memory Lifetimes and Scopes



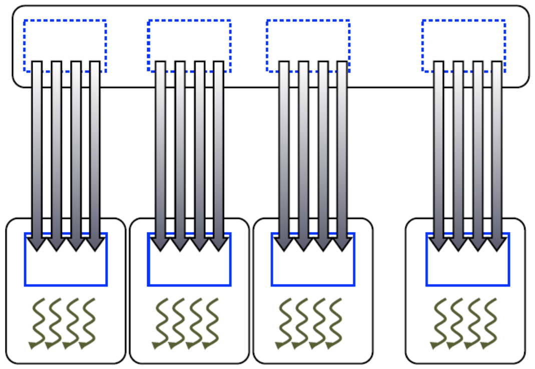
Variable Declaration	Memory	Scope	Lifetime
`int RegisterVar;`	register	thread	kernel
`__device__ __local__ int LocalVar;` `int ArrayVar[10];`	local	thread	kernel
`__device__ __shared__ int SharedVar;`	shared	block	kernel
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- Automatic variables without any qualifier reside in a register.
- Except arrays that reside in local memory
- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays and global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

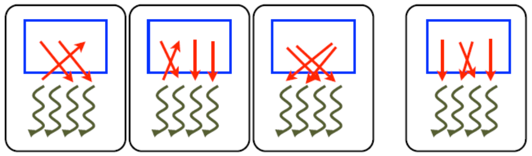
Memory Strategy

- Global memory (DRAM) is slower than shared memory.
- So, a profitable way of performing computation on the device is to **tile data** to take advantage of fast shared memory:
    - Partition data into subsets that fit into shared memory
    - Handle each data subset with one thread block by:
        - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
        - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element.
        - Copying results from shared memory to global memory

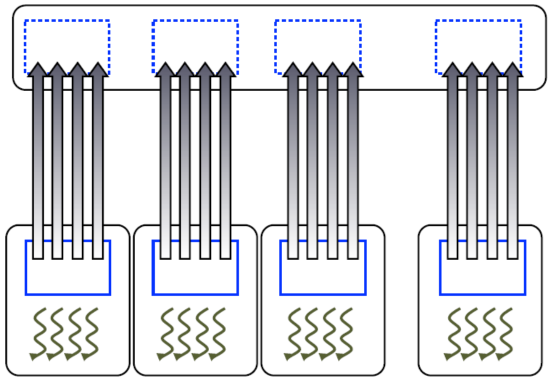
Programming Strategy



- Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.



- Perform the computation on the subset from shared memory; each thread

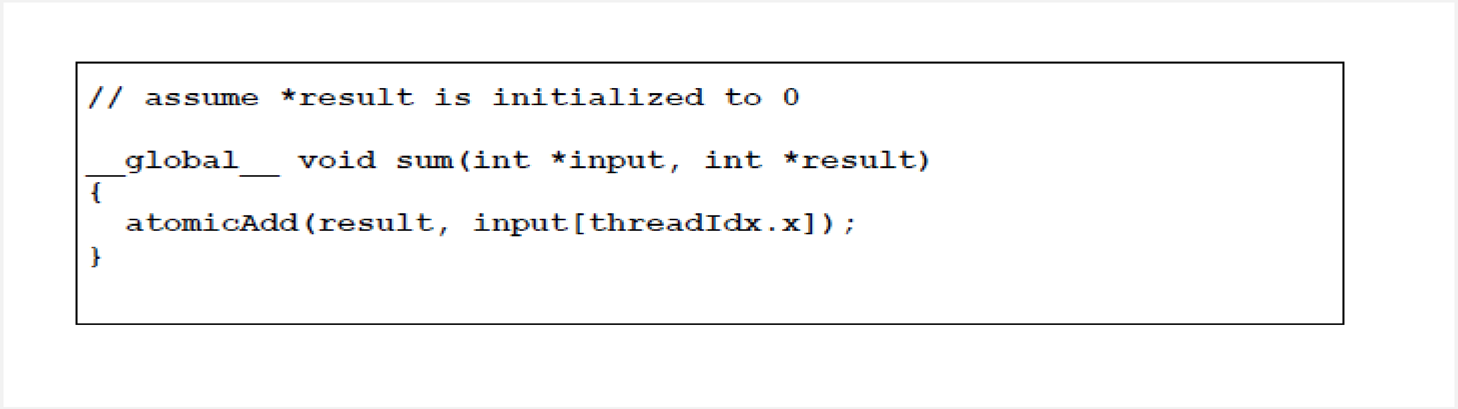


- Copy the results from shared memory back to global memory.

Atomics

- CUDA provides atomic operations to deal with race conditions.
- An atomic operation guarantees that only a single thread has access to a piece of memory while an operation completes.
- Atomic operations only work with signed and unsigned integers (except AtomicExch)
- Different types of atomic instructions:
    - Addition/subtraction: atomicAdd, atomicSub
    - Minimum/maximum: atomicMin, atomicMax
    - Conditional increment/decrement: atomicInc, atomicDec
    - Exchange/compare-and-swap: atomicExch, atomicCAS

Atomic operations



```
// assume *result is initialized to 0

__global__ void sum(int *input, int *result)
{
  atomicAdd(result, input[threadIdx.x]);
}
```

- An atomic operation is capable of reading, modifying, and writing a value back to memory without the interference of any other threads, which guarantees that a race condition won't occur.
- Atomic operations in CUDA generally work for both shared memory and global memory.
- Atomics are slower than normal load/store.
    - fast for data in shared memory
    - slower for data in device memory
- Atomic operations in shared memory are generally used to prevent race conditions between different threads within the same thread block.
- Atomic operations in global memory are used to prevent race conditions between two different threads regardless of which thread block they are in.
- After this kernel exits, the value of *result will be the sum of the inputs.
- Atomic operations are expensive; they imply serialized access to a variable.

Streams

- CUDA largely focus on massively data-parallel execution on GPUs.
- Task parallelism is also possible on GPUs: rather than simultaneously computing the same function on lots of data (data parallelism), task parallelism involves doing two or more completely different tasks in parallel.
- Task parallelism is not as flexible as data parallelism, but it allows the extraction

of even more optimization from GPU based implementation of algorithms.

- A CUDA Stream is a sequence of operations (commands) that are executed in order.
- CUDA streams can be created and executed together and interleaved although the "program order" is always maintained within each stream.
- Streams proved a mechanism to overlap memory transfer and computations operations in different stream for increased performance if sufficient resources are available.

Creating Streams

• Done by creating a stream object and associated it with a series of CUDA commands that then becomes the stream. CUDA commands have a stream pointer as an argument.

• Cannot use regular cudaMemcpy with streams, need asynchronous commands for concurrent operation.

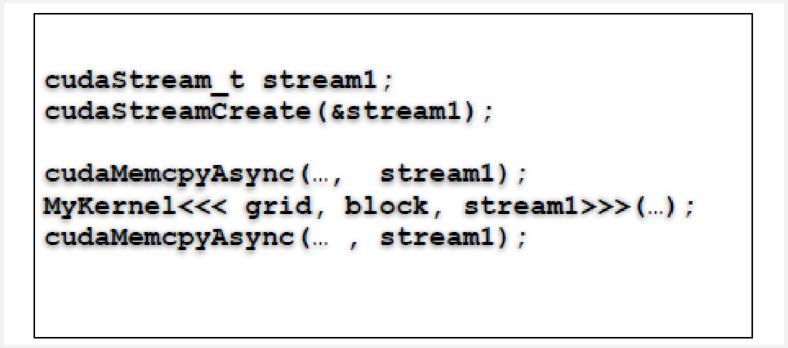
• cudaMemcpyAsync is an asynchronous version of cudaMemcpy that copies date to/from host and the device.

• May return before copy complete

• A stream argument specified.

• Needs “page-locked” memory.

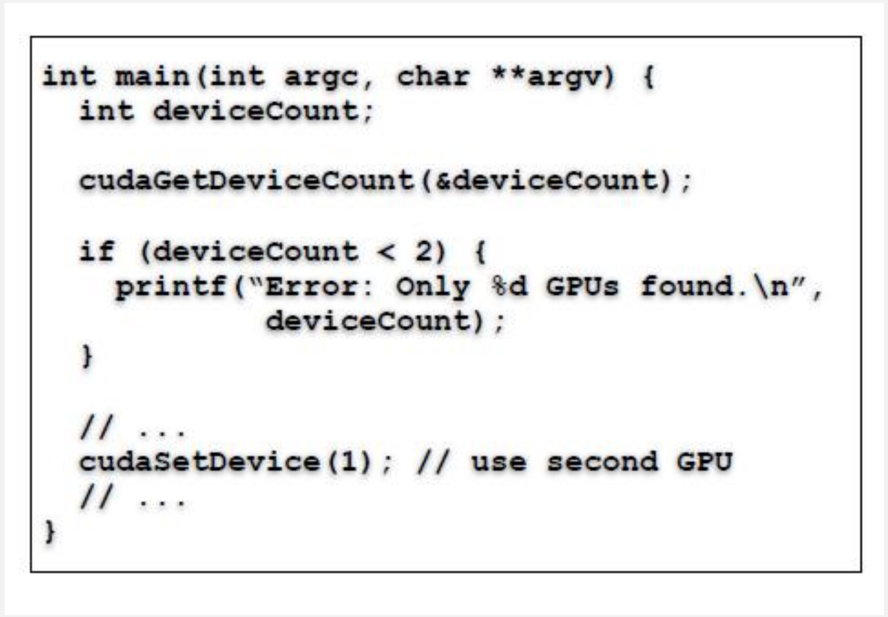
• Multiple calls to cudaStreamCreate can create multiple streams. Then multiple kernels (even different ones!) can be executed on each stream.



```
cudaStream_t stream1;
cudaStreamCreate(&stream1);

cudaMemcpyAsync(…,  stream1);
MyKernel<<< grid, block, stream1>>>(…);
cudaMemcpyAsync(… , stream1);
```

Multi-GPU Programming

- GPUs do not share global memory.
- A kernel executing on one GPU cannot access memory on another GPU.
- Inter-GPU communication:
  - Application code is responsible for transferring data between GPUs as necessary.
  - Data travels across the PCIe bus, even when GPUs are connected to the same PCIe switch.



```
int main(int argc, char **argv) {
  int deviceCount;

  cudaGetDeviceCount(&deviceCount);

  if (deviceCount < 2) {
    printf("Error: Only %d GPUs found.\n",
            deviceCount);
  }

  // ...
  cudaSetDevice(1); // use second GPU
  // ...
}
```