

# CSC 447: Parallel Programming for Multi-Core and Cluster Systems

CUDA Memory and Blocks

Instructor: Haidar M. Harmanani

Spring 2018



## CONCEPTS

# Heterogeneous Computing

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

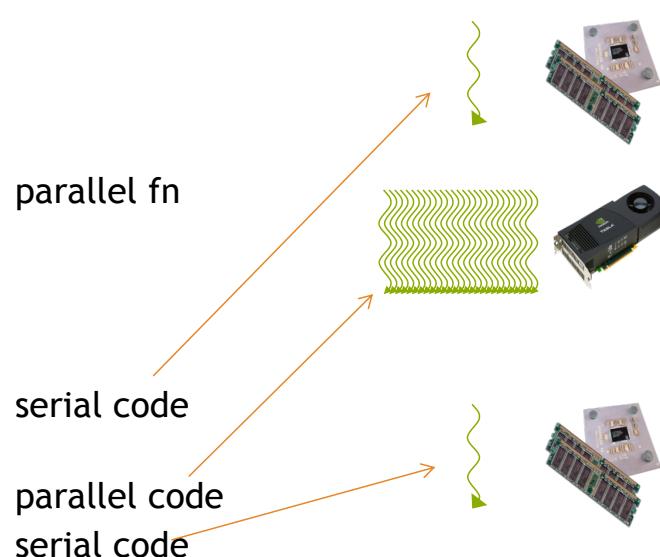


Spring 2018

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

3 | LAU  
للمحة الامريكية  
Lebanese American University

# Heterogeneous Computing

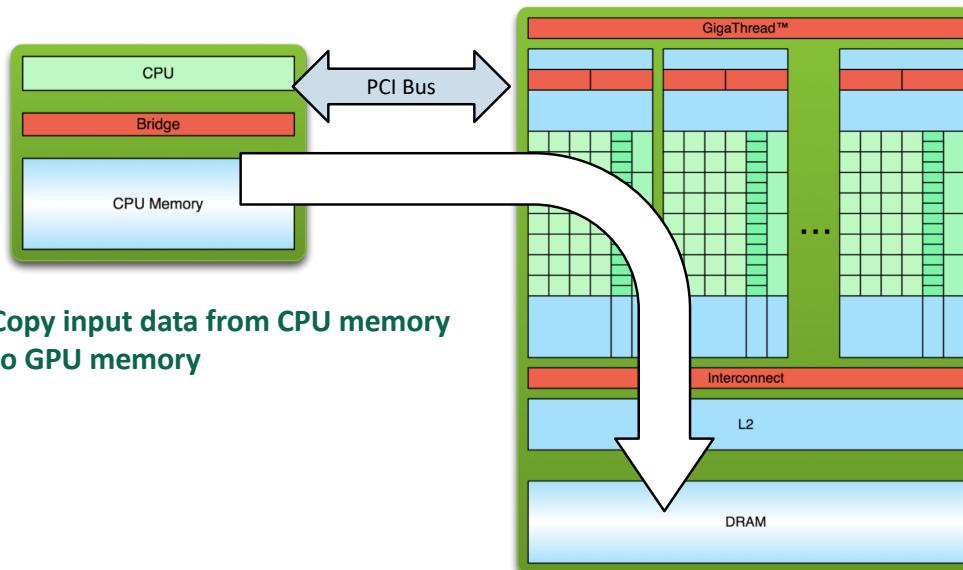


Spring 2018

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

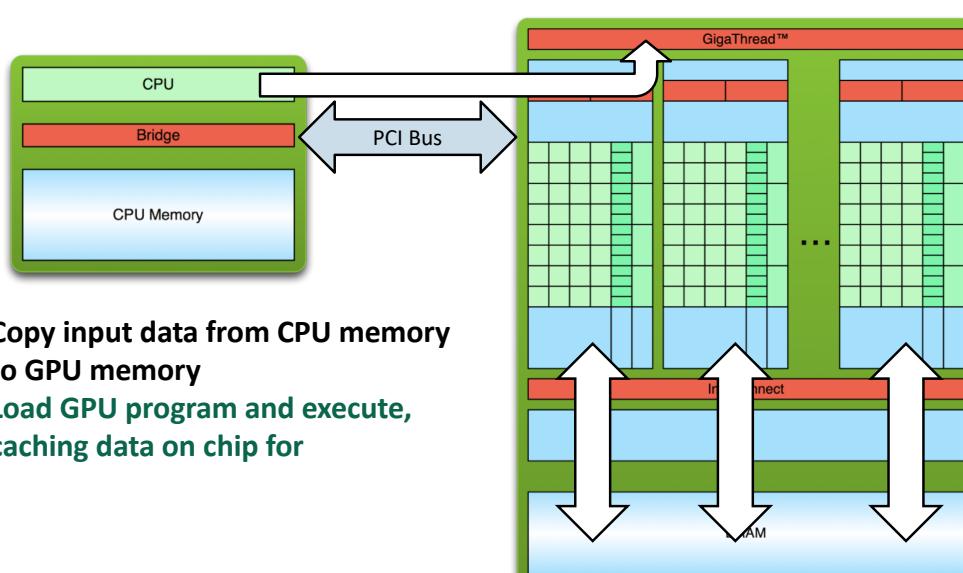
4 | LAU  
للمسيحية الأمريكية  
Lebanese American University

# Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

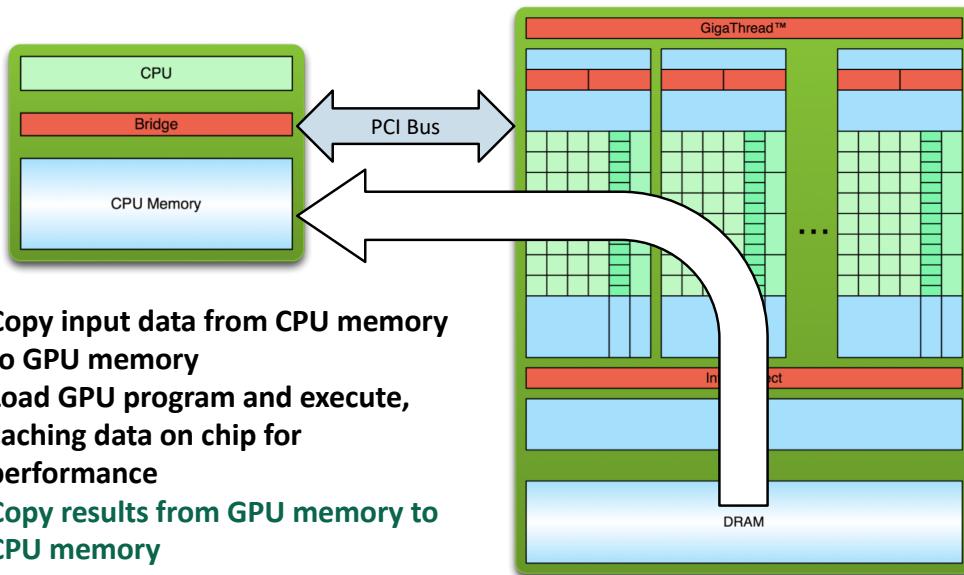
# Simple Processing Flow



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



# Simple Processing Flow



Hello World!

Heterogeneous Computing  
Hello World!  
Compiling

# CUDA

- CUDA provides a simple grid model
  - Can decompose into CUDA blocks
- A linear distribution of work within a single block leads to an ideal decomposition into CUDA blocks
  - Can assign up to sixteen blocks per SM and we can have up to 16 or 32 SMs

# Blocks, Threads, and Grids

- CUDA C allows the definition of a group of blocks in two dimensions
- Use two-dimensional indexing for problems such as matrix math or image processing
  - Avoid annoying translations from linear to rectangular indices
- A collection of parallel blocks is called a grid
- No dimension of a launch of blocks may exceed 65,535

# CUDA Hello World

- We will discuss over the following few slides a couple of simple examples
- We will start with the classical `hello_world` example

## Hello World (Device Code)

- Recall
  - Host—The CPU and its memory (host memory)
  - Device—The GPU and its memory (device memory)

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

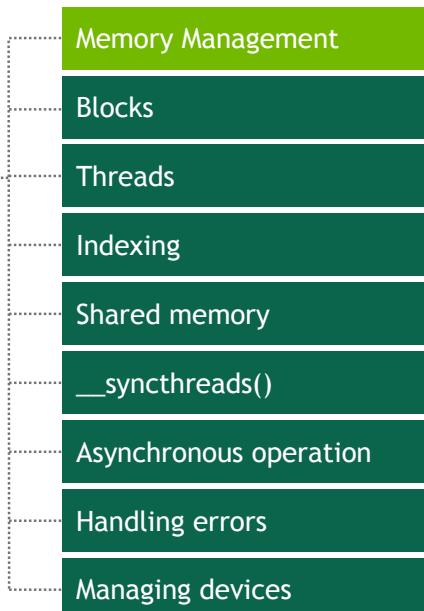
### Output:

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

# Hello World (Device Code)

- CUDA C keyword **`__global__`** indicates that a function
    - Runs on the device
    - Called from host code
  - **`kernel<<< 1, 1 >>>();`**
    - Triple angle brackets mark a call from hostcode to devicecode
    - Sometimes called a “kernel launch”
  - nvcc splits the source file into host and device components
    - NVIDIA’s compiler handles device functions like `kernel()`
    - Standard host compiler handles host functions like `main()`
      - o gcc
      - o Microsoft Visual C

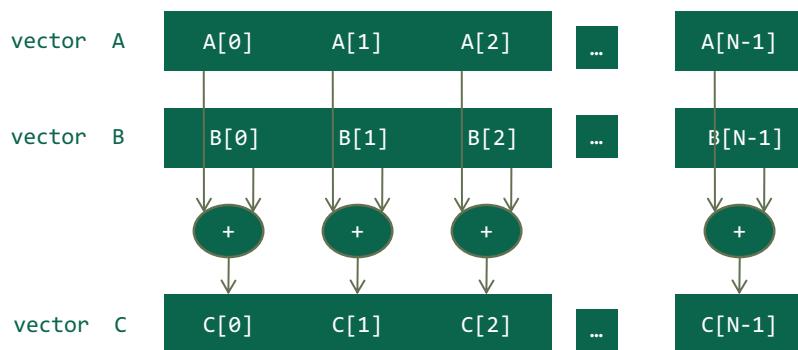
# Introduction to CUDA C



# Memory Management

- **host** and **device** memory are distinct entities in CUDA:
  - Device pointers point to GPU memory
  - May be passed to and from host code
  - May not be dereferenced from host code
- Host pointers point to CPU memory
  - May be passed to and from device code
  - May not be dereferenced from device code
- Basic CUDA API for dealing with device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`

# Vector Addition



# Vector Addition: Typical Solution

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++)
        h_C[i] = h_A[i] + h_B[i];
}
int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

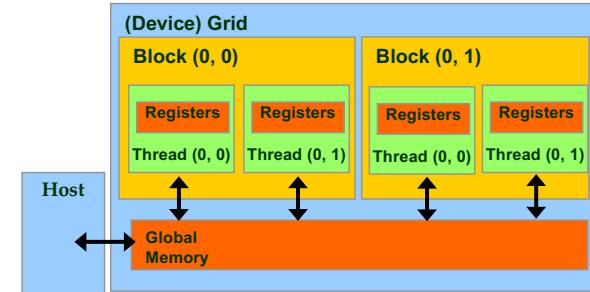
## Memory Management

- 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()`



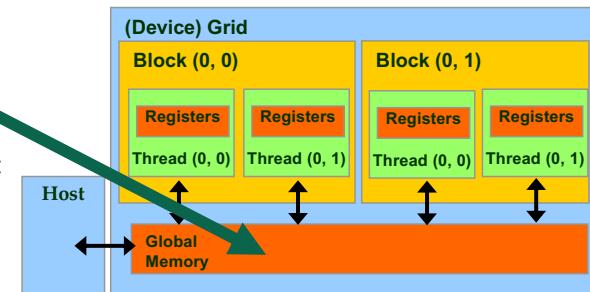
# Partial Overview of CUDA Memories

- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory



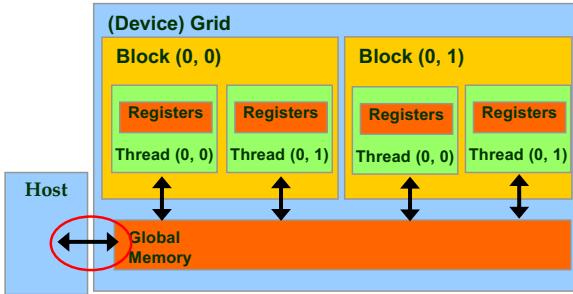
# CUDA Device Memory Management API functions

- `cudaMalloc()`
  - Allocates an object in the device global memory
  - Two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object in terms of bytes
- `cudaFree()`
  - Frees object from device global memory
  - One parameter
    - Pointer to freed object



# Host-Device Data Transfer API functions

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer



- Transfer to device is asynchronous

## Vector Addition on the GPU

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

- `vecAdd()`
  - `__global__` indicates that the function runs on the device and called from host code
  - ...so a, b, and c must point to device memory

# Adding two numbers on the GPU

```
int main( void )
{
    int a, b, c;
    int *dev_a, *dev_b, *dev_c;
    int size = sizeof( int );
                                // host copies of a, b, c
    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );
    a = 2;lock
    b = 7;
    // copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );
                                // device copies of a, b, c
                                // we need space for an integer

    // launch add() kernel on GPU, passing parameters
    vecAdd<<< 1, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

We can do without  
this as of CUDA 6

## *Kernel Launch for one block with one thread*

# In Practice, Check for API Errors in Host Code

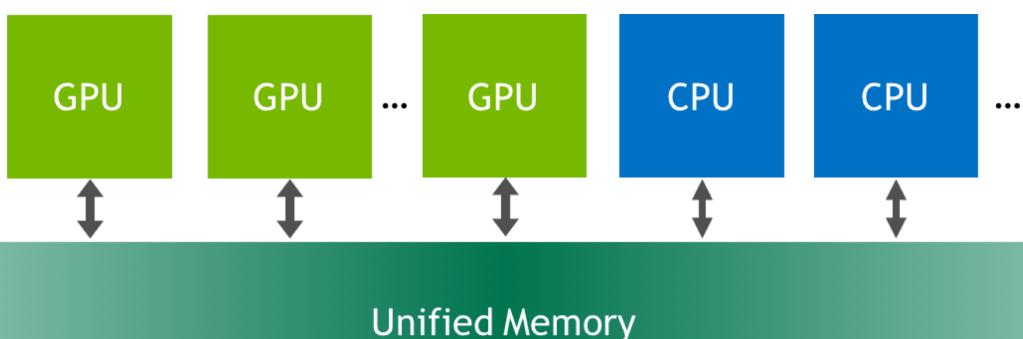
```
cudaError_t err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

# Unified Memory in CUDA 6

- GPU and CPU memories are physically distinct and separated by the PCI-Express bus.
- As of CUDA 6 and starting with the Kepler, NVidia bridged the CPU-GPU divide
  - Unified Memory creates a pool of managed memory that is shared between the CPU and GPU
  - Managed memory is accessible to both the CPU and GPU using a single pointer.
  - The system automatically migrates data between the CPU and the GPU memories

# Unified Memory



# “cudaMallocManaged” vs. “cudaMalloc”

- Some have benchmarked both and noted that is slightly “cudaMallocManaged” slower than “cudaMalloc”
- The following can be noted:
  - The main bottleneck is on the PCIE bus
  - `cudaMallocManaged` is of interest to a non-proficient CUDA programmer
    - Simpler learning curve
  - The *Maxwell* and *Pascal* architectures provide HW support for unified memory so `cudaMallocManaged()` provides better performance on those architectures

## Back to Our Addition Example: Unified Memory (CUDA 6 and beyond)

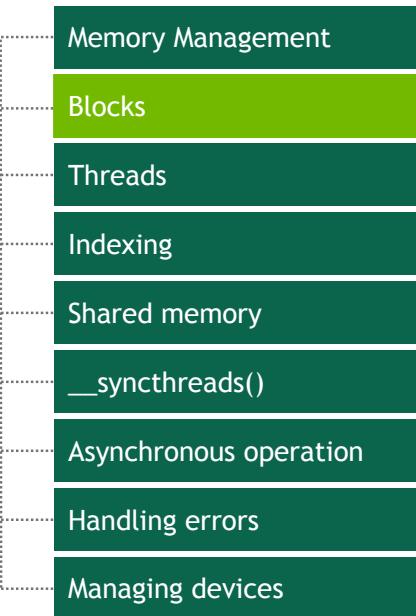
```
__global__ void vecAdd( int *a, int *b, int *c ) {
    *c = *a + *b;
}

int main( void )
{
    int *a, *b, *c;           // host copies of a, b, c
    int size = sizeof (int);   // The total number of bytes per vector
    // Allocate memory
    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);
    // launch add() kernel on GPU, passing parameters
    vecAdd<<< 1, 1 >>> ( a, b, c );
    cudaDeviceSynchronize(); // Wait for the GPU to finish
    // Free all our allocated memory
    cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

*Run one block with one thread*

*Blocks until the device has completed all preceding requested tasks.*

## CUDA Blocks



## Moving to Parallel

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

```
vecAdd<<< 1, 1 >>>();  
                  |  
                  v  
vecAdd<<< N, 1 >>>();
```

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

# Vector Addition on the GPU

- Well, since GPUs are about massive parallelism we will rewrite the add() method such that
  - Each invocation of add() is referred to as a *block*
  - Kernel can refer to its block's index with the variable `blockIdx.x`
  - Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`
- By using `blockIdx.x` to index into the array, each block handles a different index

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

# Vector Addition on the GPU

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

$$C[3] = a[3] + b[3]$$

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

# Vector Addition on the GPU: N Blocks

```
#define N 512
int main( void )
{
    int *a, *b, *c;                      // a, b, c
    int size = N * sizeof( int);          // The total number of bytes per vector

    // Allocate memory
    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);

    random_ints( a, N );
    random_ints( b, N );

    // launch add() kernel on GPU using N Parallel Blocks
    vecAdd<<< N, 1 >>>( a, b, d);

    cudaDeviceSynchronize(); // Wait for the GPU to finish

    // Free all our allocated memory
    cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

*Launch N blocks with 1 thread each*