LAU
الجامعة اللبنانية الأميركية
Lebanese American University

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

CUDA Memory and Blocks

Instructor: Haidar M. Harmanani

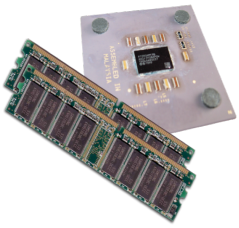
Spring 2018

CONCEPTS

- Heterogeneous Computing
- Hello World!
- Compiling

Heterogeneous Computing

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



Host

Device

Heterogeneous Computing

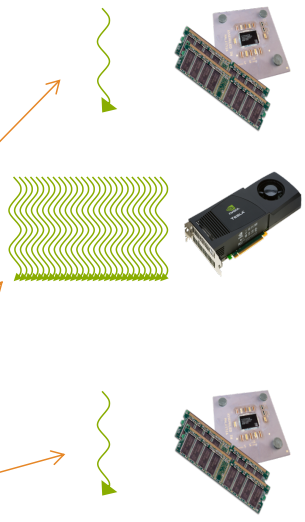
```
#include <iostream>
#include <algorithm>

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

parallel fn

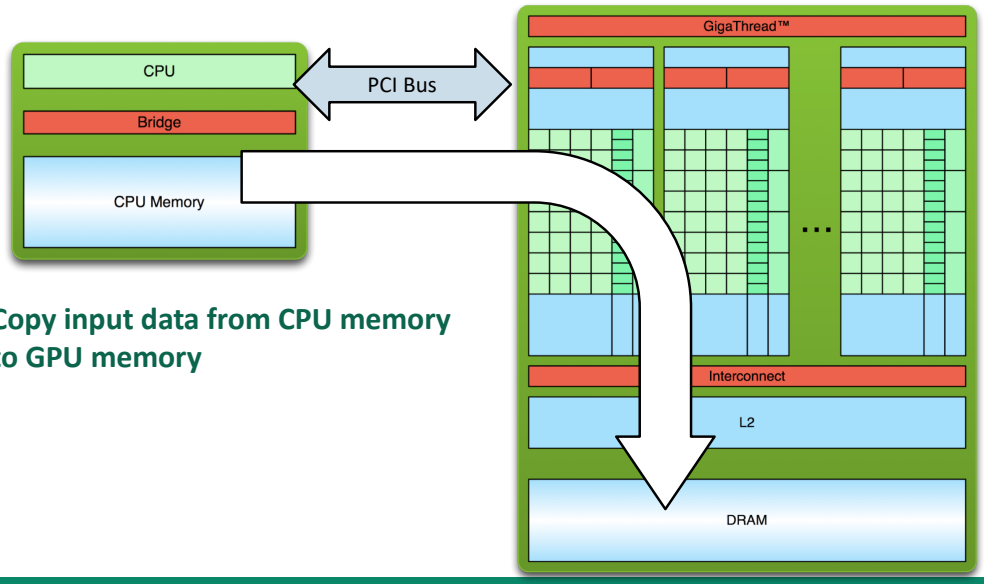
serial code

parallel code

serial code

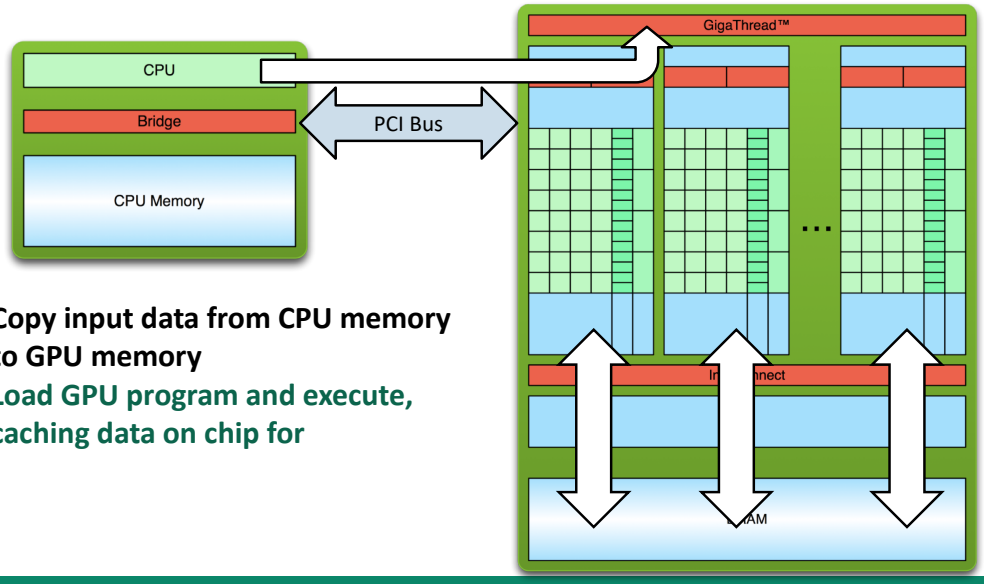


Simple Processing Flow



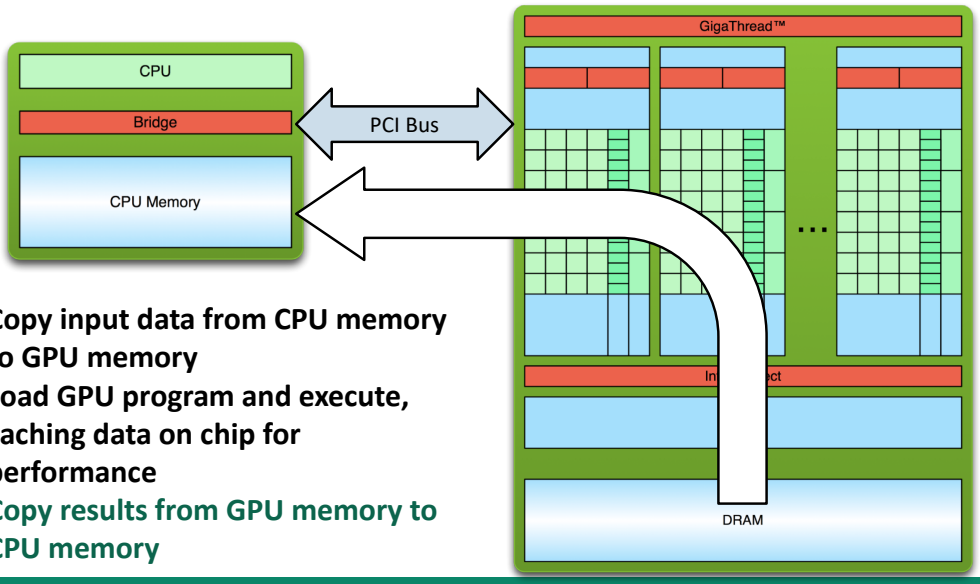
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

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!

- 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 32 SMs
- A block can have up to 1024 threads


```
vlsi:/usr/local/cuda/extras/demo_suite # ./deviceQuery
./deviceQuery Starting...


 CUDA Device Query (Runtime API) version (CUDART static linking)


Detected 1 CUDA Capable device(s)


Device 0: "GeForce GTX TITAN X"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 12202 MBytes (12794789888 bytes)
  (24) Multiprocessors, (128) CUDA Cores/MP:     3072 CUDA Cores
  GPU Max Clock rate:                            1076 MHz (1.08 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 3145728 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
```

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()
    - gcc
    - Microsoft Visual C

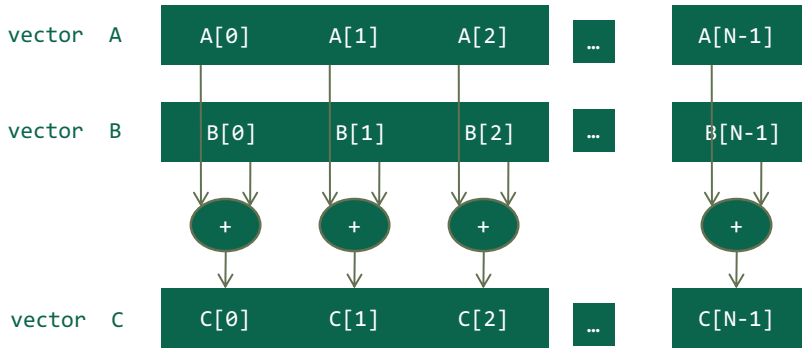
Introduction to CUDA C

- Memory Management
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

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);
}
```

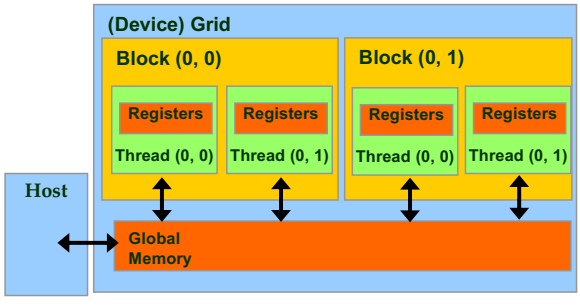
Memory Management

- 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()`

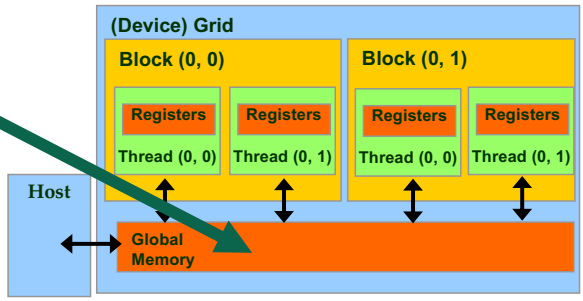
Partial Overview of CUDA Memories

- 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



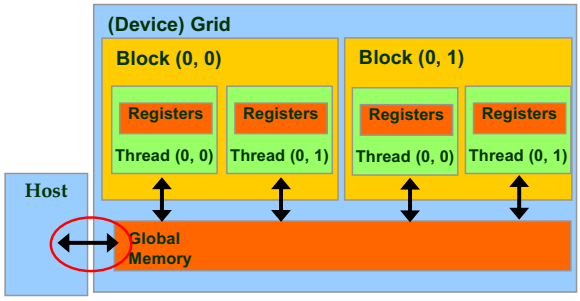
CUDA Device Memory Management API functions

- 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;                          // host copies of a, b, c
      int *dev_a, *dev_b, *dev_c;           // device copies of a, b, c
      int size = sizeof( int);              // we need space for an integer

      // 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);

    // 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

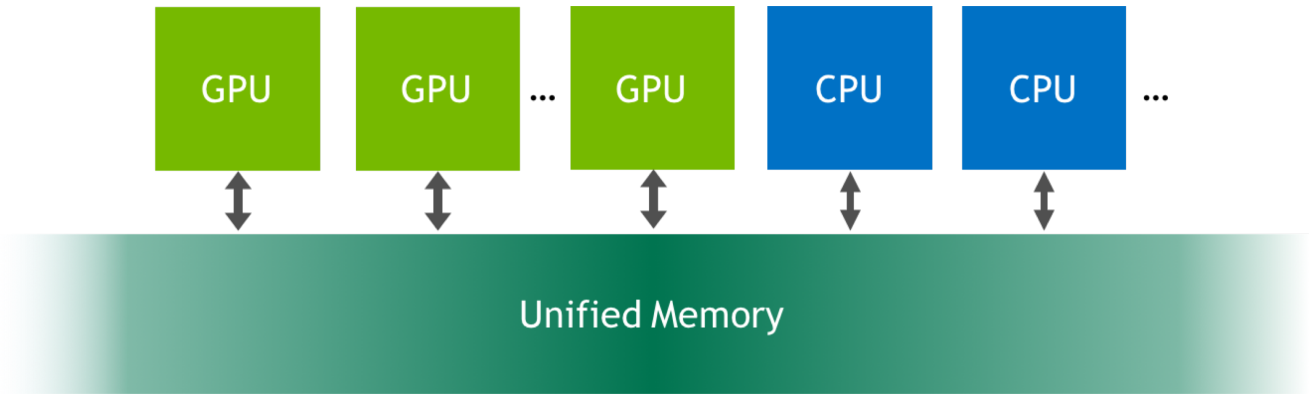
In Practice, Check for API Errors in Host Code

```
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 memproes 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.