# CUDA Kernels

Lecture 13
April 15, 2025

# To Dos

Reading for next time (GPUs!)

Program #5 due Thursday

# Issues That Impact GPU Performance

- Global Memory Bandwidth
  - Need to keep parallel compute resources fed with data
  - Need to have enough computation to use all resources and hide memory latency

# Global Memory Bandwidth

Ideal

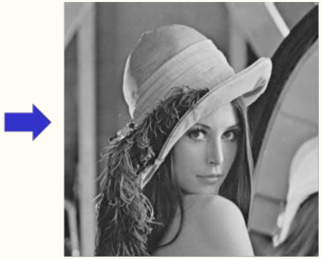
Reality

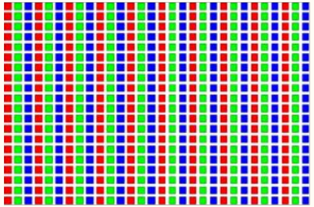
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

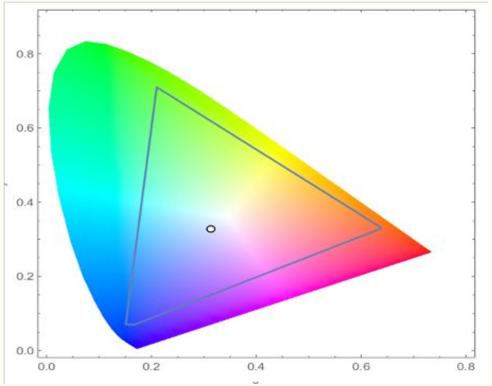
# Issues That Impact GPU Performance

- Global Memory Bandwidth
  - Need to keep parallel compute resources fed with data
  - Need to have enough computation to use all resources and hide memory latency
- Need to prevent load imbalance
  - Need to decrease likelihood of divergent control paths
- Need to prevent serialization due to locks

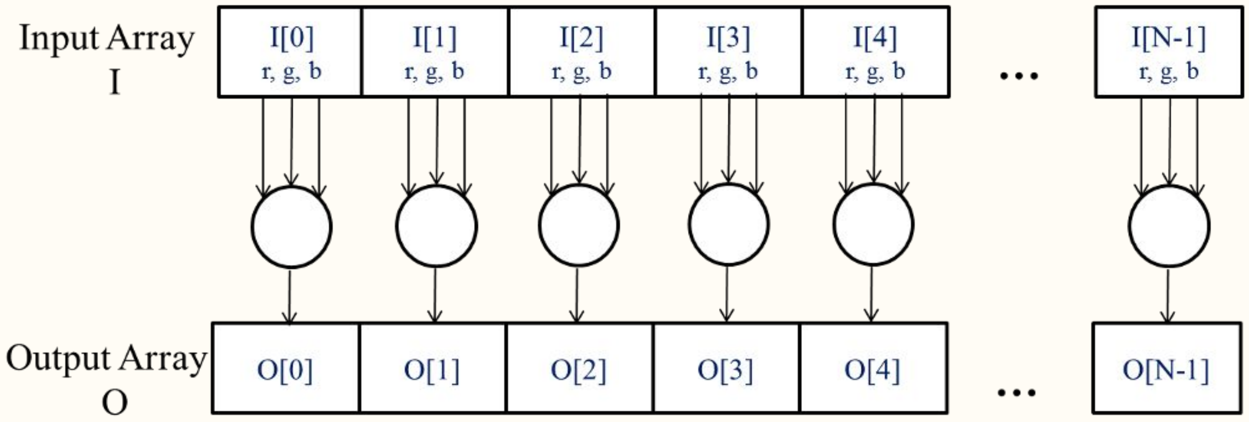
# Conversion of a color image to grey–scale image







# Pixels can be calculated independently




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

# CUDA/OpenCL – Execution Model

Integrated host+device app C program

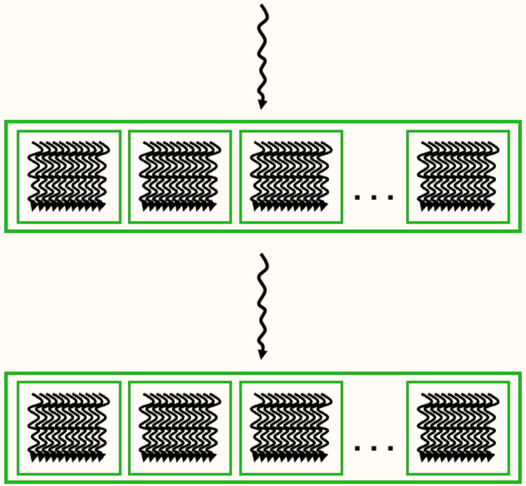
- Serial or modestly parallel parts in host C code
- Highly parallel parts in device SPMD kernel C code

**Serial Code (host)**

**Parallel Kernel (device)**
**KernelA<<< nBlk, nTid >>>(args);**

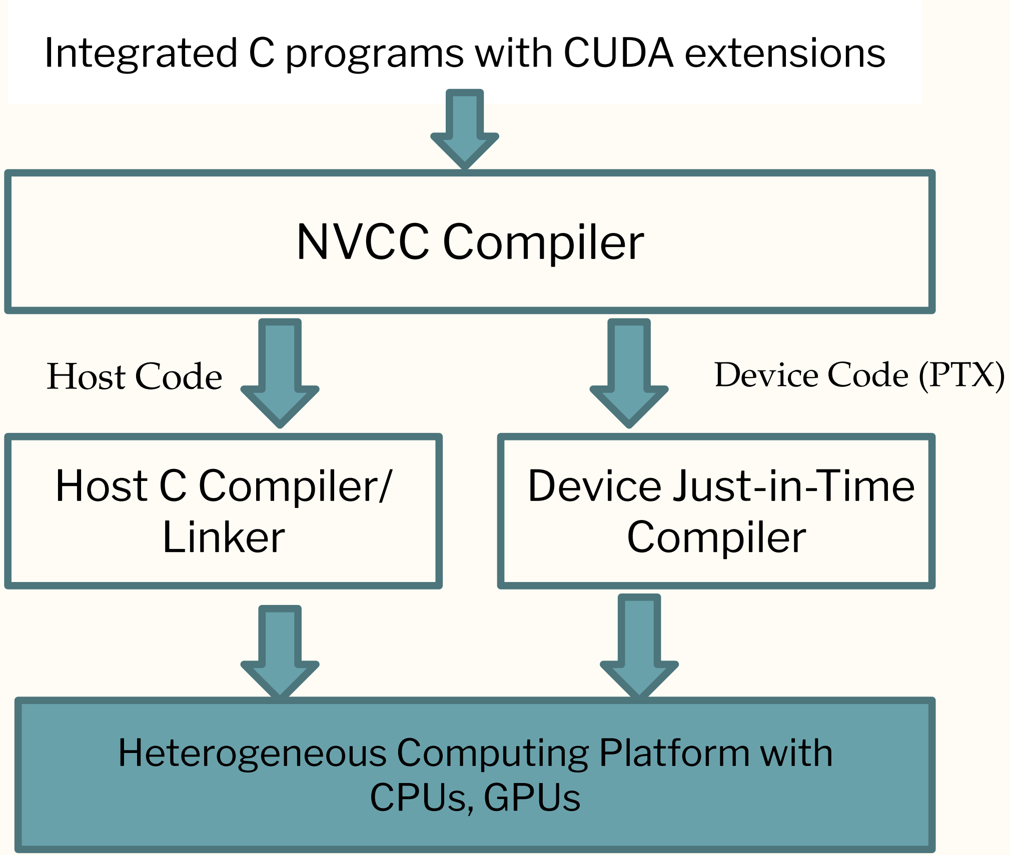
**Serial Code (host)**

**Parallel Kernel (device)**
**KernelB<<< nBlk, nTid >>>(args);**



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Compiling A CUDA Program




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

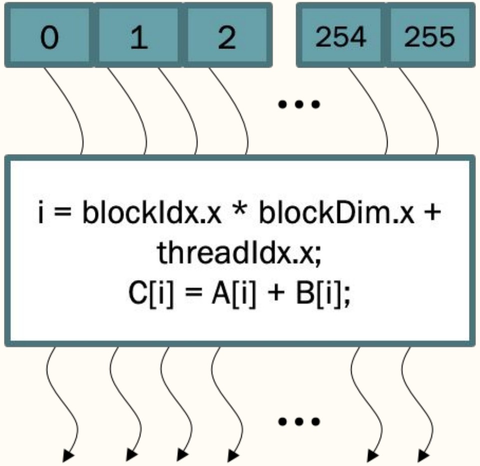
# Hierarchy of Computation

- A kernel describes the work performed on a single set of data by a single thread of control
    - Each thread will have its own registers in HW
- Many threads execute a single kernel simultaneously (data parallelism)
- CUDA provides a mechanism for organizing and naming threads
    - All threads involved in a kernel are collectively part of a grid
    - A grid is composed of many blocks, organized in 3 dimensional space
    - Each block contains many threads, organized in 3 dimensional space
    - Specify configuration of threads on kernel invocation

# Arrays of Parallel Threads

A CUDA kernel is executed by a grid (array) of threads

- All threads in a grid run the same kernel code (SPMD)
- Each thread has an index that it uses to compute memory addresses and make control decisions

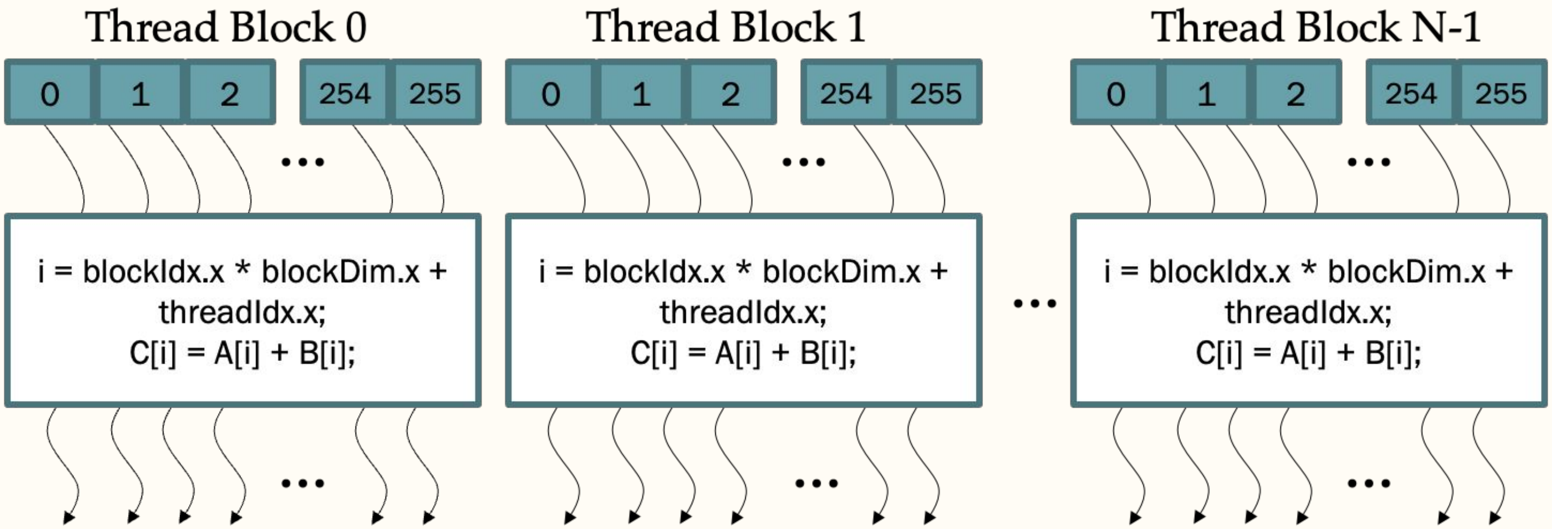



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Thread Blocks: Scalable Cooperation

Divide thread array into multiple blocks

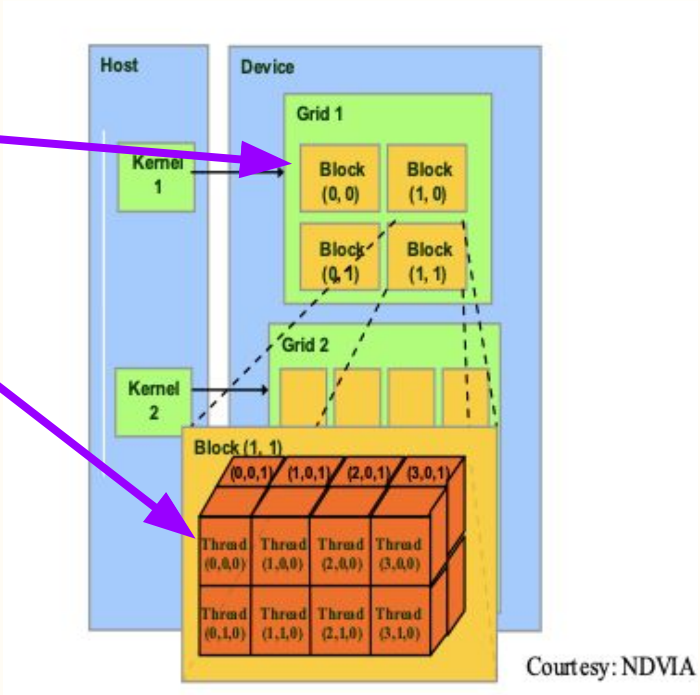
- Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
- Threads in different blocks cannot cooperate
- Blocks contain same number of threads (up to 1024)




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

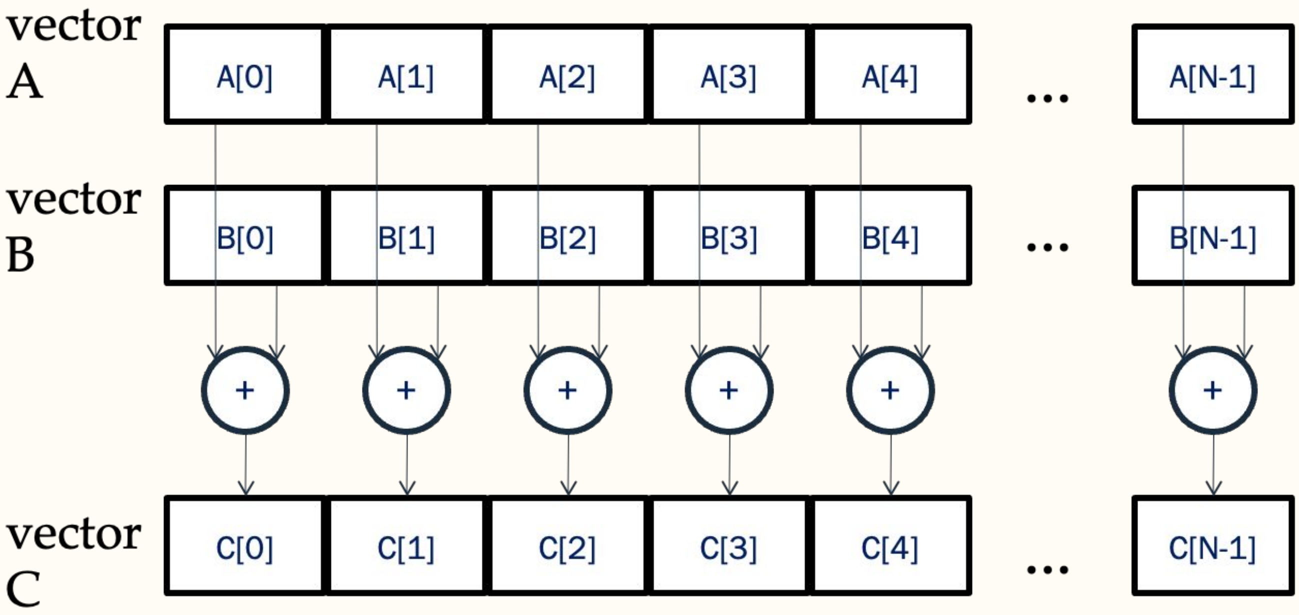
# blockDim, gridDim, blockIdx, and threadIdx variables

- Defined in kernel
- gridDim – config of blocks in grid
  - struct w/ 3 unsigned ints: x, y, z
  - Values set by kernel invocation
- blockDim – config of threads in block
  - struct w/ 3 unsigned ints: x, y, z
  - Values set by kernel invocation call
- Thread identifiers: blockIdx and threadIdx
  - blockIdx – gives all threads in block a common block coordinate (e.g. area code)
    - x, y, and z fields depending on dimensions specified by blockDim
  - threadIdx – gives all threads in block a way to distinguish themselves (e.g. local phone #)

# Calculating thread ids

- 1D grid of 1D blocks
  - blockIdx.x * blockDim.x + threadIdx.x
- 1D grid of 2D blocks
  - blockIdx.x*blockDim.x*blockDim.y + threadIdx.y*blockDim.x + threadIdx.x
- 2D grid of 1D blocks
  - blockId = blockIdx.y * gridDim.x + blockIdx.x
  - threadId = blockId *blockDim.x + threadIdx.x
- It's like calculating array index offsets with multidimensional arrays

# Vector Addition – Conceptual View




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Vector Addition – Traditional C Code

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
      C[i] = A[i] + B[i];
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Vector Addition – Traditional C Code

```
int main()
{
    // Memory allocation for A_h, B_h, and C_h
    float *A_h = (float*)malloc(N*sizeof(float));
    float *B_h = (float*)malloc(N*sizeof(float));
    float *C_h = (float*)malloc(N*sizeof(float));

    // I/O to read A_h and B_h, N elements
    // note memset is for chars/ints not floats
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Heterogeneous Computing vecAdd Host Code

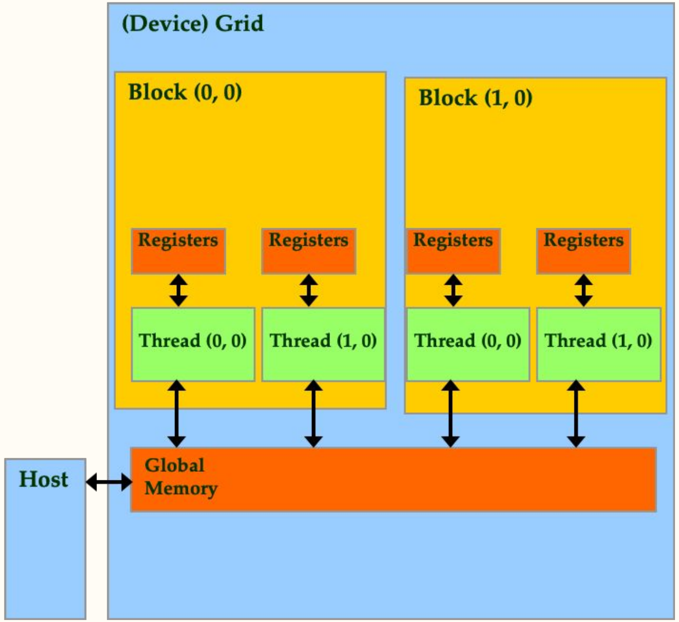
```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
   int size = n* sizeof(float);
   float* A_d, B_d, C_d;

1.  // Allocate device memory for A, B, and C
    // copy A and B to device memory

2. // Kernel launch code – to have the device
    // to perform the actual vector addition

3. // copy C from the device memory
    // Free device vectors
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Partial Overview of CUDA Memories

- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory

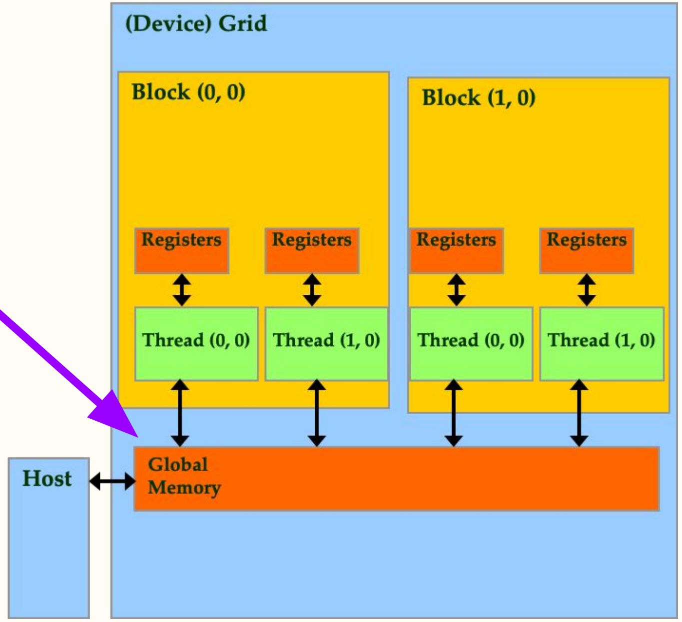
- Host code can
  - Transfer data to/from per grid global memory




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# CUDA Device Memory Management API functions

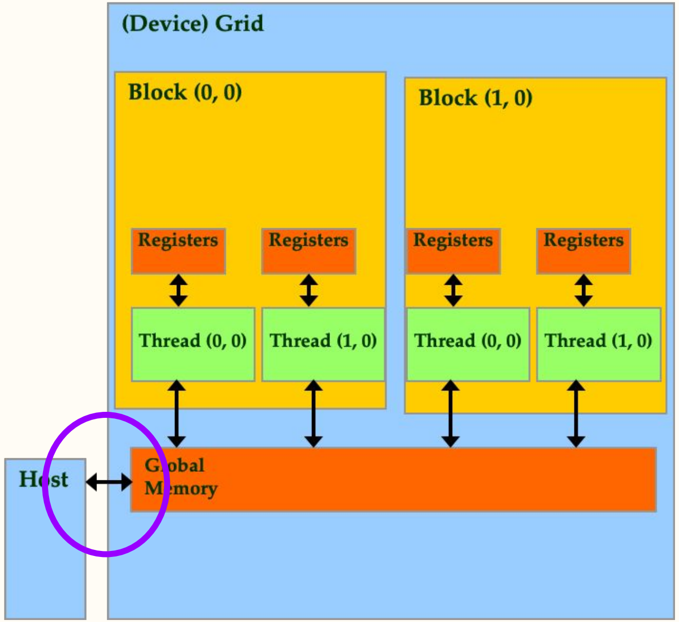
- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - Address of a pointer to the allocated object (void**)
    - Size of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# 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 synchronous




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

```
void vecAdd(float* A, float* B, float* C, int n)
{
   int size = n* sizeof(float);
   float* A_d, B_d, C_d;

1.  // Allocate device memory for A, B, and C
    cudaMalloc((void **) &A_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // copy A and B to device memory
    cudaMalloc((void **) &C_d, size);

2. // Kernel launch code – to be shown later
```

Missing Error Checking Code!

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

```
3. // copy C from the device memory
   cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

   // Free device vectors
   cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);

}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Error Checking CUDA Calls

```
__host__ __device__ cudaError_t cudaMalloc ( void** devPtr , size_t size )
```

Allocate memory on the device.

**Parameters**

`devPtr`
- Pointer to allocated device memory

`size`
- Requested allocation size in bytes

**Returns**

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

**Description**

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. cudaMalloc() returns cudaErrorMemoryAllocation in case of failure.

The device version of cudaFree cannot be used with a `*devPtr` allocated using the host API, and vice versa.

**Note:**

- Note that this function may also return error codes from previous, asynchronous launches.
- Note that this function may also return cudaErrorInitializationError, cudaErrorInsufficientDriver or cudaErrorNoDevice if this call tries to initialize internal CUDA RT state.
- Note that as specified by cudaStreamAddCallback no CUDA function may be called from callback. cudaErrorNotPermitted may, but is not guaranteed to, be returned as a diagnostic in such case.

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g37d37965bfb4803b6d4e59ff26856356

# Example: Vector Addition Kernel

Device code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i < n)
        C_d[i] = A_d[i] + B_d[i];
}
```

Host code

```
int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<< ceil(n / 256.0), 256 >>>(A_d, B_d, C_d, n);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016 ECE408/CS483, University of Illinois, Urbana-Champaign

# Specifying Thread Organization

- Specified after kernel name, before argument list
  - kernelName<<< B, T>>>>(argument list)

- B: configuration of blocks in grid
  - 1D: can just use a number indicating number of blocks
  - 2D or 3D: need to specify dimensions using dim3 variable
- T: configuration of threads in blocks
  - 1D: can just use a number indicating number of threads
  - 2D or 3D: need to specify dimension using dim3 variable

# dim3 type

- Integer vector type based on uint3
- Any dimension not specified initialized to 1
- Components accessible as variable.x, variable.y, variable.z
- Examples
    - dim3 block1D(5);
    - dim3 block2D(5, 5);
    - dim3 block3D(5, 5, 5);
- blockDim and gridDim are both of dim3 type

# More on Kernel Launch

Host code

```
int vecAdd(float* A, float* B, float* C, int n)
{
  // A_d, B_d, C_d allocations and copies omitted
  // Run ceil(n/256) blocks of 256 threads each
  dim3 DimGrid(n/256, 1, 1);
  if (n % 256)
        DimGrid.x++;
  dim3 DimBlock(256, 1, 1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

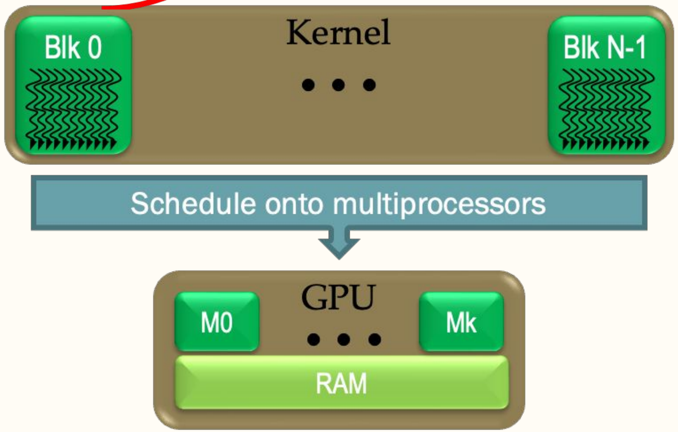
- Any call to a kernel function is asynchronous, explicit synch needed for blocking

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g37d37965bfb4803b6d4e59ff26856356

# Kernel execution in a nutshell

```
__host__
Void vecAdd()
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);

vecAddKernel<<<DimGrid,DimBlock>>>
(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
     float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```




https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g37d37965bfb4803b6d4e59ff26856356

# More on 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
  - Each "__" consists of two underscore characters
- A kernel function must return void
  - __device__ and __host__ can be used together

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g37d37965bfb4803b6d4e59ff26856356