ECE408 /CS483/CSE408 Spring 2018

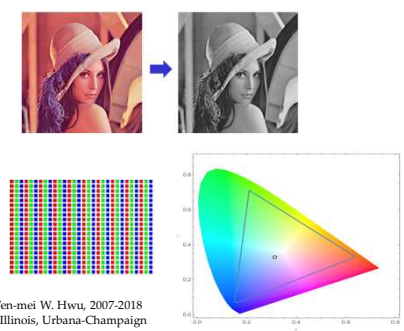
Applied Parallel Programming

Lectures 2: Introduction to CUDA C and Data Parallel Programming

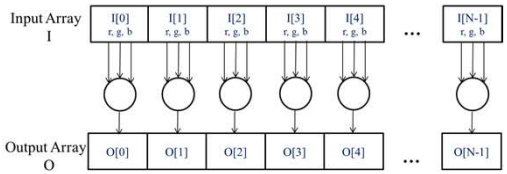
Objective

- To learn the basic concept of data parallel computing
- To learn the basic features of the CUDA C programming interface

A Data Parallel Computation Example: Conversion of a color image to grey–scale image

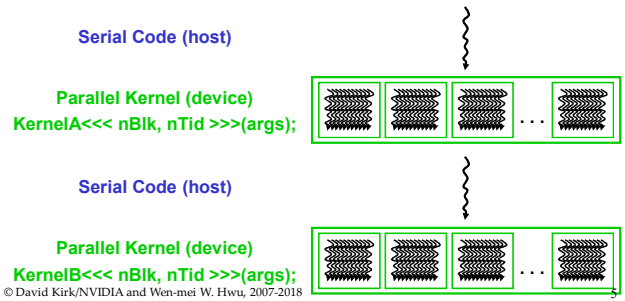


The pixels can be calculated independently of each other



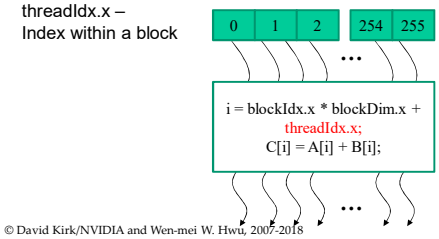
CUDA/OpenCL – Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code



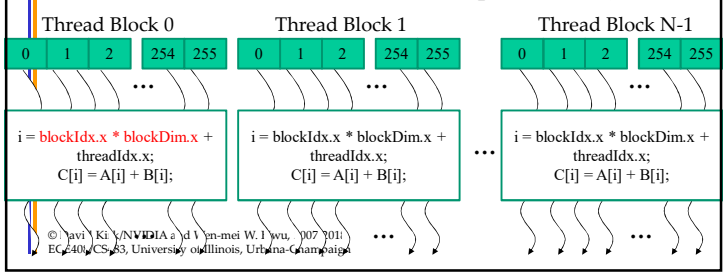
Arrays of Parallel Threads

- 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



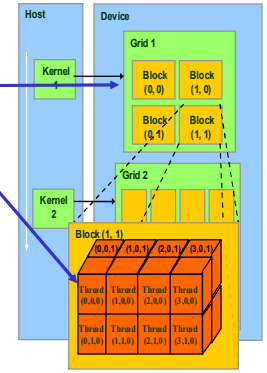
Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization** (to be covered later)
  - Threads in different blocks cooperate less (later)

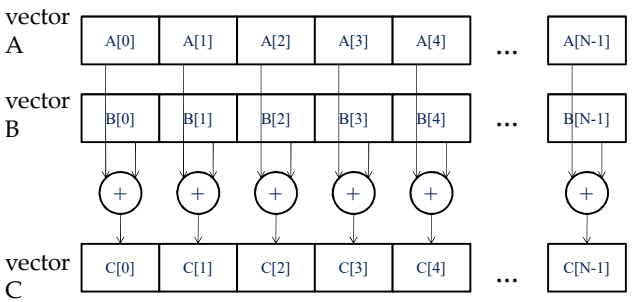


blockIdx and threadIdx

- 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
  - Vectors, matrices, tensors
  - Solving PDEs on volumes
  - …



Vector Addition – Conceptual View



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];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

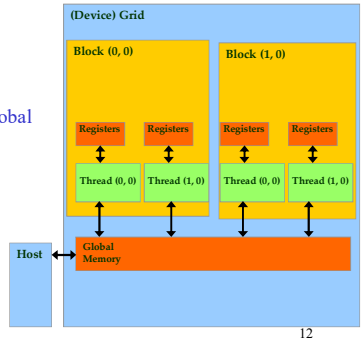

Heterogeneous Computing vecAdd Host Code

```
#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
}
```


Partial Overview of CUDA Memories

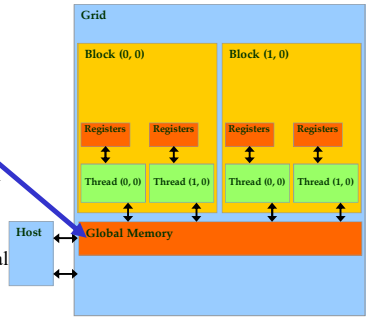
- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory
- Host code can
  - Transfer data to/from per grid global memory

We will cover more later.



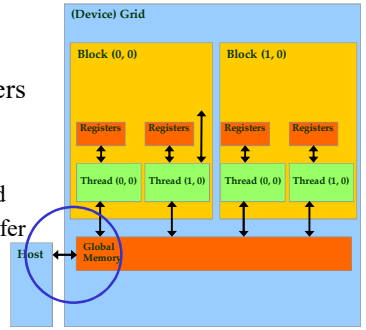
CUDA Device Memory Management API functions

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




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

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

    // Allocate device memory for
     cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code – to be shown later
       …
3. // Transfer C from device to host
     cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
     cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```


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 = blockIdx.x * blockDim.x + threadIdx.x ;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
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);
}
```

Example: Vector Addition Kernel

```
// 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 = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

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
  vecAddKernel<<<ceil(n/256.0),256>>>(A_d, B_d, C_d, n);
}
```

17

17

More on Kernel Launch

Equivalent 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 from CUDA 1.0 on, explicit synch needed for blocking

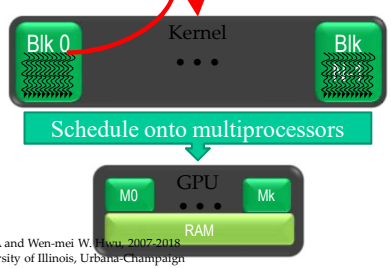
18

18

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];
}
```



19

19

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

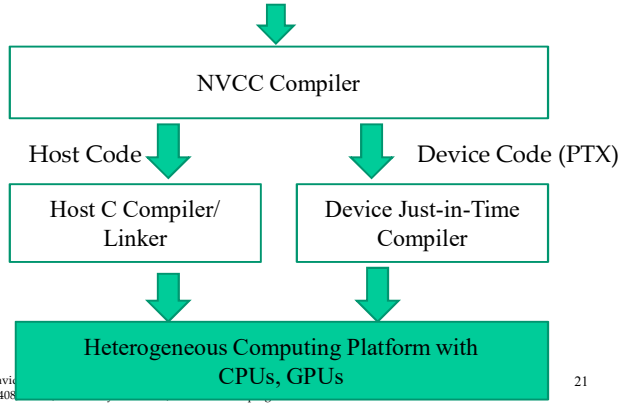
- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together

20

20

Compiling A CUDA Program

Integrated C programs with CUDA extensions



QUESTIONS?