Fundamentals of CUDA 2

Prof. Seokin Hong

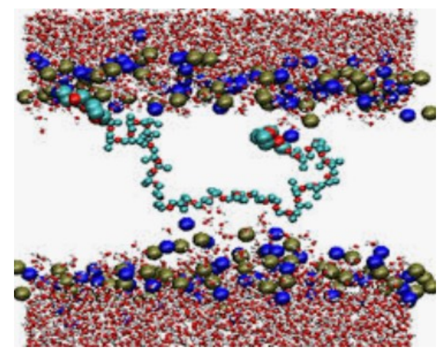
Agenda

- What is CUDA?
- Device Global Memory and Data Transfer
- Error Checking
- A Vector Addition Kernel
- Kernel Functions and Threading
- Kernel Launch

A Vector Addition Kernel

Review: Data Parallelism (aka Data-level Parallelism)

- Parallel execution of the same instruction stream on multiple data
  - **Image processing** : deal with individual pixels in a image
  - **Molecular dynamics** : simulate interactions between thousands to millions of atoms
  - **Airline scheduling** : handles thousands of flights, crews, and airport gates.
  - **Starcraft** : control hundreds to thousands of tanks, marines, zergling, etc.



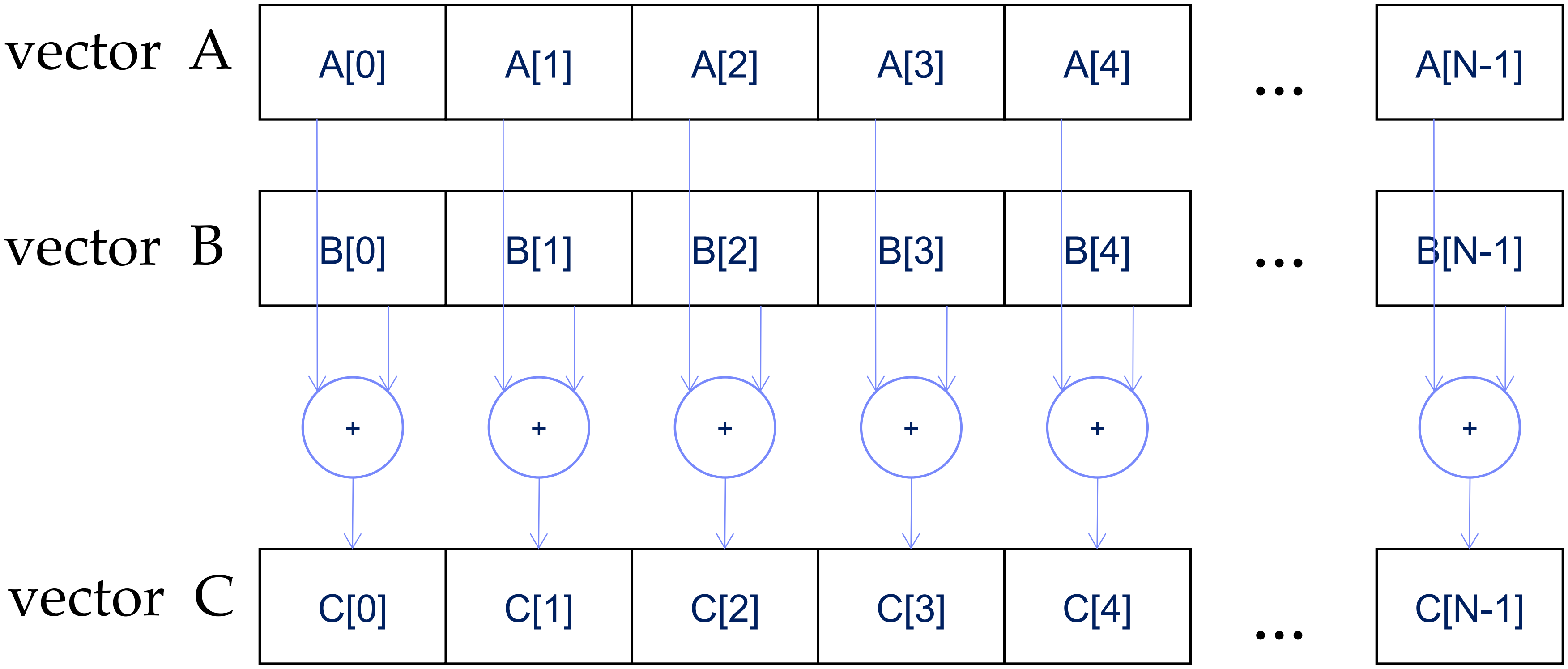
Vector Addition

- **Scalar vs Vector**
  - **Scalar** : a single number ex) 1
  - **Vector** : an array of numbers ex) [1,2,3]

- **Vector** : represented as 1D array
  - const int a[SIZE];
  - const int b[SIZE];
  - int c[SIZE];

- **Vector addition** : c[…] = a[…] + b[…]

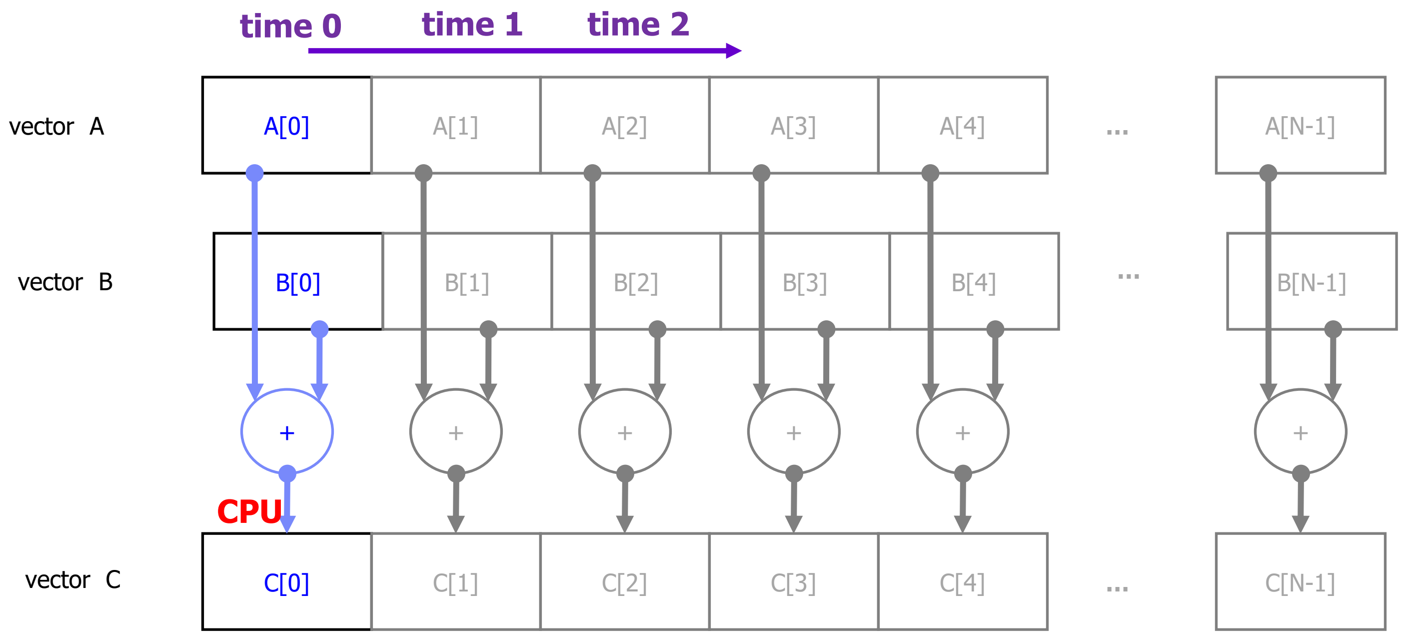
Vector Addition – Conceptual View



Slide Credit: Slides are modified from Prof Baek's slides

CPU-based Vector Addition

- a single CPU does an addition. then, the next addition
  - ➔ Serial execution



Slide Credit: Slides are modified from Prof Baek's slides

CPU-based Vector Addition (cont'd)

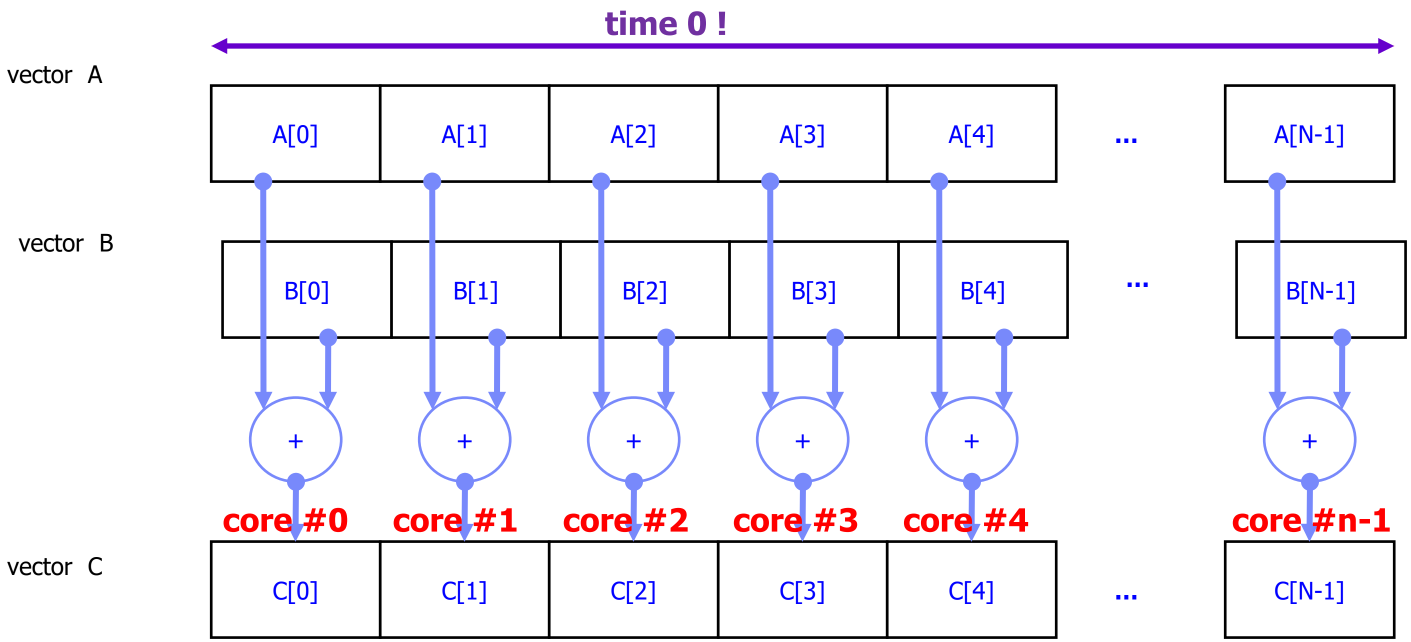
- **Traditional C Code**

```
// 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);
}
```

CUDA-based Vector Addition

- many GPU cores do the addition at the same time !

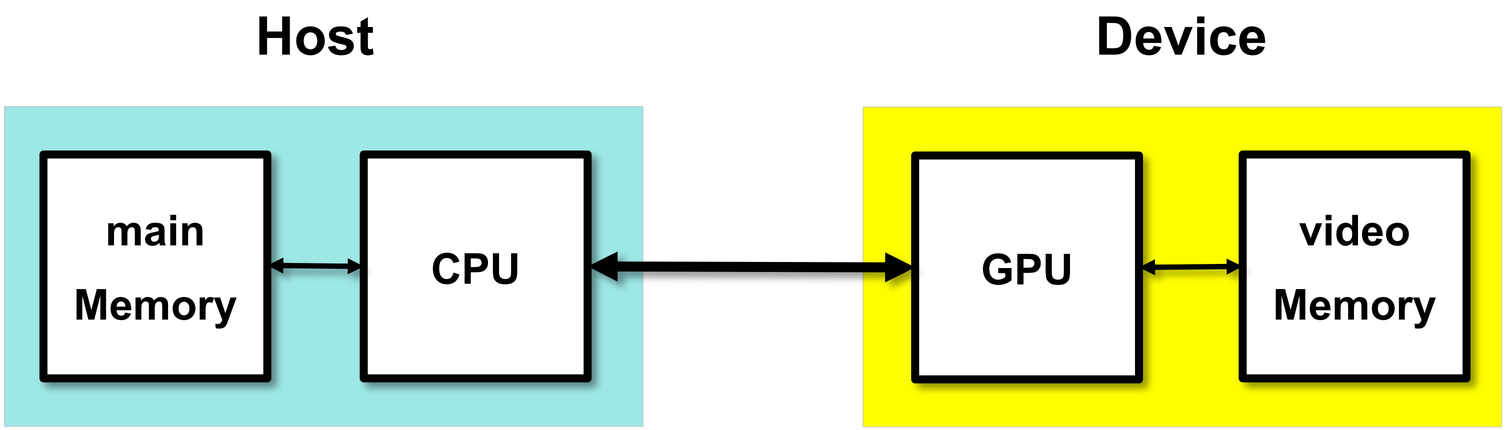
  ➔ Parallel execution



Slide Credit: Slides are modified from Prof Baek's slides

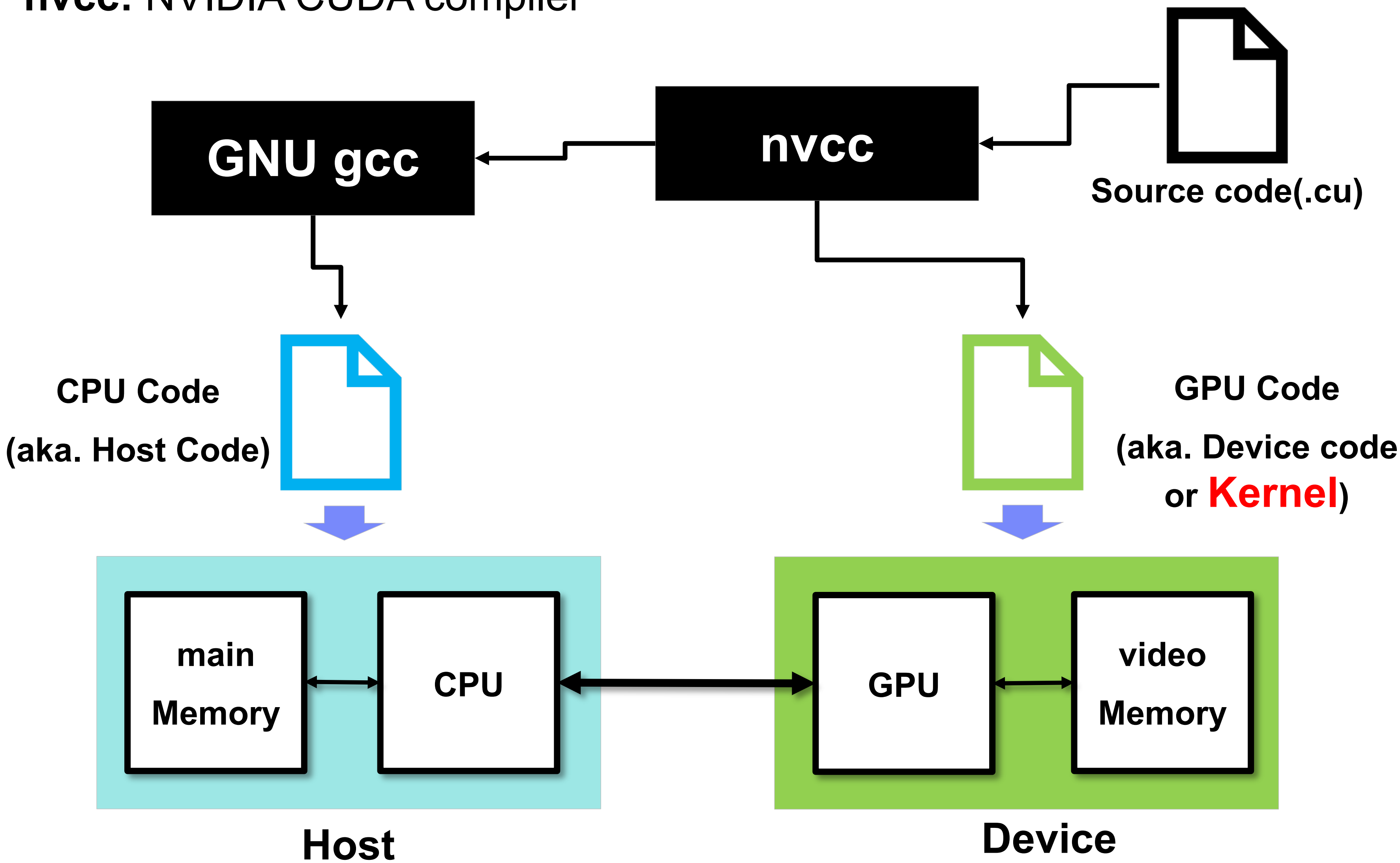
Review: CUDA Programming Model

- **Host** : CPU + main memory (host memory)
- **Device** : GPU + video memory (device memory)



Review: CUDA Programming Model (Cont'd)

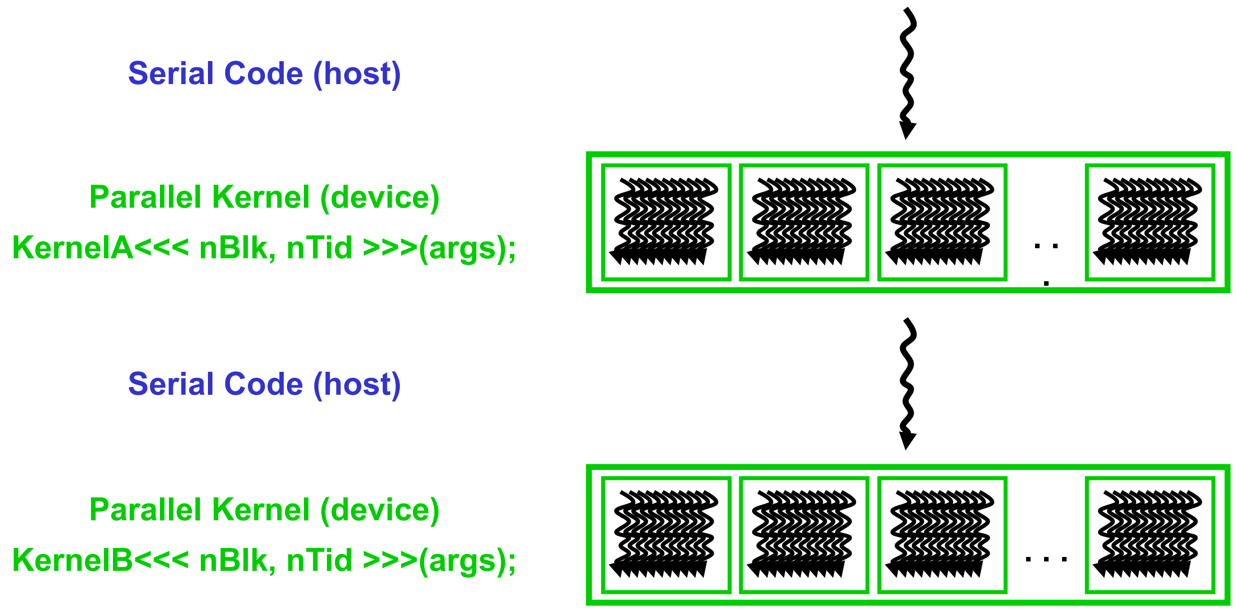
- **GNU gcc** : linux c compiler
- **nvcc:** NVIDIA CUDA compiler



Review: Execution of CUDA Program

- CUDA Program: Integrated host+device app C program
  - Serial or modestly parallel parts in **host C code**
  - Highly parallel parts in **device C code (kernel)**

- A **kernel** is executed (launched) by **a large number of threads**
  - **grid :** A group of all threads that are generated by a kernel launch



CUDA-based Vector Addition (cont'd)

- **vecAdd (Host code)**

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

CUDA-based Vector Addition (cont'd)

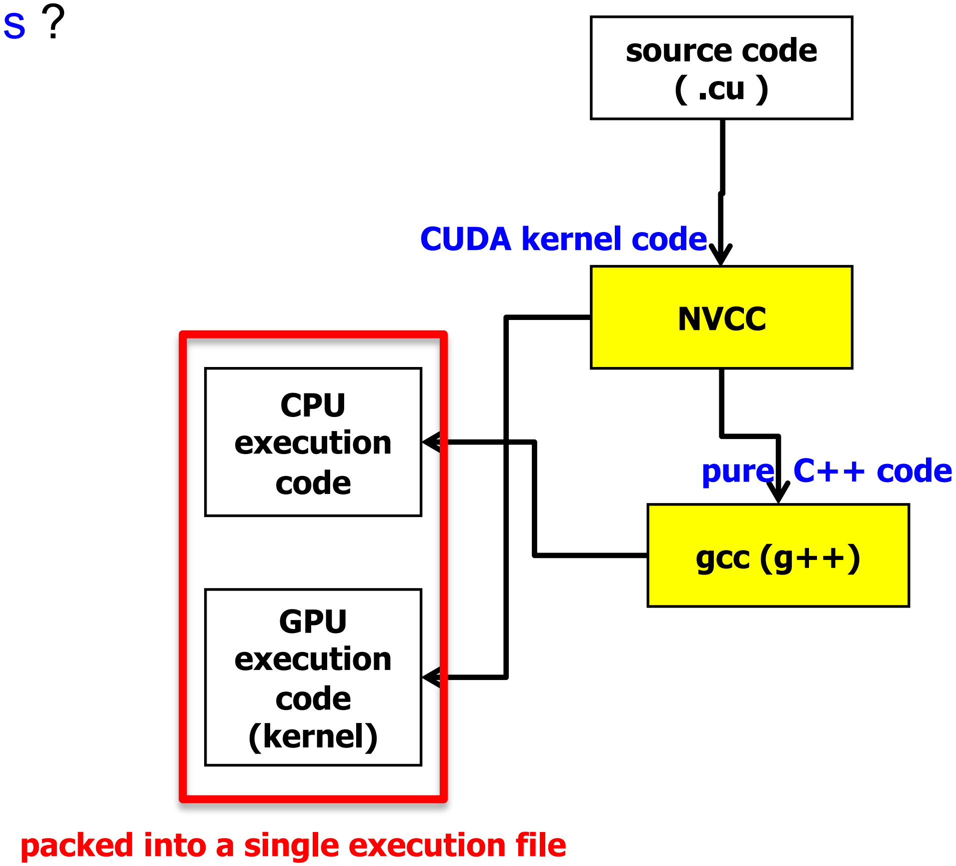
- **vecAdd (Host code)**

```
#include <cuda.h>
void vecAdd(int* A, int* B, int* C, int n)
{
    int size = n * sizeof(int);
    int* A_d=0;
    int* B_d=0;
    int* C_d=0;
    // Allocate device memory
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);
   // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
   // Kernel invocation code – to be shown later
     …
   // 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);
}
```

Kernel Functions and Threading

Review: CUDA programming model

- At the source code level,
  - How can we distinguish those source codes?
  - And, what is the unit of compilations ?



Slide Credit: Slides are modified from Prof Baek's slides

CUDA Function Declarations

- **Compilation unit ?: Functions**
  - Each function will be assigned to CPU and/or GPU
- **How to distinguish them?**
  - **use PREFIX for each function**
  - **__host__** : can be called by CPU (default, can be omitted)
    - Each "__" consists of two underscore characters
  - **__device__** : called from other GPU functions,
    cannot be called by the CPU
  - **__global__** : launched by CPU,
    cannot be called from GPU, must return void
  - **__device__** and **__host__** can be used together
    - Compiler generates two versions (host and device).
      - One is executed on the host and can be called from a host function.
      - The other is executed on the device and can be called from a device function

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

Slide Credit: Slides are modified from Prof Baek's slides

Example of a Kernel: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void addKernel(int* A_d, int* B_d, int* C_d)
{
     // each thread knows its own index
     int i = threadIdx.x;
    C_d[i] = A_d[i] + B_d[i];
}
```

- **threadIdx:** Built-in variable that gives each thread a unique coordinate within a block

CUDA Kernel Function and Threading

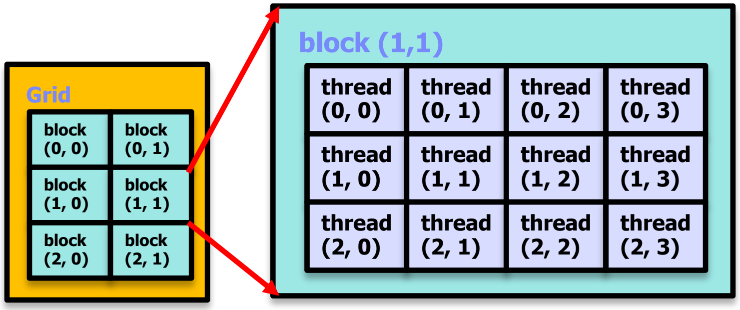
- All threads will execute the same kernel function on different cores in parallel

  ➔SPMD: Single Program Multiple Data

- All threads within a warp will execute the same instructions on different cores of a SM in parallel

  ➔ SIMT: Single Instruction Multiple Thread

- When a host code launches a kernel, **a grid of threads is generated**
  - Each grid is organized as an array of thread blocks (aka block)
  - All blocks of a grid are of the same size (up to 1024 threads)
  - The total number of threads in each thread block is specified in the host code where launching the kernel
    - The same kernel can be launched with different numbers of threads



Kernel Launch

CUDA Kernel Launch

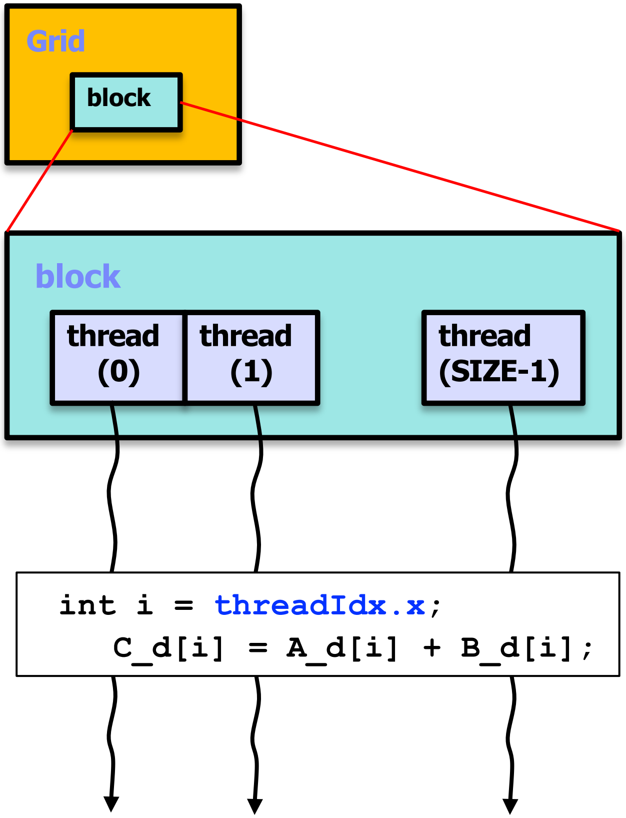
- **Kernel launch syntax**

  addKernel<<<1, SIZE>>>(A_d, B_d, C_d);

  Kernel name

  The number of blocks

  Threads per block



- CUDA view
  - a thread executes addKernel( ) **with threadIdx.x = 0**
  - a thread executes addKernel( ) **with threadIdx.x = 1**
  - a thread executes addKernel( ) **with threadIdx.x = 2**
  - ...
  - a thread executes addKernel( ) **with threadIdx.x = SIZE-1**

A Complete Version : vectAdd.cu

Device Code (Kernel)

```
#include <cuda.h>

#include <iostream>


// Compute vector sum C = A+B

// Each thread performs one pair-
   wise addition

__global__

void addKernel(int* A_d, int* B_d, int*
   C_d)

{

     // each thread knows its own index

     int i = threadIdx.x;

    C_d[i] = A_d[i] + B_d[i];

}
```

Host Code

```
void vecAdd(int* A, int* B, int* C, int n)

{

    int size = n * sizeof(int);

    int* A_d=0;

    int* B_d=0;

    int* C_d=0;

    // Allocate device memory

    cudaMalloc((void **) &A_d, size);

    cudaMalloc((void **) &B_d, size);

    cudaMalloc((void **) &C_d, size);


   // Transfer A and B to device memory

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);

    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);


    addKernel<<<1, n>>>(A_d, B_d, C_d);


   // 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);

}
```

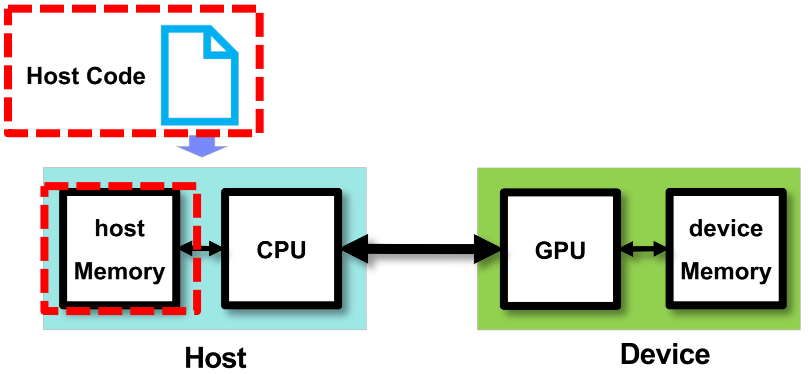
A Complete Version : vectAdd.cu (Cont’d)

Host Code

```
int main(void) {
    // host-side data
   const int SIZE = 5;
   int a[SIZE] = { 1, 2, 3, 4, 5 };
   int b[SIZE] = { 10, 20, 30, 40, 50 };
   int c[SIZE] = { 0 };

   vecAdd(a,b,c, SIZE);

    // print the result
   printf("{%d,%d,%d,%d,%d} + {%d,%d,%d,%d,%d} "
              "= {%d,%d,%d,%d,%d}\n",
                   a[0], a[1], a[2], a[3], a[4],
                   b[0], b[1], b[2], b[3], b[4],
                   c[0], c[1], c[2], c[3], c[4]);
    // done
   return 0;
}
```

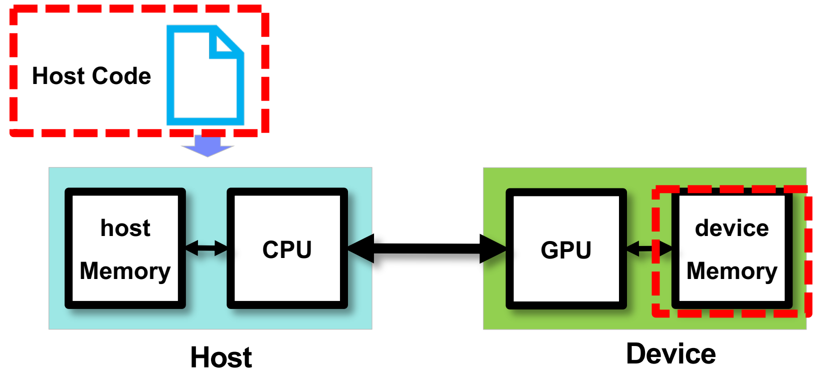
Execution of vectAdd.cu

```
int main(void) {
    // host-side data
   const int SIZE = 5;
   int a[SIZE] = { 1, 2, 3, 4, 5 };
   int b[SIZE] = { 10, 20, 30, 40, 50 };
   int c[SIZE] = { 0 };

   vecAdd(a,b,c,SIZE);

    // print the result
   printf("{%d,%d,%d,%d,%d} + {%d,%d,%d,%d,%d} "
            "= {%d,%d,%d,%d,%d}\n",
                  a[0], a[1], a[2], a[3], a[4],
                  b[0], b[1], b[2], b[3], b[4],
                  c[0], c[1], c[2], c[3], c[4]);
    // done
   return 0;
}
```



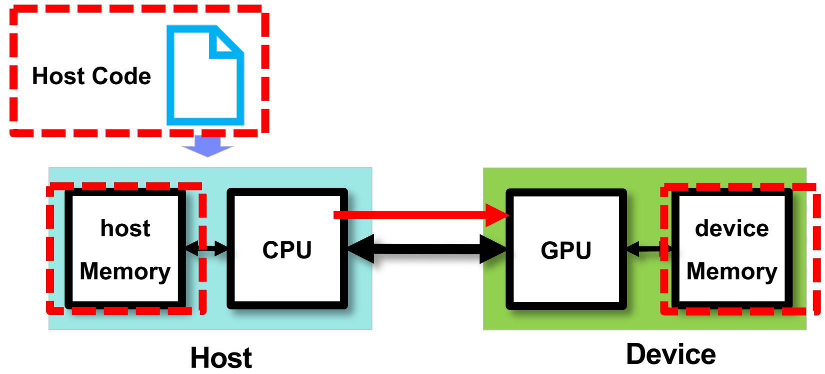
Execution of vectAdd.cu (cont'd)

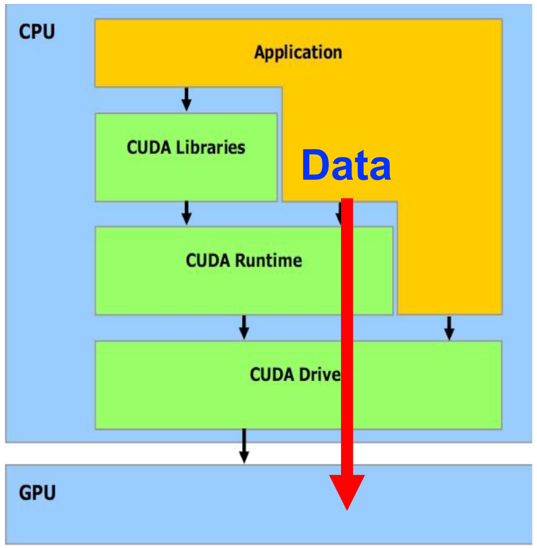
```
void vecAdd(int* A, int* B, int* C, int n)
{
    int size = n * sizeof(int);
    int* A_d=0;
    int* B_d=0;
    int* C_d=0;
    // Allocate device memory
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    addKernel<<<1, n>>>(A_d, B_d, C_d);

    // 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);
}
```



Execution of vectAdd.cu (cont'd)

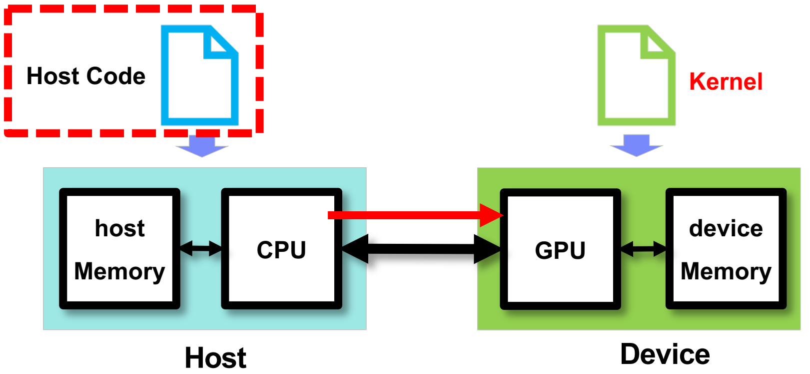
```
void vecAdd(int* A, int* B, int* C, int n)
{
    int size = n * sizeof(int);
    int* A_d=0;
    int* B_d=0;
    int* C_d=0;
    // Allocate device memory
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

   // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    addKernel<<<1, n>>>(A_d, B_d, C_d);

   // 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);
}
```

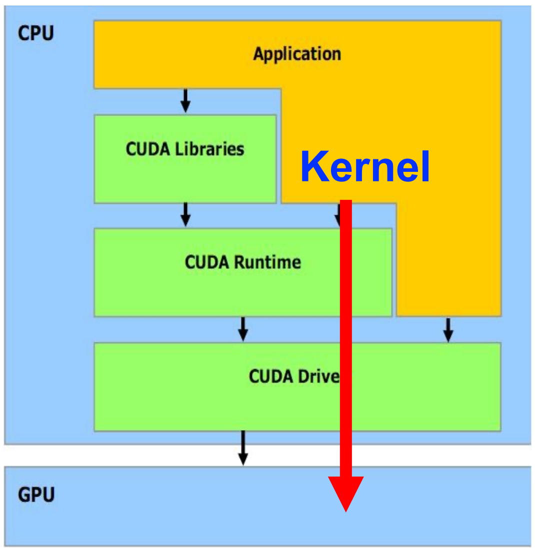




Execution of vectAdd.cu (cont'd)

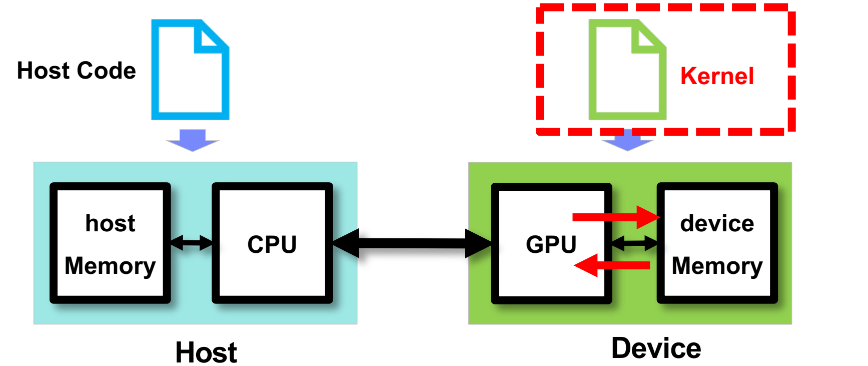
```
void vecAdd(int* A, int* B, int* C, int n)
{
    int size = n * sizeof(int);
    int* A_d=0;
    int* B_d=0;
    int* C_d=0;
    // Allocate device memory
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

   // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    addKernel<<<1, n>>>(A_d, B_d, C_d);

   // 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);
}
```

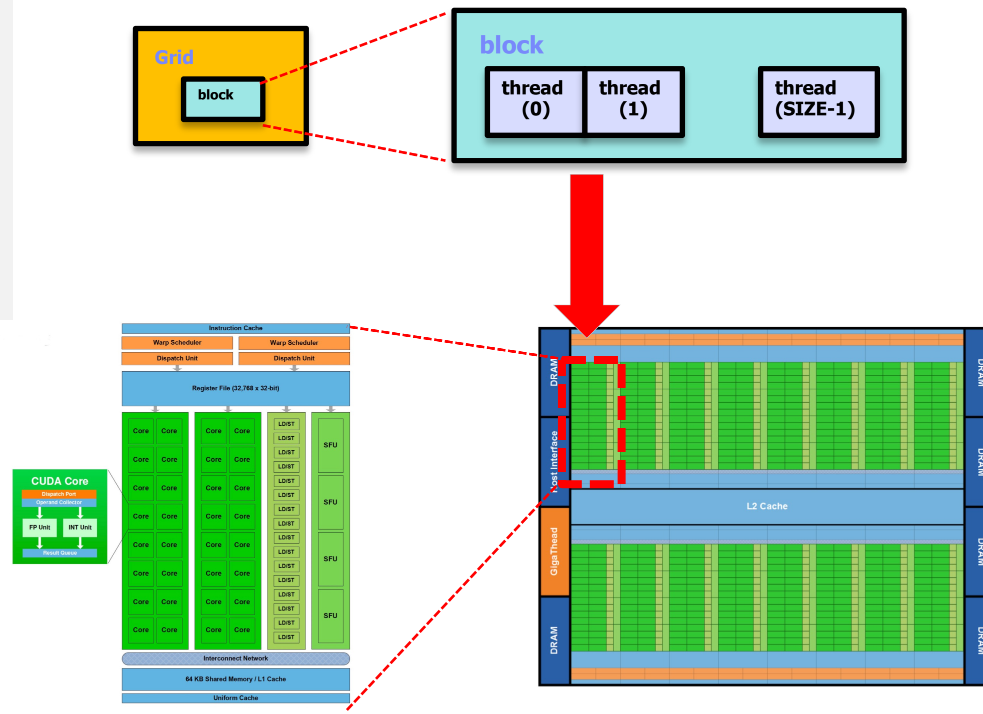




Execution of vectAdd.cu (cont'd)

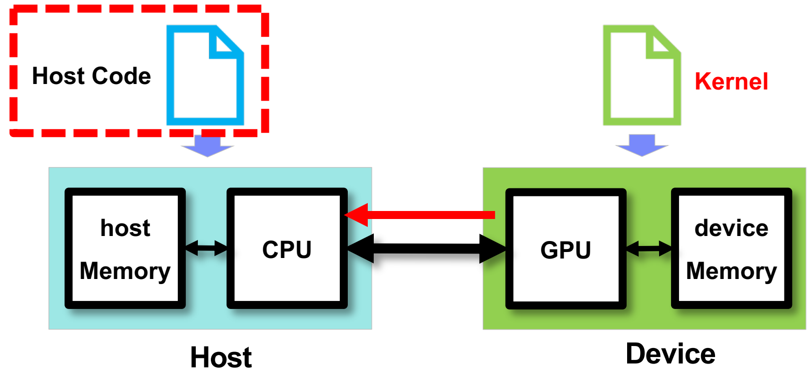
```
#include <cuda.h>

#include <iostream>


// Compute vector sum C = A+B

// Each thread performs one pair-
   wise addition

__global__

void addKernel(int* A_d, int* B_d, int*
   C_d)

{

      // each thread knows its own index

     int i = threadIdx.x;

    C_d[i] = A_d[i] + B_d[i];

}
```

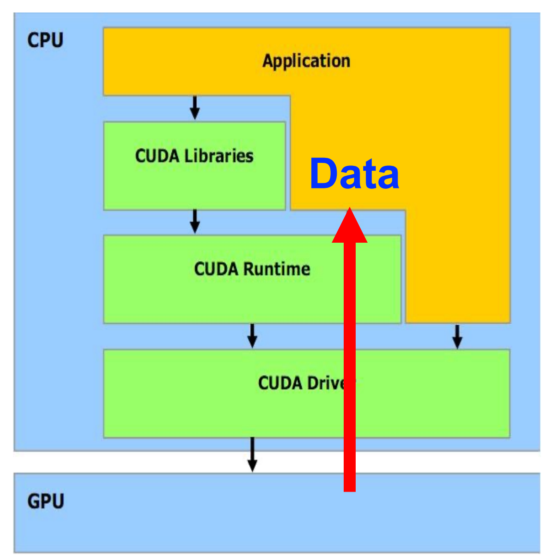




Execution of vectAdd.cu (cont'd)

```
void vecAdd(int* A, int* B, int* C, int n)
{
    int size = n * sizeof(int);
    int* A_d=0;
    int* B_d=0;
    int* C_d=0;
    // Allocate device memory
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

   // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    addKernel<<<1, n>>>(A_d, B_d, C_d);

   // 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);
}
```





Next

- CUDA threads for scalable parallel execution