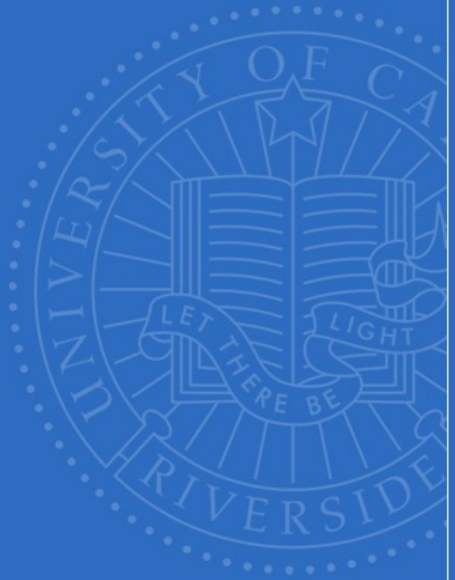
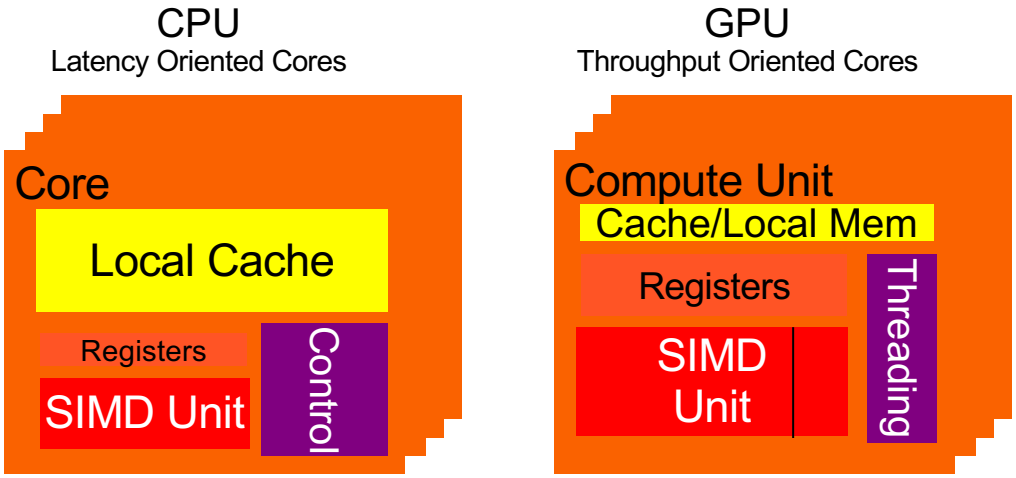


Introduction to CUDA C

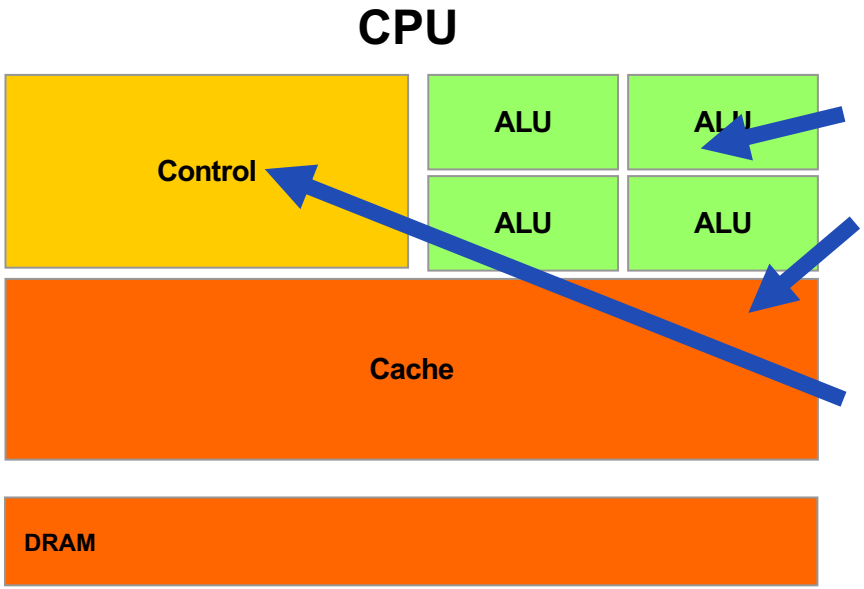
UNIVERSITY OF CALIFORNIA, RIVERSIDE

Slide credit: Slides adapted from

CPU and GPU are designed very differently

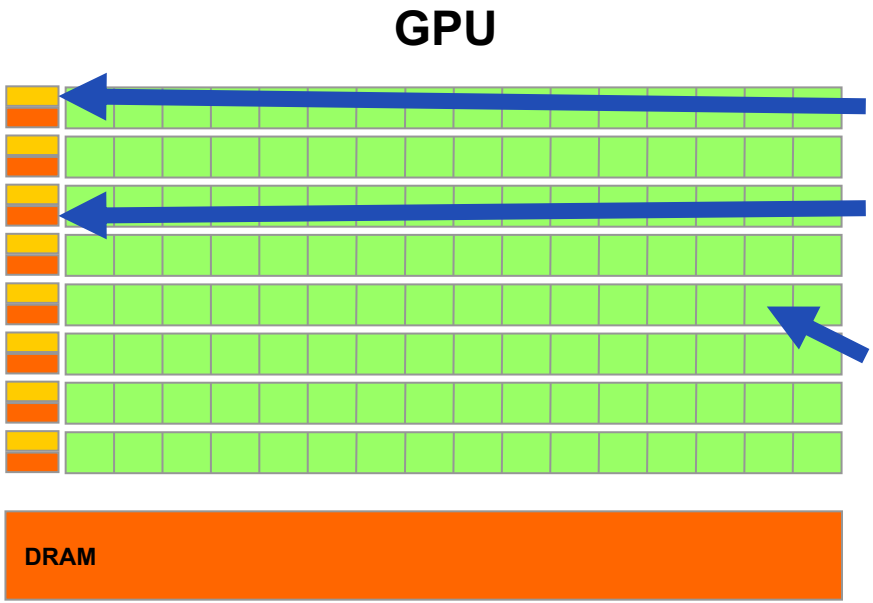


CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Applications Use Both CPU and GPU

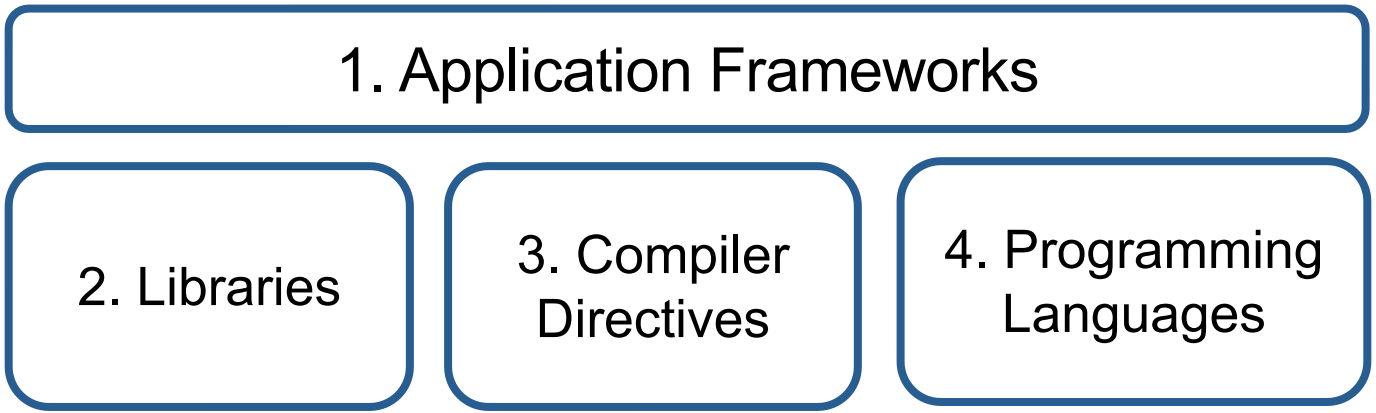
- CPUs best used for *sequential parts* where *latency* matters
- CPUs can be >10X faster than GPUs for sequential code
- GPUs best used for *parallel parts* where *throughput* wins
- GPUs can be >10X faster than CPUs for parallel code

Parallelizing code is easy. Making parallel code fast is hard.

Objective

- To learn the main venues and developer resources for GPU computing
  - Where CUDA C fits in the big picture

4 Ways to Accelerate Applications



1. Application Frameworks: Use GPUs without knowing it!

- **Ease of use:** Using application frameworks enable GPU acceleration *without in-depth knowledge of GPU programming*.
- **Domain Specific:** Many application frameworks are designed for specific domains. For example, *Keras/TensorFlow/PyTorch* for machine learning,
- Application frameworks typically use high-quality implementations of functions (libraries) in the backend.

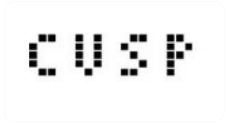
2. Libraries: Easy, High-Quality Acceleration

UCR

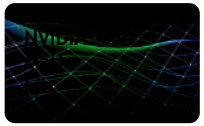
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming.
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

GPU Accelerated Libraries

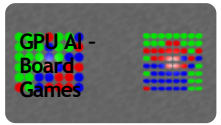
Linear Algebra
FFT, BLAS,
SPARSE, Matrix



Numerical & Math
RAND, Statistics



Data Struct. & AI
Sort, Scan, Zero Sum



Visual Processing
Image & Video

NVIDIA
Video
Encode

3. Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement.
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

Example: OpenACC

Compiler directives for C, C++, and FORTRAN:

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
  copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[i];
    }
```

4. Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** *Programmer* has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types.
- **Verbose:** The *programmer* often needs to express more details

Most of class focuses on this!

GPU Programming Languages

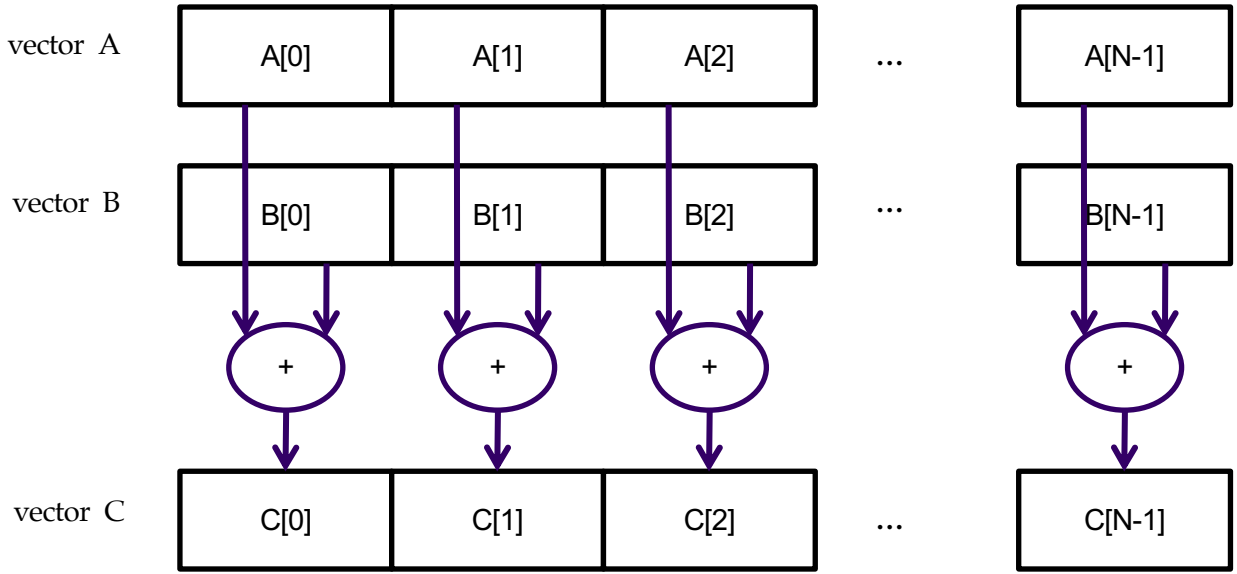
Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran
C	**CUDA C**
C++	**CUDA C++, Modern C++ (C++17 onwards)**
Python	Numba, PyCUDA
F#	Alea.cuBase

MEMORY ALLOCATION AND DATA MOVEMENT API FUNCTIONS

Objective

- To learn the basic API functions in CUDA host code
  - Device Memory Allocation
  - Host-Device Data Transfer

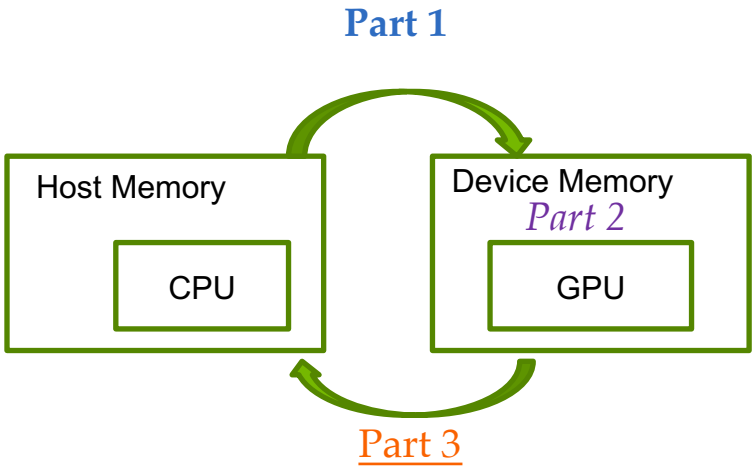
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

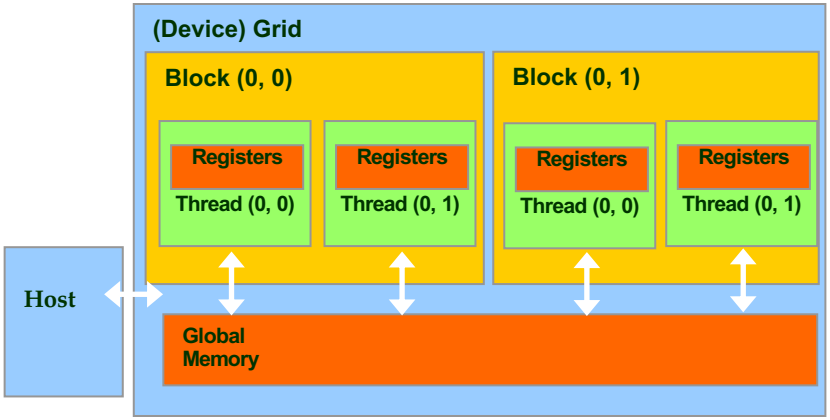
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++)
     h_C[i] = h_A[i] + h_B[i];
}

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);
}
```

Heterogeneous Computing vecAdd CUDA Host Code




```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n* sizeof(float);
   float *d_A, *d_B, *d_C;
   // Part 1
   // Allocate device memory for A, B, and C
   // copy A and B to device memory

   // Part 2
   // Kernel launch code – the device performs the actual vector
addition

   // Part 3
   // copy C from the device memory
}
```

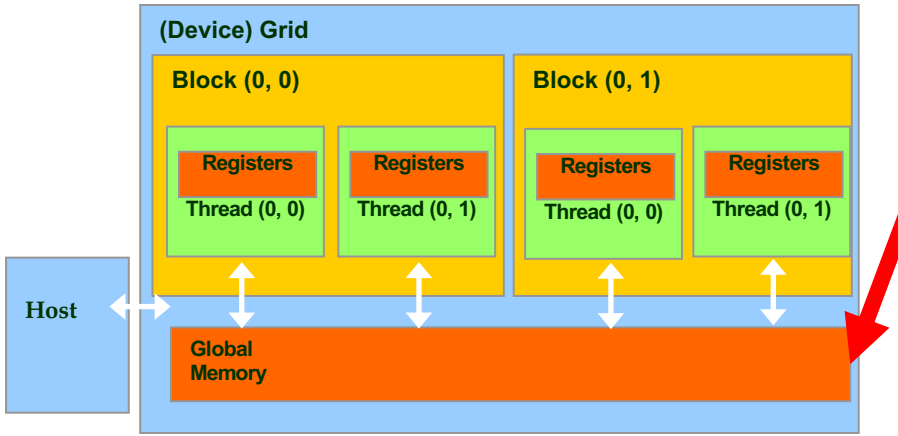
Partial Overview of CUDA Memories



- 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

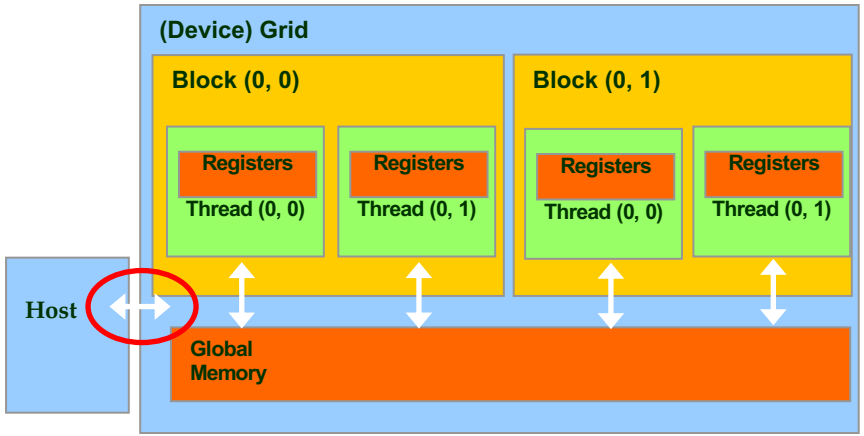
We will cover more memory types and more sophisticated memory models later.

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 Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_B, size);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_C, size);

     // Kernel invocation code – to be shown later

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

In Practice, Check for API Errors in Host Code

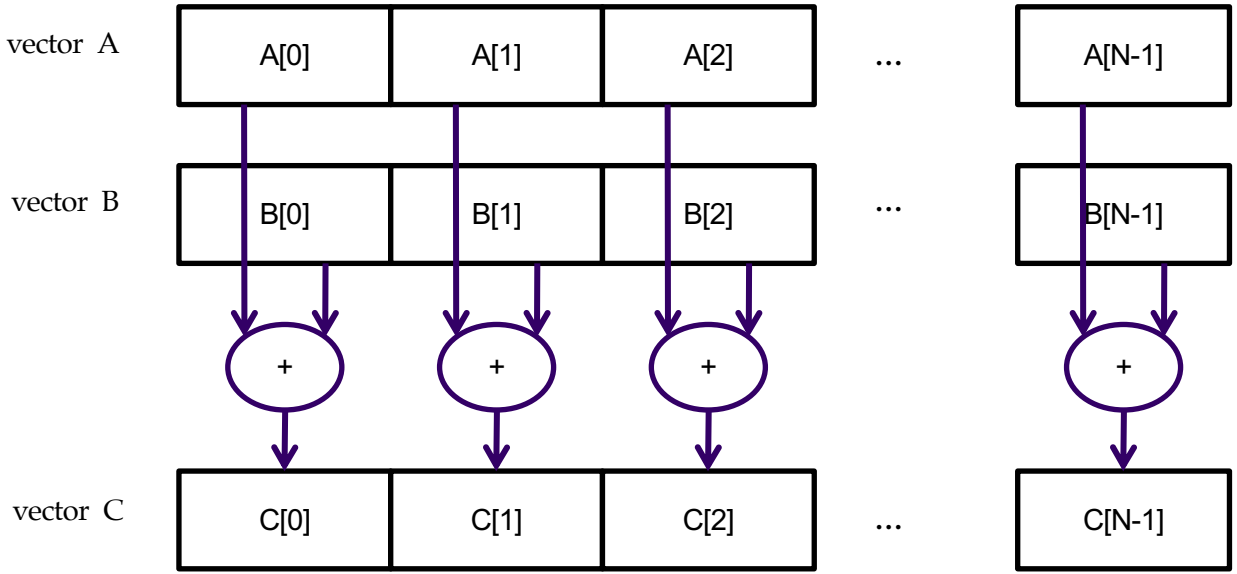
```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

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

THREADS AND KERNEL FUNCTIONS

Objective

- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping

Data Parallelism - Vector Addition Example

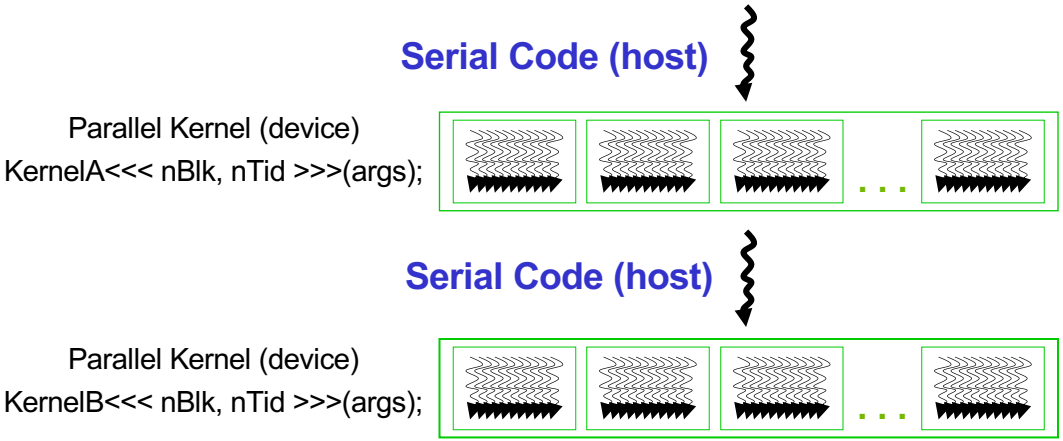


C[i] = A[i] + B[i];

How do we get i without for loops?

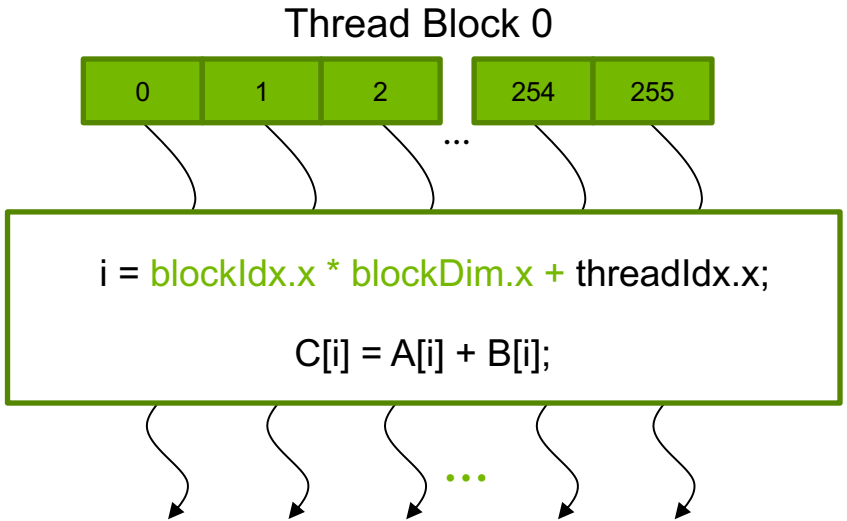
CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** kernel code

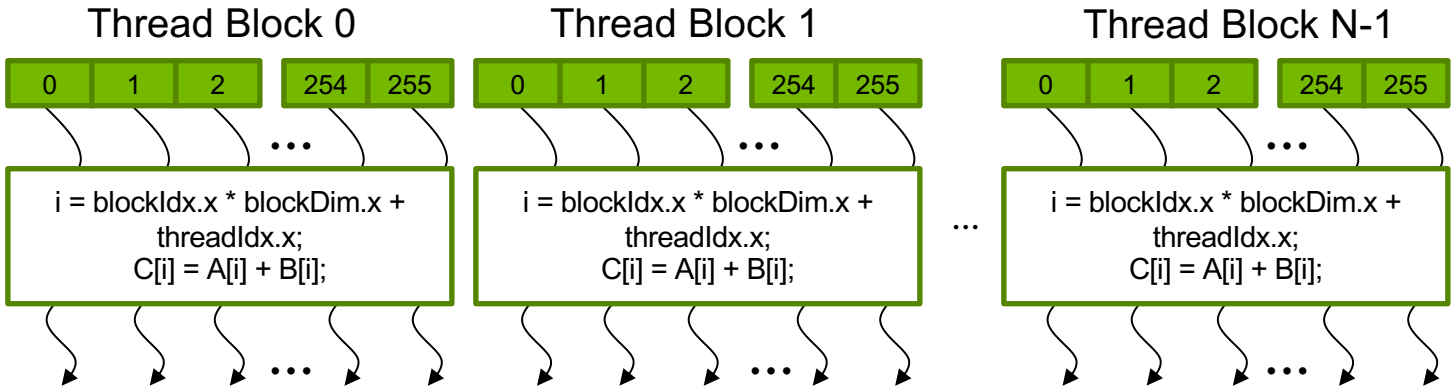


Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has *indexes* 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**
    - (We will learn more of this throughout class)
  - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …