CUDA Programming

Lecture 1

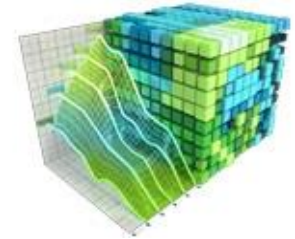
S.-Kazem Shekofteh

kazem.shekofteh@ziti.uni-heidelberg.de

Post-Doc Research Fellow, ZITI, Institute of computer Engineering, ***Heidelberg University, Germany***

Faculty Member, Department of Computer Engineering, ***Shandiz Institute of Higher Education, Iran***

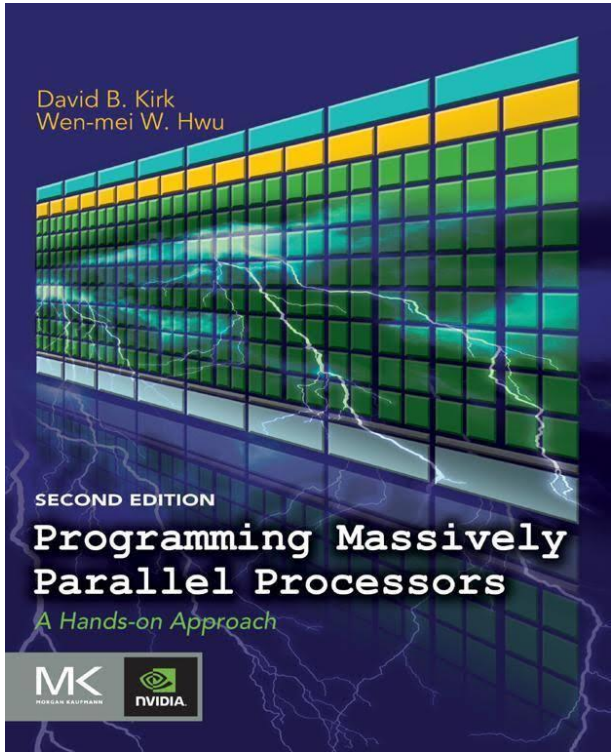
PhD in Computer Engineering, ***Ferdowsi University of Mashhad, Iran***

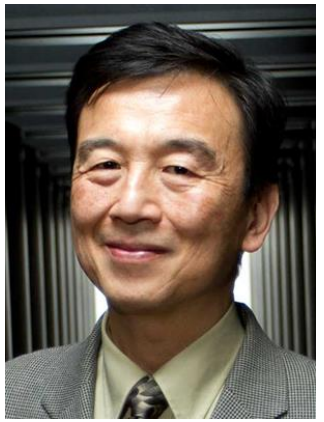


NVIDIA Visual Profiler

Reference

- David B. Kirk and Wen-mei W. Hwu, ”Programming Massively Parallel Processors,” 2nd Ed. 2012





Outline

- Introduction to GPU and its applications
- Basics of CUDA programming
- Kernel definition
- Thread hierarchy

CPU or GPU? Which one is better?!

- A car or a bus? Which one is better?

CPU or GPU? Which one is better?!

- A car or a bus? Which one is better?

To do what? To go where? With how many people?

CPU or GPU? Which one is better?!

- A car or a bus? Which one is better?

CPU

GPU

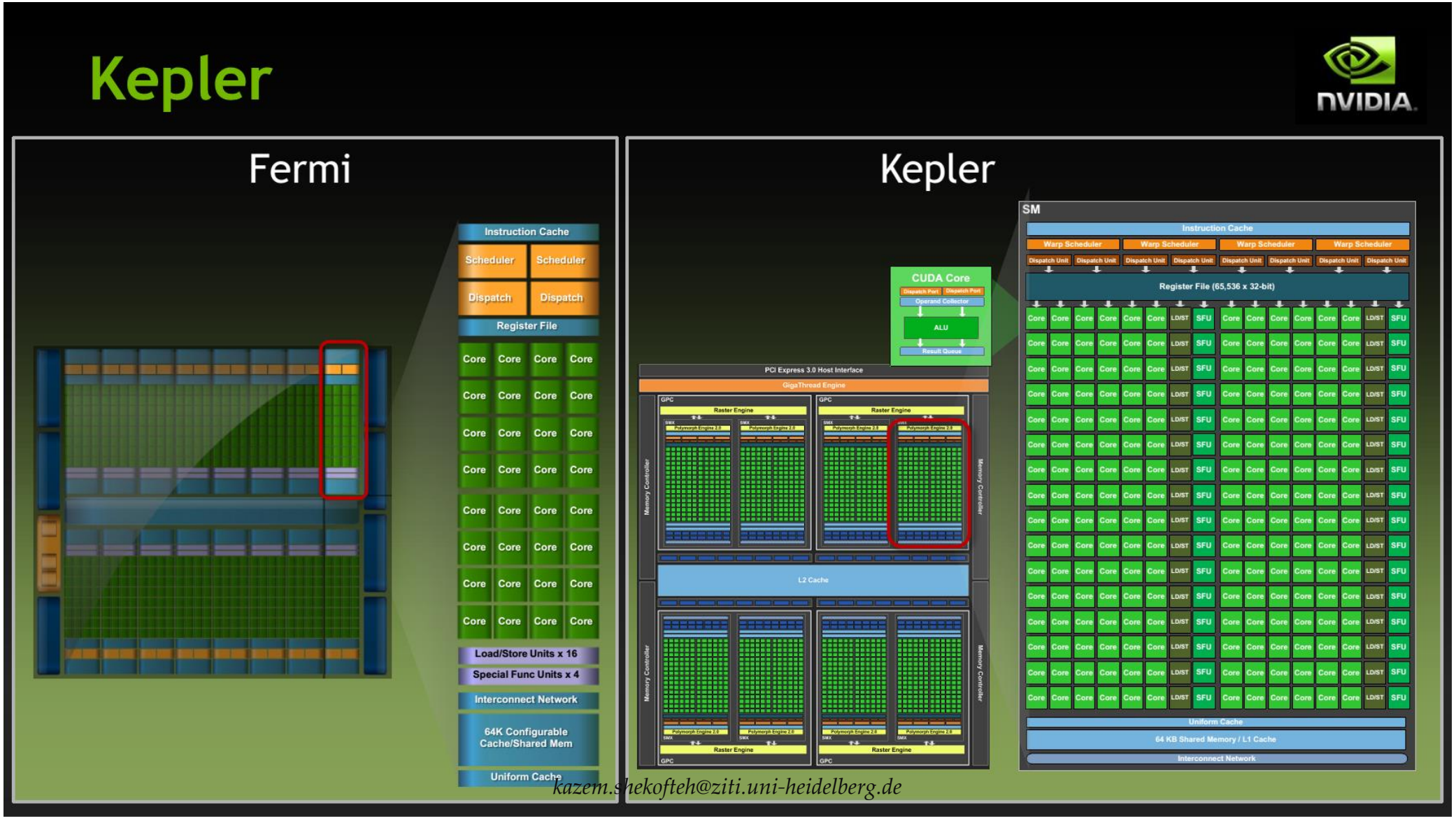
CPU or GPU? Which one is better?!

- CPU
  - Optimized for low-latency operations such as access to cached data sets
  - Control logic for out-of-order and speculative execution
- GPU
  - Optimized for data-parallel through computation
  - Architecture tolerant of memory latency

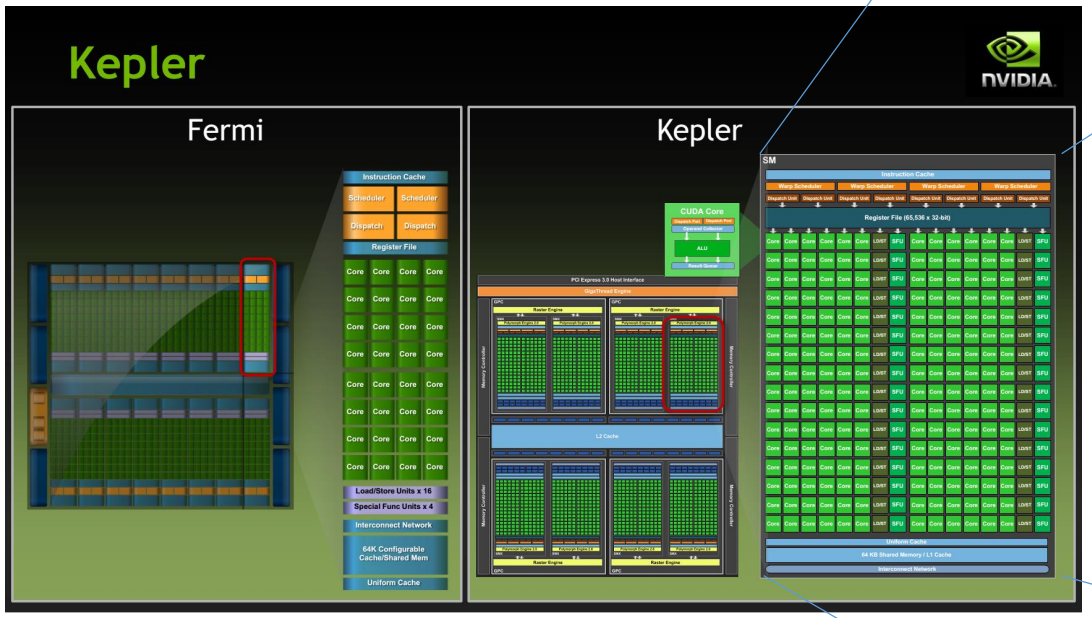
A brief look at the GPU architecture

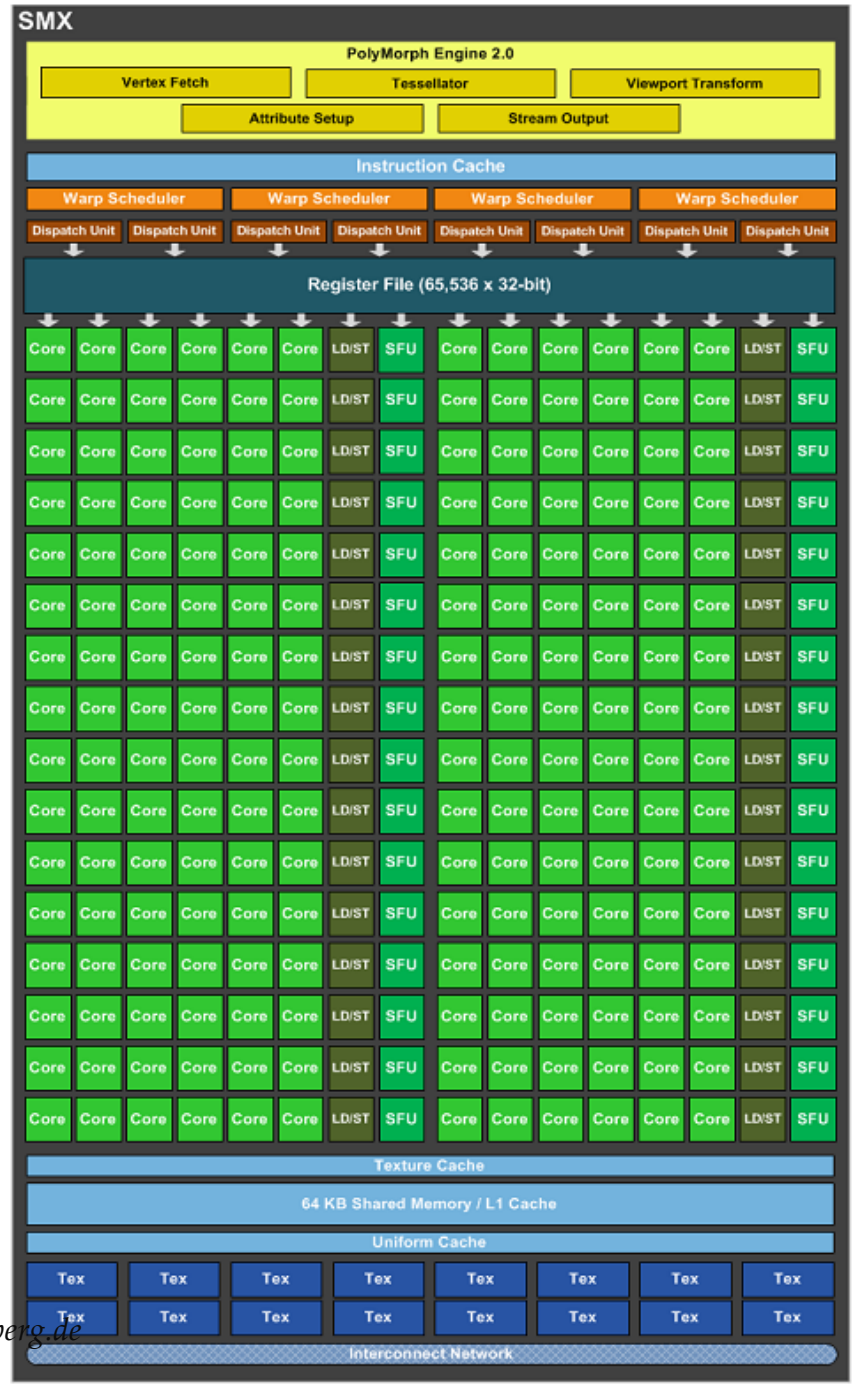
- Two main components
  - Global Memory
    - Analogous to RAM in a CPU server
    - Accessible by both CPU and GPU
  - Streaming Multiprocessor (SM, SMX, SMM, …)
    - Performs the actual computations
    - Has its own control units, registers, execution pipelines, cached, …

A brief look at the GPU architecture



A brief look at the GPU architecture





Architecture History

Microarchitecture Name	Production Date
Tesla	2007
Fermi	2009
Kepler	2013
Maxwell	2015
Pascal	2016
Volta	2017
Turing	2018
Ampere	2020
Hopper	2022

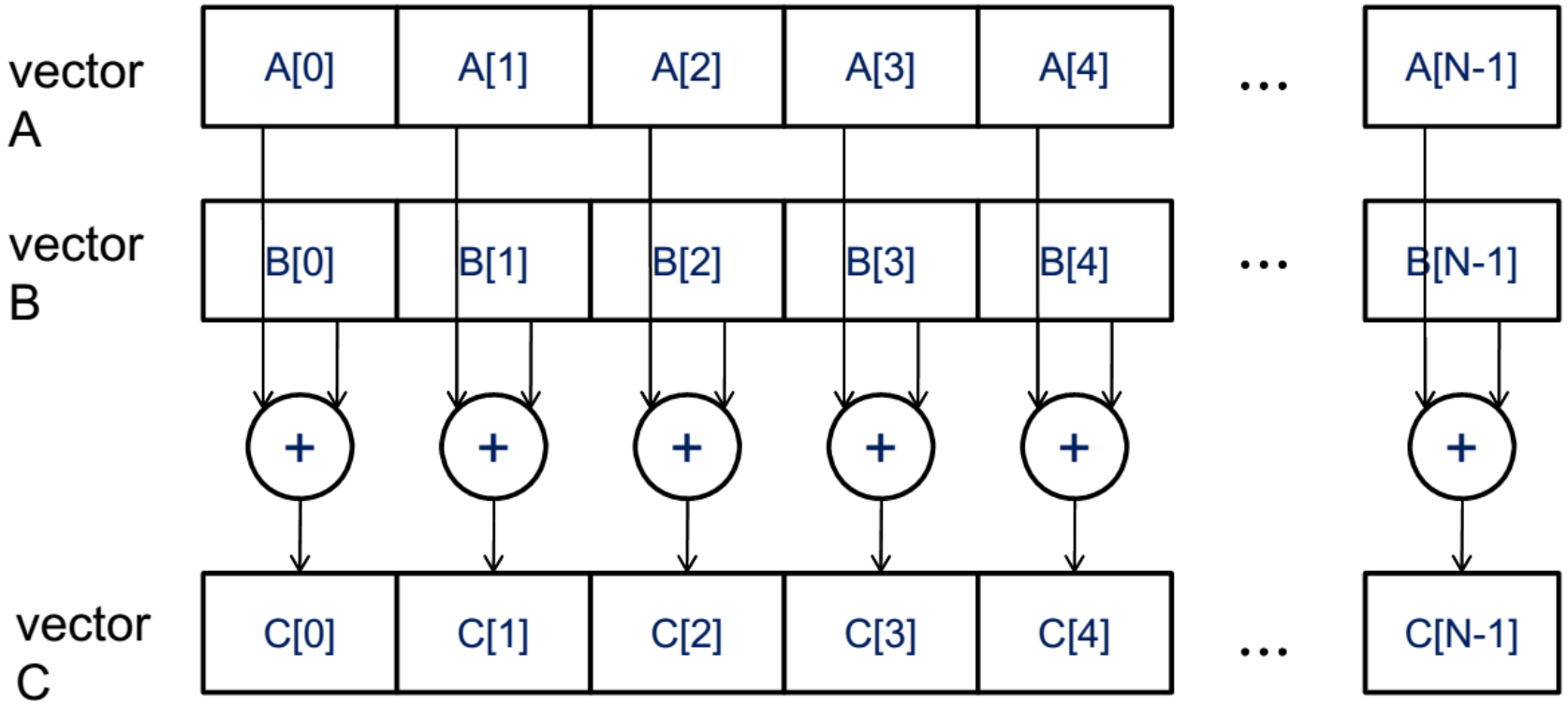
Task Parallelism vs. Data Parallelism

- Task parallelism is typically exposed through task decomposition of applications.
- For example, a simple application may need to do a vector addition and a matrix-vector multiplication.
- Task parallelism exists if the two tasks can be done independently.

Data Parallelism

- Example: vector addition
- each element of the sum vector C is generated by adding an element of input vector A to an element of input vector B.
  - C[0] is generated by adding A[0] to B[0]
- All additions can be performed in parallel.
- Data parallelism in real applications can be more complex!!

Data Parallelism



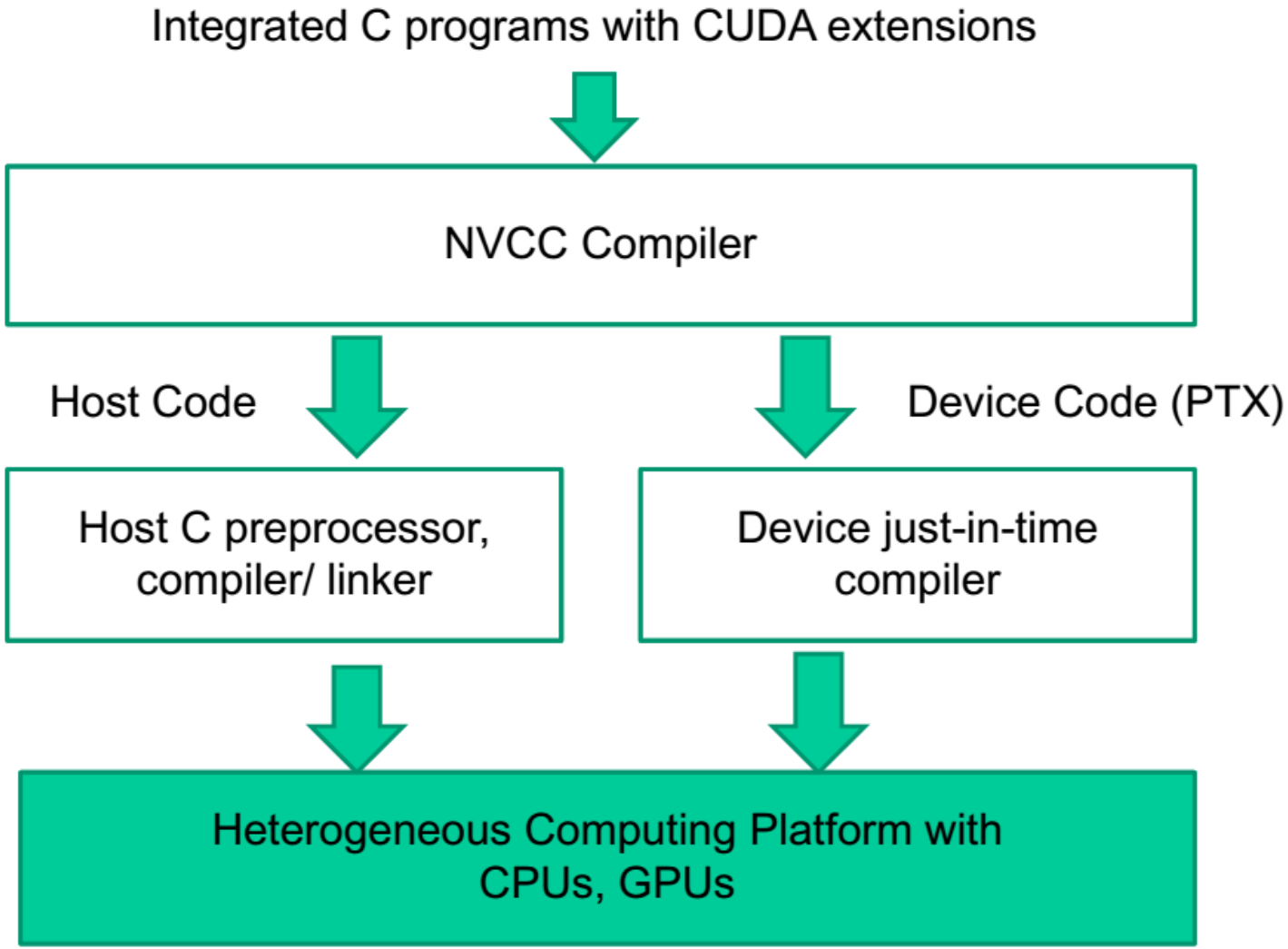
GPU Programming

- Totally, the programmer is capable of
  - Partitioning the problem into coarse sub-problems that can be solved independently in parallel by **blocks of threads**.
  - Partitioning each sub-problem into finer pieces that can be solved cooperatively in parallel by all **threads** within the block.
- Each block of threads can be scheduled on any of the available multiprocessors

CUDA Program Structure

- The structure of a CUDA program reflects the coexistence of a ***host*** (CPU) and
  one or more ***devices*** (GPUs) in the computer.
- Each CUDA source file (.cu) can have a mixture of both host and device code, function, variable, etc.
- The function or data declarations for the device are clearly marked with special CUDA keywords.
- The code needs to be compiled by a compiler that recognizes and understands these additional declarations.
- The device code is marked with CUDA keywords for labeling data-parallel functions, called ***kernels***, and their associated data structures.

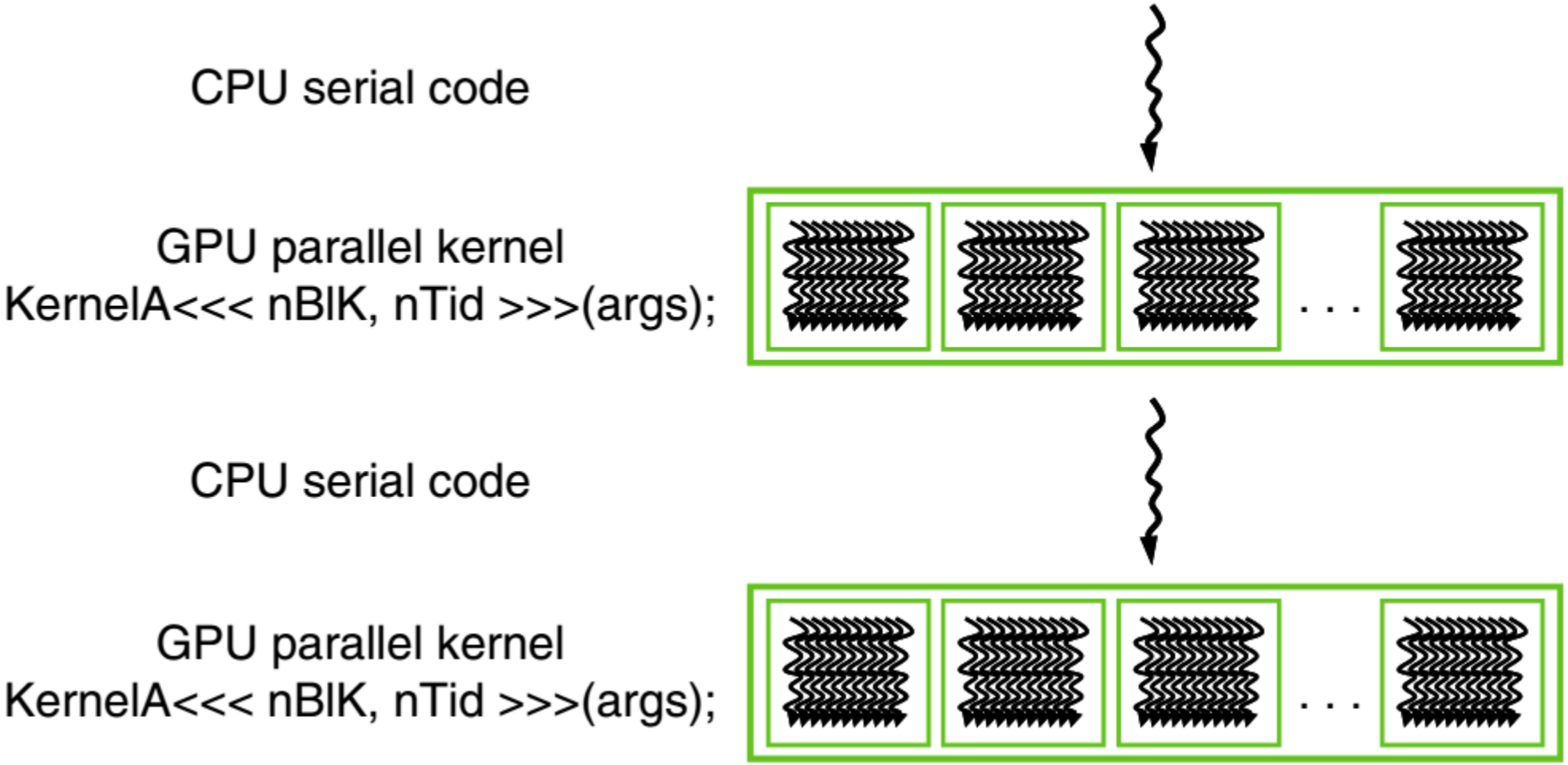
CUDA Program Structure



How a CUDA Program executes

- The execution starts with host (CPU) execution. When a kernel function is called, or ***launched***, it is executed by a large number of threads on a device.
- All the threads that are generated by a kernel launch are collectively called a ***grid***.
- When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is launched.

How a CUDA Program executes



How a CUDA Program executes

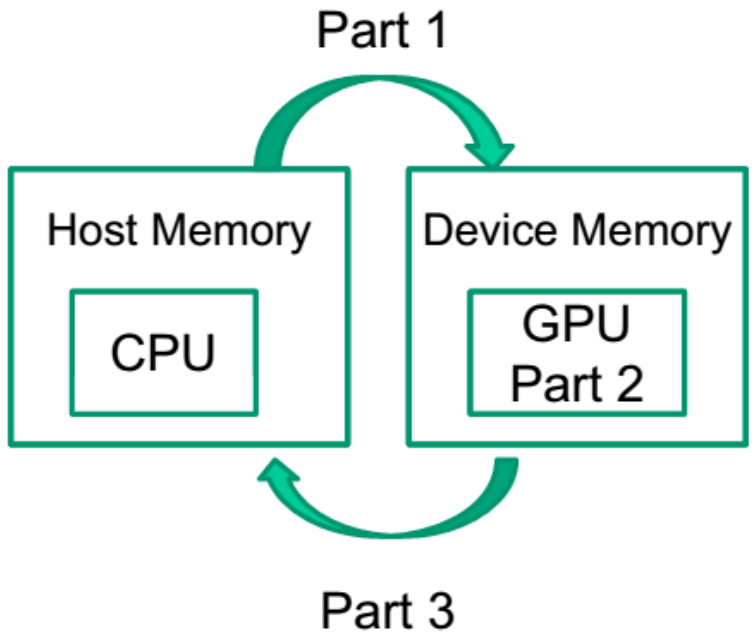
- Launching a kernel typically generates a large number of threads to exploit data parallelism.
- In the vector addition example, each thread can be used to … ?
  - How many threads?

A Vector Addition Kernel (C Program)

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  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 each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

A Vector Addition Kernel

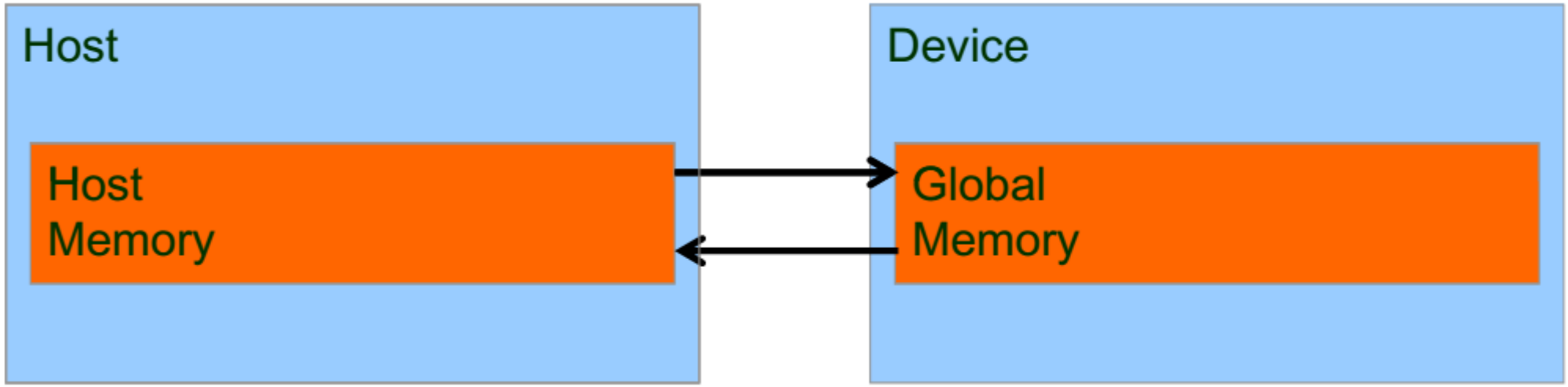
```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
   int size = n* sizeof(float);
    float *A_d, *B_d, *C_d;
   …
1. // Allocate device memory for A, B, and C
    // copy A and B to device memory

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

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



Device Global Memory

- In CUDA, host and devices have separate memory spaces.
- Devices are often hardware cards that come with their own DRAM.
- Global memory is also referred as device memory.
- To execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer pertinent data from the host memory to the allocated device memory (Part 1).
- After device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed (Part 3).

Device Global Memory



- There are more device memory types than shown.

Device Global Memory

- Function ***cudaMalloc*()** can be called from the host code to allocate a piece of device global memory for an object.

```
float *d_A
int size = n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

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

Device Global Memory

- Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device. This is accomplished by calling ***cudaMemCpy*** function.

cudaMemcpy()

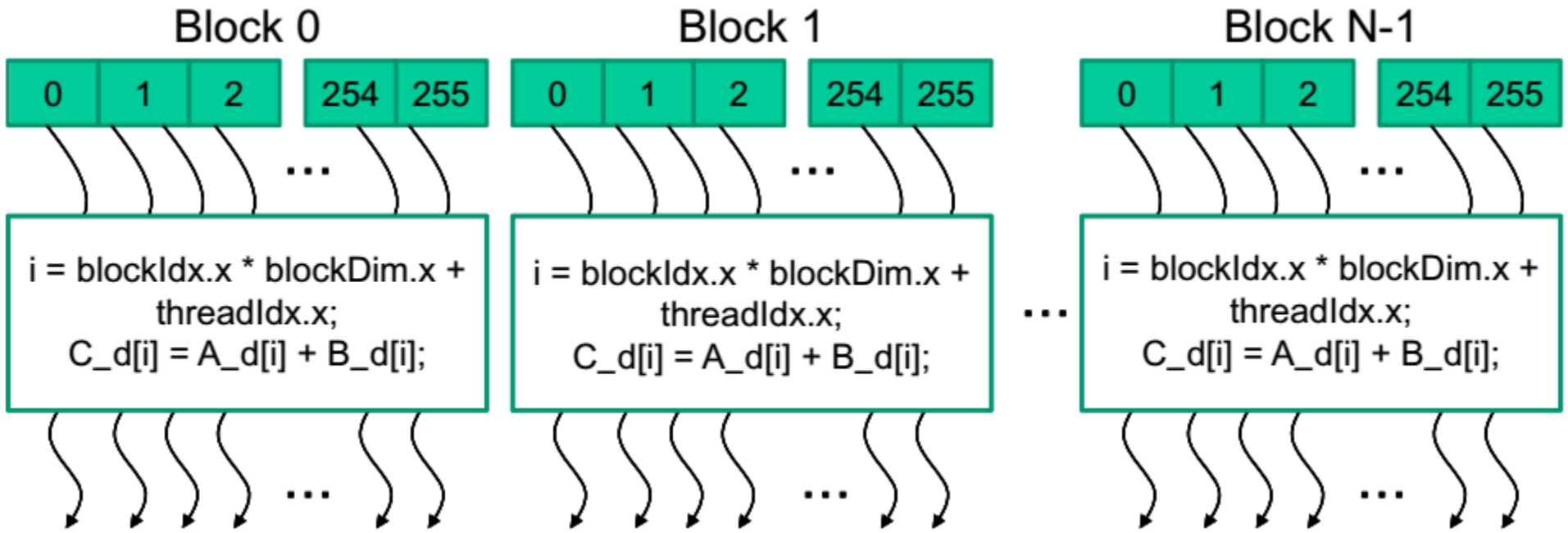
- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

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

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

Kernel Functions and Threading

- In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase.
- This is an instance of SPMD (Single Program Multiple Data) parallel programming style.
- When a host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized in a two-level hierarchy.
  - Each grid is organized into an array of thread blocks.
  - Each thread block consists of several threads.

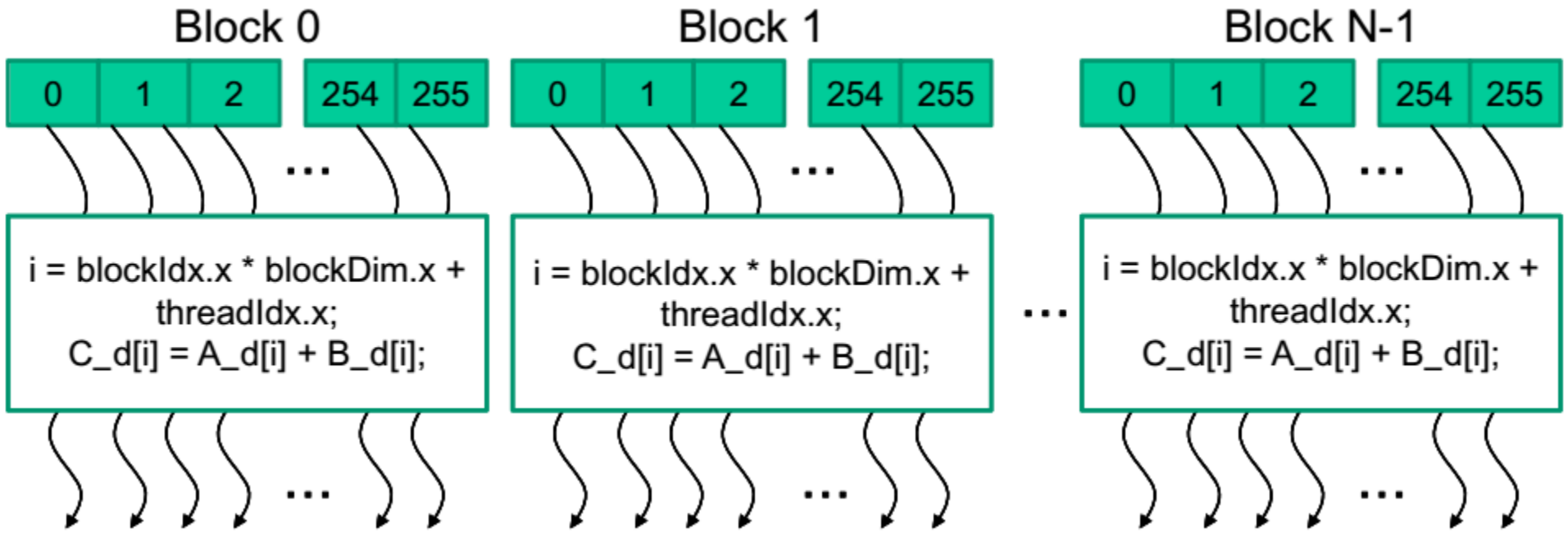
Kernel Functions and Threading



Kernel Functions and Threading

- For a given grid of threads, the number of threads in a block is available in the **`blockDim`** variable.
- A variable of type **`Dim3`**.
  - **`blockDim.x`**
  - **`blockDim.y`**
  - **`blockDim.z`**
- In general, the dimensions of thread blocks should be multiples of 32 due to hardware efficiency reasons.

Kernel Functions and Threading

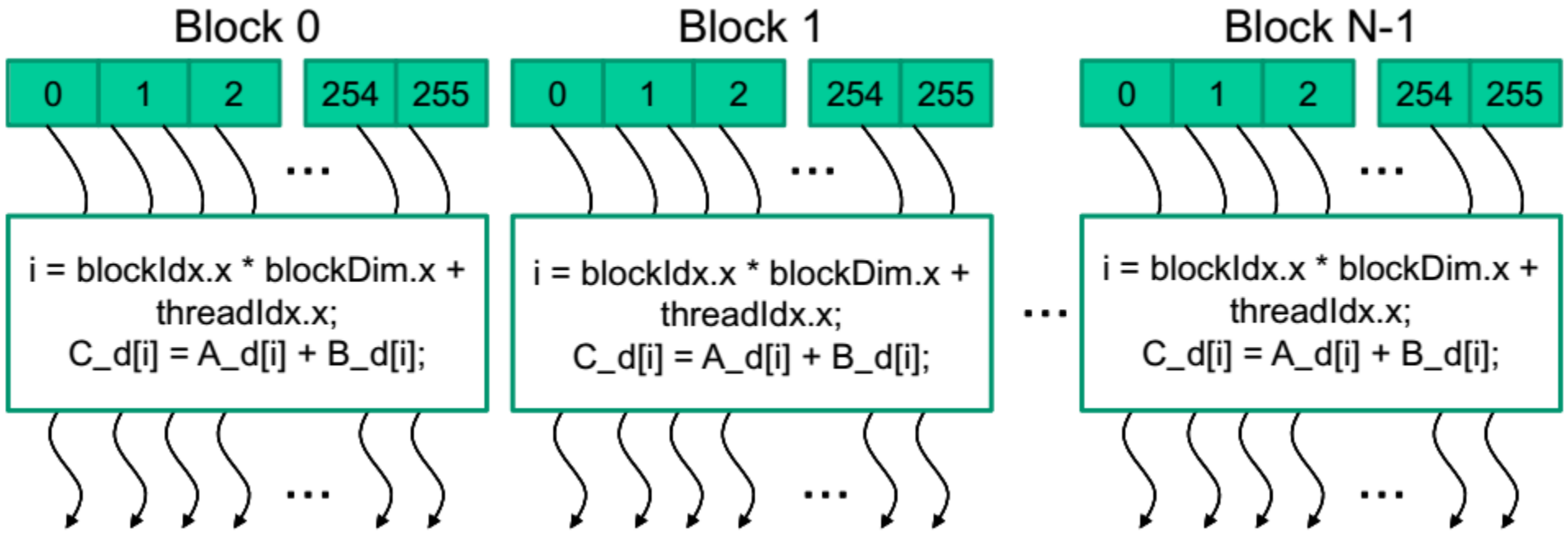


`blockDim.x = 256`

Kernel Functions and Threading

- Each thread in a block has a unique **`threadIdx`** value.
- For example,
  - The first thread in block 0 has value 0 in its **`threadIdx`** variable.
  - The second thread has value 1
  - The third thread has value 2.
  - …
- This allows each thread to combine its **`threadIdx`** and **`blockIdx`** values to create a unique global index for itself with the entire grid.

Kernel Functions and Threading




```
i = blockIdx.x * blockDim.x + threadIdx.x
```

Kernel Functions and Threading

`blockDim.x = 256`

The *i* values of threads in block 0 ranges from 0 to 255

The *i* values of threads in block 1 ranges from 256 to 511

Kernel Functions and Threading

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

What's wrong with this code??!

Kernel Functions and Threading

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

The previous code may access to invalid addresses!

Kernel Functions and Threading

A CUDA specific keyword

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

`__global__` keyword indicates that the function is a kernel and that it can be called from a host function to generate a grid of threads on a device.

Kernel Functions and Threading

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

Kernel Functions and Threading

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}
```

Compare these two codes!

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Kernel Functions and Threading

- How to call **`vecAddKernel`**?
- When the host code launches a kernel, it sets the grid and thread block dimensions via ***execution configuration*** parameters.
  - Number of blocks
  - Number of threads per block
- The ***execution configuration*** parameters are given between the <<< and >>> before the traditional C function arguments.
  - The first configuration parameter gives the number of thread blocks in the grid.
  - The second specifies the number of threads in each thread block.

```
Kernel<<<no. of blocks , no. of threads per block>>>(args);
```

Kernel Functions and Threading

- How to call **`vecAddKernel`**?

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}

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

Kernel Functions and Threading

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

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

    vecAddKernel<<<ceil(n/2560), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
      // Free device memory for A, B, C
     cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

Summary

- Introduction to GPU devices and their application
- Function Declaration using CUDA keywords `__global__`, `__device__`, etc.
- Kernel launch using execution configuration parameters in <<< , >>>
- Predefined variables: `threadIdx`, `blockDim`, etc.
- Runtime API: `cudaMalloc()`, `cudaFree()`, and `cudaMemCpy()`

More to Read

- CUDA C Programming Guide, NVIDIA, March 2019
- Tuning CUDA Applications for Kepler
- Tuning CUDA Applications for Maxwell
- Tuning CUDA Applications for Pascal
- Tuning CUDA Applications for Volta
- Tuning CUDA Applications for Turing