UNIT-4

GPU

UNIT-IV 8

Hours Introduction: GPUs as Parallel Computers, Architecture of a Model GPU, Why More Speed or Parallelism? GPU Computing. Introduction to CUDA: Data Parallelism, CUDA Program Structure, A Vector Addition Kernel , Device Global Memory And Data Transfer, Kernel Functions and Threading.

Self-Study: GPUs History of GPU Computing: Evolution of Graphics Pipelines, Parallel Programming Languages and Models, GPU Memory

Heterogeneous Parallel Computing

CPU drove rapid performance increase and cost reduction in computer applications for more than two decades. i.e. GFLOPS, or giga (1012) floating-point operations per second, to the desktop and TFLOPS, or tera (1015) floating-point operations per second, to cluster servers.

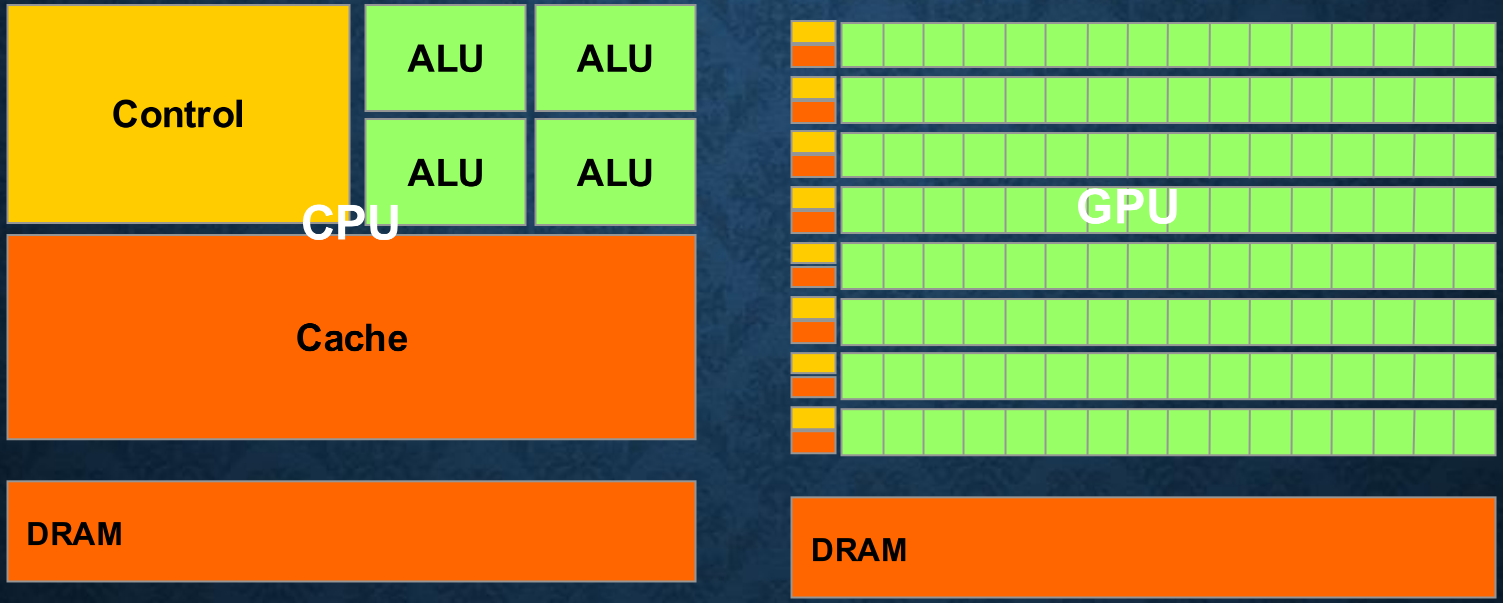
This drive, however, has slowed since 2003 due to energy consumption and heat dissipation issues that limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU.

Heterogeneous Parallel Computing

The semiconductor industry has settled on two main trajectories for designing microprocessors

- The multicore trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores.
- In contrast, the many-thread trajectory focuses more on the execution throughput of parallel applications.

CPUS AND GPUS HAVE FUNDAMENTALLY DIFFERENT DESIGN PHILOSOPHIES



Multicore CPU

Design of a CPU is optimized for sequential code performance.

Use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution.

large cache memories are provided to reduce the instruction and data access latencies of large complex applications

Memory bandwidth is another important issue. The speed of many applications is limited by the rate at which data can be delivered from the memory system into the processors.

Latency-oriented design

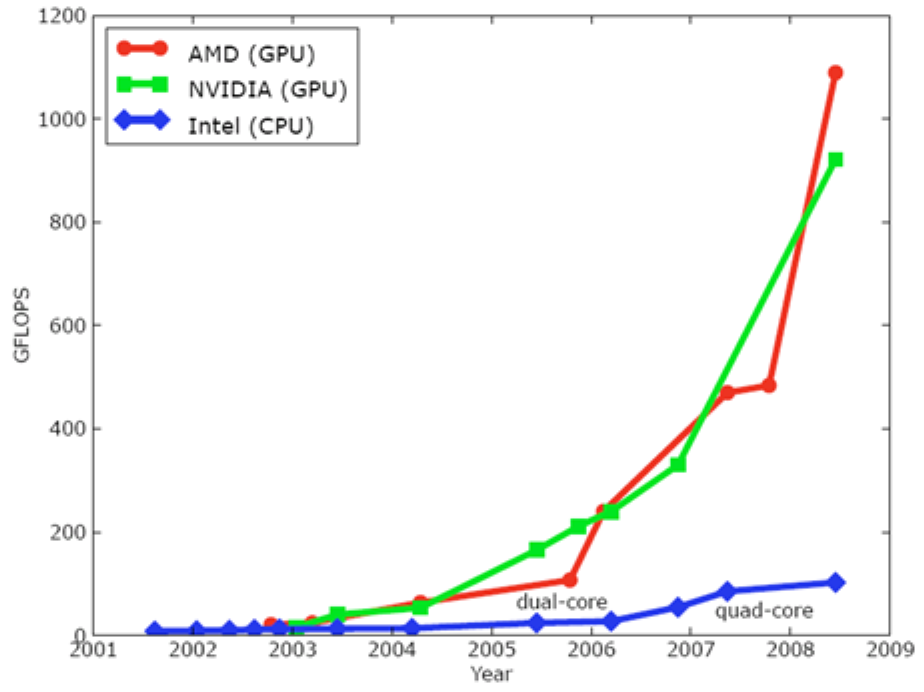
CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time

Many-core GPU

- Shaped by the fast-growing video game industry that expects tremendous massive number of floating-pint calculations per video frame.
- Motive to look for ways to maximize the chip area and power budget dedicated to floating-point calculations : Optimize for the execution throughput of massive number of threads.
- The design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency.
- The reduce area and power on memory and arithmetic allows designers to have more cores on a chip to increase the execution throughput.
- Throughput-oriented design

CPU + GPU

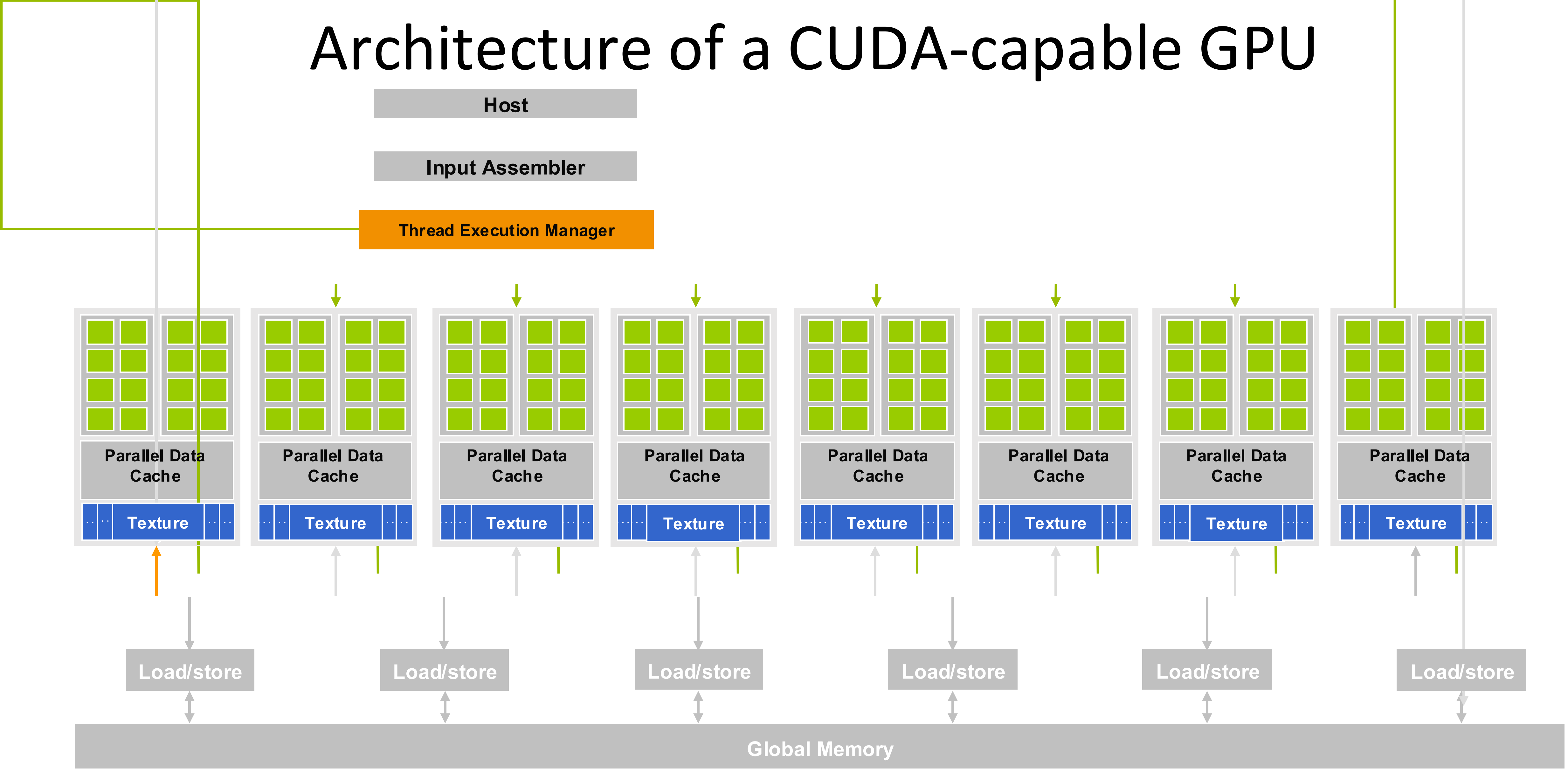
- GPU will not perform well on tasks on which CPUs are design to perform well. For program that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs.
- When a program has many threads, GPUs with higher execution throughput can achieve much higher performance than CPUs.
- Many applications use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.



WHY MASSIVELY PARALLEL PROCESSOR

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until last year, programmed through graphics API
  - GPU in every PC and workstation – massive volume and potential impact

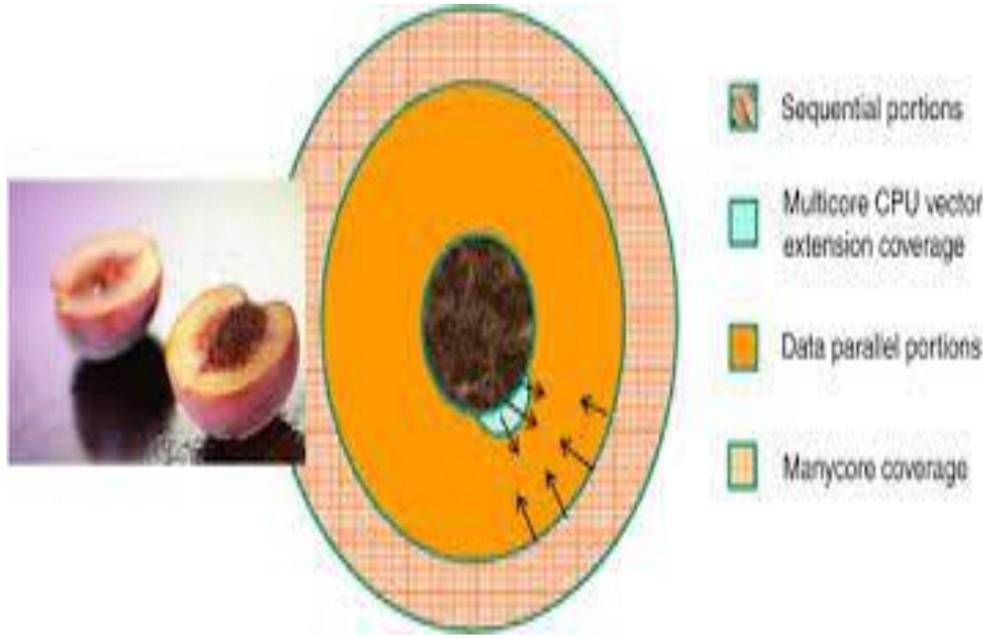
Architecture of a CUDA-capable GPU



It is organized into an array of highly threaded streaming multiprocessors (SMs).

- Two streaming multiprocessors form a building block.
- The number of SMs in a building block can vary from one generation of CUDA GPUs to another generation
- Each SM has few streaming processors (SPs) that share control logic and an instruction cache.
- Each GPU comes with multiple gigabytes of DRAM (global memory).
- A good application runs 5k to 12k threads. CPU support 2 to 8 threads.

Stretching Traditional Architectures



The centre core of the peach represent the sequential portions of application.

The sequential portions have been target of modern instruction level parallelism.
In modern CPU that include cache memories ,branch prediction, data forwarding are important in preserving instruction level parallelism in these portions.

Orange portion is data parallel portion of application

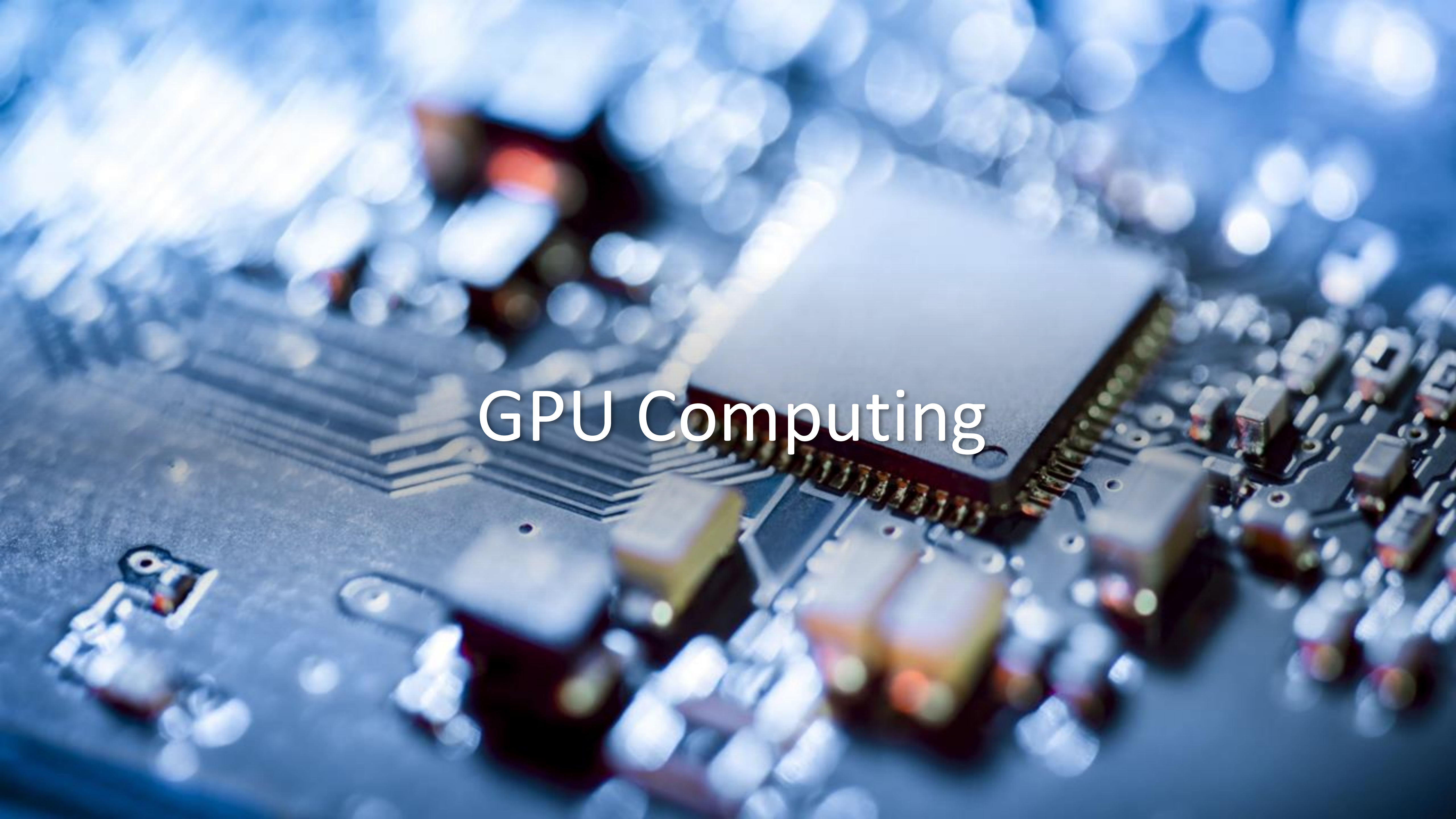
These have large data sets whose elements are to be processed on parallel.
Traditional CPUs do not have sufficient amount of execution resources to achieve dramatic increase in performance.

Meshed layer represents types of data parallel applications that can be efficiently covered by manycore architectures today.

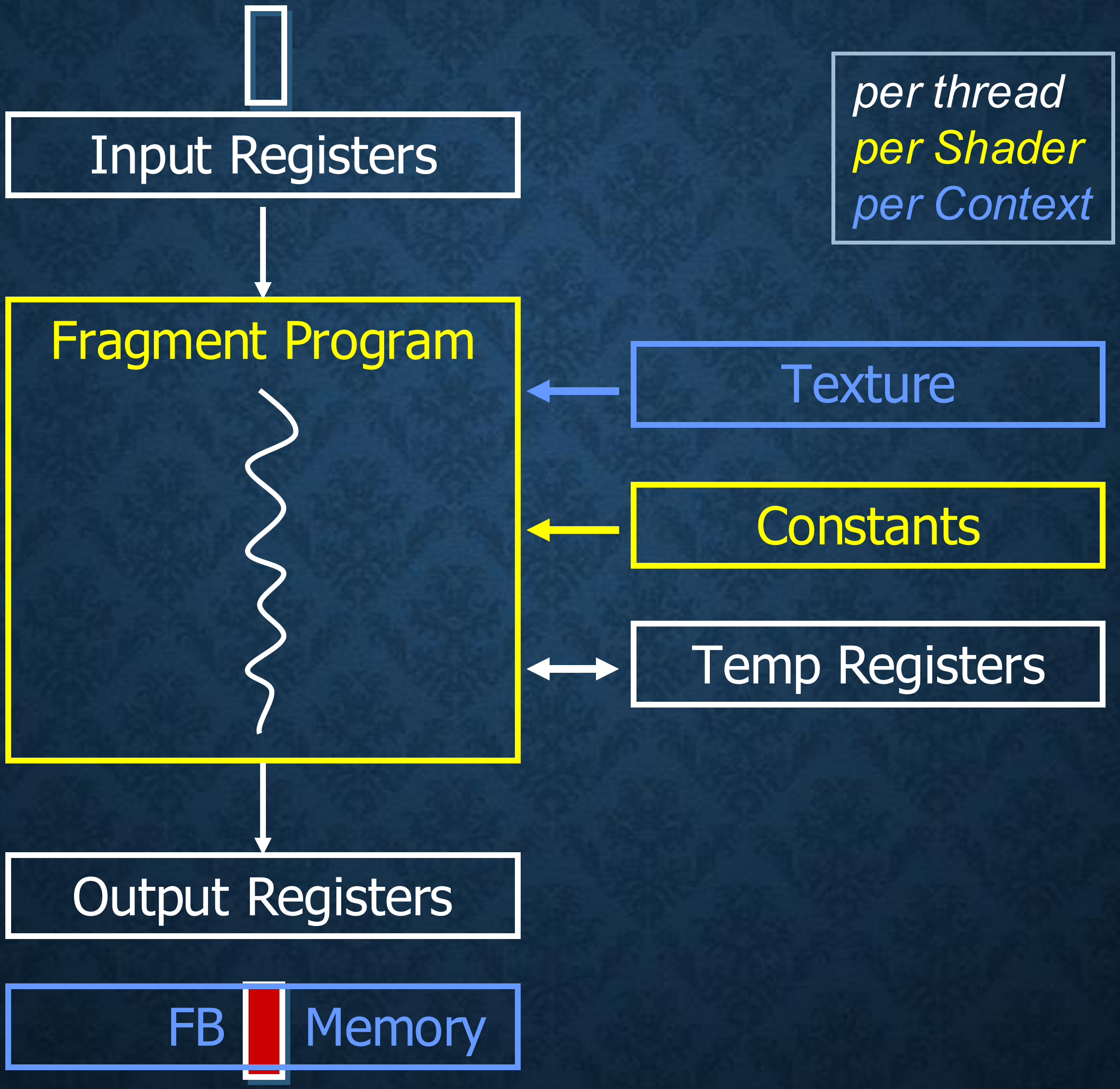
The figure explains us that there is a large population of parallel application that are currently neither covered by CPUs nor many core processors.

WHY MORE SPEED OR PARALLELISM?

- The main motivation for massively parallel programming is for applications to enjoy continued speed increase in future hardware generations.
- When an application is suitable for parallel execution, a good implementation on a GPU can achieve more than 100 times (100X) speedup over sequential execution on a single CPU core.
- If the application includes data parallelism, it’s often a simple task to achieve a 10X speedup with just a few hours of work



GPU Computing



The restricted input and output capabilities of a shader programming model.

GPU beyond Graphics

Architecture of a GPU

Same components as a typical CPU

However,…

- More computing elements
- More types of memory

Original GPUs had vertex and pixel shaders

- Specifically for graphics

Modern GPUs are slightly different

- CUDA – Compute Unified Device Architecture

Computational Elements of a GPU

Streaming Processor – Core of the design

- Place where all of the computation takes place

Streaming Multiprocessor

- Groups of streaming multiprocessors
- In addition to the SPs, these also contain the Special Function Units and Load/Store Units

Instructional Schedulers

Complex Control Logic

Types of GPU Memory

Global

DRAM

Slowest Performance

Texture

Cached Global Memory

“Bound” at runtime

Constant

Cached Global Memory

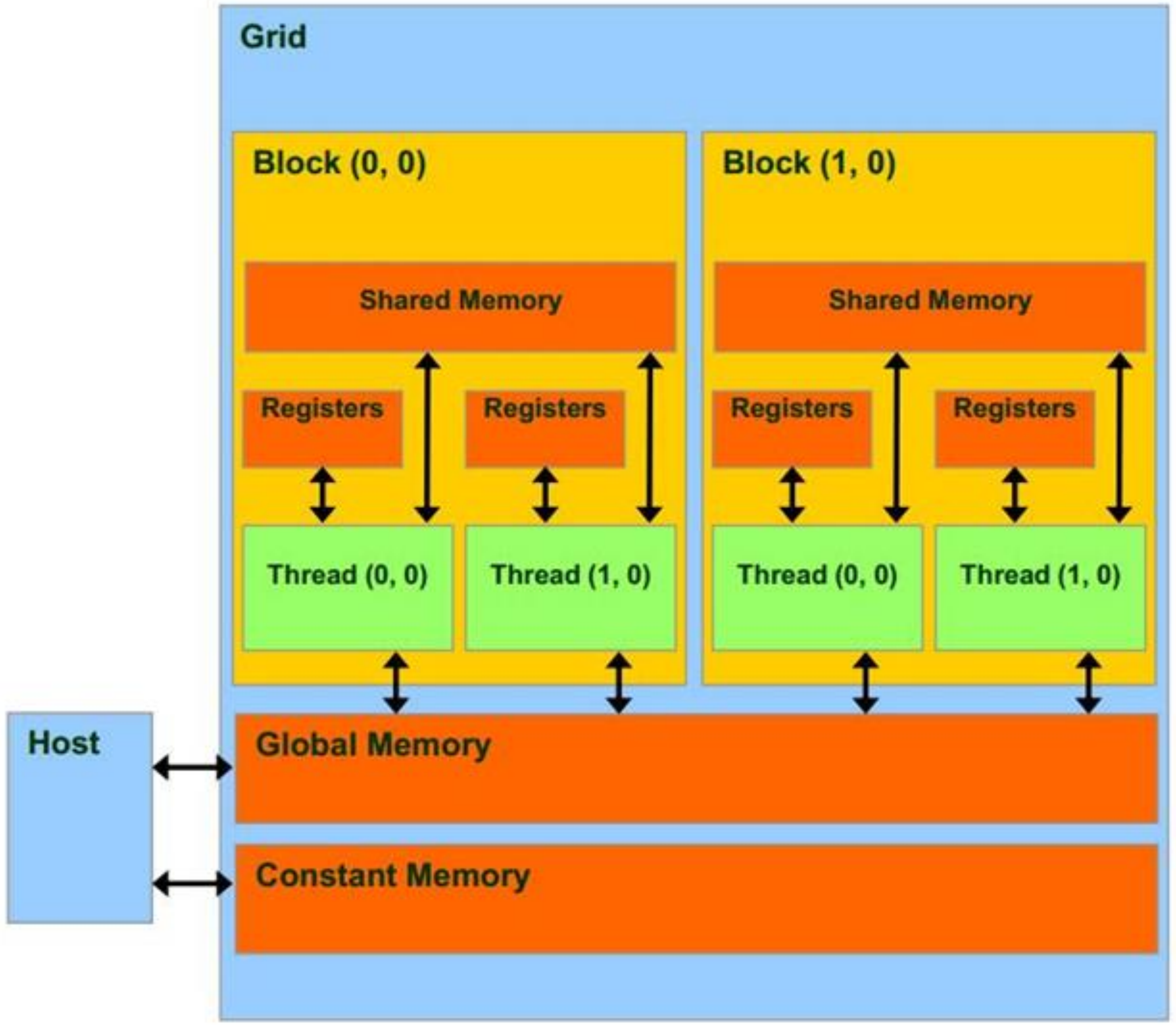
Shared

Local to a block of threads

- Evolution of Graphics Pipelines — mcs572 0.7.8 documentation (uic.edu)

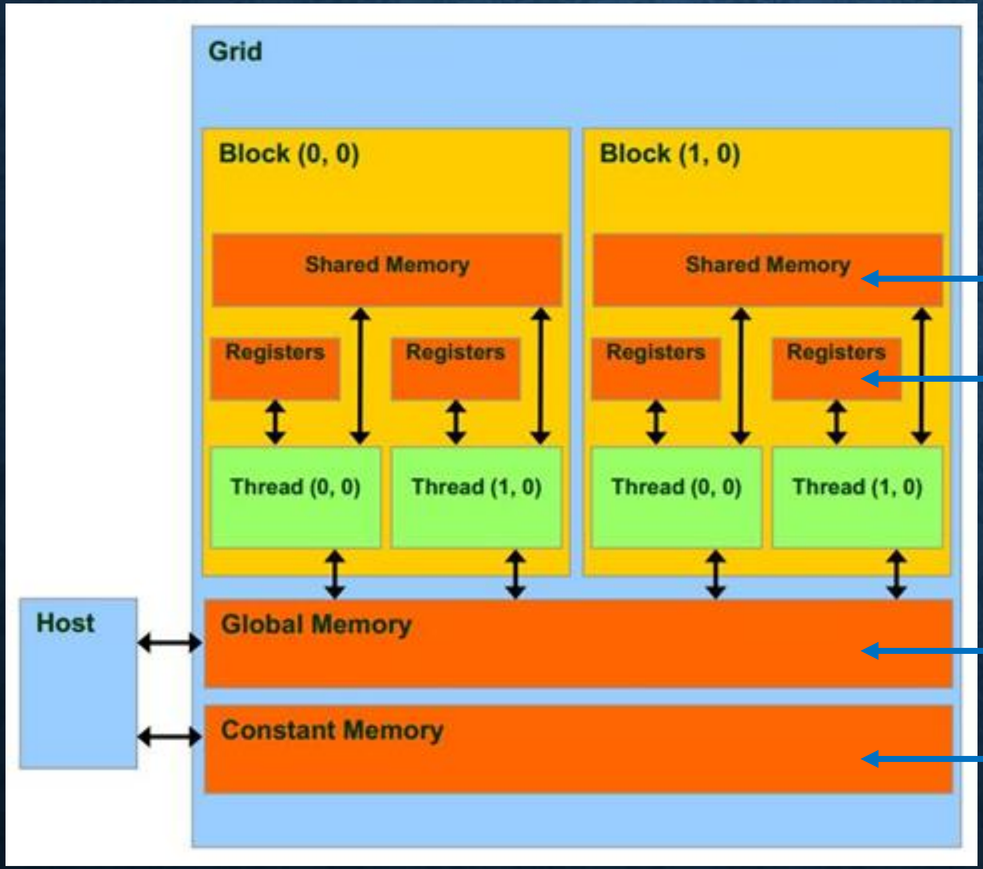
Terminology

Thread	Thread – The smallest grain of the hierarchy of device computation
Block	Block – A group of threads
Grid	Grid – A group of blocks
Warp	Warp – A group of 32 threads that are executed simultaneously on the device
Kernel	Kernel - The creator of a grid for GPU execution



Grids, Blocks, and Threads

CUDA MEMORY

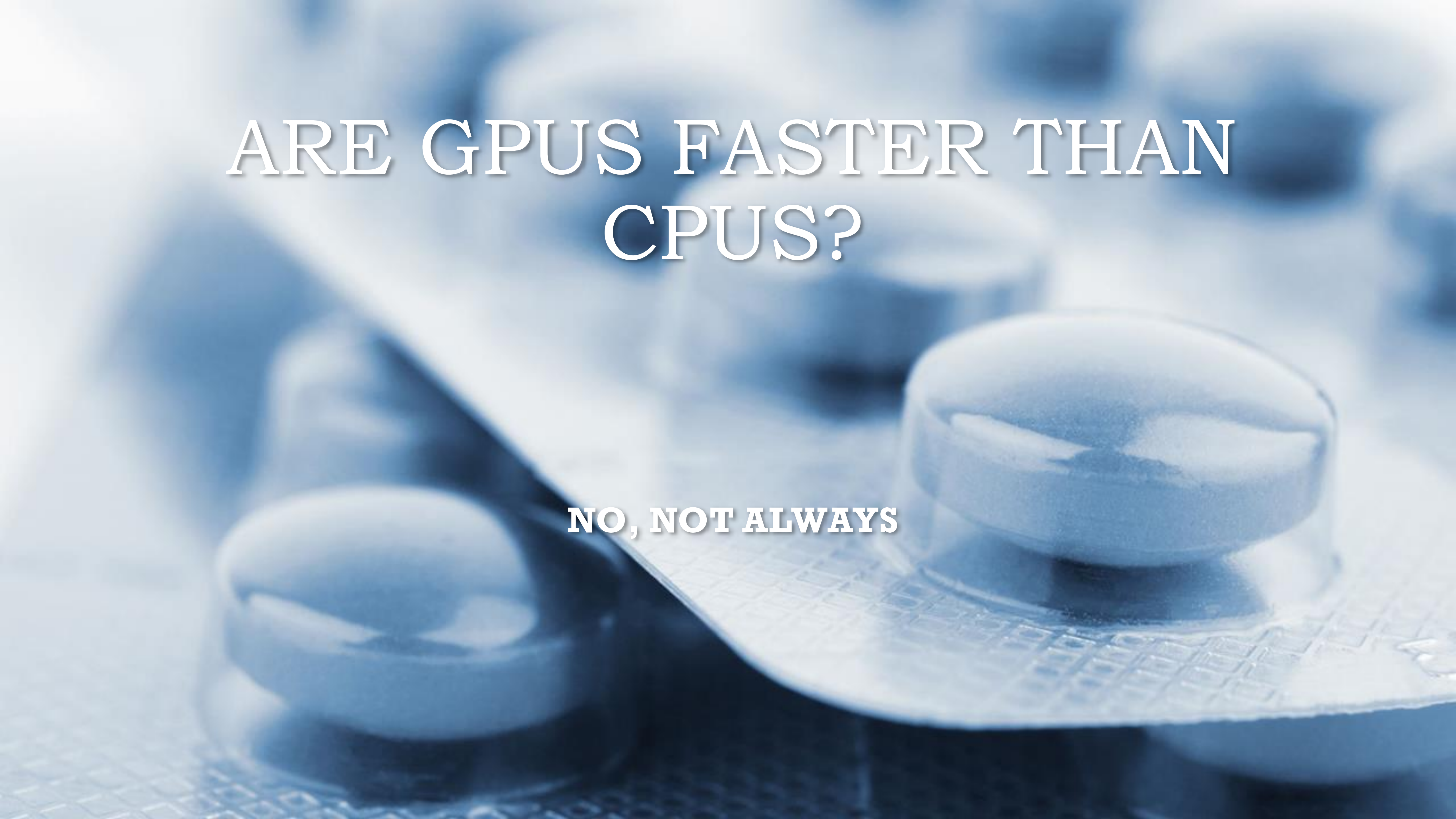


Faster, per-block

Fastest, per-thread

Slower, global

Read-only, cached



ARE GPUS FASTER THAN CPUS?

NO, NOT ALWAYS

Most general-purpose computing, a CPU performs much better than a GPU.

That's because CPUs are designed with fewer processor cores that have higher clock speeds than the ones found on GPUs, allowing them to complete series of tasks very quickly.

GPUs, on the other hand, have much greater number of cores and are designed for a different purpose

HETEROGENEOUS COMPUTING

Host: the CPU and its memory

Device: the GPU and its memory

Introduction to CUDA

CUDA C is an extension to the popular C programming languages with new keywords and application programming interfaces for programmers to take advantage of heterogeneous computing systems that contain both CPUs and massively parallel GPU’s.

To a CUDA programmer, the computing system consists of a host that is a traditional CPU, such as an Intel architecture microprocessor in personal computers today, and one or more devices that are processors with a massive number of arithmetic units.

CUDA

A CUDA device is typically a GPU.

CUDA devices accelerate the execution of these applications by applying their massive number of arithmetic units to these data-parallel program sections.

DATA PARALLELISM

Modern software applications often process a large amount of data and incur long execution time on sequential computers.

In general, data parallelism is the main source of scalability for parallel programs.

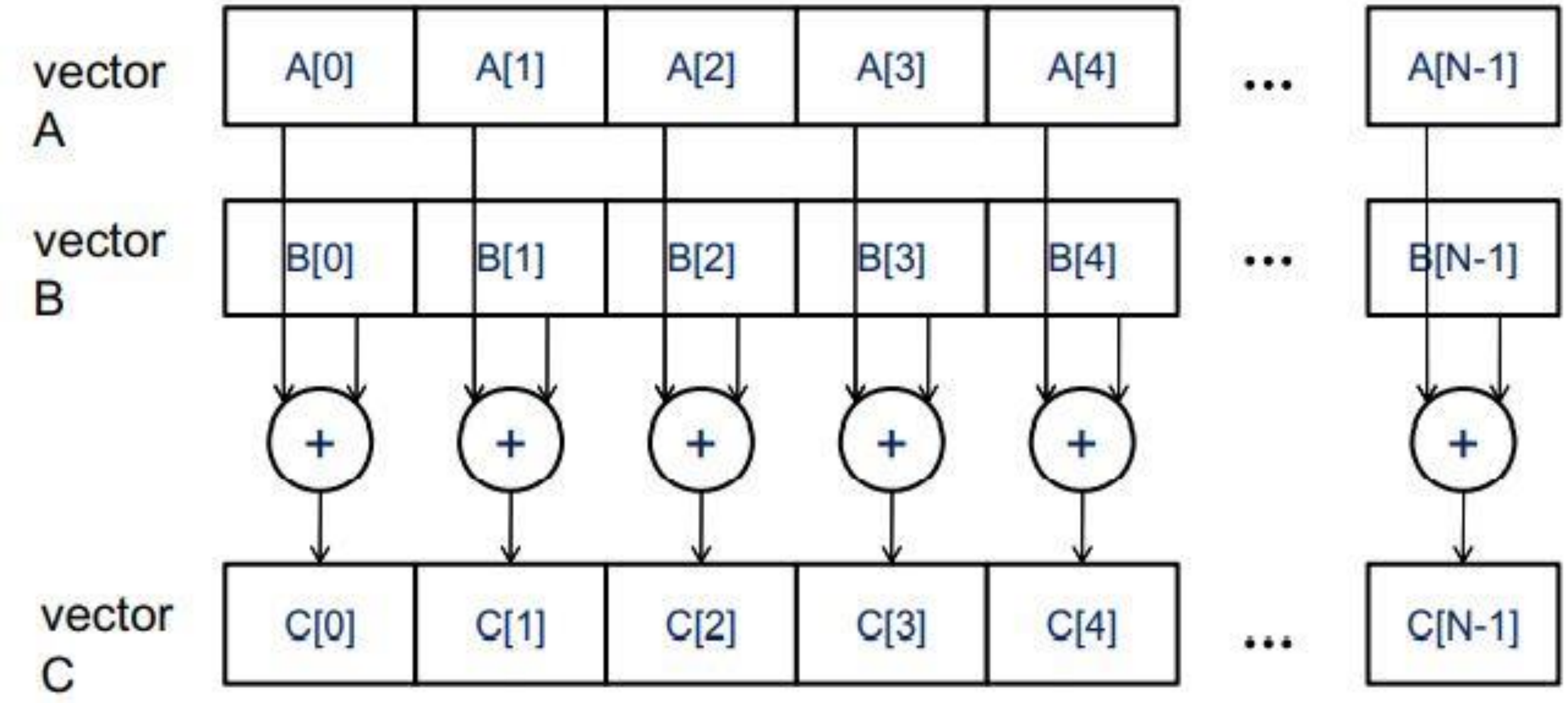
Task parallelism

- 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. Each of these would be a task. Task parallelism exists if the two tasks can be done independently

CUDA –provides Data & Task Paralleism

Data Parallelisms	Task Parallelisms
1. Same task are performed on different subsets of same data.	1. Different task are performed on the same or different data.
2. Synchronous computation is performed.	2. Asynchronous computation is performed.
3. As there is only one execution thread operating on all sets of data, so the speedup is more.	3. As each processor will execute a different thread or process on the same or different set of data, so speedup is less.
4. Amount of parallelization is proportional to the input size.	4. Amount of parallelization is proportional to the number of independent tasks is performed.
5. It is designed for optimum load balance on multiprocessor system.	5. Here, load balancing depends upon on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.

Example of data parallelism : Vector addition



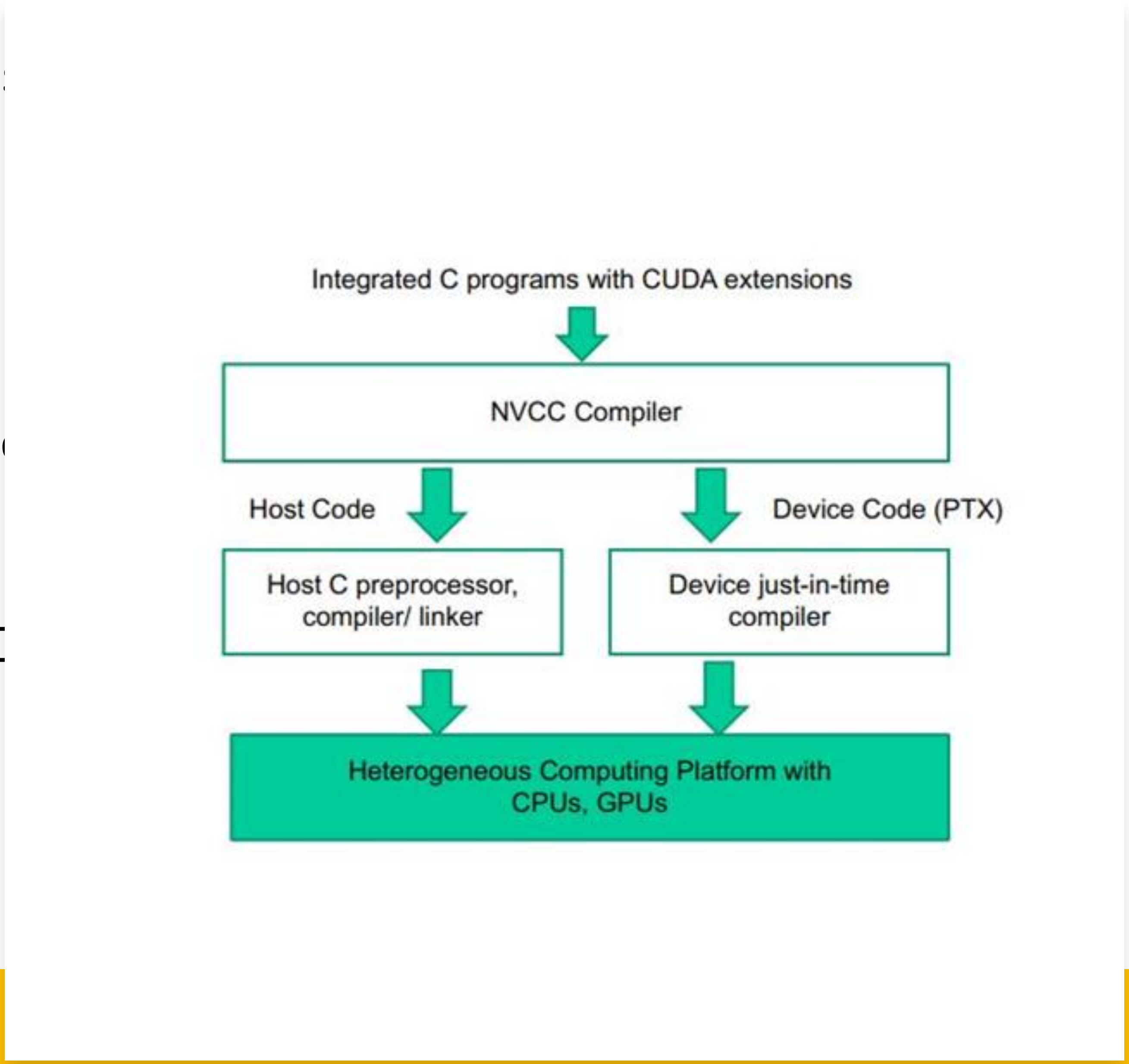
CUDA PROGRAM STRUCTURE

- 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 can have a mixture of both host and device code.
- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device functions and data declarations into any C source file.
- The function or data declarations for the device are clearly marked with special CUDA keywords.
- These are typically functions that exhibit a rich amount of data parallelism

- Once device functions and data declaration are added to a source file, it is no longer acceptable to a traditional C compiler.

- The code needs to be compiled by a compil that recognizes and understands these additional declaration.
- We will be using a CUDA C compiler by NVIL called NVCC.

CUDA keywords are used to separate the host code and device code.

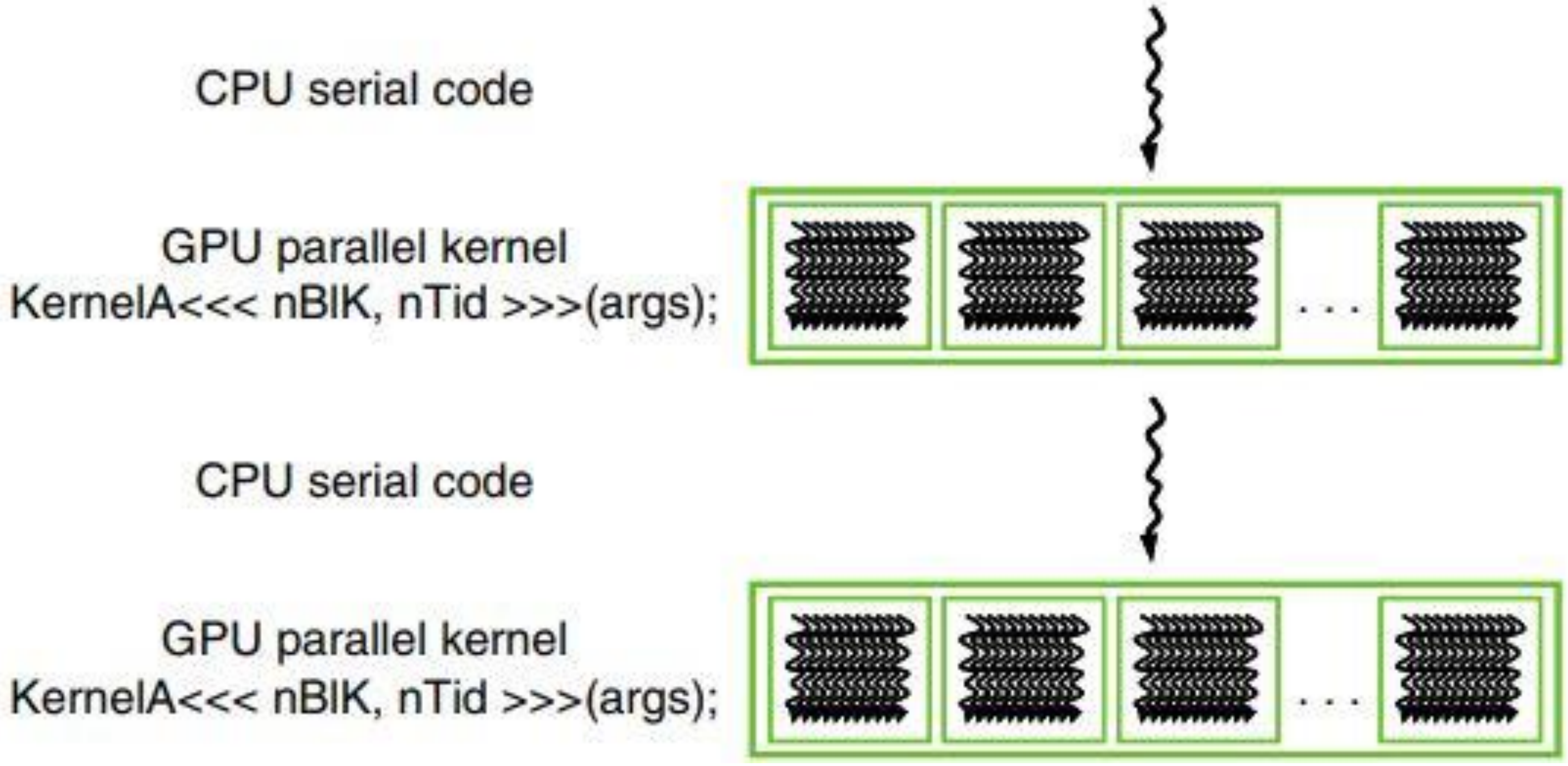

Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/ linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

The device code is marked with CUDA keywords for labelling data-parallel functions, called kernels, and their associated data structures.

The device code is further compiled by a runtime component of NVCC and executed on a GPU device.

In situations where there is no device available or a kernel can be appropriately executed on a CPU, one can also choose to execute the kernel on a CPU using tools like MCUDA.

Execution of a CUDA program

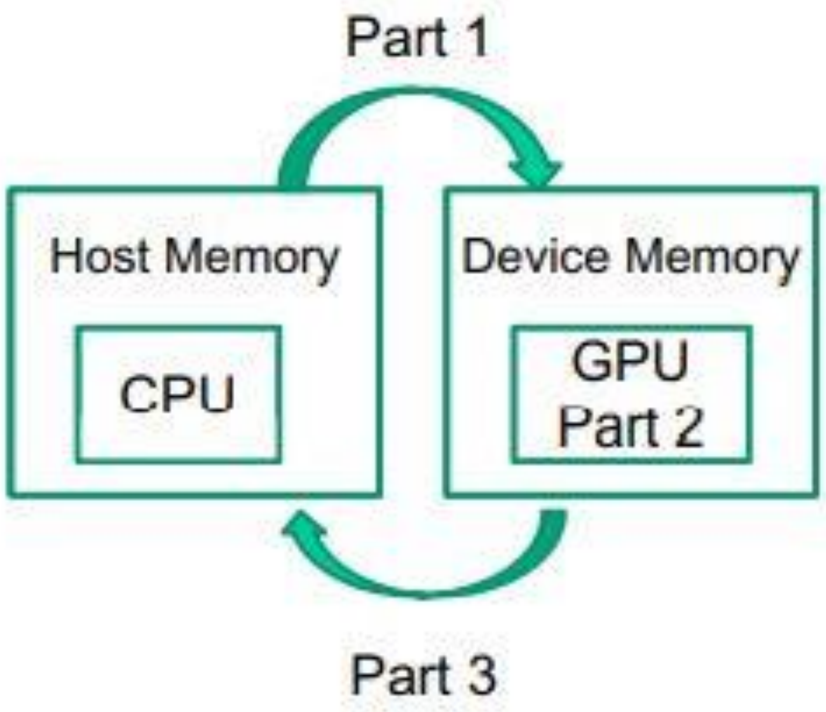


A VECTOR ADDITION KERNEL : C program

```
// 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);
}
```

- In each piece of host code, we will prefix the names of variables that are mainly processed by the host with h_ and those of variables that are mainly processed by a device d_

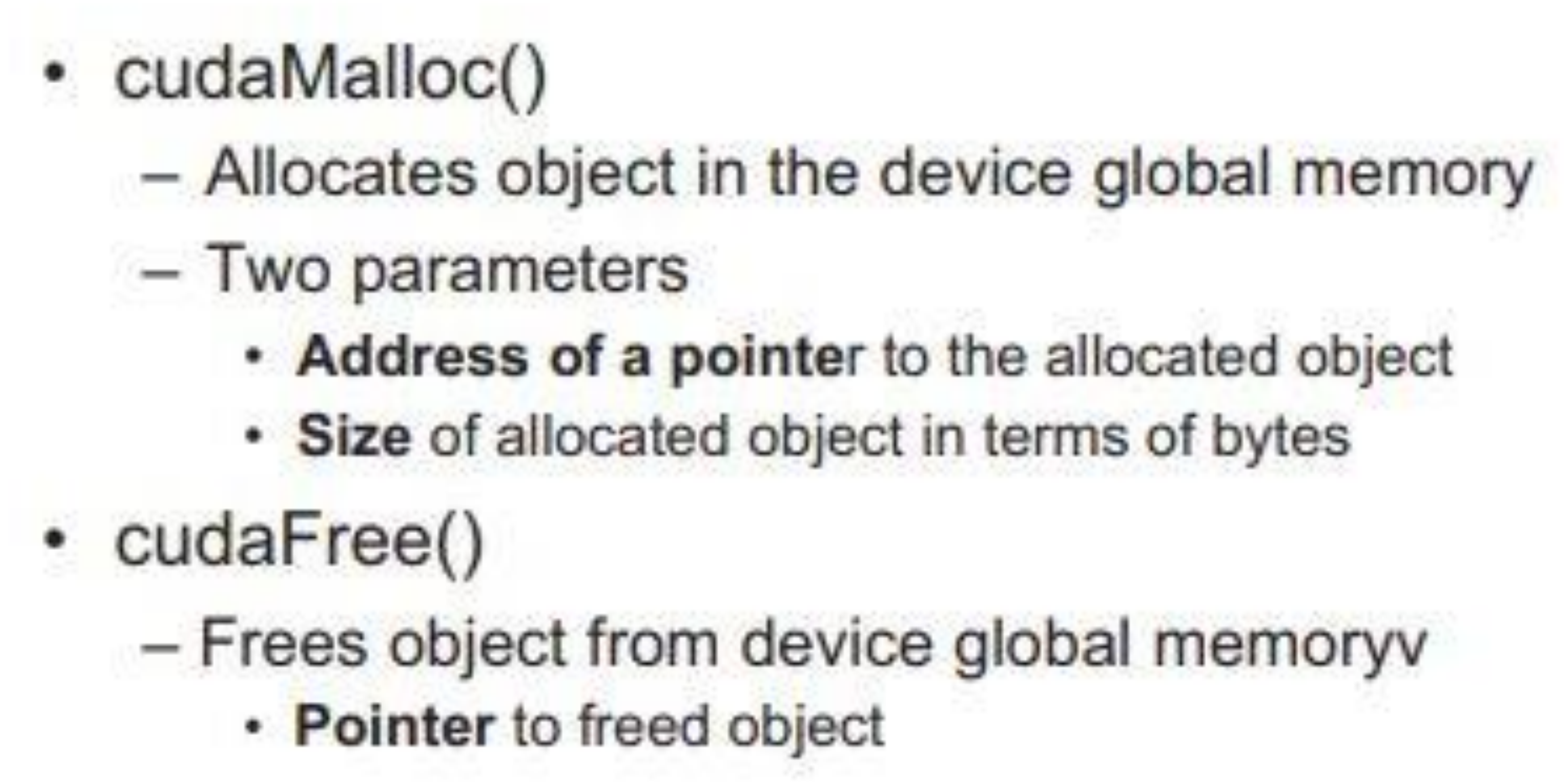
```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
  int size = n* sizeof(float);
   float *A_d, *B_d, *C_d;
   …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

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 AND DATA TRANSFER

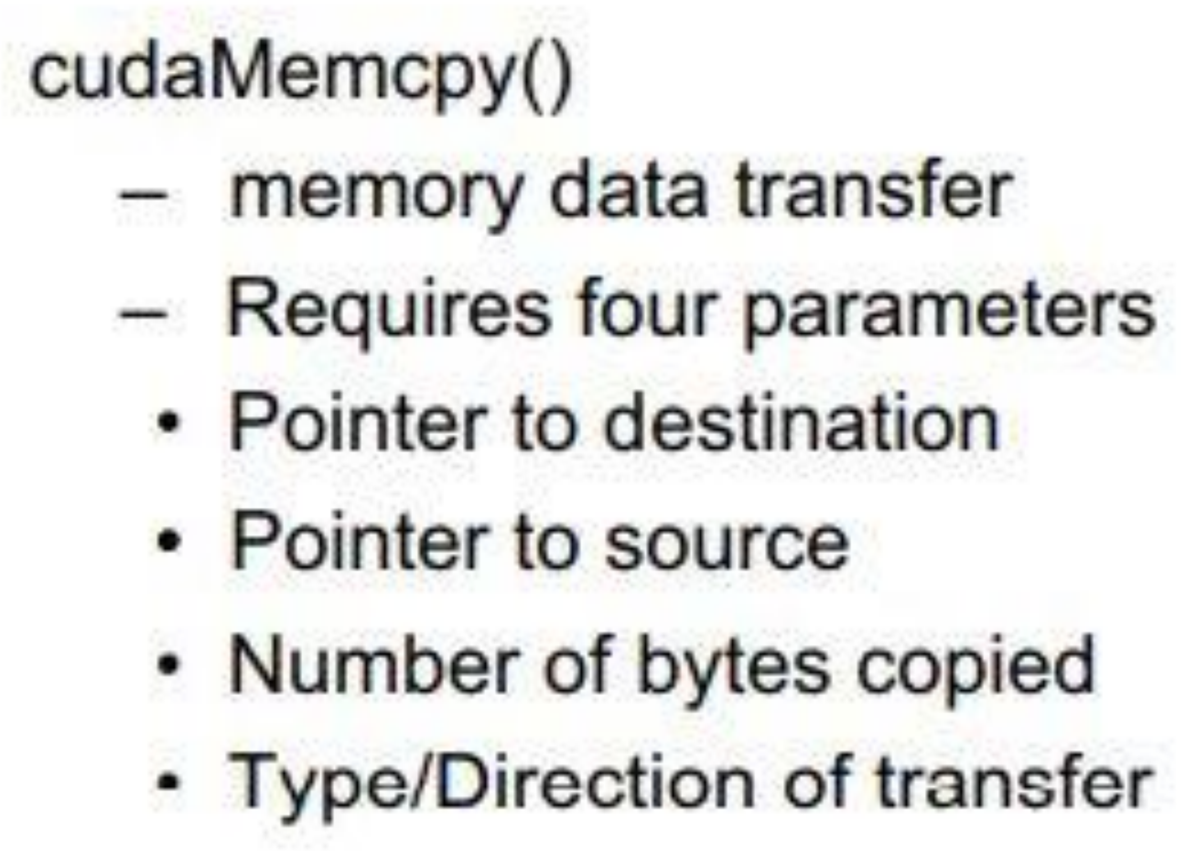
- In CUDA, host and devices have separate memory spaces.
- To execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer pertinent data from the host memory to the allocated device memory.
- After device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed.
- The CUDA runtime system provides Application Programming Interface (API) functions to perform these activities on behalf of the programmer

API functions for allocating and freeing device global memory.



- 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

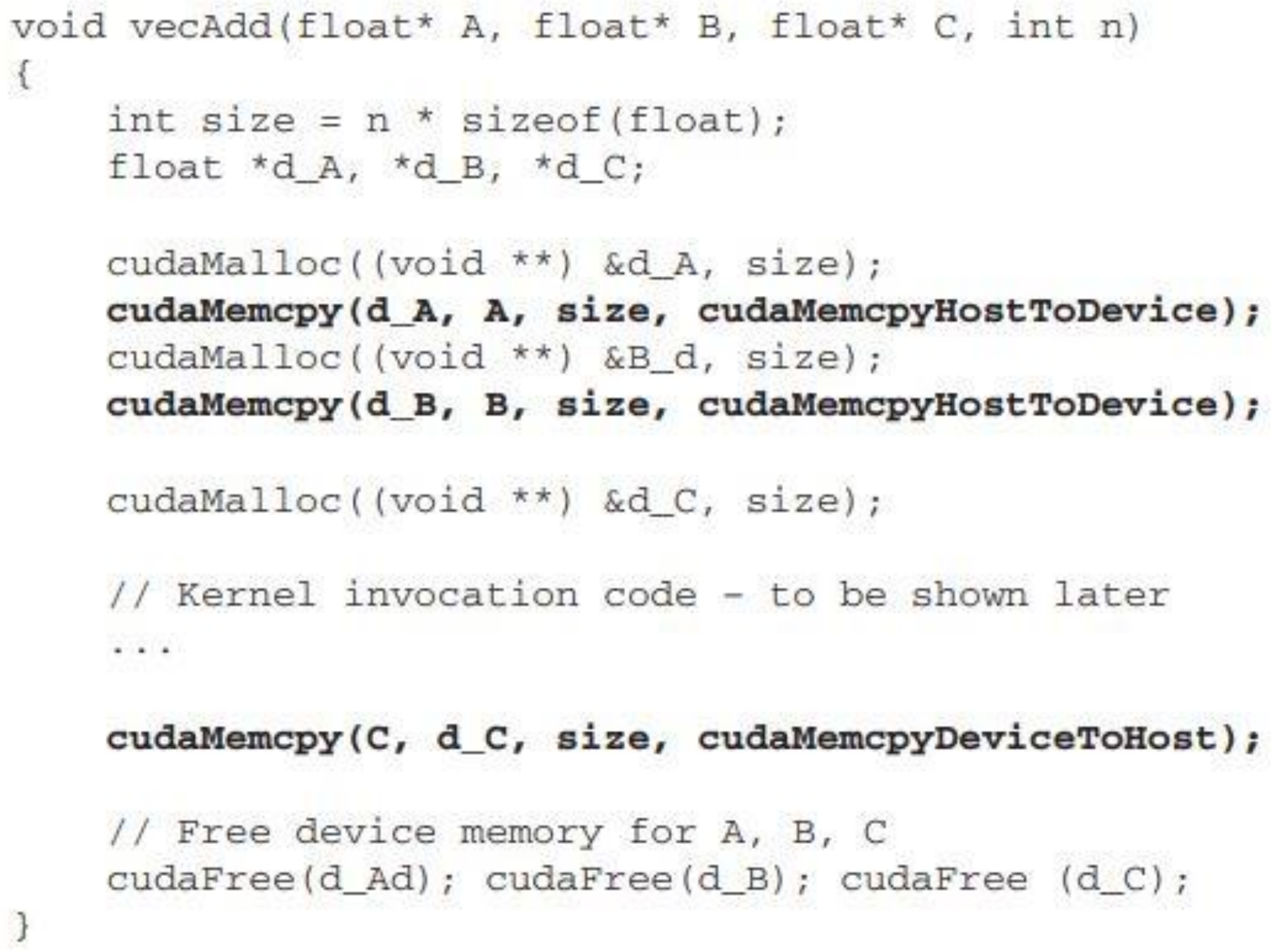
API functions for allocating and freeing device global memory.



cudaMemcpy()

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

Example

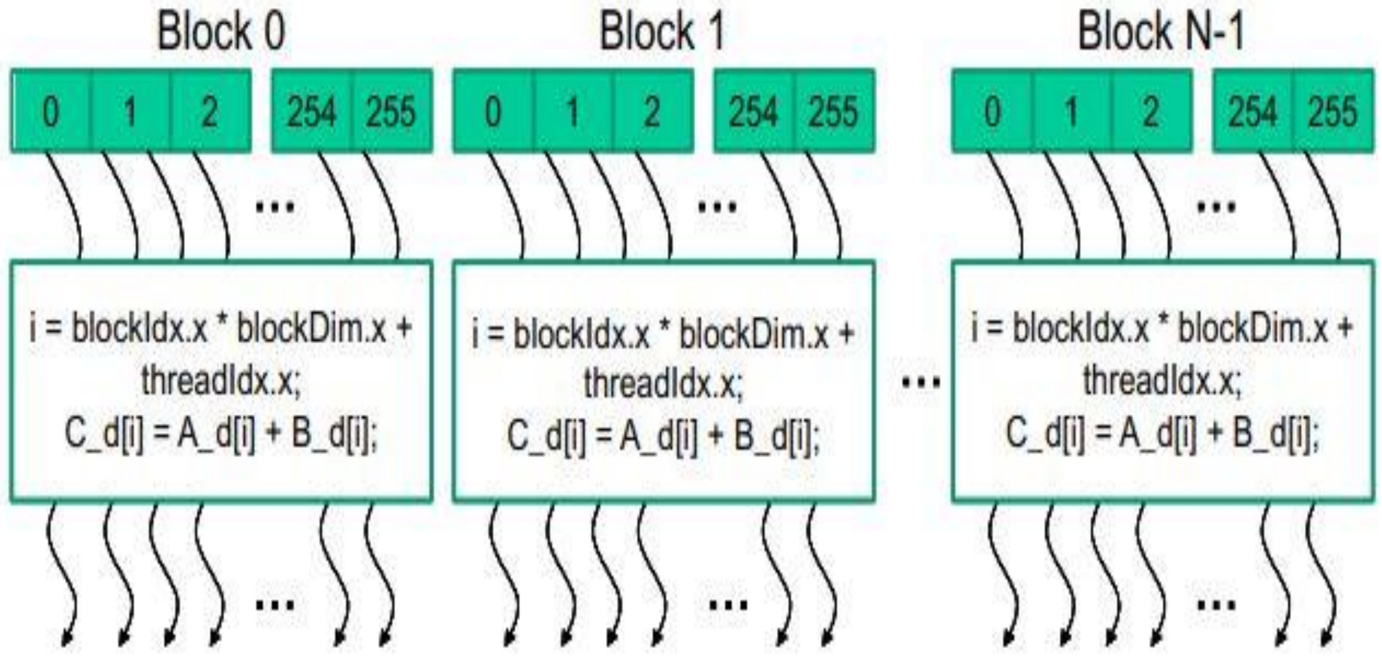


```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    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

- CUDA programming is an instance of the well-known SPMD, since all these threads execute the same code.
- 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, which will be referred to as blocks.
- All blocks of a grid are of the same size; each block can contain up to 1,024 threads.

- The number of threads in each thread block is specified by the host code when a kernel is launched.
- For a given grid of threads, the number of threads in a block is available in the blockDim variable.
- The same kernel can be launched with different numbers of threads at different parts of the host code.
- The value of the blockDim.x variable is 256.
- In general, the dimensions of thread blocks should be multiples of 32 due to hardware efficiency reasons



Block 0
Block 1
Block N-1
0
1
2
254
255
...
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];

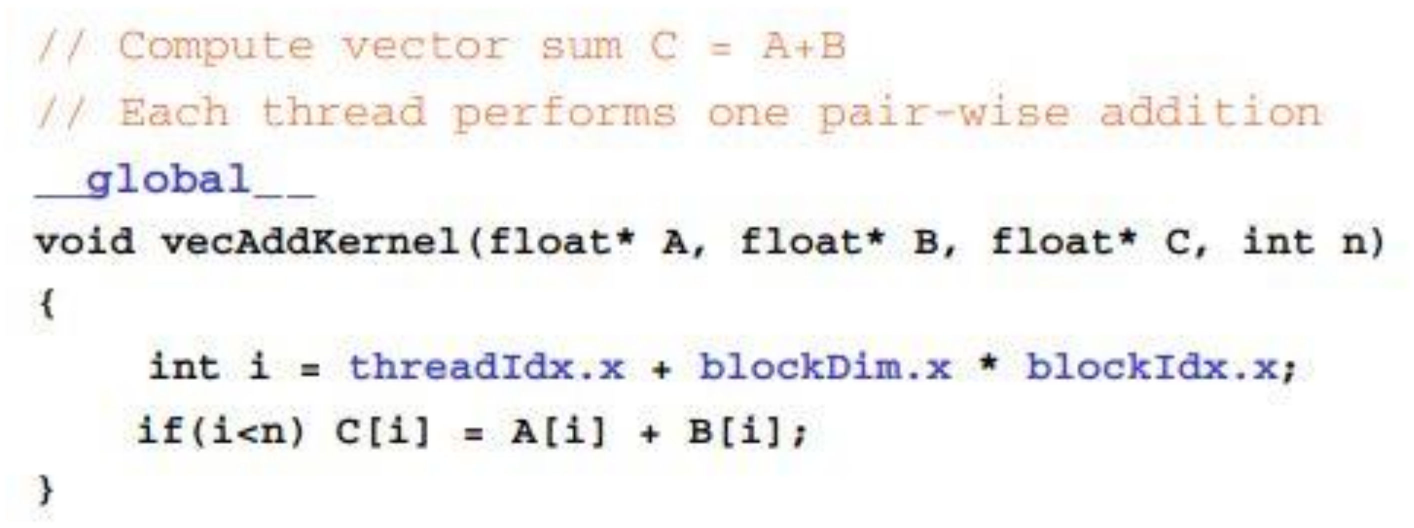
- Each thread in a block has a unique threadIdx value.
- For example, the first thread in block 0 has value 0 in its threadIdx variable, the second thread has value 1, the third thread has value 2, etc.
- This allows each thread to combine its threadIdx and blockIdx values to create a unique global index for itself with the entire grid.
- a data index i is calculated as

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

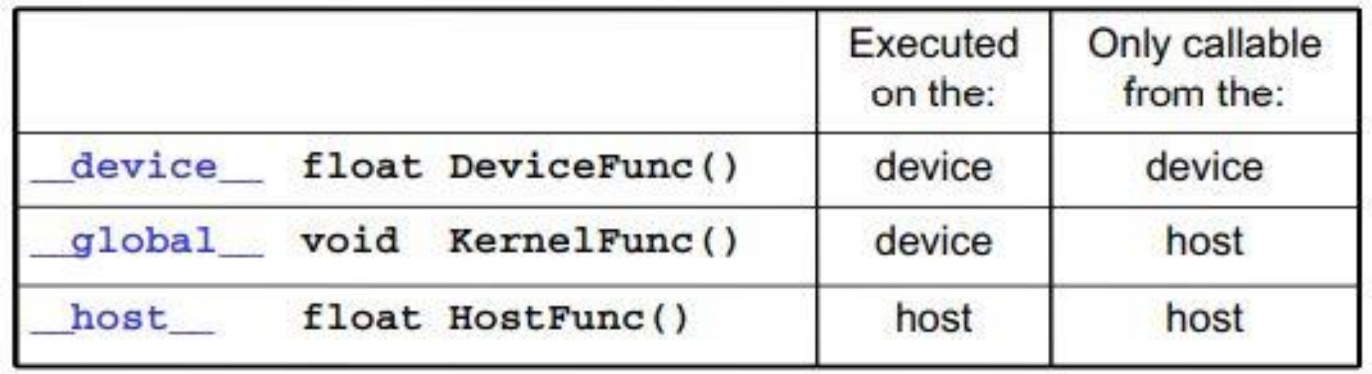
- since blockDim is 256 in our example, the i values of threads in block 0 ranges from 0 to 255.
- The i values of threads in block 1 range from 256 to 511. The i values of threads in block 2 range from 512 to 767. That is, the i values of the threads in these three blocks form a continuous coverage of the values from 0 to 767.

- By launching the kernel with a larger number of blocks, one can process larger vectors. By launching a kernel with n or more threads, one can process vectors of length.

Note that all threads execute the same kernel code



```
// 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];
}
```



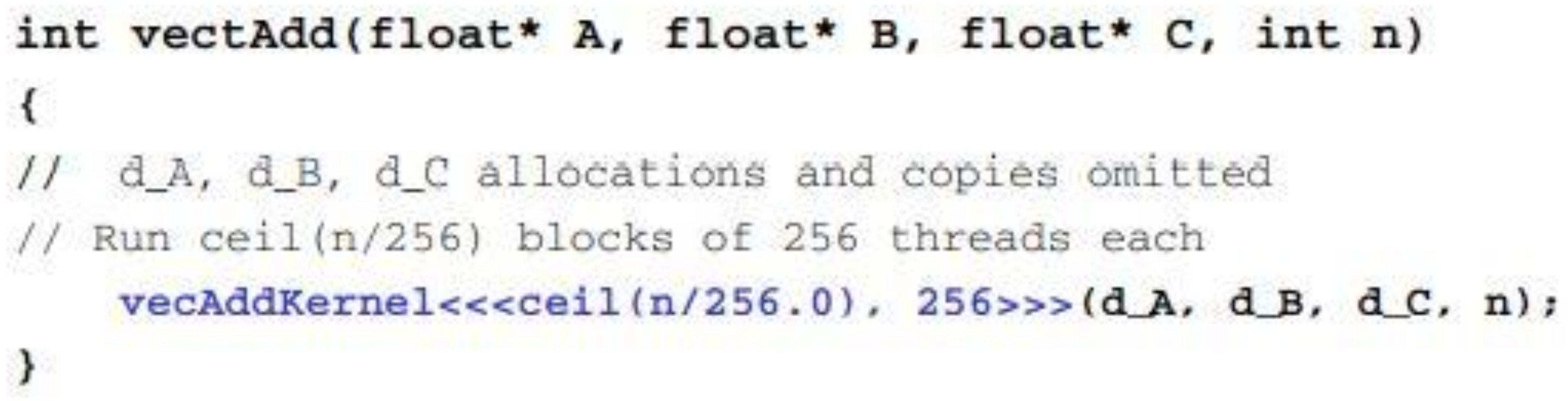
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

When the host code launches a kernel, it sets the grid and thread block dimensions via execution configuration parameters.

The configuration parameters are given between the <<< and >>> before the traditional C function arguments.

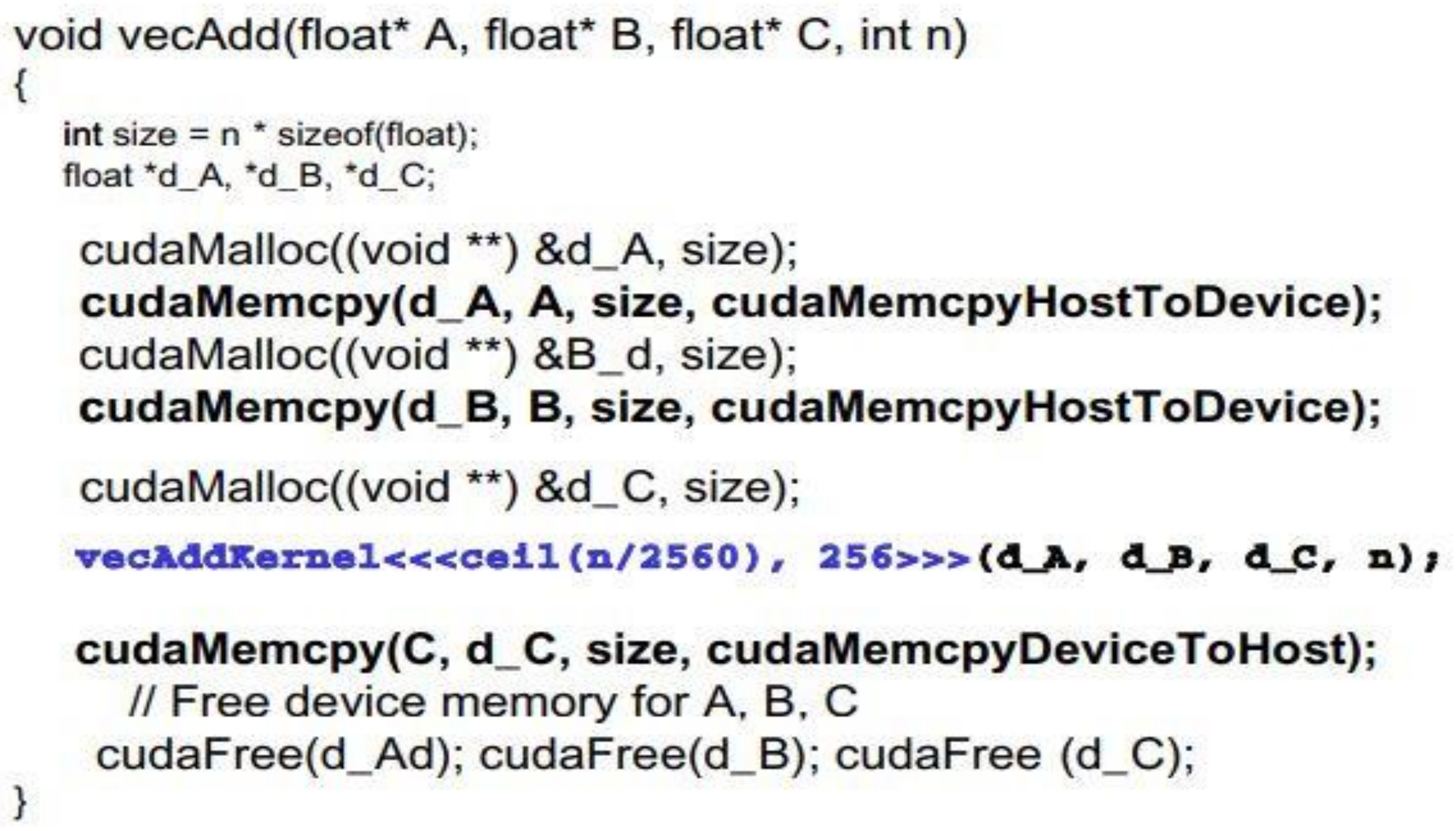
The first configuration parameter gives the number of thread blocks in the grid. The second specifies the number of threads in each thread block.

To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0.



```
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);
}
```

Example



```
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);
}
```