Unit 5

CUDA Architecture

- Introduction to GPU Architecture overview, Introduction to CUDA C-
- CUDA programming model, write and launch a CUDA kernel, Handling Errors, CUDA memory
- model, Manage communication and synchronization, Parallel programming in CUDA- C.

Objective:-To know the operating system requirements to qualify in handling the parallelization

Outcome:-Build the logic to parallelize the programming task

Reference Book:-Shane Cook, “CUDA Programming: A Developer's Guide to Parallel Computing with GPUs”

Motivation for GPU computing

- Moore's law states that the number of transistors in an integrated circuit will double every two years.
- Moore made this prediction in 1965!
- Over the last decade, we have begun to see an end to this motivates the need for other ways to increase computational performance.
- This is where Heterogeneous computing (specifically for us, GPU computing) comes in.

High level hardware overview

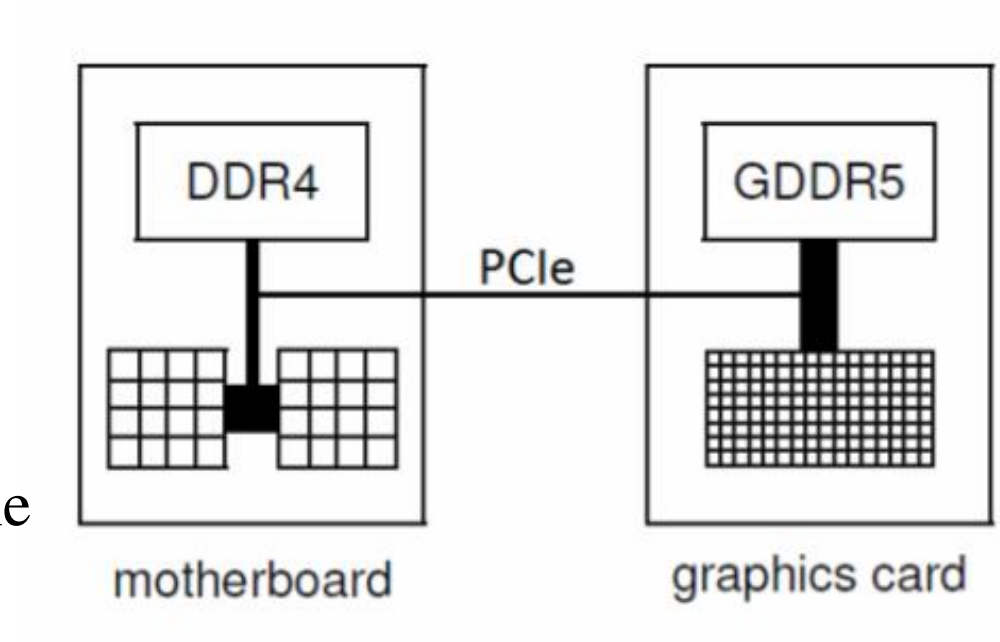
- Heterogeneous computing makes use of more than one type of computer processor (for example a CPU and a GPU).
- Heterogeneous computing is sometimes called hybrid computing or accelerated computing.
- Some of the motivations for employing heterogeneous technologies are:
    - Significant reductions in floor space needed.
    - Energy efficiency.
    - Higher throughput.

High level hardware overview

A typical server configuration is to have one or more CPUs communicating with one or more GPUs through the PCIe bus.

The CPU has 10's of cores, the GPU has 1000's.

The server in which the GPU sits is often referred to as the "host"The GPU itself is often called the "device"



Generations of GPUs

- Several generations of GPUs have been released now.Starting in 2007 with the first GeneralPurpose Graphics Processing Unit (GPGPU) called Tesla.
- The “Tesla” name has been subsequently used as the identifier for all NVIDIA HPC cards.
- Currently, 5 generations of hardware cards are in use, although the Kepler and Maxwell generations are becoming more scarce.

Kepler (compute capability 3.x):
• first released in 2012, including HPC cards.
• excellent double precision arithmetic (DP or fp64).
• our practicals will use K40s and K80s.
Maxwell (compute capability 5.x):
• first released in 2014.
• an architecture for gaming, so poor DP.
Pascal (compute capability 6.x):
• first released in 2016.
• many gaming cards and several HPC cards.
Volta (compute capability 7.x):
• first released end 2017 / start 2018.
• only HPC cards, excellent DP.
Turing (compute capability 7.5):
• first released Q3 2018.
• this is the gaming version of Volta.
• some HPC cards (specifically for AI inference). Poor DP.

Outline

- GPU
- CUDA Introduction
  - What is CUDA
  - CUDA Programming Model
  - CUDA Library
  - Advantages & Limitations
- CUDA Programming

GPU

- GPUs are massively multithreaded many core chips
  - Hundreds of scalar processors
  - Tens of thousands of concurrent threads
  - 1 TFLOP peak performance
  - Fine-grained data-parallel computation

- Users across science & engineering disciplines are achieving tenfold and higher speedups on GPU

Outline

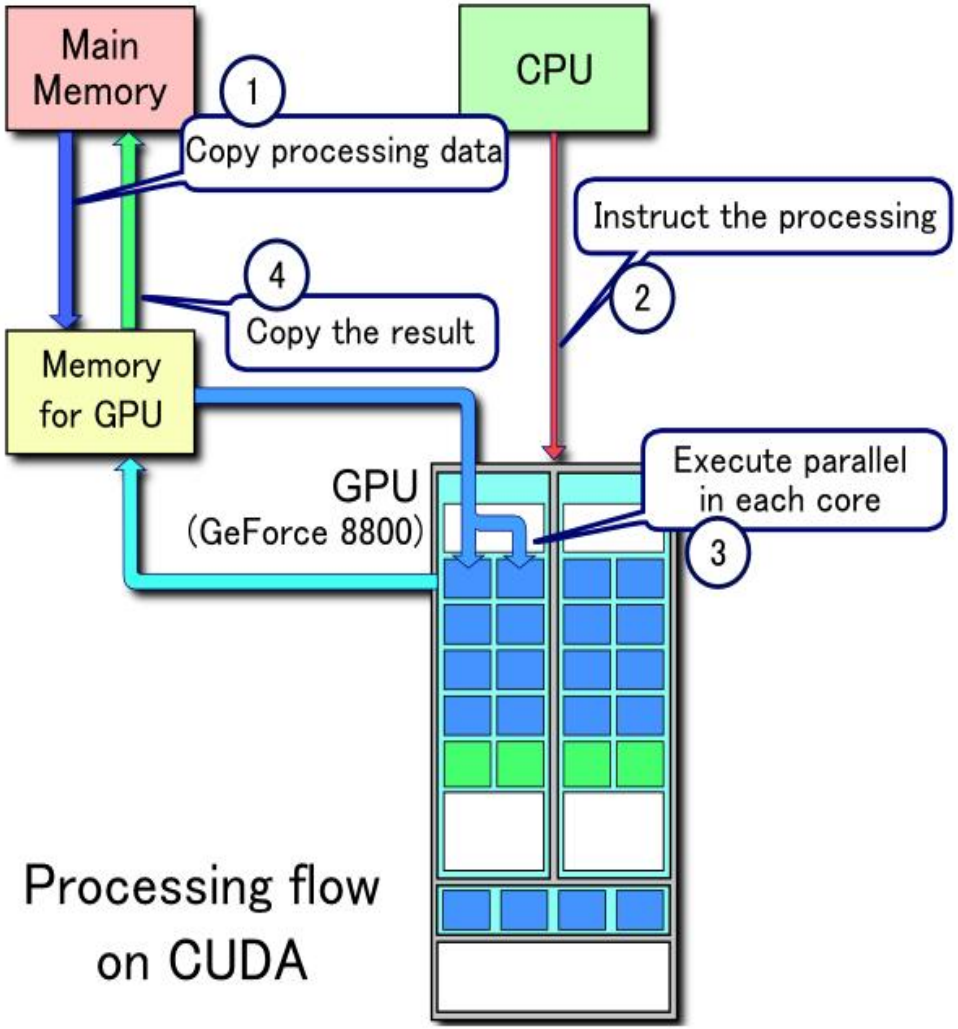
- GPU
- CUDA Introduction
  - What is CUDA
  - CUDA Programming Model
  - CUDA Library
  - Advantages & Limitations
- CUDA Programming
- Future Work

What is CUDA?

- **CUDA is the acronym for Compute Unified Device Architecture.**
  - **A parallel computing architecture developed by NVIDIA.**
  - **The computing engine in GPU.**
  - **CUDA can be accessible to software developers through industry standard programming languages.**
- **CUDA gives developers access to the instruction set and memory of the parallel computation elements in GPUs.**

Processing Flow

- **Processing Flow of CUDA:**
  - **Copy data from main mem to GPU mem.**
  - **CPU instructs the process to GPU.**
  - **GPU execute parallel in each core.**
  - **Copy the result from GPU mem to main mem.**

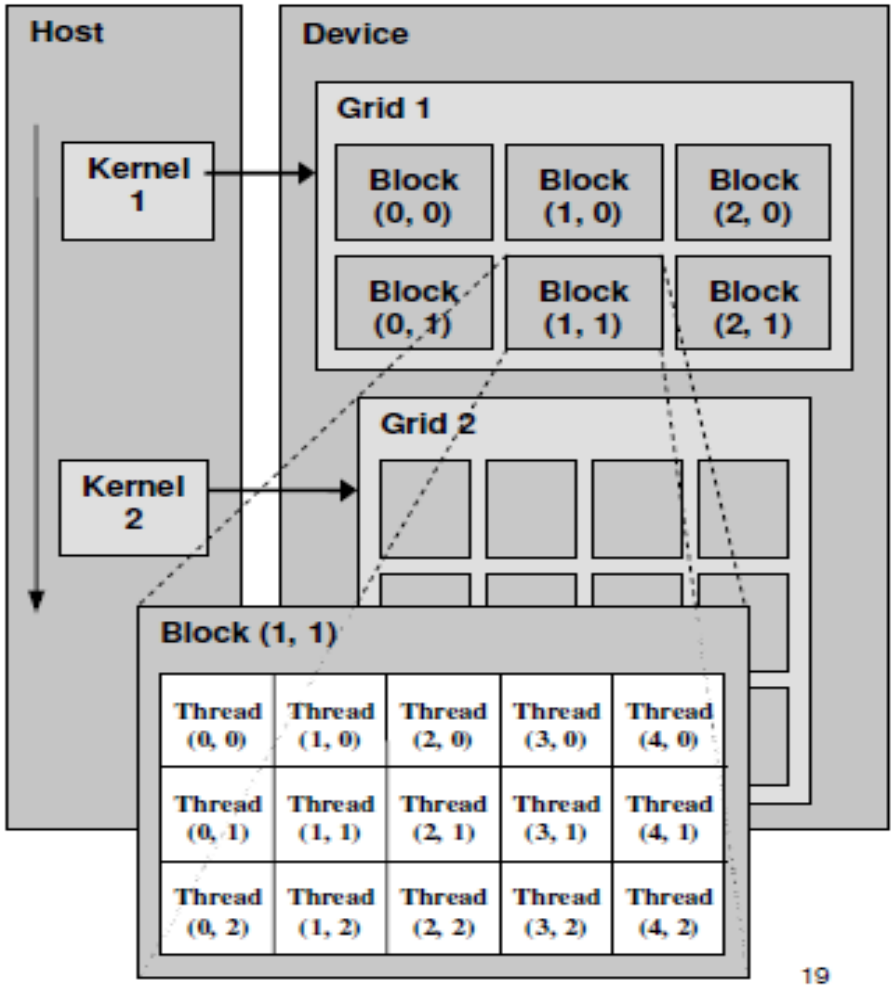


Outline

- GPU
- CUDA Introduction
  - What is CUDA
  - CUDA Programming Model
  - CUDA Library
  - Advantages & Limitations
- CUDA Programming
- Future Work

CUDA Programming Model

Device = GPU
Host = CPU
Kernel = function that runs on the device



CUDA Programming Model

A kernel is executed by a grid of thread blocks

- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution

- Threads from different blocks cannot cooperate

CUDA Kernels and Threads

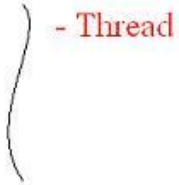
- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel

- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

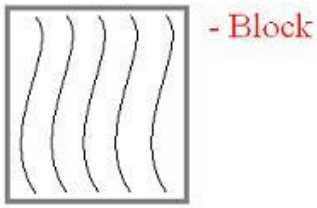
Thread Hierarchy

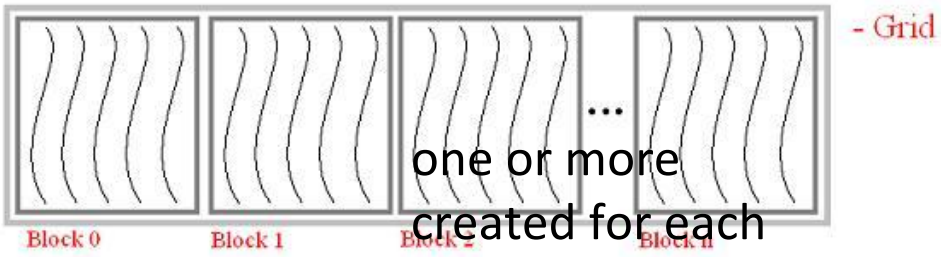
Thread – Distributed by the CUDA runtime (identified by threadIdx)

Warp – A scheduling unit of up to 32 threads



Block – A user defined group of 1 to 512 threads. (identified by blockIdx)



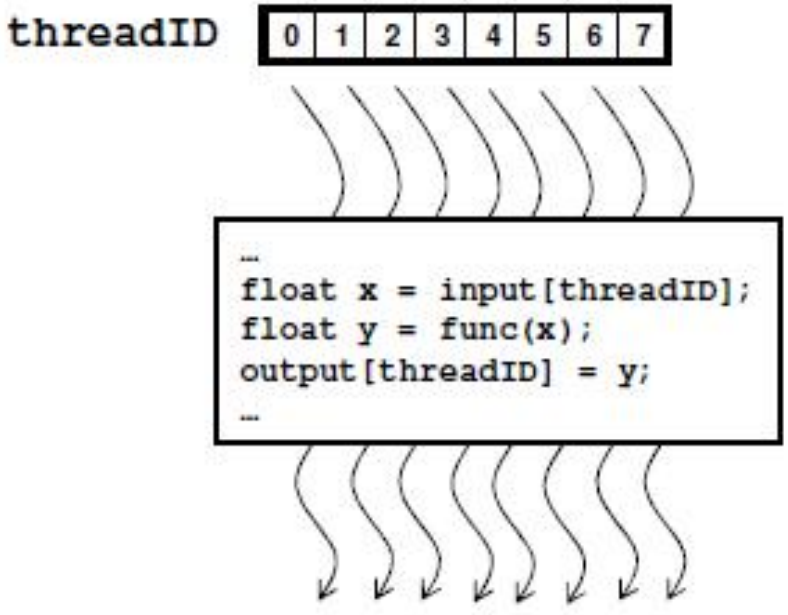


one or more created for each function

Grid – A group of blocks. A grid is CUDA kernel

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Minimal Kernels

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;
will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}

void main()
{
   .....
      increment_cpu(a, b, N);
}
```

CUDA program

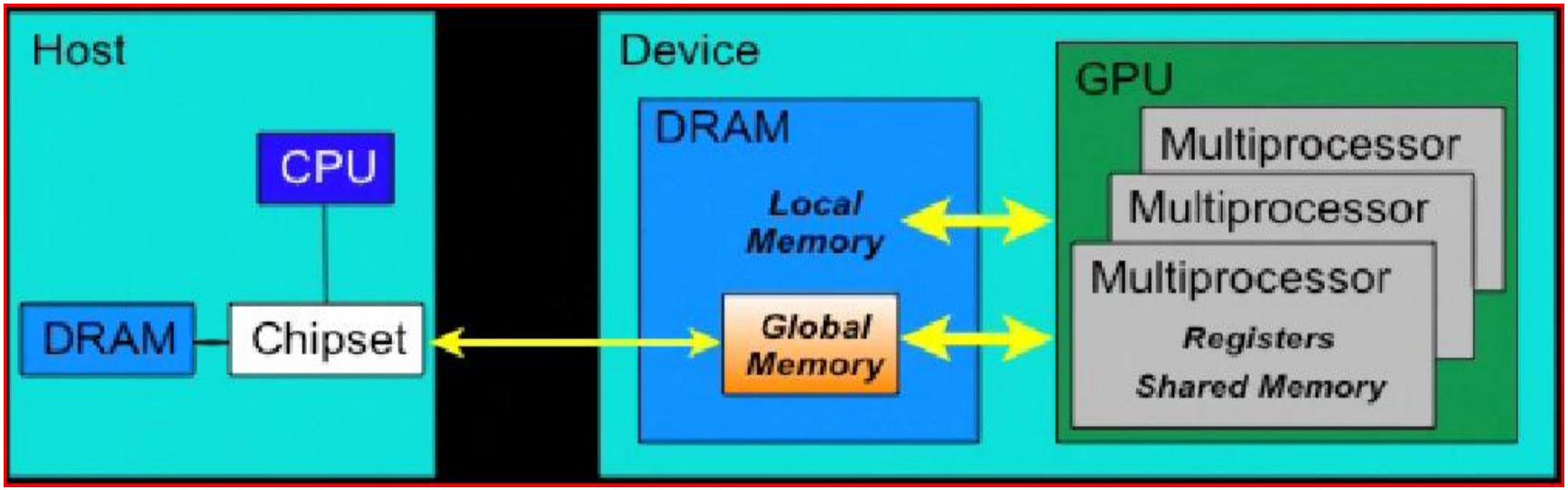
```
__global__ void increment_gpu(float *a, float b, int N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N)
          a[idx] = a[idx] + b;
}

void main()
{
   ...
      dim3 dimBlock (blocksize);
      dim3 dimGrid( ceil( N / (float)blocksize) );
      increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA

Manage memory



Moving Data…

CUDA allows us to copy data from one memory type to another.

This includes dereferencing pointers, even in the host's memory (main system RAM)

To facilitate this data movement CUDA provides **cudaMemcpy()**

```
cudaError_t cudaMemcpy ( void *                  dst,
                         const void *            src,
                         size_t                  count,
                         enum cudaMemcpyKind     kind
                       )
```

Parameters:

dst	- Destination memory address
src	- Source memory address
count	- Size in bytes to copy
kind	- Type of transfer

enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

cudaMemcpyHostToHost	Host -> Host.
cudaMemcpyHostToDevice	Host -> Device.
cudaMemcpyDeviceToHost	Device -> Host.
cudaMemcpyDeviceToDevice	Device -> Device.

Advantages of CUDA

- CUDA has several advantages over traditional general purpose computation on GPUs:
  - Scattered reads – code can read from arbitrary addresses in memory.
  - Shared memory - CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads.

Limitations of CUDA

- CUDA has several limitations over traditional general purpose computation on GPUs:
  - A single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.
  - The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.
  - CUDA-enabled GPUs are only available from NVIDIA.

Cuda Programming

- Cuda Specifications
  - Function Qualifiers
  - CUDA Built-in Device Variables
  - Variable Qualifiers
- Cuda Programming and Examples
  - Compile procedure
  - Examples

Function Qualifiers

- _global__ : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- __device__ : called from other GPU functions,
  - cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host
- __host__ and __device__ qualifiers can be combined
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
- dim3 gridDim;
  - Dimensions of the grid in blocks (at most 2D)
- dim3 blockDim;
  - Dimensions of the block in threads
- dim3 blockIdx;
  - Block index within the grid
- dim3 threadIdx;
  - Thread index within the block

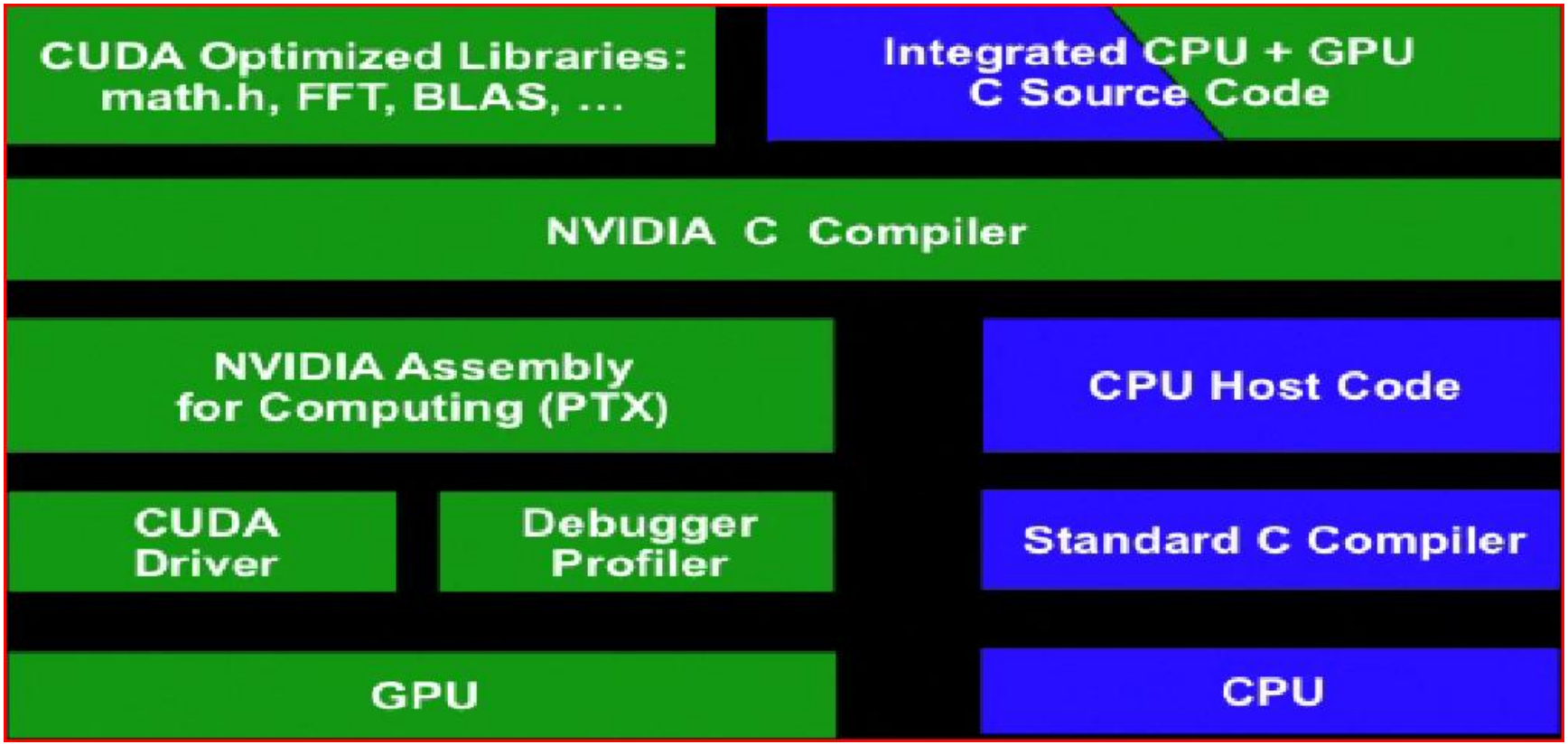
Variable Qualifiers (GPU code)

- __device__
  - Stored in device memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: kernel execution
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - Arrays of more than 4 elements stored in device memory

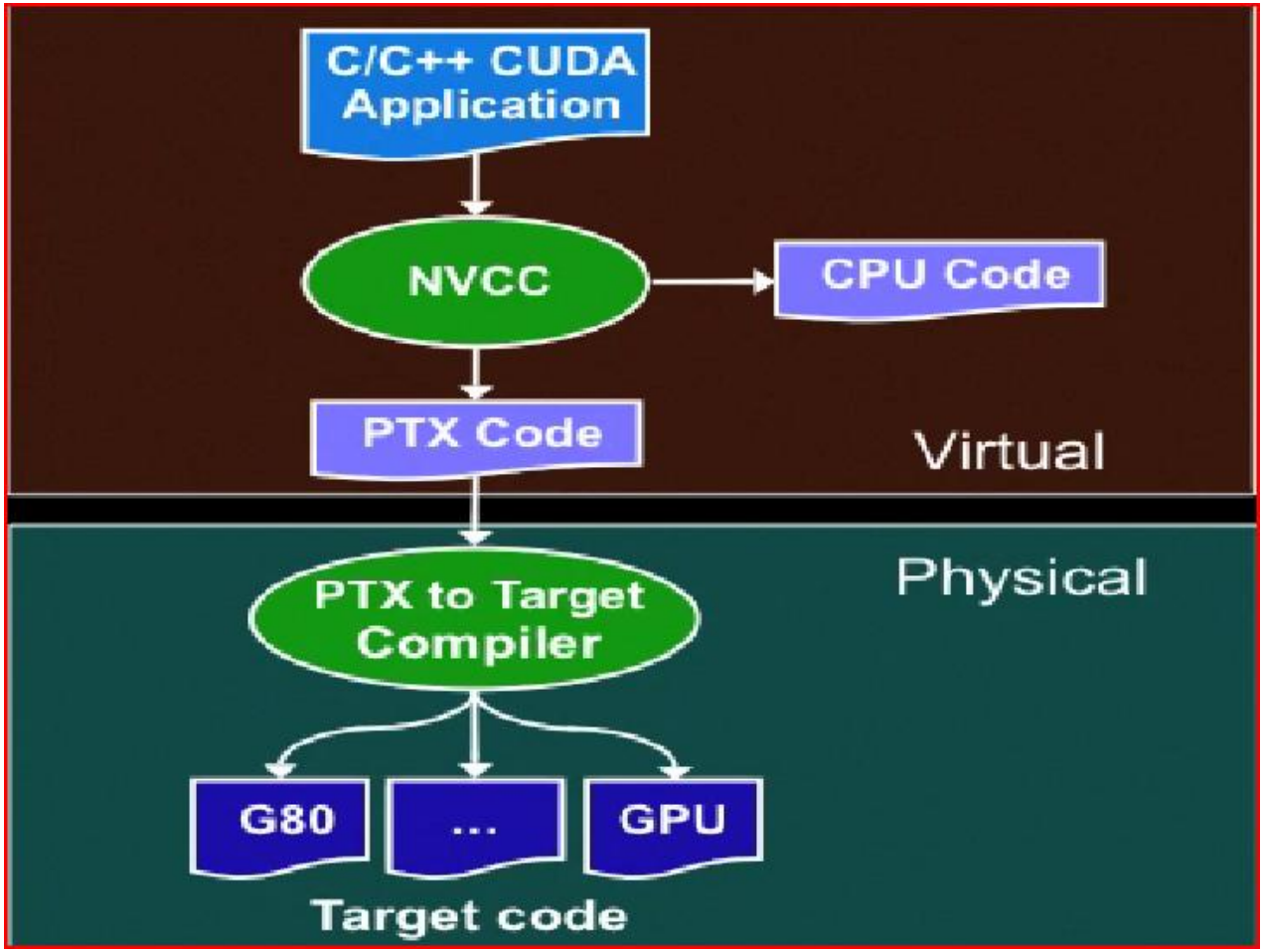
Cuda Programming

- **Kernels are C functions with some restrictions**
  - **Can only access GPU memory**
  - **Must have void return type**
  - **No variable number of arguments (“varargs”)**
  - **Not recursive**
  - **No static variables**
- **Function arguments automatically copied from CPUto GPU memory**

Cuda Compile



Cuda Compile_cont



Cuda Compile_cont

nvcc <filename>.cu [-o <executable>]

- Builds release mode

nvcc -g <filename>.cu

- Builds debug mode
- Can debug host code but not device code

nvcc -deviceemu <filename>.cu

- Builds device emulation mode
- All code runs on CPU, no debug symbols

nvcc -deviceemu -g <filename>.cu

- Builds debug device emulation mode
- All code runs on CPU, with debug symbols