Unit 5 : CUDA Architecture

Introduction to GPU and CUDA

1. Compute Unified Device Architecture is parallel computing architecture for NVIDIA.
2. This parallel computing platform which is implemented by GPU.
3. GPUs → **Parallel throughput architecture that emphasises executing many concurrent threads.**
4. GPUs for general purpose programming → GPGPU
5. CUDA → Virtual instructions set and memory of the parallel elements in CUDA GPUs.
6. NVIDIA developed CUDA → To let programmers write parallel programs using extension of C language.
7. CUDA → C, C++, Forton
8. C/C++ programs are compiled with nvcc, which is NVIDIA's compiler.

Benefits of CUDA:

1. Efficiently process thousands of elements in parallel.
2. Gives us a way that tasks can communicate and collaborate.
3. Good for lots of computation and lots of data.

CUDA Architecture Overview

1. Consists of C language extensions that execute on the compute device.
2. 2 Parts → Host and device

3. Host → Computer, machine, CPU running Windows, Linux, MacOS
4. Device → GPU, hardware that plugged into an onboard chip.
5. Fast single thread execution → CPU is optimised
6. High multi thread throughput → GPU

1. CPU
    a. Host → CPU and Memory
    b. Host code is executed by CPU
    c. Small data size tasks → CPU

d. 3 main parts :

    i. CUDA Libraries → Include BLAS, FFT and other functions for CUDA architecture.

    ii. CUDA Runtime → Its a higher level API implemented on top of driver API. Each function of runtime API is broken down into more basic operations and then issued to driver. Handles dynamic behavior

    iii. CUDA Driver → Low level API , provides more control on how the GPU device is used.

e. Expanded version includes

    a. Control Unit → Key component in HPC, usually called the core.

    b. ALU

    c. Cache

    d. DRAM → DRAM is commonly used as the main memory (RAM) in computers, where it stores data that is actively being used or processed by the CPU. This includes program instructions, application data, and the operating system itself.

2. GPU

    a. Highly parallel, multithreaded, many core processor with high computational power.

    b. Contains many processors.

    c. GPU → Viewed as compute device, co processor to CPU and it has its own DRAM.

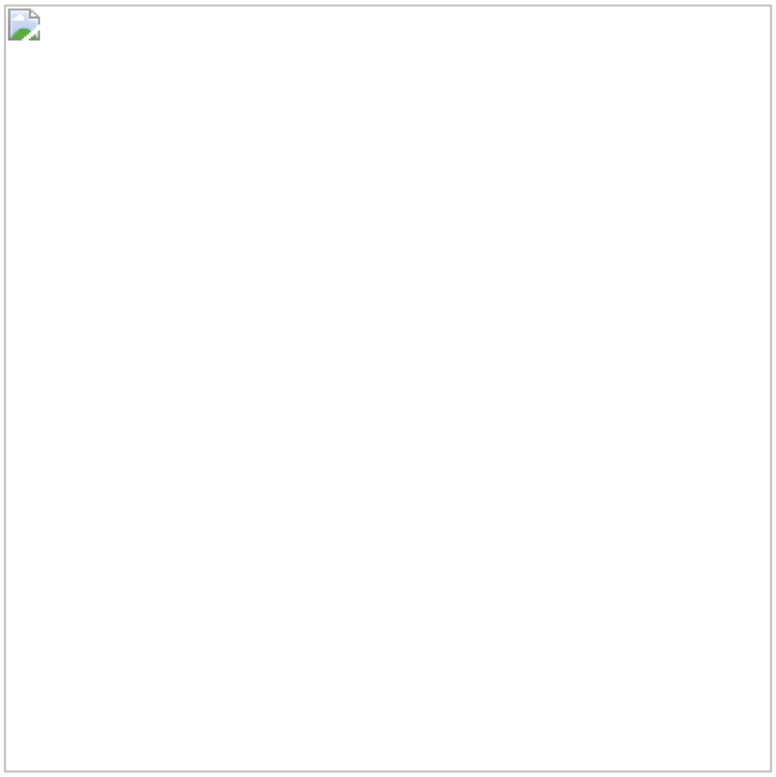
    d. Device → GPU and memory

    e. NVIDIA introduced a programming model called **Unified Memory, bridging the divide between host and device memory spaces. This allows access to both the CPU and GPU memory using a single pointer.**

CUDA Processing Flow

- **Data Transfer from Main Memory to GPU Memory:** The first step involves copying the data that needs to be processed by the GPU from the computer's main memory (also known as host memory) to the GPU's memory.
- **Instructing the Processing:** The CPU then sends instructions, or in CUDA terminology, a kernel, to the GPU. This kernel includes the program that the GPU will execute to process the data.
- **Executing the Kernel in Parallel:** The GPU receives the kernel and breaks it down into smaller tasks called threads. These threads are then grouped into grids and thread blocks. The GPU executes these threads in parallel on its streaming multiprocessors (SMs). This is where the power of CUDA comes from, as thousands of threads can be executed simultaneously, which can significantly accelerate certain tasks.

- **Transferring Results Back to Main Memory:** Once the GPU has finished processing the data, the results are copied back from the GPU's memory to the CPU's main memory.

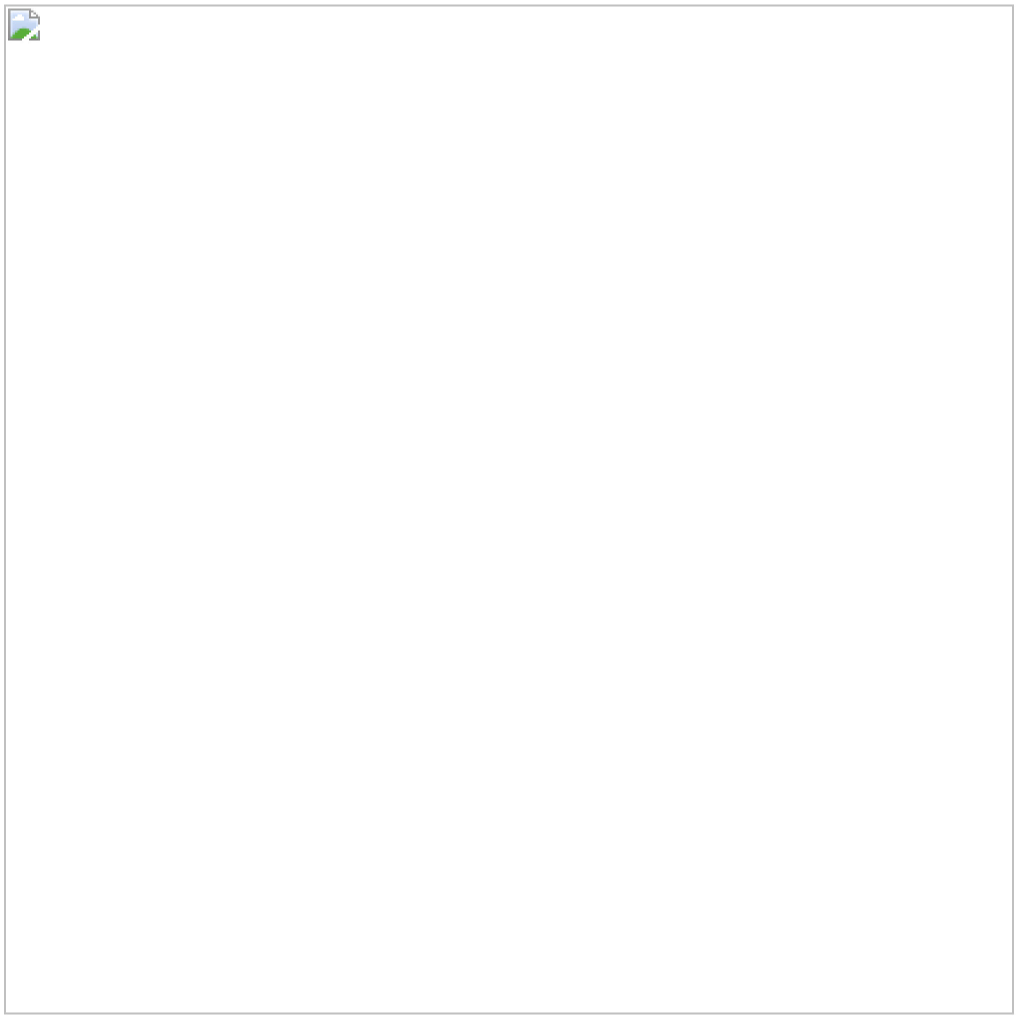


CUDA Programming Models for HPC Architecture

1. Boundary between program and model implementation is communication abstraction.
2. Compilers and libraries → Use system calls to access hardware services.
3. Abstraction layers depend on → Way to organise threads on GPU, Way to access memory on GPU

CUDA Programming Structure:

1. CUDA programming is a combination of serial and parallel executions.
2. Make an interaction between CPU and GPU models.



3. Serial execution → executes in a host thread.
4. Parallel execution → executes in concurrent threads across multiple elements.
5. CPU → Responsible for management of enviornment, code, and data for the device before loading compute intensive tasks.

Advantages , Limitation and Applications of CUDA

Advantages

1. Good for lots of computation and lots of data.
2. CUDA → High level languages like C to develop applications that can take advantage of high level of performance.
3. CUDA → Fast shared memory
4. Compiled code will run directly on GPU.

Limitations

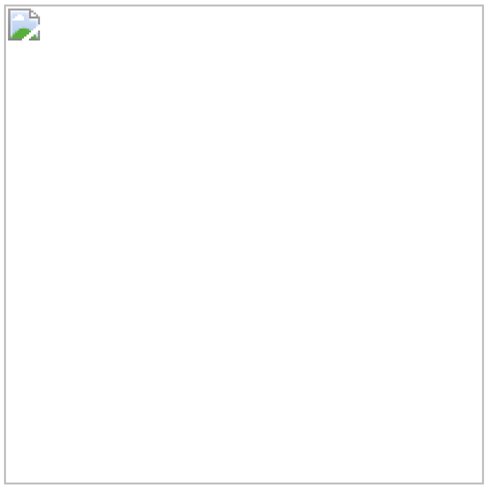
1. CUDA code only supported on NVIDIA hardware.
2. No support of recursive function.
3. Bus bandwidth and latency between GPU and CPU creates a bottleneck.

Applications

1. Computational fluid dynamics:
    a. NVIDIA's cuda architecture is used to accelerate computational fluid dynamics.
    b. These initial investigations indicate acceptable levels of performances using GPU powered, workstations.
    c. This helped in design and modeling of highly efficient rotors and blades thus effectively. modelling devices around complex movement of air and fluids.

Write and launch CUDA kernel and handling errors

1. CUDA contains host program (Program control) and device code (GPU) combined in single C program.
2. CUDA C code can run on any number of processors without the need of recompilation and you can map CUDA threads to GPU threads or to CPU vectors.



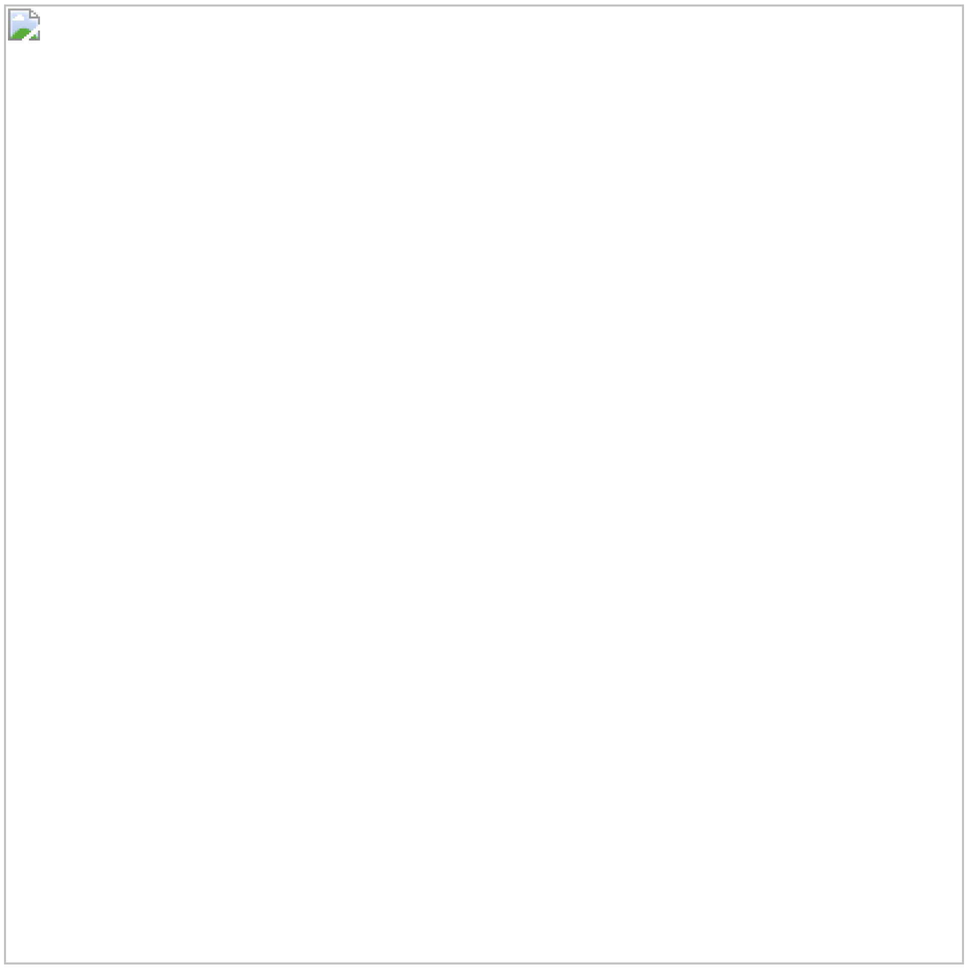
Program execution

First CUDA C program

Normal Hello world

CUDA hello world program

1. CUDA C adds the **global** qualifier to standard C. This alerts the compiler that a function should be compiled to run on the device. (GPU)
2. NVCC gives the function kernel() to the compiler and handles the device code.
3. Launching the kernel is done by Host. (CPU)
4. mykernel<<<1,1>>>();
5. syntax : kernelname<<<grid, block>>>(argument list);
6. The angle brackets make a call from host to device.



1. We can pass parameters to a kernel.
2. We need to allocate memory in the device.
3. cudaMalloc () → It tells CUDA runtime to allocate memory on the device.
4. void ** & dev_c → hold the address of the newly allocated memory
    a. &dev_c → address of the the device variable (It gives a pointer to dev_c, which holds the memory address where "dev_c" is stored.
    b. This part is a typecast that converts the address of the pointer `dev_c` to a pointer to a void pointer. The address of `dev_c` is passed by reference to `cudaMalloc`, so that `cudaMalloc` can update the value of `dev_c` to point to the allocated device memory.
5. To free memory allocated for cudaMalloc(), we need to call cudaFree().

6. cudaMemCpy() → accessing device pointers within device code.
7. cudaMemcpyDeviceToHost → Instructs the runtime that the source pointer is a device pointer and the destination pointer is a host pointer.
8. cudaMemcpyHostToDevice → Indicate that opposite situation, where source data is on the host and the destination is an address on the device.
9. cudaMemcpyDevicetoDevice → Both pointers are on the device.

CUDA kernels

CUDA kernels are functions written in CUDA C/C++ that are designed to be executed on the GPU. These kernels define the computation that will be performed in parallel by multiple threads on the GPU.

1. **Parallel Execution**: CUDA kernels are designed to be executed by multiple threads concurrently on the GPU. Each thread executes the same code but operates on different data elements. This enables parallel processing of data, which can lead to significant performance improvements compared to sequential execution on the CPU.
2. **Thread Hierarchy**: CUDA kernels are organized into a hierarchy of threads, blocks, and grids. Threads are the smallest unit of parallel execution and are organized into blocks. Blocks, in turn, are organized into a grid. The organization of threads, blocks, and grids allows for flexible control over parallel execution and facilitates efficient utilization of the GPU's computational resources.
3. **Thread Indexing**: Within a CUDA kernel, each thread is assigned a unique thread index, which can be used to determine the data elements that the thread will operate on. CUDA provides built-in variables such as `threadIdx.x`, `blockIdx.x`, and `blockDim.x` to access the thread index, block index, and block dimensions, respectively.
4. **Memory Access**: CUDA kernels can access both global memory (device memory) and shared memory (on-chip memory shared by threads within a block). Efficient memory access patterns are crucial for maximizing

performance, and CUDA provides mechanisms such as memory coalescing and shared memory usage to optimize memory access.

5. **Kernel Invocation**: CUDA kernels are invoked from the host (CPU) code using special syntax, typically using the `<<<...>>>` syntax to specify the grid and block dimensions. The host code launches the kernel, and the GPU executes the kernel code in parallel.
6. **Error Handling**: CUDA provides error-checking mechanisms to detect and handle errors that occur during kernel execution. Developers can use CUDA API functions like `cudaGetLastError()` and `cudaDeviceSynchronize()` to check for errors and synchronize the host and device execution, respectively.

1. Executed by an array of threads.
2. All threads run the same code, and each thread has an ID using to compute memory addresses.
3. kernel_function <<< num_blocks, num_threads >>>(param 1, param 2)

Basic terms:

1. Thread : More limited than java threads
2. Kernel : Code executed by a thread
3. Thread block : Group of threads that execute simultaneously on the same processor.
4. Warps : Set of 32 concurrent threads in a block.
5. Grid : Group of thread blocks, each block executing on different processor.
6. Stream : Grid in the process of executing a kernel.

Threads within a block cooperate via shared memory, can they can synchronise in different blocks.

Allow programs to transparently scale to different GPUs.

Manage CUDA memory model

1. Device pointers → Point to GPU memory
2. Host pointers → Point to CPU memory
3. CPU manages GPU memory

- **Host Memory (DRAM):** This is the main memory of the CPU. It is slower than device memory but has a much larger capacity.
- **Device Memory (Global Memory):** This is the main memory of the GPU. It is much faster than host memory but has a smaller capacity.
- **Registers:** These are very small and fast memories located within the Streaming Multiprocessors (SMs) of the GPU. Threads can only access data in their own register file.
- **Shared Memory:** This is a type of memory that can be shared by all the threads in a block. It is faster than global memory but has a limited capacity.

4. Host → Allocates or frees memory, copies data to and from the device, applies to global device memory (DRAM)
5. Host can read and write global memory but not shared memory.
6. Each thread → private local memory, each thread block → shared memory visible to threads of the block
7. Processor registers → Variables we declare in kernel will use registers unless we run out or they cant be stored in registers.
8. Local memory → not shared, read write
9. Shared memory → shared by all threads in the same block, enables fast communication between threads, read write
10. Global memory → Shared by all threads, main means of communicating read write between host and device.
11. Slower to access than other memories like shared and registers.
12. Constant memory → Only read, no write, sets values in constant memory before launching the kernel
13. Texture memory → Read by only GPU, CPU sets it up.
14. Has extra addressing tricks, uses CUDA APIs for dealing with device memory.

Communication between threads

1. Need to shared data between threads to compute the final result or to combine the output.
2. Communication of threads is possible due to → shared memory
3. Variables in shared memory → copy is created for each block of the variable launched on the GPU, and each thread shared that memory, but threads cannot see or modify the copy of the variable.
4. Latency is low and they reside physically on the GPU. This is why fast communication takes place and sometimes it uses global memory if it wants to

share large data among threads.

5. CUDA C makes a region of shared variables using keyword __ shared __
6. Synchronisation between threads → Mechanism for synchronising between threads is required.
7. Main use of synchronisation → prevent
    a. RAW Read after write
    b. WAR write after read
    c. WAW write after write
    d. **RAW (Read After Write)**:
        - This hazard occurs when a read operation depends on the outcome of a preceding write operation from another thread.
        - Example: Thread A writes a value to a shared memory location, and then Thread B reads from the same location before Thread A's write operation is complete.
        - To mitigate RAW hazards, synchronization mechanisms such as locks, mutexes, or barriers can be used to ensure that read operations occur only after write operations have completed.
    e. **WAR (Write After Read)**:
        - This hazard occurs when a write operation depends on the outcome of a preceding read operation from another thread.
        - Example: Thread A reads a value from a shared memory location, and then Thread B writes to the same location before Thread A's read operation is complete.
        - To mitigate WAR hazards, synchronization mechanisms can also be used to ensure that write operations occur only after read operations have completed.
    f. **WAW (Write After Write)**:
        - This hazard occurs when multiple write operations are performed concurrently by different threads on the same memory location without

proper synchronization.

- Example: Thread A writes a value to a shared memory location, and then Thread B writes a different value to the same location without any synchronization between the two operations.
- To mitigate WAW hazards, synchronization mechanisms are crucial to ensure that write operations are serialized or properly coordinated to avoid conflicting writes.

and synchronize to commit all memory writes and reads and computation.

8. To make sure the threads are synchronised, we used 2 types of synchronisation:
   a. Implicit → functions are implicitly synchronised, (1 or more threads must complete before proceeding to the next section)
   b. Barrier → Synchronisation barrier is explicitly added

Parallel Programming in CUDA C

1. dev_a and dev_b → inputs, dev_c → result
2. kernel <<< 2,1 >>>() → runtime creating 2 copies of the kernel and running them in parallel.
3. <<<256, 1>>> → 256 Blocks running on the GPU.
4. How to know within the code which block is currently running ? blockIdx.