KEY CONCEPTS IN GPU PROGRAMMING:

GPU programming has revolutionized parallel computing, enabling efficient execution of tasks through massive parallelism. In this report, we will delve into key concepts related to GPU programming, specifically focusing on the Kernel, CUDA Block, and Thread, which are fundamental elements of NVIDIA's CUDA (Compute Unified Device Architecture) platform.

Kernel, Thread Block (CUDA Block in NVIDIA GPUs), Warp and Threads:

A group of **threads** is called a CUDA block. **CUDA blocks** are grouped into a grid. A **kernel** is executed as a **grid** of blocks of threads.

- A **Kernel** is a function that runs on the GPU and represents a single instance of a computational task. Kernels are written in CUDA C/C++ and are launched by the CPU. They are designed to be executed in parallel by multiple threads on the GPU. Kernels operate on a data set, where each thread processes a distinct portion of the data. The kernel function is launched with a certain number of threads, and each thread executes a portion of the kernel code. CUDA kernels are designed to work in parallel, which means that each thread can execute the same code with different data. For example, a kernel could be used to perform matrix multiplication, image processing, or simulations.

- A **CUDA block** (also known as a thread block) is a grouping of threads that execute a kernel in parallel. Threads within a block can collaborate and share data using **shared memory**, which is much faster than global memory. The number of threads within a block is determined by the developer but is usually a multiple of 32, as this aligns with the GPU's hardware architecture. Blocks are scheduled to run on multiprocessors within the GPU.

- A **Thread** is the smallest unit of execution within a GPU. Threads within a block run the same kernel code, but each thread can be assigned a unique identifier to determine its data access and processing tasks. Threads within a block can communicate through shared memory, making them suited for collaborative tasks. Threads are organized into warps, where each warp consists of 32 threads that execute in lockstep, executing the same instruction simultaneously.

- A **Warp** is a fixed-size group of threads, typically consisting of 32 threads on most modern GPUs. Threads within a warp execute the same instruction simultaneously, enabling efficient SIMD (Single Instruction, Multiple Data) processing. Different warps can execute independently, allowing for instruction-level parallelism.

Process:

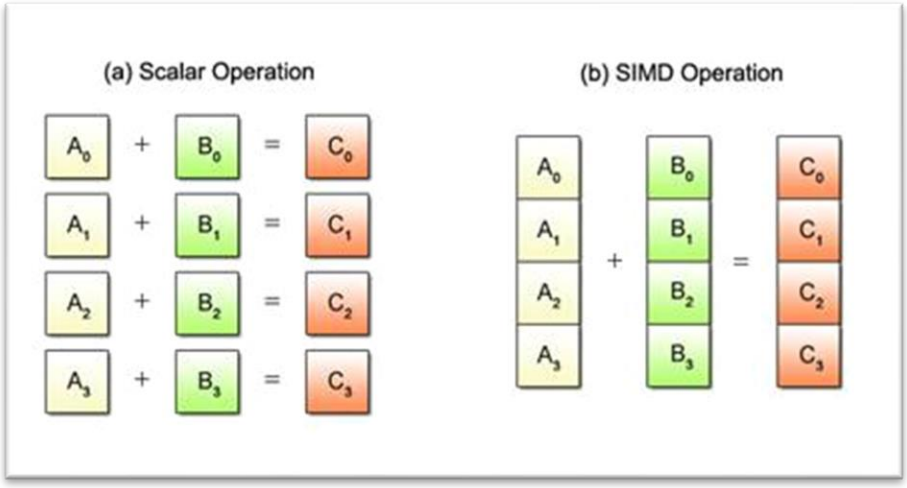
A CUDA program starts by creating a CUDA context, either explicitly using the driver API or implicitly using the runtime API, for a specific GPU. The context encapsulates all the hardware resources necessary for the program to be able to manage memory and launch work on that GPU.

In a typical CUDA program, data are first sent from main memory to the GPU memory, then the CPU sends instructions to the GPU, then the GPU schedules and executes the kernel on the available parallel hardware, and finally results are copied back from the GPU memory to the CPU memory.

SIMD:

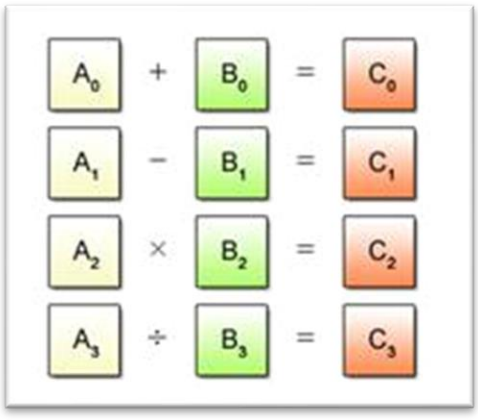
SIMD is short for Single Instruction/Multiple Data, while the term SIMD operations refers to a computing method that enables processing of multiple data with a single instruction. In contrast, the conventional sequential approach using one instruction to process each individual data is called scalar operations.

Using a simple summation as an example, the difference between the scalar and SIMD operations is illustrated below.



With conventional scalar operations, four add instructions must be executed one after another to obtain the sums as shown in above figure. Meanwhile, SIMD uses only one add instruction to achieve the same result, as shown in Figure. Requiring fewer instructions to process a given mass of data, SIMD operations yield higher efficiency than scalar operations.

Despite the advantage of being able to process multiple data per instruction, SIMD operations can only be applied to certain predefined processing patterns. SIMD operations cannot be used to process multiple data in different ways. A typical example is given in the below figure where some data is to be added and other data is to be deducted, multiplied or divided.



Example:

SIMD is primarily geared towards graphics applications and physics calculations that require simple, repetitive calculations of enormous amounts of data. Capable of processing multiple data with a single instruction, SIMD operations are widely used for 3D graphics and audio/video processing in multimedia applications.

GPU Memory Hierarchy - SRAM/ DRAM, Shared memory, Constant memory:

Efficient memory management is crucial in optimizing performance in GPU computing.

- SRAM (Static Random-Access Memory):

  SRAM is a type of high-speed, low-capacity memory that provides extremely fast access times. It is typically used as cache memory in GPUs to store frequently accessed data and instructions. SRAM is more expensive to manufacture than DRAM and has lower storage capacity but offers **lower latency** and **higher bandwidth**. Due to its high-speed nature, SRAM is ideal for storing critical data that requires rapid access.

  Example: Program counters and loop variables.

- DRAM (Dynamic Random-Access Memory):

  DRAM is a type of memory that offers higher storage capacity compared to SRAM but with slightly **higher latency**. It is used as the main memory in GPUs. DRAM is slower than SRAM but provides a larger pool of memory for storing a broader range of data. It is critical for executing large-scale parallel computations and storing data structures.

  Example: Program code, data, and intermediate results.

- Shared Memory:

  Shared memory is a special type of memory that is physically located on the GPU chip and is shared among threads within the same thread block. It offers **low-latency, high-bandwidth access**, making it ideal for communication and data sharing between threads within a block. Threads can cooperate and exchange data through shared memory, allowing for efficient synchronization and parallel processing.

  Example: Shared memory is frequently used for storing intermediate results and facilitating data exchange in parallel algorithms.

- Constant Memory:

  Constant memory is a read-only memory space optimized for storing data that remains constant throughout the execution of a kernel. It offers faster access than global memory but has limitations in terms of storage capacity. By using constant memory, developers can reduce memory access latency for frequently used constants, improving overall performance.

  Example: Constants, lookup tables, and other read-only data that is shared among multiple threads in a block

Scheduler:

The GPU scheduler is a crucial component in modern graphics processing units (GPUs) that plays a vital role in managing the allocation of hardware resources and orchestrating the execution of tasks.
The functionalities of a GPU Scheduler include Task Management, Resource Allocation, Pipelining and Overlapping and Context switching.

The GPU scheduler is responsible for managing and scheduling tasks, often represented as kernels, to be executed on the GPU. These tasks are launched by the CPU and are executed in parallel by multiple threads on the GPU. The scheduler ensures fair distribution of resources among various tasks, preventing any single task from monopolizing the GPU's processing power.

The scheduler allocates hardware resources, such as Streaming Multiprocessors (SMs) and memory, to executing tasks. It makes decisions about how to partition available resources among the active tasks to maximize parallelism and performance

Modern GPUs utilize pipelining to keep multiple tasks in different stages of execution simultaneously. The scheduler ensures that tasks are scheduled in such a way that different stages of multiple tasks can overlap, effectively hiding memory latency and keeping the GPU's execution units busy.

Context switching involves switching the GPU's execution context from one task to another. The scheduler handles context switching efficiently, saving and restoring the necessary context information for each task. This allows the GPU to seamlessly switch between different tasks without disrupting overall performance.

Example: Gaming Graphics, Scientific Computing and Deep learning.

BY
MATTA ROHIT
21CS01022
COMPUTER ORGANIZATION AND ARCHITECTURE
COMPUTER SCIENCE AND ENGINEERING
IIT BHUBANESWAR