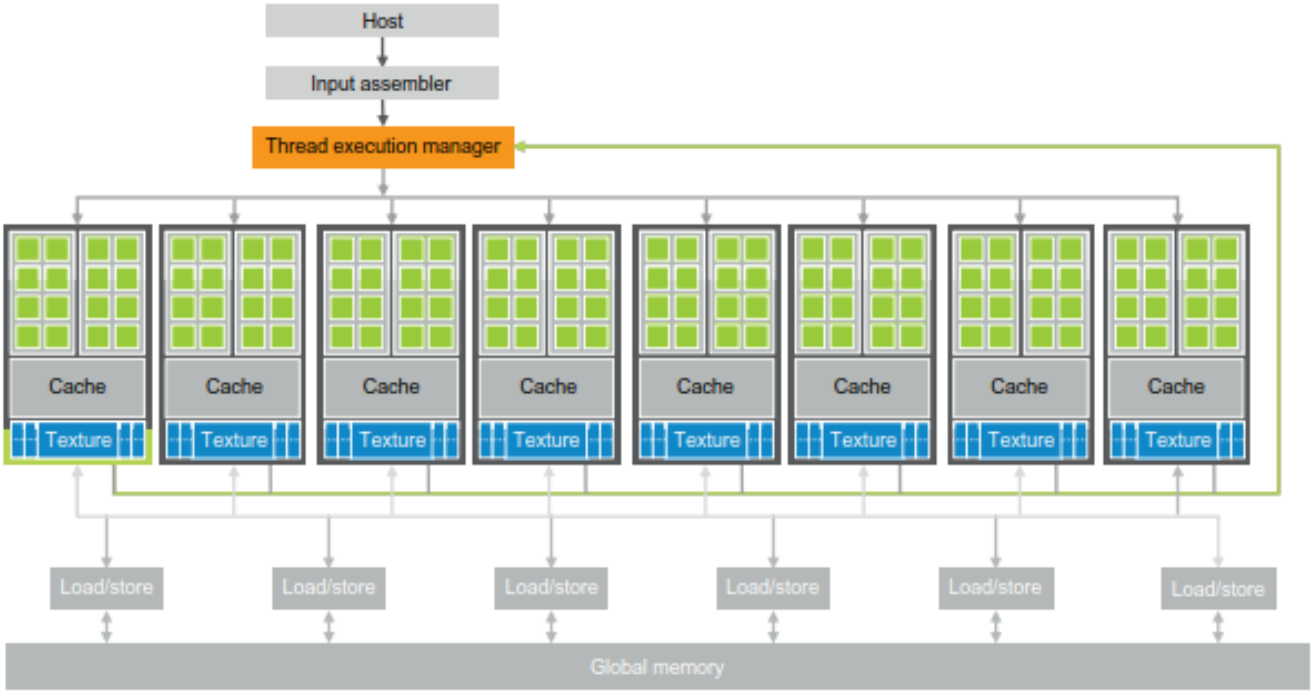
GPU Series – II

WHAT IS CUDA!



Architecture of CUDA Enabled GPU

(Img Src: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu)

CUDA (Compute Unified Device Architecture) is a parallel computing platform created by NVIDIA. It allows developers to use NVIDIA GPUs for general-purpose processing, known as GPGPU (General-Purpose computing on Graphics Processing Units).

CUDA Program structure is as follows:

- Allocate memory on the GPU for the data required by the computation.
- Transfer data from the CPU memory to the GPU memory.
- Launch a kernel, a function written in CUDA that runs on the GPU, specifying the number of threads and blocks.
- Transfer the results from the GPU memory back to the CPU memory.
- Free the allocated memory on the GPU.

What is GPGPU? Why do we need it?

PGPU (General-Purpose computing on Graphics Processing Units) is the use of a GPU, which usually handles only computation for graphics rendering, to perform computation in applications traditionally handled by the CPU. This is useful for tasks that can be parallelized, allowing significant speed-ups in computation-intensive applications like scientific simulations, machine learning, and data processing.

How is CUDA Programming Different from Regular programming for CPU and GPU?

Aspect	Regular CPU Programming	CUDA Programming
Parallel Execution Model	Optimized for low-latency sequential processing; few powerful cores.	Optimized for high-throughput parallel processing; thousands of simpler cores.
Architecture	Few powerful cores for complex tasks with sophisticated control logic and large caches.	Hundreds to thousands of simpler cores for high parallelism.

Memory Management	Managed by the OS with complex memory hierarchies and large multi-level caches (L1, L2, sometimes L3).	Requires explicit management with various memory types (global, shared, constant, texture).
Programming Model	Uses general-purpose languages like C, C++, Python, with libraries optimized for sequential or moderate parallel tasks.	Extends C/C++ with specific keywords and functions for parallel execution on the GPU.
Execution Flow	Sequential execution flow with occasional branching and multi-threading.	Involves launching kernels that run on the GPU, organized in grids of thread blocks.
Parallelism Focus	Instruction-level parallelism (ILP) and simultaneous multithreading (SMT).	Data-level parallelism (DLP) and thread-level parallelism (TLP).
Control Unit	Sophisticated control units with branch prediction, out-of-order execution, and advanced instruction pipelining.	Simpler control logic optimized for parallel processing rather than complex decision-making.
Cache System	Large multi-level caches to minimize memory access latency.	Smaller specialized caches for managing massive data throughput.
Execution Model	Optimized for low-latency execution, handling fewer tasks quickly.	Optimized for high throughput, capable of executing many tasks concurrently.
Memory Access	Complex memory hierarchy to ensure fast access to data.	High-bandwidth memory systems designed to handle large volumes of data efficiently.

What is the difference between CUDA enabled GPU and GPU without CUDA?

Aspect	**CUDA Enabled GPU**	**GPU Without CUDA**
General Purpose Computing	Supports GPGPU (General-Purpose computing on Graphics Processing Units) via CUDA API.	Primarily focused on traditional graphics rendering.
Programming Model	Uses CUDA programming model with extensions to C/C++ for parallel processing.	Limited to graphics APIs like OpenGL or Direct3D; no support for general-purpose computing.
Parallel Computing	Designed for high-throughput parallel computing with extensive support for parallel algorithms.	Primarily optimized for graphics parallelism, not general-purpose parallel tasks.
Development Tools	Provides CUDA Toolkit with development tools, libraries, and debugging support.	Lacks development tools for general-purpose parallel programming.
Memory Management	Explicit control over memory allocation and transfer between CPU and GPU.	Memory management is primarily automated and focused on graphics tasks.
Compute Capability	Can execute complex compute tasks like scientific simulations, machine learning, and data processing.	Limited to graphics-related computations; general-purpose tasks are inefficient or unsupported.
Execution Flexibility	Supports custom kernel functions to run directly on the GPU.	Restricted to shader programs and fixed-function pipelines for graphics rendering.
Use Cases	Widely used in HPC (High-Performance Computing), AI, scientific research, and real-time data processing.	Primarily used in rendering graphics for games, video playback, and 3D applications.

To Summarize the difference:

CUDA Enabled GPU:

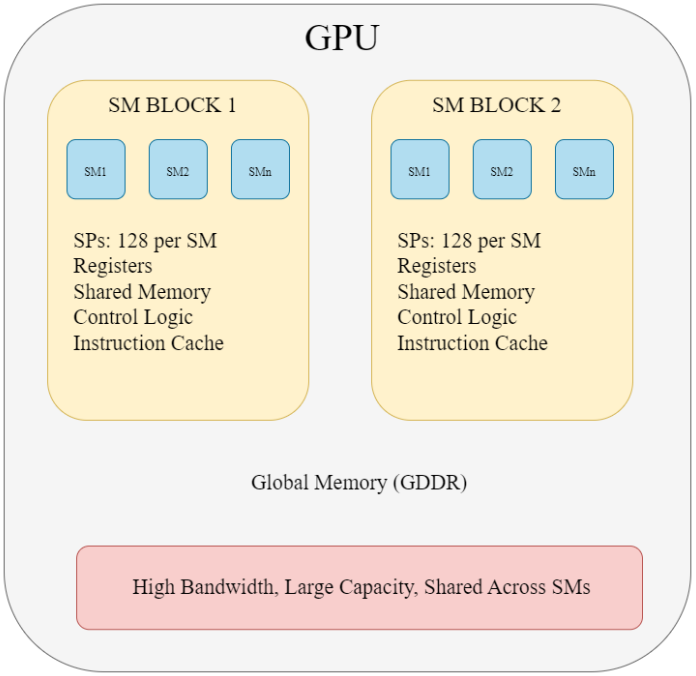
- Supports general-purpose computing, enabling developers to write custom parallel algorithms using CUDA.
- Provides a rich set of development tools and libraries for debugging, profiling, and optimizing parallel applications.
- Requires explicit management of threads and memory, giving developers fine-grained control over the execution.

GPU Without CUDA:

- Limited to graphics rendering tasks, with no support for general-purpose parallel computing.
- Uses standard graphics APIs, which abstract much of the parallelism and memory management.
- Development focuses on optimizing graphical performance, with less flexibility for general-purpose computation.

The **Primary Components of CUDA** architecture include Streaming Multiprocessors (SMs), Streaming Processors (SPs), and various types of memory, including GDDR (Graphics Double Data Rate) memory.

Streaming Multiprocessors (SMs): SMs are the primary computational units in a CUDA-capable GPU, responsible for managing and executing thousands of threads concurrently. Each SM contains multiple Streaming Processors (SPs), shared memory, control logic, and instruction cache. The exact number of SMs per GPU and SPs per SM can vary across different GPU generations.



Streaming Processors (SPs): SPs are the individual cores within an SM that execute the actual instructions of a CUDA kernel. They are simpler than CPU cores and are optimized for executing a high number of threads in parallel. Each SP handles arithmetic and logical operations and works in conjunction with other SPs within the same SM to perform large-scale parallel computations.

Control Logic and Instruction Cache: Control Logic Manages the execution of threads within the SM, including scheduling and coordinating the operations of the SPs. Instruction Cache Stores instructions to be executed by the SPs, enabling efficient instruction fetch and reducing latency.

Memory Hierarchy

- **Global Memory (GDDR SDRAM):** The main memory of the GPU, which is accessible by all SMs and is used for storing data that needs to be processed. GDDR memory is designed for high bandwidth to handle large volumes of data, which is essential for tasks like graphics rendering and scientific simulations.
- **Shared Memory:** A smaller, faster memory located within each SM, used for inter-thread communication and temporary storage. Shared memory allows threads within the same block to quickly share data without going through the slower global memory.
- **Registers:** Fast, small-sized storage within each SP for holding temporary variables and intermediate results during computation.
- **Constant and Texture Memory:** Read-only memory spaces optimized for specific access patterns, such as repeated reading of constants and texture sampling.

Now, how does the GDDR work?

GDDR (Graphics Double Data Rate) SDRAM is essential in storing video images and texture information, which are crucial for 3D rendering. GDDR memory holds data like color, depth, stencil, and accumulation data required for rendering scenes in graphics applications.

- **Color Data:** Information about the color of each pixel.
- **Depth Data:** Information about the distance of each pixel from the viewer, used for 3D rendering.
- **Stencil Data:** Used for masking certain parts of the rendering process.
- **Accumulation Data:** Used for effects like motion blur.

GDDR memory's high bandwidth is crucial for transferring large amounts of data quickly, ensuring smooth and realistic rendering of video and graphics. The frame buffer, a section of GDDR memory, stores the image data currently being displayed or about to be displayed on the screen.

In addition to its role in graphics rendering, GDDR memory is also critical in massively parallel applications, where it supports the high data throughput required for efficient parallel processing.

Bandwidth: Refers to the amount of data that can be transferred per unit time. High bandwidth is critical for efficiently handling large datasets common in parallel computing tasks.

Latency: The time taken to initiate a data transfer. In massively parallel applications, high latency can be mitigated by high bandwidth.

How does high bandwidth compensate for longer latency?

In massively parallel applications, many data transfers can occur simultaneously. High bandwidth allows these parallel transfers to proceed quickly, reducing the impact of latency. By maximizing the data throughput, the overall performance of the application can be enhanced despite individual data transfer latencies.

GDDR memory plays a crucial role not only in graphics rendering but also in supporting the high data throughput necessary for massively parallel applications. Its high bandwidth allows for efficient handling of large datasets and compensates for longer latencies by enabling multiple parallel data transfers, thereby enhancing overall application performance.

~~ To be Continued ~~

Reference: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu