# Module 1 ( Introduction )

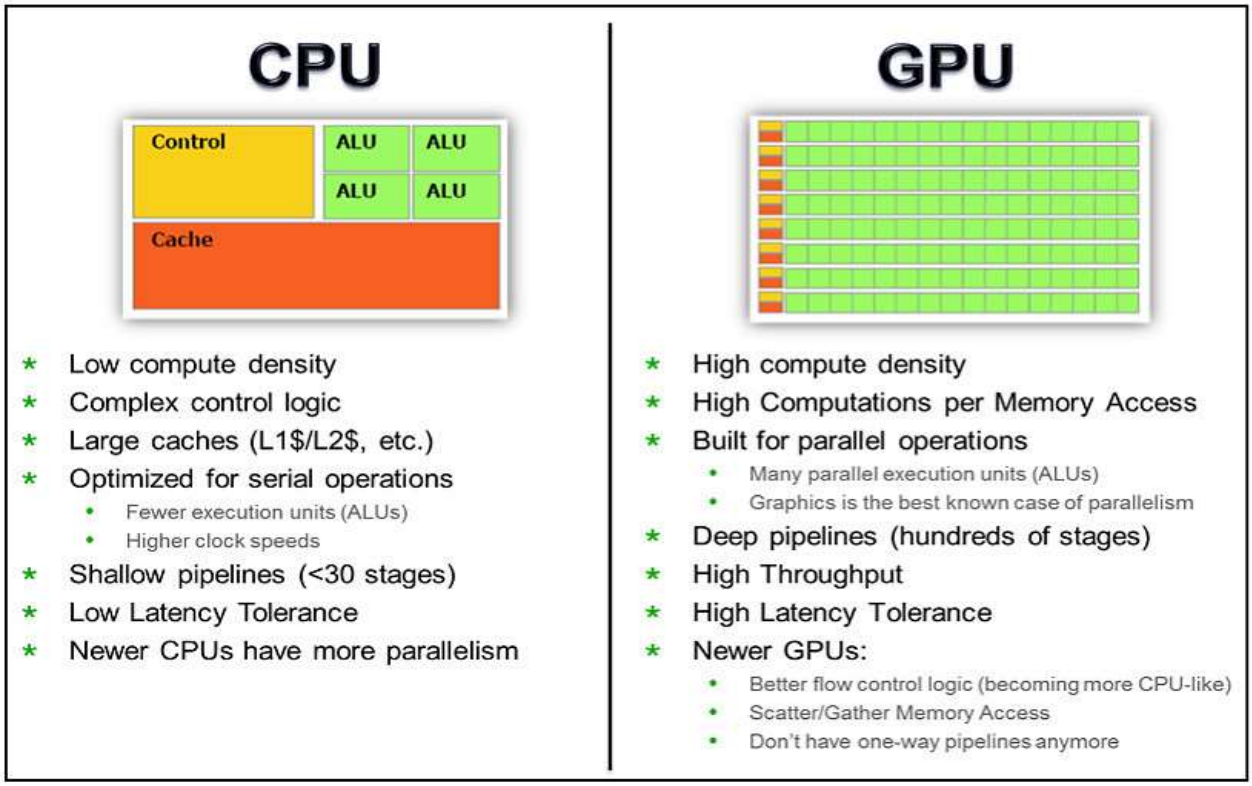
**Syllabus**

Introduction - GPUs as Parallel Computers- Architecture of a Modern GPU- Why More Speed or Parallelism? - Parallel Programming Languages and Models.

History of GPU Computing- Evolution of Graphics Pipelines - The Era of Fixed-Function Graphics Pipelines- Evolution of Programmable Real-Time Graphics- Unified Graphics and Computing Processors- GPU Computing- Scalable GPUs

## GPUs as Parallel Computers

Graphics Processing Units (GPUs) are specialized hardware designed to accelerate the processing of images and videos, but their capabilities extend far beyond that, making them powerful tools for parallel computation. Over the past decade, GPUs have evolved into general-purpose parallel processors capable of executing a wide range of computational tasks, from scientific simulations to machine learning and data analysis.




**Features**

**Many Cores:**

Unlike CPUs with a smaller number of powerful cores, GPUs have a large number of smaller cores, allowing them to execute many calculations concurrently.

**Data Parallelism:**

GPUs are optimized for data parallelism, where the same operation is performed on large amounts of data simultaneously across multiple cores.

**CUDA Programming:**

To harness the parallel processing power of a GPU, developers typically use a programming framework like NVIDIA CUDA, which allows them to write code that efficiently utilizes the GPU's architecture.

**Applications of GPU parallel computing.**

**Graphics rendering:**

The primary function of a GPU, where complex 3D graphics are rendered quickly by distributing calculations across many cores.

**Scientific computing:**

Simulations, data analysis, and complex mathematical calculations can be significantly accelerated with GPUs due to their parallel processing capabilities.

**Machine learning:**

Training large neural networks, a key component of AI, heavily benefits from the parallel processing power of GPUs.

## Architecture of a Modern GPU

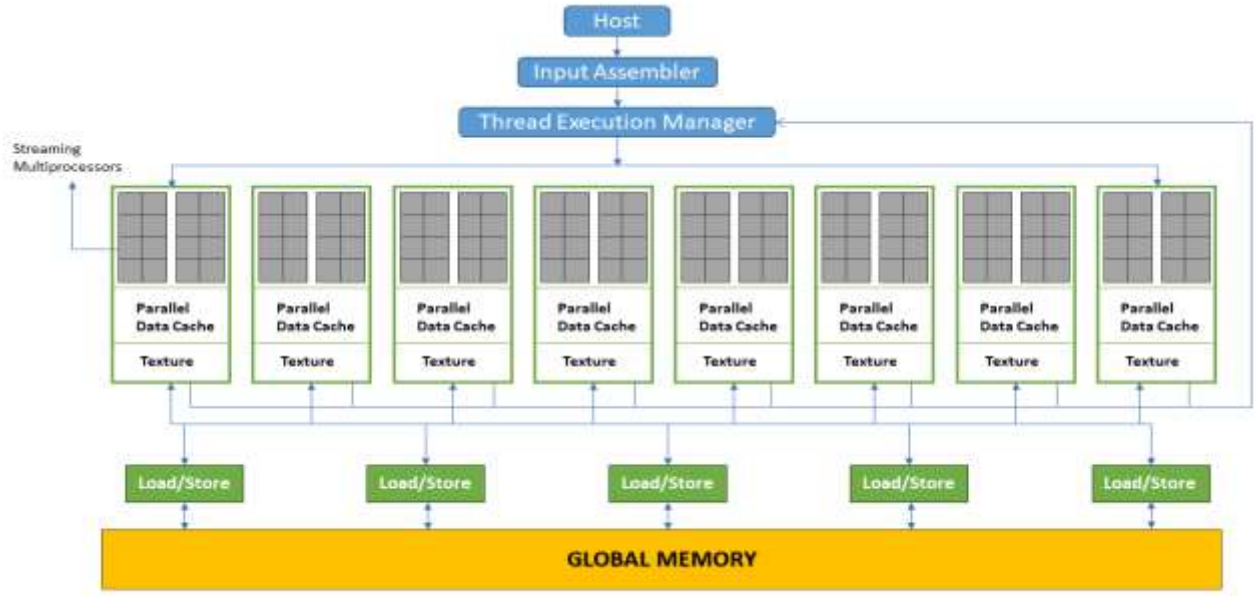



Figure shows the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). two SMs form a building block. However, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. Each SM has a number of streaming processors (SPs) that share control logic and an instruction cache. Each GPU currently comes with multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM, referred to as global memory.

**Stream Processors**

Responsible for executing instructions in parallel, making GPUs highly efficient at tasks that can be split into many smaller parts.

**Parallel Processing**

**SIMD (Single Instruction, Multiple Data):** GPUs are designed with a SIMD architecture, meaning that they can execute the same operation on multiple pieces of data simultaneously. This is particularly useful in tasks like rendering graphics, where the same set of operations (e.g., shading, transformations) needs to be applied to many pixels or vertices.

**SIMT (Single Instruction, Multiple Threads):** NVIDIA's CUDA architecture uses SIMT, which is an extension of SIMD. In SIMT, threads are grouped together in warps (groups of threads), and all threads in a warp execute the same instruction simultaneously but operate on different data.

**Memory Hierarchy**

- **Global Memory:** This is the main memory of the GPU, shared across all cores, but it has high latency and lower bandwidth compared to other levels of memory.

- **Shared Memory:** Shared memory is a small, fast memory space shared by threads within a block. It allows for high-speed data exchange and is crucial for optimizing parallel algorithms. Shared memory is much faster than global memory but is limited in size.
- **Registers:** Registers are the fastest form of memory available to individual threads. They are small in size but are used to store temporary data that threads need to work on. Since registers are very fast, efficient use of them is key to high GPU performance.
- **L2 Cache:** The L2 cache is a shared cache that is available to all cores and helps reduce the latency of accessing global memory.
- **VRAM (Video RAM):** VRAM is a type of memory used specifically in GPUs for storing textures, shaders, and other data related to rendering. It is optimized for high throughput in rendering tasks.

## Why More Speed or Parallelism?

Parallelism is generally preferred over simply increasing speed because it allows you to tackle complex problems by distributing the workload across multiple processing units, potentially leading to significantly faster results compared to just boosting a single core's speed, especially when dealing with large datasets or computationally intensive tasks.

- HDTV vs NTSC
- iPhone vs other smart phones
- Car Gaming

As technology improves, we need more computing power for things like HD TVs, video processing, and gaming. For example, HD TVs offer much better detail than older ones, and future TVs will need even more processing for features like high-resolution videos and 3D effects. Better computing also leads to smoother and more natural user interfaces, like the iPhone's touchscreen. In gaming, more computing power allows for realistic effects, such as cars getting damaged in accidents, instead of just pre-set scenes. These improvements in graphics and simulations require a lot of computing power.

New applications often involve simulating complex systems with massive amounts of data, leading to the rise of "big data." To handle this data, computations can be done in parallel across different parts of the data, though they need to be combined later. Effective data management is crucial for speeding up parallel applications. While experts know how to manage this, most developers could benefit from a more intuitive understanding of these techniques. The goal is to teach data management in a simple way, with practical examples and exercises. CUDA provides a programming model that supports parallel implementation and efficient data management, and it has been widely tested by developers.

## Parallel Programming Languages

1. **OpenMP (Open Multi-Processing)**:
    - **Language**: C, C++, Fortran
    - **Description**: OpenMP is a set of compiler directives, library routines, and environment variables that allow the easy development of parallel programs in shared memory systems. It provides an API for parallel programming on multicore processors.

- **Key Features**:
  - Directives for loop parallelism (`#pragma omp parallel for`).
  - Shared memory model.
  - Scalable and easy to use.
  - Supports both task parallelism and data parallelism.

2. **MPI (Message Passing Interface)**:
   - **Language**: C, C++, Fortran
   - **Description**: MPI is a standard for message-passing parallel programming, primarily used in distributed memory systems. It enables processes running on different nodes or processors to communicate and synchronize via messages.
   - **Key Features**:
     - Allows for explicit data sharing between processes.
     - Suitable for large-scale systems, including supercomputers.
     - Supports both point-to-point and collective communication patterns.
3. **CUDA (Compute Unified Device Architecture)**:
   - **Language**: C, C++, Fortran (with specific extensions)
   - **Description**: CUDA is a parallel computing platform and API model created by NVIDIA for GPUs. It allows developers to write software that can harness the massive parallelism of GPUs, ideal for computation-heavy applications like deep learning, simulations, and data processing.
   - **Key Features**:
     - GPU acceleration.
     - Supports thousands of threads in parallel.
     - Allows for both low-level and high-level programming.
4. **OpenCL (Open Computing Language)**:
   - **Language**: C-based
   - **Description**: OpenCL is an open standard for parallel programming across heterogeneous platforms. It allows for programming on CPUs, GPUs, and other processors.
   - **Key Features**:
     - Suitable for a wide range of devices (CPUs, GPUs, FPGAs).
     - Portable across platforms.
     - Uses a C-based language with extensions for parallelism.
5. **Haskell (with Parallelism Extensions)**:
   - **Language**: Haskell
   - **Description**: Haskell is a functional programming language that supports parallelism through constructs like "par" and "pseq". Haskell's immutability makes it easier to reason about parallel programs.
   - **Key Features**:
     - Lazy evaluation, which can help in parallel execution.
     - High-level abstractions for parallelism.
     - Fine-grained parallelism with minimal state changes.
6. **Go (Golang)**:
   - **Language**: Go
   - **Description**: Go is a language created by Google that simplifies concurrent programming. It includes Goroutines and Channels to facilitate easy management of concurrent tasks.
   - **Key Features**:
     - Lightweight concurrency model (Goroutines).

- Channels for communication between Goroutines.
- Simplicity and ease of use for parallel tasks.

---

## Parallel Programming Models

Parallel programming models describe the way in which parallelism is organized and managed during computation. These models abstract the hardware complexities and provide structured ways to execute parallel tasks.

1. **Shared Memory Model**:
   - **Description**: In shared memory models, all processors can access a global memory space. This model is suited for systems where multiple cores share the same physical memory.
   - **Example**: OpenMP
   - **Key Characteristics**:
     - Data is shared between threads or processes.
     - Synchronization mechanisms (like locks, barriers) are often required.
     - Efficient for multicore processors or machines with shared memory.
2. **Distributed Memory Model**:
   - **Description**: In distributed memory systems, each processor has its own local memory. Communication between processors is done via message passing, usually through MPI.
   - **Example**: MPI
   - **Key Characteristics**:
     - Each processor has its own memory space.
     - Data must be explicitly shared through messaging or file I/O.
     - Suitable for large-scale clusters or distributed systems.
3. **Data Parallelism**:
   - **Description**: Data parallelism involves distributing data across multiple processors and performing the same operation on each piece of data simultaneously. It's useful in operations like matrix multiplication, image processing, and scientific computing.
   - **Example**: CUDA, OpenCL
   - **Key Characteristics**:
     - The same operation is applied to different pieces of data in parallel.
     - Easily scalable across multiple processors or cores.
     - Common in vector and matrix computations.
4. **Task Parallelism**:
   - **Description**: Task parallelism involves executing different tasks or functions concurrently, where tasks can be independent or require synchronization. It focuses on parallelizing different activities rather than data.
   - **Example**: OpenMP (with tasks), Go (Goroutines)
   - **Key Characteristics**:
     - Tasks can be scheduled independently.
     - Suitable for problems where different tasks are decoupled but still need coordination.
     - Often used in parallelizing applications like web servers or data processing pipelines.
5. **Pipeline Parallelism**:

- **Description**: Pipeline parallelism divides a computation into stages, where each stage can be processed independently and concurrently. This model is often used in streaming applications and real-time systems.
- **Example**: Media processing pipelines, video rendering
- **Key Characteristics**:
  - Tasks are broken into a series of steps or stages.
  - Each stage can be processed in parallel on different data.
  - Ideal for tasks that need sequential processing but can be done in parallel stages.

6. **Functional Parallelism**:
   - **Description**: Functional parallelism is based on the functional programming paradigm, where computation is modeled as the evaluation of mathematical functions. Parallelism arises from dividing tasks into sub-functions that can be executed in parallel.
   - **Example**: Haskell
   - **Key Characteristics**:
     - Focuses on immutability and stateless computation.
     - Parallel tasks are pure functions, making it easier to reason about concurrency.
     - High-level abstractions for parallelism and easy reasoning about parallel execution.

## History of GPU Computing

To CUDA C and OpenCL programmers, GPUs are massively parallel numeric computing processors programmed in C with extensions. One does not need to understand graphics algorithms or terminology to be able to program these processors. However, understanding the graphics heritage of these processors illuminates the strengths and weaknesses of them with respect to major computational patterns. In particular, the history helps to clarify the rationale behind major architectural design decisions of modern programmable GPUs: massive multithreading, relatively small cache memories compared to CPUs, and bandwidth-centric memory interface design. Insights into the historical developments will also likely give readers the context needed to project the future evolution of GPUs as computing devices.

## EVOLUTION OF GRAPHICS PIPELINES

- Three-dimensional (3D) graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then PC accelerators in the mid- to late 1990s.
- During this period, the performance-leading graphics subsystems declined in price from $50,000 to $500.
- During the same period, the performance increased from 50 million pixels per second to 1 billion pixels per second, and from 100,000 vertices per second to 10 million vertices per second.
- Shrinking feature sizes of semiconductor devices
- they also come from the new innovations of graphics algorithms and hardware design innovations.
- These innovations have shaThe remarkable advancement of graphics hardware performance has been driven by the market demand for high-quality real-time graphics in computer applications.

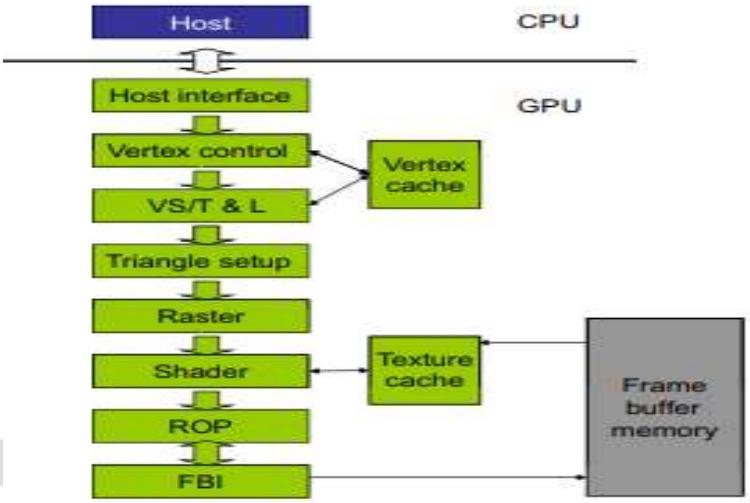
- For example, in an electronic gaming application, one needs to render evermore complex scenes at ever-increasing resolution at a rate of 60 frames per second.
- The net result is that over the last 30 years, graphics architecture has evolved from a simple pipeline for drawing wireframe diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering complex interactive imagery of 3D scenes.

## The Era of Fixed-Function Graphics Pipelines

- From the early 1980s to the late 1990s, the leading performance graphics hardware was fixed-function pipelines that were configurable, but not programmable.

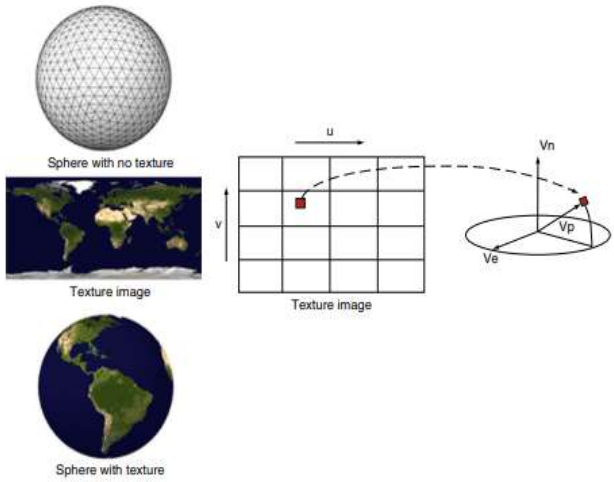
  In that same era, major graphics Application Programming Interface (API) libraries became popular. An API is a standardized layer of software, that is, a collection of library functions that allows applications (e.g., games) to use software or hardware services and functionality.

  **Example : fixed-function graphics pipeline in early NVIDIA GeForce GPUs**.




- **The host interface** receives graphics commands and data from the CPU.
  - The commands are typically given by application programs by calling an API function.
- The host interface typically contains a specialized **DMA hardware** to efficiently transfer bulk data to and from the host system memory to the graphics pipeline.
- The host interface also communicates back the status and result data of executing the commands.
- GeForce graphics pipeline is designed to render **triangles**, so vertex is typically used to refer to the corners of a triangle.

- The surface of an object is drawn as a collection of triangles. The finer the sizes of the triangles are, the better the quality of the picture typically becomes.
- **The vertex control stage** in Figure receives parameterized triangle data from the CPU.
- The vertex control stage converts the triangle data into a form that the hardware understands and places the prepared data into the **vertex cache**.
- The vertex shading, transform, and lighting (**VS/T&L**) stage transforms vertices and assigns per-vertex values (colors, normals, texture coordinates, tangents, etc.).
  - The shading is done by the **pixel shader hardware**. The vertex shader can assign a color to each vertex.
- **The triangle setup stage** further creates edge equations that are used to interpolate colors and other pervertex data (e.g., texture coordinates) across the pixels touched by the triangle.
- **The raster stage** determines which pixels are contained in each triangle.
  - For each of these pixels, the raster stage interpolates per-vertex values necessary for shading the pixel, which includes color, position, and texture position that will be shaded (painted) on the pixel.
- **The shader stage** determines the final color of each pixel. This can be generated as a combined effect of many techniques: interpolation of vertex colors, texture mapping, per-pixel lighting mathematics, reflections, and more.
- Many effects that make the rendered images more realistic are incorporated in the shader stage.




- Above fig illustrates texture mapping, one of the shader stage functionalities.
- It shows an example in which a world map texture is mapped onto a sphere object.

- Note that the sphere object is described as a large collection of triangles.
- Although the shader stage needs to perform only a small number of coordinate transform calculations to identify the exact coordinates of the texture point that will be painted on a point in one of the triangles that describes the sphere object, the sheer number of pixels covered by the image requires the shader stage to perform a very large number of coordinate transforms for each frame.
- **The ROP (raster operation) stage** performs the final raster operations on the pixels.
- It performs color raster operations that blend the color of overlapping/adjacent objects for transparency and anti-aliasing effects.
- It also determines the visible objects for a given viewpoint and discards the occluded pixels.
- A pixel becomes occluded when it is blocked by pixels from other objects according to the given viewpoint
- Figure 2.3 illustrates anti-aliasing, one of the ROP stage operations.
- There are three adjacent triangles with a black background.
- In the aliased output, each pixel assumes the color of one of the objects or the background.
- The limited resolution makes the edges look crooked and the shapes of the objects distorted.
- The problem is that many pixels are partly in one object and partly in another object or the background.
- Forcing these pixels to assume the color of one of the objects introduces distortion into the edges of the objects.
- The anti-aliasing operation gives each pixel a color that is blended, or linearly combined, from the colors of all the objects and background that partly overlap the pixel.
- **The frame buffer interface (FBI)** stage manages memory reads from and writes to the display frame buffer memory.
- For high-resolution displays, there is a very high bandwidth requirement in accessing the frame buffer.
- Such bandwidth is achieved by two strategies.
    - graphics pipelines typically use special memory designs that provide higher bandwidth than the system memories.
    - FBI simultaneously manages multiple memory channels that connect to multiple memory banks.

The combined bandwidth improvement of multiple channels and special memory structures gives the frame buffers much higher bandwidth than their contemporaneous system memories.

## Evolution of Programmable Real-Time Graphics

In 2001, the NVIDIA GeForce 3 introduced general shader programmability, marking a significant step in GPU evolution. This was paired with the release of DirectX 8 and OpenGL extensions for vertex shaders. As GPUs progressed, DirectX 9 expanded programmability to pixel shaders, with the ATI Radeon 9700 (2002) featuring a programmable 24-bit floating-point pixel shader and the GeForce FX adding 32-bit pixel processors.

The unification of vertex and pixel shader functionalities continued, exemplified by NVIDIA's GeForce 6800 and 7800 series, which separated processors for vertex and pixel tasks. However, the Xbox 360 (2005) introduced a unified processor GPU for both tasks. GPUs are designed to exploit data independence, which is key in rendering millions of triangles and pixels per frame, allowing for massive parallelism.

Key stages in the graphics pipeline are programmable, including vertex shaders (which handle the transformation of triangle vertex positions) and pixel shaders (which compute pixel colors). Geometry shaders also modify or generate new primitives. These shader programs can run in parallel due to their independent nature, driving the design of parallel processing GPUs.

The pipeline also incorporates fixed-function stages, like rasterization, which are more efficient for specific tasks. This combination of programmable and fixed-function stages optimizes performance while offering control over rendering algorithms.

GPUs are optimized for high memory bandwidth rather than low latency, as algorithms typically access memory in a coherent, efficient manner. This is reflected in the dominance of floating-point data paths and fixed-function logic on GPU chips, with memory bandwidth in recent designs exceeding 190 GB/s, far surpassing CPU memory capabilities.

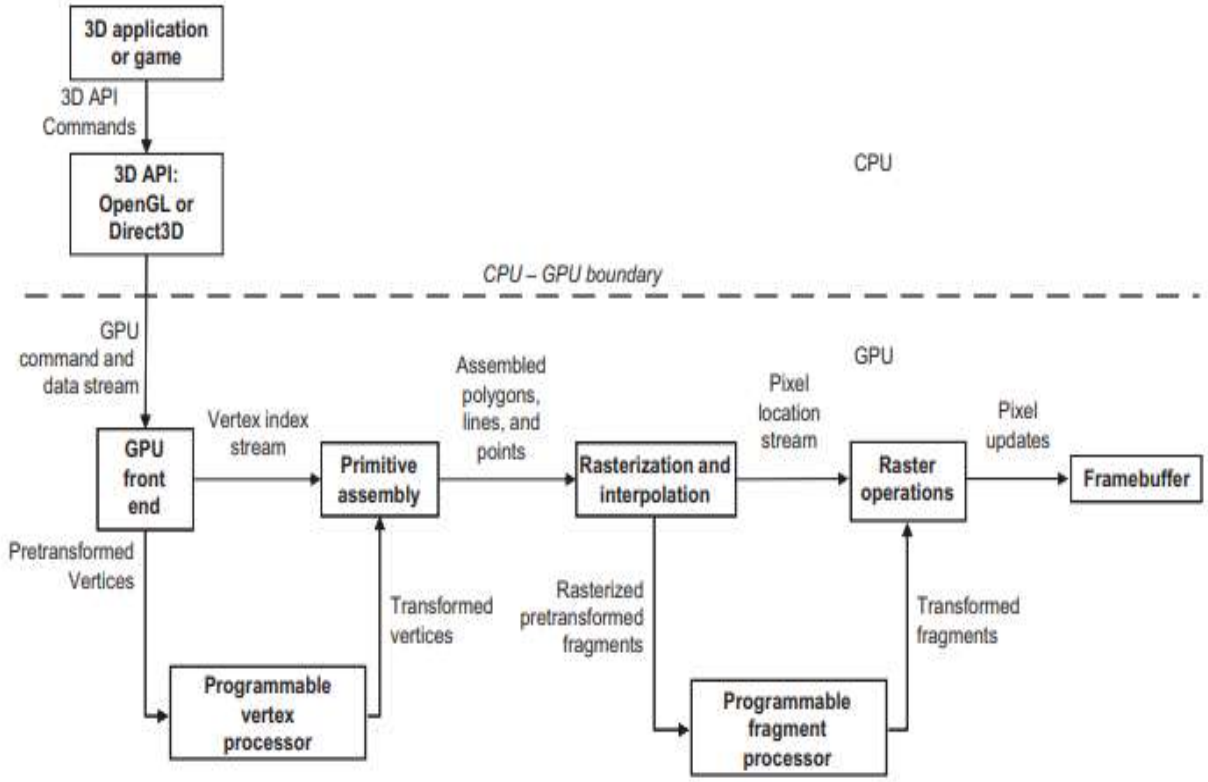



**FIGURE 2.4**

An example of a separate vertex processor and fragment processor in a programmable graphics pipeline.

## Unified Graphics and Computing Processors

Introduced in 2006, NVIDIA's GeForce 8800 GPU revolutionized graphics architecture by mapping separate programmable stages (vertex, geometry, and pixel shaders) to a unified array of processors. This unified design allows dynamic allocation of processing resources across different stages, providing better load balancing depending on the needs of specific rendering algorithms. The GPU's recirculating pipeline visits these processors multiple times with fixed-function logic interspersed, optimizing performance.

The GeForce 8800 was built with DirectX 10 in mind, which introduced the concept of identical vertex and pixel shaders, along with a new geometry shader to handle entire primitives instead of individual vertices. With growing complexity in shading algorithms, the demand for higher shader throughput, particularly floating-point operations, led NVIDIA to prioritize high operating clock speeds. This necessitated a shift toward a unified processor array to balance engineering challenges with performance efficiency.

The design of a single processor array, rather than separate ones for each stage, also opened the door to using GPUs for general-purpose numeric computing, expanding their capabilities beyond graphics rendering.


**FIGURE 2.5**

Unified programmable processor array of the GeForce 8800 GT graphics pipeline.

## GPU COMPUTING

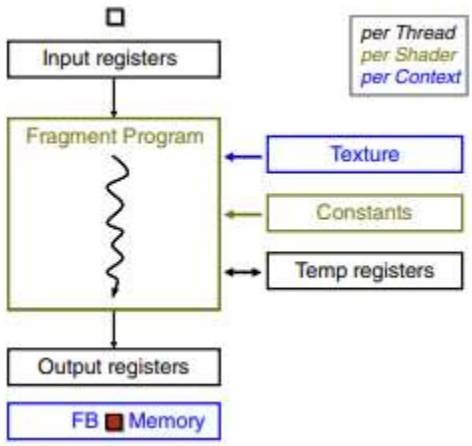
During the development of the Tesla GPU architecture, NVIDIA recognized that its potential could be greatly expanded if programmers could think of the GPU as a general-purpose processor. To achieve this, NVIDIA adopted a programming model where programmers explicitly declared the data-parallel aspects of their workloads.

For the DirectX 10 generation, NVIDIA had already begun working on high-efficiency floating-point and integer processors to support the graphics pipeline. The Tesla architecture took this a step further by transforming shader processors into fully programmable processors with their own instruction memory, instruction cache, and control logic for sequencing instructions. These added features were made more cost-effective by allowing multiple shader processors to share resources like instruction cache and sequencing logic. This design proved effective for graphics, where the same shader program is applied to large numbers of vertices or pixels.

To further broaden the architecture's applicability, NVIDIA introduced memory load and store instructions with random byte addressing, enabling support for compiled C programs. The Tesla architecture also provided a more generic parallel programming model, including parallel thread hierarchies, barrier synchronization, and atomic operations, making it easier to manage highly parallel workloads. The introduction of CUDA (Compute Unified Device Architecture) C/C11 compilers,

libraries, and runtime software allowed developers to access the GPU's parallel computing power without needing to use the graphics API.

The Tesla-based G80 chip powered NVIDIA's GeForce 8800 GTX and was later succeeded by GPUs like the G92, GT200, Fermi, and Kepler, marking a significant shift in the role of GPUs from primarily graphics rendering to general-purpose parallel computing.




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

## Scalable GPUs

1. **Early Graphics System Scalability:**
   - Early workstation graphics allowed scalability by adding more pixel processor circuit boards for higher pixel performance.
   - Before the mid-1990s, PC graphics had limited scalability with only the VGA controller as an option.
2. **Emergence of 3D Accelerators:**
   - With the advent of 3D-capable accelerators, there was room for a range of packaging and chip designs, such as GeForce 2 GTS (high performance) and GeForce 2 MX (lower cost).
   - Today, a single architecture typically requires 4-5 chip designs to cover different desktop performance and price points.
3. **Introduction of Multi-GPU Scaling:**
   - In 2004, NVIDIA adopted multi-GPU scalability (SLI) after acquiring 3dfx, starting with the GeForce 6800.
   - SLI provides transparent scalability to both programmers and users, meaning applications run unchanged across various GPU configurations.
4. **CPU Scalability via Multicore Design:**
   - CPUs scale by increasing the number of cores (from quad-core to oct-core), requiring programmers to find parallelism to fully utilize the additional cores.

- Applications often need to be rewritten with more parallel tasks for each doubling of core count.

5. **GPU Scalability and Fine-Grained Parallelism:**
   - GPUs embrace highly multithreaded architectures, encouraging fine-grained data parallelism via CUDA.
   - This enables massive parallelism, and each doubling of GPU core count leads to more hardware execution resources and higher performance.
6. **Portable and Transparent Scalability:**
   - The GPU parallel programming model is designed for transparent and portable scalability, meaning programs written once can run efficiently on GPUs with varying core counts.
   - This scalability model ensures that applications utilize available hardware resources without needing significant changes.
7. **Historical Scaling Initiatives:**
   - In 1998, 3dfx introduced multi-board scaling with its original SLI (Scan Line Interleave) on the Voodoo2.
   - NVIDIA created distinct product variants based on a single architecture, such as the Riva TNT Ultra (high performance) and Vanta (low cost), using speed binning.