GPU硬件架构概述

GPU主要由显存、GPU和接口、电源等组成。

从显存的角度，可以将显卡分为集成显卡和独立显卡。其中，集成显卡指的是显卡使用CPU的一部分内存作为显存，GPU和CPU用不同的虚拟地址对CPU中的同一个物理地址寻址。使用集成显卡时，CPU和GPU共享总线。在渲染时，CPU将顶点等数据存入主存，然后GPU可以通过UMA来进行数据传输。所以相比独立显卡，在使用集成显卡时CPU可以更快速的访问显存。对独立显卡来说，GPU可以使用专门的显存条，并使用显存条的物理地址进行寻址，这是最常见的显卡类型。在独立显卡结构中，GPU可以直接从显存中读写信息。而CPU访问显存条中的储存空间时，需要映射一部分GPU储存空间到CPU地址空间，典型大小为256MB或512MB。

GPU是显卡上最重要的核心处理芯片。虽然GPU和CPU一样是主力计算元件，但CPU重在实时响应，对单任务速度要求高，需要针对延迟优化，所以晶体管数量和能耗都需要用在分支预测、乱序执行、低延迟缓存等控制部分；GPU主要使用于具有极高可预测性和大量相似运算的批处理，以及高延迟、高吞吐的架构运算，对缓存的要求相对很低，顺序运算效率很高，同时相对的乱序处理效率很低。

CPU除了负责浮点和整型运算，还有很多其它的指令集的负载，如多媒体解码和硬件解码，CPU注重单线程性能，保证指令流不中断，需要消耗更多晶体管和能耗用在控制部分，于是CPU分配在浮点运算的功耗会减少；GPU基本只进行浮点运算，设计结构简单，效率更高，GPU注重吞吐率，单指令能驱动更多的计算，相比较GPU消耗在控制的能耗就少得多，因此可以将资源留给浮点运算使用。GPU的浮点运算能力比CPU高10~12倍。

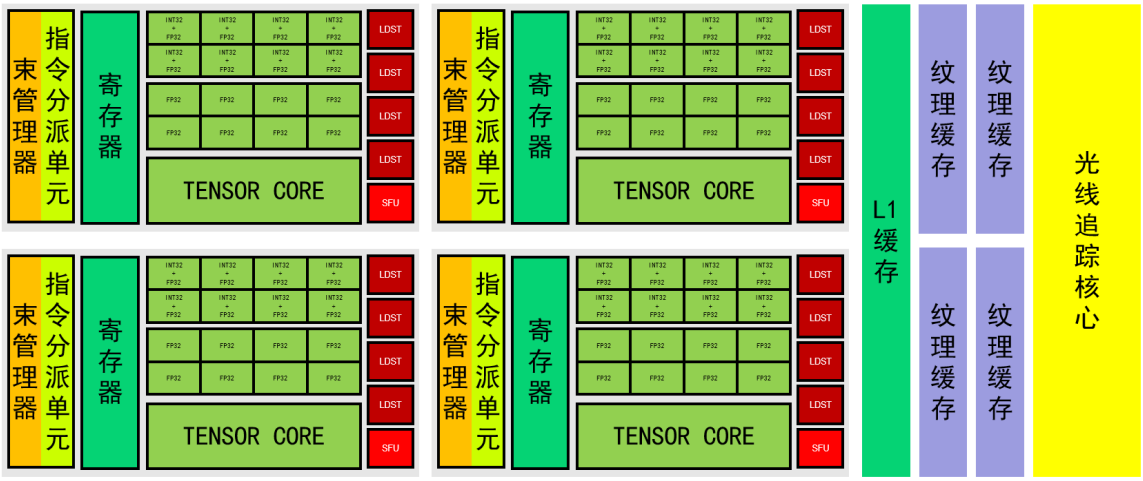
GPU的最小计算单元被称为流处理器(SP)，有时也被称为核心(Core)。在GPU诞生初期，其运算单元被分为顶点处理单元和像素处理单元，顾名思义，早期GPU的结构中使用了两个不同的硬件单元来分别处理顶点着色器和像素着色器，这种分离渲染架构存在严重的资源分配不均的问题，两种单元渲染任务量不同，效率低下。从DirectX10开始，两种处理单元被统一为了流处理器，在这两个阶段中的运算都是在流处理器中进行的，这种将顶点处理单元和像素处理单元合并的概念又被称作统一着色器架构。

因为渲染过程中以三维空间运算为主，而三维空间运算又基本都是浮点数运算，所以流处理器早期仅支持浮点数运算，但现在由于GPU开始进入人工智能领域发挥作用，最新的GPU也开始使用支持浮点数运算的SP。

根据显卡厂商的不同，SP的大小也不同。英伟达生产的显卡使用的SP通常是对1D数据(也就是浮点数或整数)进行计算的，而AMD的显卡中的SP通常是一些向量计算单元，通常可以一次对一个4D或4D+1D数据进行计算。A卡的理论计算能力远超N卡，但实际执行效率并不高，一旦进入GPU的图形信息是1D或3D形式这一的非标准数据形式，A卡的执行效率最低可降至25%至20%。N卡在软件上具有明显优势，包括微软在内的软件商都为N卡开发优化，使得大量工具软件和游戏在N卡环境下有更好的表现。

将大约16个、32个或64个SP分为一组，它们并行的执行一次任务，被称为执行一个GPU线程束。线程束是被一个称为流多重处理器(SM)的硬件结构进行管理的，如下图所示：

流多处理器 SM



目前的SM采用单指令多线程(SIMT)架构，也就是说，在同一个SM中的一组SP会共用一个指令。SM中存在一个很大的寄存器空间，在RTX 3080中这个寄存器的空间为16384×32-bit。在分配指令时，这个寄存器会被均分给每个SP，每个SP会获得一块连续的寄存器空间，就好像每个SP拥有一个自己的寄存器那样。

在GPU执行并行任务时，每个最小的并行计算任务被称为一个线程。如在进行蒙皮解算时，这个最小的计算单元就是对一个顶点的解算，而在渲染时最小计算单元则是对一个片元的光照计算。整个并行任务，如解算整个网格上所有的顶点，被分配给一个或多个SM进行计算，SM接受的这些线程被称为线程组。

在SM接受一个线程组后，由束管理器来将线程组分配成若干线程束，如一个100个线程的浮点数运算线程组会被分为4个线程束，前三个线程束有32个线程，而最后一个只有4个线程。一个SM中可能包括若干个线程束，如在上图的架构中，这个SM就可以一次执行4个线程束。在运行时，指令分派单元会同时向所有SP发送同一个指令，而SP则同时进行运算。其中，当遇到寻址或取址指令时，指令分派单元要向不同SP发送不同的地址。如果单个线程需要的寄存器空间过大，可能使每束的最多线程数减少，影响并行性。

在一个线程束中，如果线程中不存在动态分支语句，那么它们的所有行为都是可预测相同的。但一旦存在动态分支语句，因为SIMT架构中所有SP必须执行同一指令的特性，但凡有一个线程执行另一个分支，那么整个线程束就不得不被执行两遍，将两个分支的结果都运行一次，并让每个线程扔掉它们各自不需要的结果。如果在GPU编程中出现嵌套的分支语句或复杂的循环语句，每个线程束的执行次数可能呈指数级递增，这个效应被称为线程分歧。可见，使用循环语句运行多次来读取某常量数组中的信息并不会导致线程分歧，线程分歧的严重与否关键在于相邻的元素通过动态分支语句能否得到基本相似的分支。最近的一些GPU厂商已经开始着手实现在SP中执行动态分支语句的功能，但由于旧的硬件设备还没有彻底离开市场，所以无限制的使用动态分支语句还存在较高的风险。

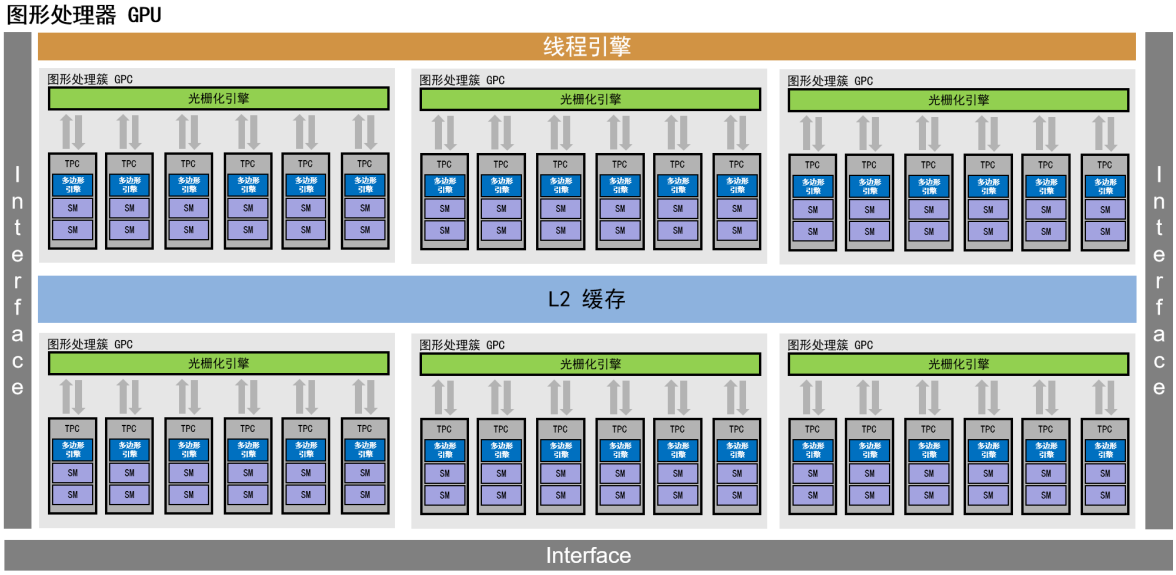
同一束中不同线程间可以通过本地缓存共享通道LDS(Local Data Share)进行通信，这也给了GPU对同一组值进行并行读写操作的可能性。在LDS上的通信根据显卡厂商和其使用的架构不同，可能是有锁的或无锁的。

在执行每束的计算过程中，还具有一些辅助计算单元，比如Nvidia架构中专门用于矩阵运算的TENSOR Core，以及用于进行三角函数和指对数运算的特殊运算单元SFU。

除此之外，SM还具有供整个线程组使用的L1缓存和纹理缓存。GPU通常采用大带宽的缓存策略，也就是一次缓存更多的数据，这使得SP缓存命中的概率大大增加，但同时任何一次缓存丢失都需要更高的代价。以纹理采样为例，由于纹理数据储存在显存中，在每次遇到采样语句时，为了将显存中的数据块调入纹理缓存(与纹理在GPU中的压缩方式有关，因为与本书关联不大，所以在此不详解)，可能需要使处理器阻塞几百上千个时钟周期。为了缓解这样的问题，GPU使用换入换出操作来隐藏延迟。每当有一批片元在等待采样纹理数据时，可以将此时SP组的上下文备份保存在寄存器中，然后导入下一批片元进行

处理，这个操作被称为换出。换出使得核心不需要空等采样结果，可以继续执行更多的片元。当显存中的数据被拷贝到纹理缓冲时，SM可以将SP上下文从寄存中拷贝回SP中，继续进行这个片元采样之后的运算，直到它再次被换出，这个操作被称为换入。计算本身占用的寄存器越多，换入换出能进行的最大次数也就越少，如果寄存器已经达到储存上线后再遇到采样丢失，就只能让线程空转来等待采样结果。

在Nvidia的Ampere架构中，由两个SM组成的纹理处理簇TPC是最小的任务分配单位，而每六个TPC组成的图形处理簇GPC是一次渲染中的最小任务分配单位。RTX 3080芯片中则由六个这样的GPC组成。如下图所示：



在GPC中存在光栅化引擎，其作用是在顶点阶段结束像素阶段开始前对所有顶点属性，包括坐标和其他顶点属性，进行扫描变换和属性插值。在顶点阶段结束后，需要插值的顶点属性会按照指定格式储存到L1缓存中，然后经过光栅化引擎生成片元后，重新送回L1缓存中。

TPC中存在多边形引擎，其作用是实现顶点裁剪、曲面细分和属性装配等，在外界传入的顶点属性或片元属性会经过多边形引擎装配到指定的寄存器或缓存中。