

# GPU基础知识

---

## 一、导言

世界上大多数事物的发展规律是相似的，在最开始往往都会出现相对通用的方案解决绝大多数的问题，随后会出现为某一场景专门设计的解决方案，这些解决方案不能解决通用的问题，但是在某些具体的领域会有极其出色的表现。

对于GPU而言，这套理论也同样适用，希望通过此文能让你真正了解GPU的特性、用途、GPU与CPU之间的存在哪些异同、GPU的底层架构，以及为什么深度学习需要使用GPU而不是CPU。

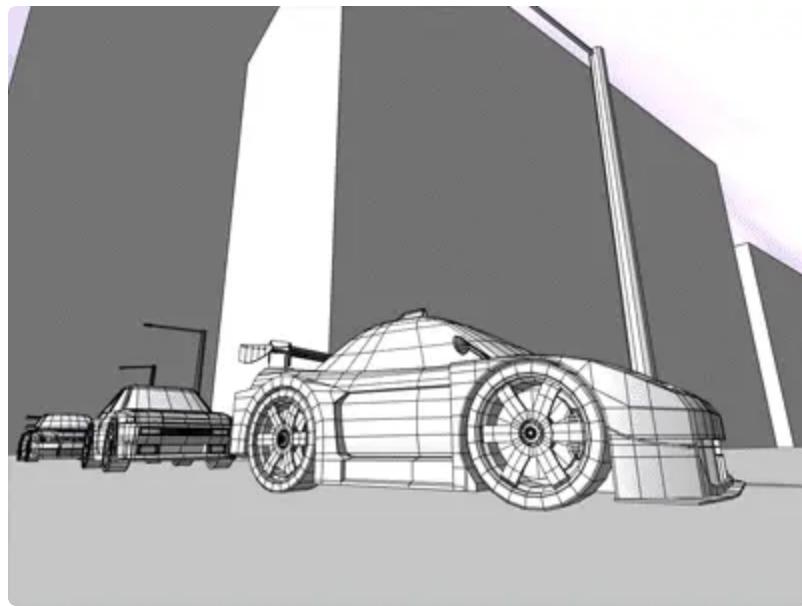
## 二、GPU概述

### 1、什么是GPU

GPU全称是Graphics Processing Unit，图形处理单元，又称视觉处理器、图形显示卡。GPU负责渲染出2D、3D、VR效果，主要专注于计算机图形图像领域。后来人们发现，GPU非常适合并行计算，可以加速现代科学计算，GPU也因此不再局限于游戏和视频领域。



NVIDIA GPU芯片实物图



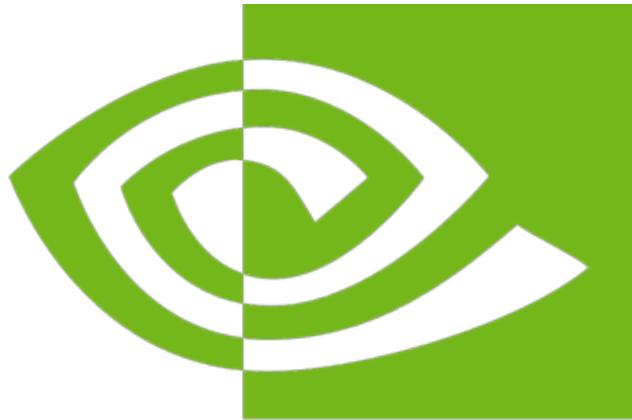
GPU渲染图

我们日常讨论GPU和显卡时，经常混为一谈，严格来说是有所区别的。GPU是显卡（Video card、Display card、Graphics card）最核心的部件，但除了GPU，显卡还有扇热器、通讯元件、与主板和显示器连接的各类插槽。

对于PC桌面，生产GPU的厂商主要有两家：

- NVIDIA:

英伟达，是当今首屈一指的图形渲染技术的引领者和GPU生产商佼佼者。NVIDIA的产品俗称N卡，代表产品有GeForce系列、GTX系列、RTX系列等。



**nVIDIA**®

- AMD:

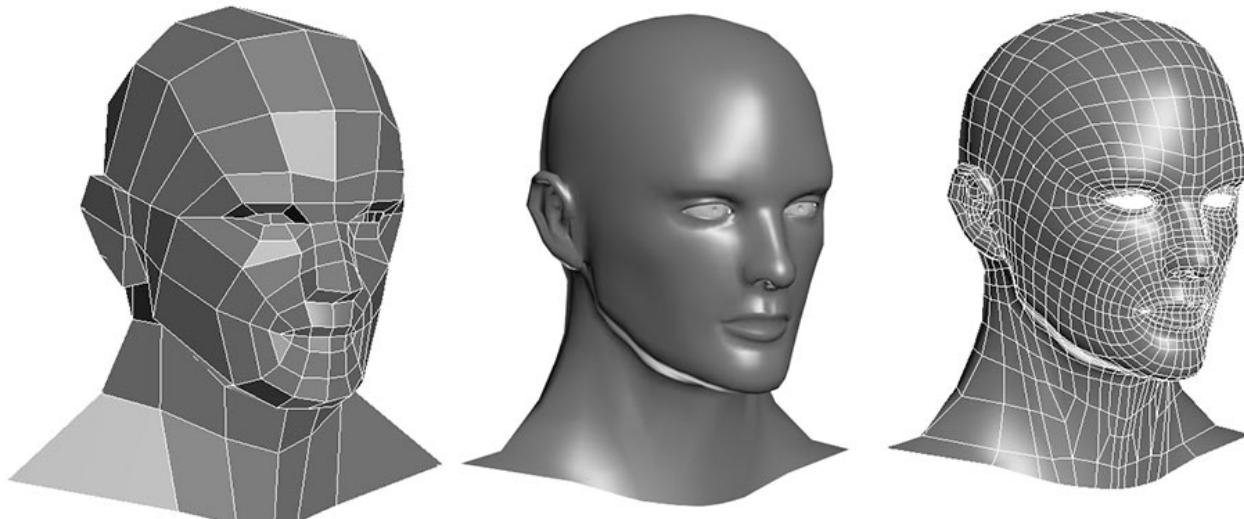
既是CPU生产商，也是GPU生产商，它家的显卡俗称A卡。代表产品有Radeon系列。



当然，NVIDIA和AMD也都生产移动端、图形工作站和数据中心类型的GPU。

除了NVIDIA和AMD，国内也有很多GPU厂商，如华为昇腾、壁仞科技、燧原科技、登临科技、天数智芯、沐曦、摩尔线程、景嘉微、芯动科技等等。

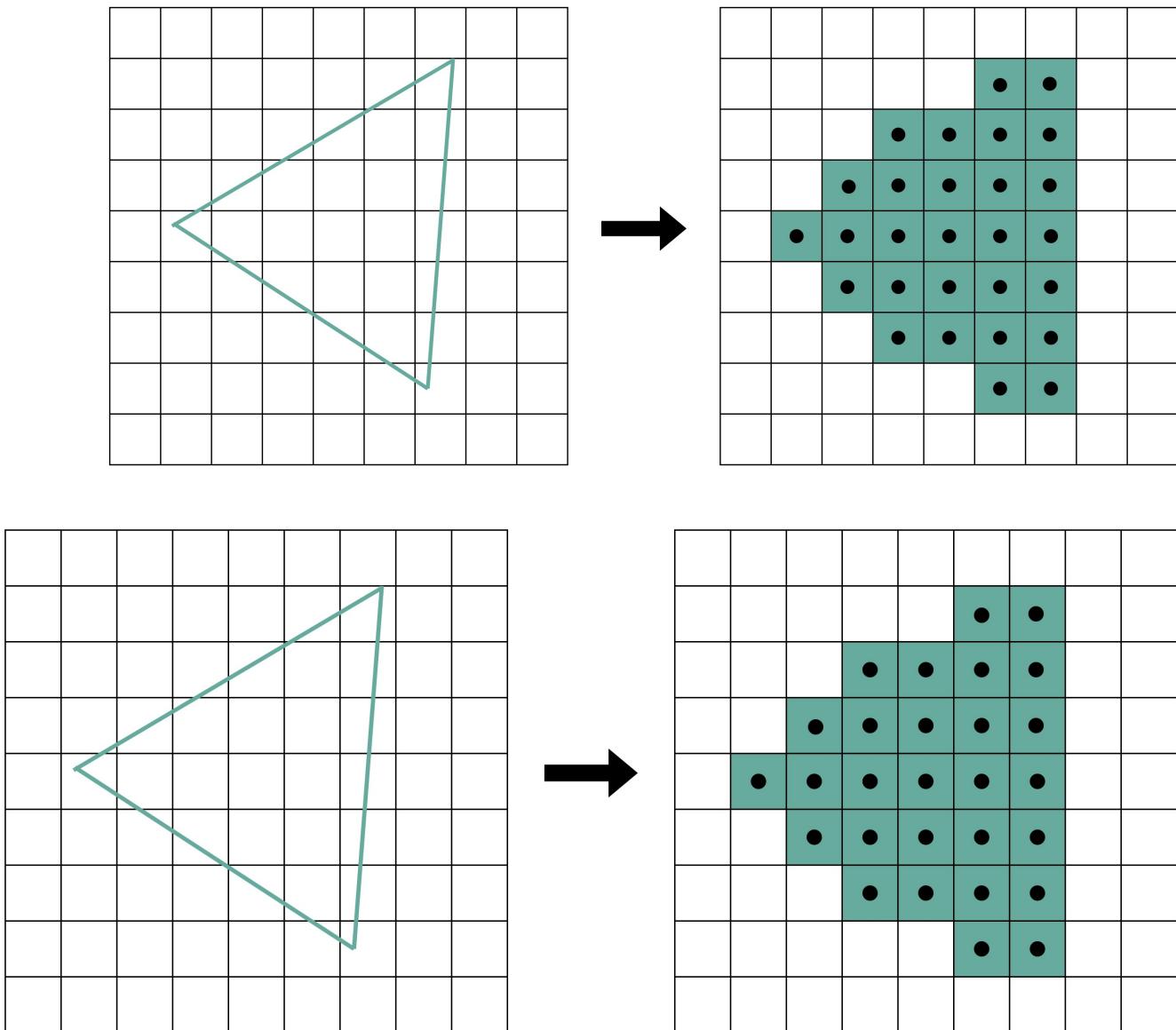
各种游戏里面的人物的脸，并不是那个相机或者摄像头拍出来的，而是通过多边形建模（Polygon Modeling）创建出来的。而实际这些人物在画面里面的移动、动作，乃至根据光线发生的变化，都是通过计算机根据图形学的各种计算，实时渲染出来的。



图像进行实时渲染的过程，可以被分解成下面这样 5 个步骤：

- 顶点处理 (Vertex Processing)
  - 构成多边形建模的每一个多边形呢，都有多个顶点（Vertex）。这些顶点都有一个在三维空间里的坐标。但是我们的屏幕是二维的，所以在确定当前视角的时候，我们需要把这些顶点在三维空间里面的位置，转化到屏幕这个二维空间里面。这个转换的操作，就被叫作顶点处理。这样的转化都是通过线性代数的计算来进行的。可以想见，我们的建模越精细，需要转换的顶点数量就越多，计算量就越大。而且，这里面每一个顶点位置的转换，互相之间没有依赖，是可以并行独立计算的。
- 图元处理
  - 把顶点处理完成之后的各个顶点连起来，变成多边形。其实转化后的顶点，仍然是在一个三维空间里，只是第三维的 Z 轴，是正对屏幕的“深度”。所以我们针对这些多边形，需要做一个操作，叫剔除和裁剪（Cull and Clip），也就是把不在屏幕里面，或者一部分不在屏幕里面的内容给去掉，减少接下来流程的工作量。
- 棚格化
  - 我们的屏幕分辨率是有限的。它一般是通过一个个“像素（Pixel）”来显示出内容的。对于做完图元处理的多边形，把它们转换成屏幕里面的一个个像素点。每一个图元都可以并行独立地棚

格化。

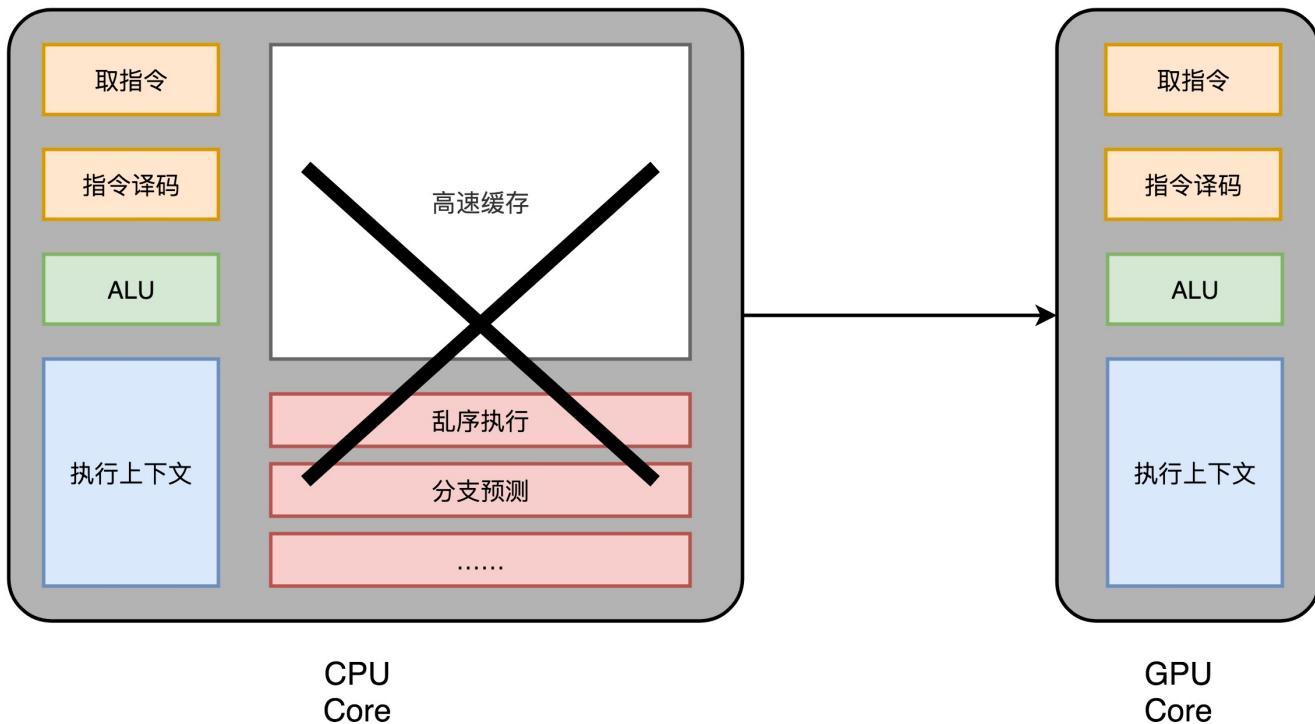


- 片段处理
  - 在栅格化变成了像素点之后，我们的图还是“黑白”的。我们还需要计算每一个像素的颜色、透明度等信息，给像素点上色。
- 像素操作
  - 把不同的多边形的像素点“混合（Blending）”到一起。可能前面的多边形可能是半透明的，那么前后的颜色就要混合在一起变成一个新的颜色；或者前面的多边形遮挡住了后面的多边形，那么我们只要显示前面多边形的颜色就好了。最终，输出到显示设备。

经过这完整的 5 个步骤之后，完成了从三维空间里的数据的渲染，变成屏幕上你可以看到的 3D 动画了。称之为图形流水线（Graphic Pipeline）。



现代 CPU 里的晶体管变得越来越多，越来越复杂，其实已经不是用来实现“计算”这个核心功能，而是拿来实现处理乱序执行、进行分支预测，以及高速缓存部分。而在 GPU 里，这些电路就显得有点多余了，GPU 的整个处理过程是一个流式处理（Stream Processing）的过程。因为没有那么多分支条件，或者复杂的依赖关系，我们可以把 GPU 里这些对应的电路都可以去掉，做一次小小的瘦身，只留下取指令、指令译码、ALU 以及执行这些计算需要的寄存器和缓存就好了。

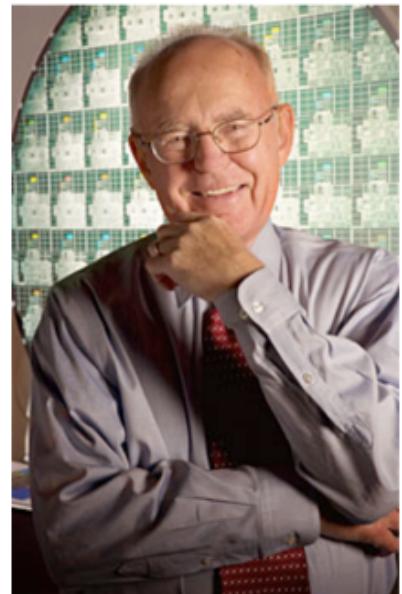
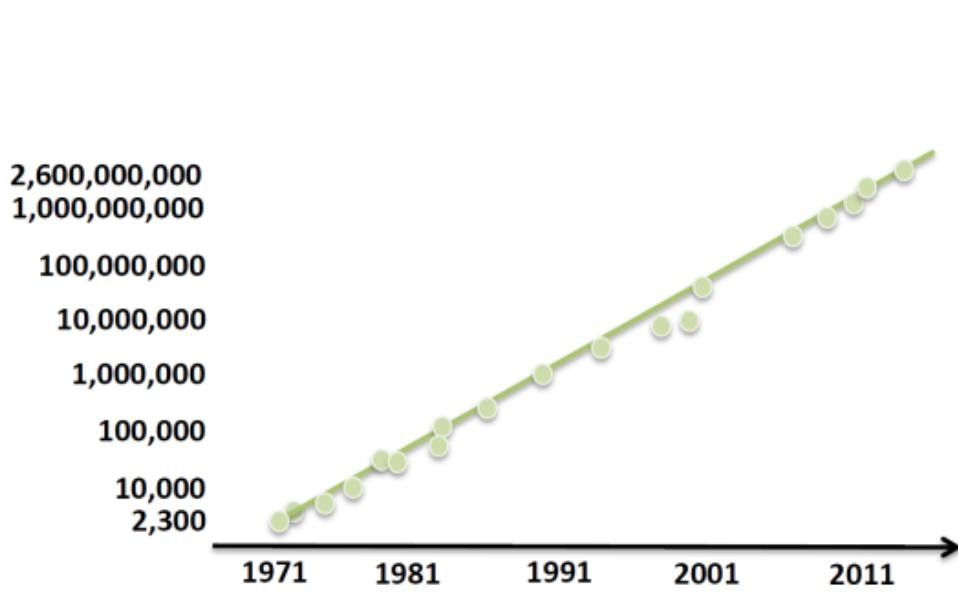


于是，我们就可以在一个 GPU 里面，塞很多个这样并行的 GPU 电路来实现计算，就好像 CPU 里面的多核 CPU 一样。和 CPU 不同的是，我们不需要单独去实现什么多线程的计算。因为 GPU 的运算是天然并行的。无论是对多边形里的顶点进行处理，还是屏幕里面的每一个像素进行处理，每个点的计算都是独立的。

一方面，GPU 是一个可以进行“通用计算”的框架，我们可以通过编程，在 GPU 上实现不同的算法。另一方面，现在的深度学习计算，都是超大的向量和矩阵，海量的训练样本的计算。整个计算过程中，没有复杂的逻辑和分支，非常适合 GPU 这样并行、计算能力强的架构。

## 2、GPU的历史

众所周知，CPU的发展符合摩尔定律：每18个月速度翻倍。

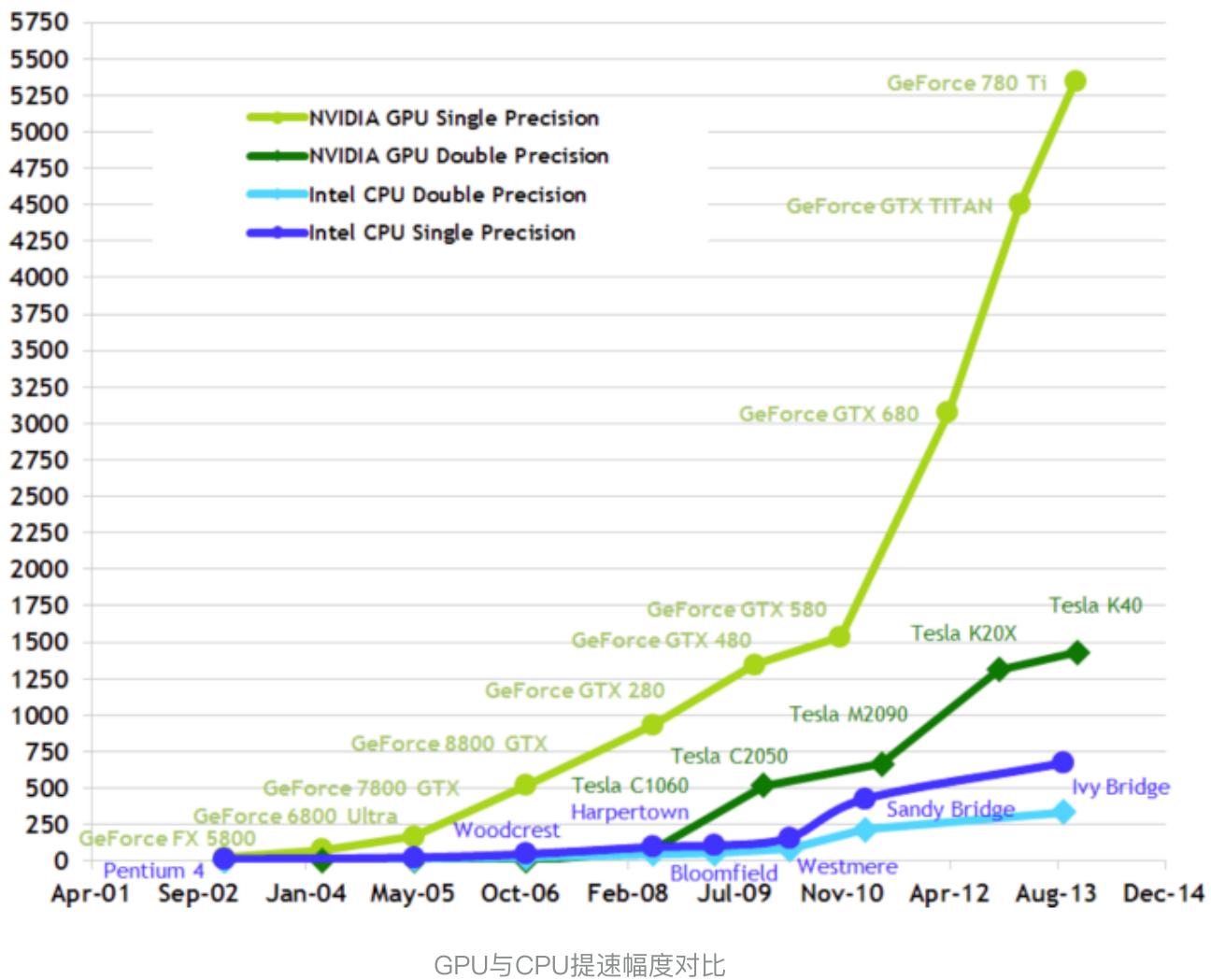


Gordon Moore, co-Founder  
Intel Corporation

处理芯片晶体管数量符合摩尔定律，图右是摩尔本人，Intel的创始人

而NVIDIA创始人黃仁勋在很多年前曾信誓旦旦地说，GPU的速度和功能要超越摩尔定律，每6个月就翻一倍。NV的GPU发展史证明，他确实做到了！GPU的提速幅率远超CPU：

## Theoretical GFLOP/s



NVIDIA GPU架构历经多次变革，从起初的Tesla发展到最新的Turing架构，发展史可分为以下时间节点：

- 2008 – Tesla
  - 2006年,英伟达发布首个通用GPU计算架构Tesla。它采用全新的CUDA架构,支持使用C语言进行GPU编程,可以用于通用数据并行计算。Tesla架构具有128个流处理器,带宽高达86GB/s,标志着GPU开始从专用图形处理器转变为通用数据并行处理器。
  - Tesla最初是给计算处理单元使用的,应用于早期的CUDA系列显卡芯片中,并不是真正意义上的普通图形处理芯片。
- 2010 – Fermi
  - 2009年, 英伟达发布Fermi架构,是第一款采用40nm制程的GPU。Fermi架构带来了重大改进,包括引入L1/L2快速缓存、错误修复功能和 GPUDirect技术等。Fermi GTX 480拥有480个流处理器

器,带宽达到177.4GB/s,比Tesla架构提高了一倍以上,代表了GPU计算能力的提升。

- Fermi是第一个完整的GPU计算架构。首款可支持与共享存储结合纯cache层次的GPU架构, 支持ECC的GPU架构。

- **2012 – Kepler**

- 2012年,英伟达发布Kepler架构,采用28nm制程,是首个支持超级计算和双精度计算的GPU架构。Kepler GK110具有2880个流处理器和高达288GB/s的带宽,计算能力比Fermi架构提高3–4倍。Kepler架构的出现使GPU开始成为高性能计算的关注点。
- Kepler相较于Fermi更快, 效率更高, 性能更好。

- **2014 – Maxwell**

- 2014年,英伟达发布Maxwell架构,采用28nm制程。Maxwell架构在功耗效率、计算密度上获得重大提升,一个流处理器拥有128个CUDA核心,而Kepler仅有64个。GM200具有3072个CUDA核心和336GB/s带宽,但功耗只有225W,计算密度是Kepler的两倍。Maxwell标志着GPU的节能计算时代到来。
- 全新的立体像素全局光照 (VXGI) 技术首次让游戏 GPU 能够提供实时的动态全局光照效果。基于 Maxwell 架构的 GTX 980 和 970 GPU 采用了包括多帧采样抗锯齿 (MFAA)、动态超级分辨率 (DSR)、VR Direct 以及超节能设计在内的一系列新技术。

- **2016 – Pascal**

- 2016年,英伟达发布Pascal架构,采用16nm FinFETPlus制程,增强了GPU的能效比和计算密度。Pascal GP100具有3840个CUDA核心和732GB/s的显存带宽,但功耗只有300W,比Maxwell架构提高50%以上。Pascal架构使GPU可以进入更广泛的人工智能、汽车等新兴应用市场。
- Pascal 架构将处理器和数据集成在同一个程序包内, 以实现更高的计算效率。1080系列、1060系列基于Pascal架构

- **2017 – Volta**

- 2017年,英伟达发布Volta架构,采用12nm FinFET制程。Volta 架构新增了张量核心,可以大大加速人工智能和深度学习的训练与推理。Volta GV100具有5120个CUDA 核心和900GB/s的带宽,加上640个张量核心, AI计算能力达到112 TFLOPS,比Pascal架构提高了近3倍。Volta的出现标志着AI成为GPU发展的新方向。
- Volta 配备640 个Tensor 核心, 每秒可提供超过100 兆次浮点运算(TFLOPS) 的深度学习效能, 比前一代的Pascal 架构快5 倍以上。

- **2018 – Turing**

- 2018年,英伟达发布Turing 架构,采用12nm FinFET制程。Turing架构新增了Ray Tracing核心(RT Core),可硬件加速光线追踪运算。Turing TU102具有4608个CUDA核心、576个张量核心和72个RT核心,支持GPU光线追踪,代表了图形技术的新突破。同时,Turing架构在人工智能方面性能也有较大提升。
  - Turing 架构配备了名为 RT Core 的专用光线追踪处理器, 能够以高达每秒 10 Giga Rays 的速度对光线和声音在 3D 环境中的传播进行加速计算。Turing 架构将实时光线追踪运算加速至上一代 NVIDIA Pascal™ 架构的 25 倍, 并能以高出 CPU 30 多倍的速度进行电影效果的最终帧渲染。2060系列、2080系列显卡也是跳过了Volta直接选择了Turing架构。
- 2020 – Ampere
    - 2020年,英伟达发布Ampere架构,采用Samsung 8nm制程。Ampere GA100具有6912个CUDA核心、108个张量核心和72个RT核心,比Turing架构提高约50%。Ampere 架构在人工智能、光线追踪和图形渲染等方面性能大幅跃升,功耗却只有400W,能效比显著提高。

从Tesla架构到Ampere架构,英伟达GPU在体系结构、性能和功能上不断创新,不断推动图形渲染、高性能计算和人工智能等技术发展。随着GPU架构的不断演进, GPU已经从图形渲染的专用加速器,发展成通用的数据并行处理器,在人工智能、自动驾驶、高性能计算等领域获得广泛应用。

从Tesla架构到Ampere架构,全面而深入地剖析了英伟达GPU近15年来的架构演变历史,详细介绍了各GPU架构的制程、规格、特性以及技术意义。

通过这些分析可以清晰地看出,英伟达GPU架构在不断演进中实现了跨越式的提高,不但加强了图形渲染和通用计算功能,也在人工智能和光线追踪等新兴技术上作出了持续创新,成就了GPU在各领域的广泛应用。

### 3、GPU的用途

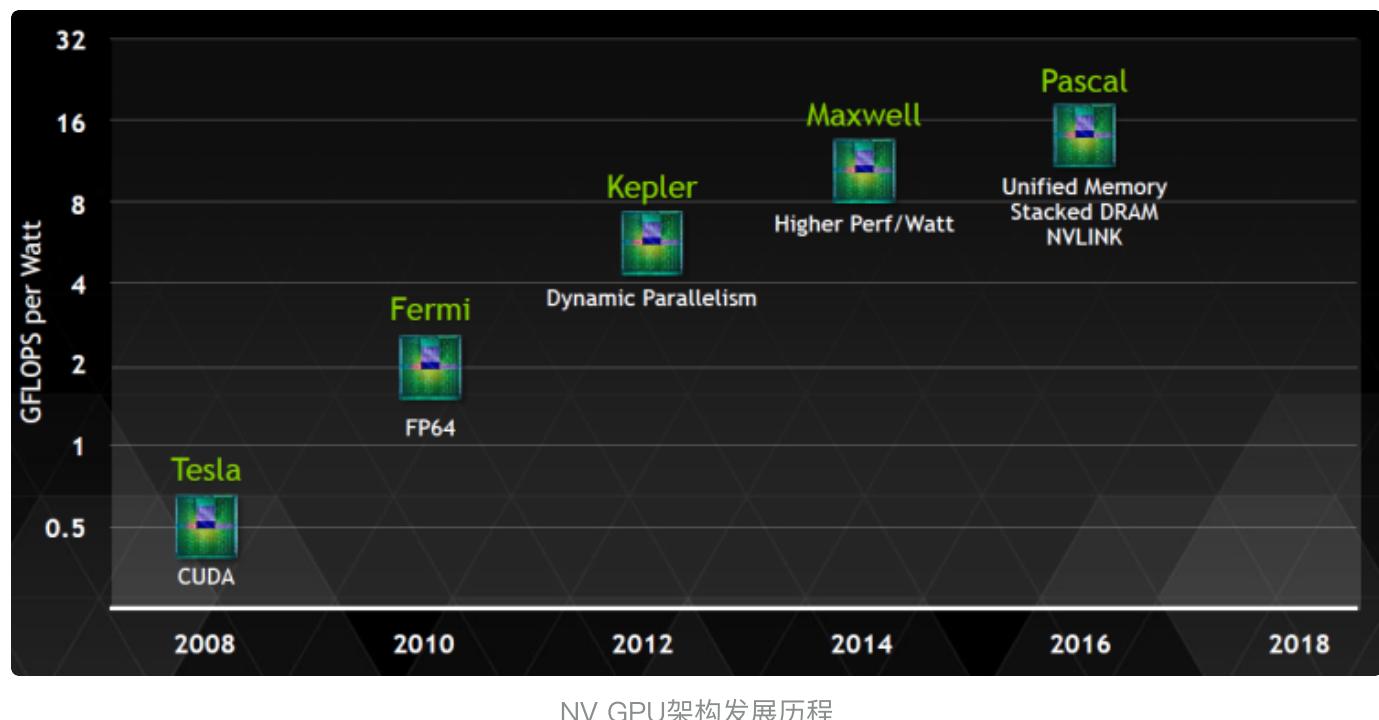
现代GPU除了绘制图形外和游戏渲染外, 还担当了很多额外的功能, 综合起来如下几方面:

- 图形绘制。
  - 这是GPU最传统的拿手好戏, 也是最基础、最核心的功能。为大多数PC桌面、移动设备、图形工作站提供图形处理和绘制功能。
- 物理模拟。
  - GPU硬件集成的物理引擎 (PhysX、Havok) , 为游戏、电影、教育、科学模拟等领域提供了成百上千倍性能的物理模拟, 使得以前需要长时间计算的物理模拟得以实时呈现。

- 海量计算。
  - 计算着色器及流输出的出现，为各种可以并行计算的海量需求得以实现，CUDA就是最好的例证。
- AI运算。
  - 近年来，特别是从ChatGPT推出后，人工智能的崛起推动了GPU集成了AI Core运算单元，反哺AI运算能力的提升，给各行各业带来了计算能力的提升。
- 其它计算。
  - 音视频编解码、加解密、科学计算、离线渲染等等都离不开现代GPU的并行计算能力和海量吞吐能力。

## 三、GPU架构发展

### 1、概述



# NVIDIA的GPU架构发展史

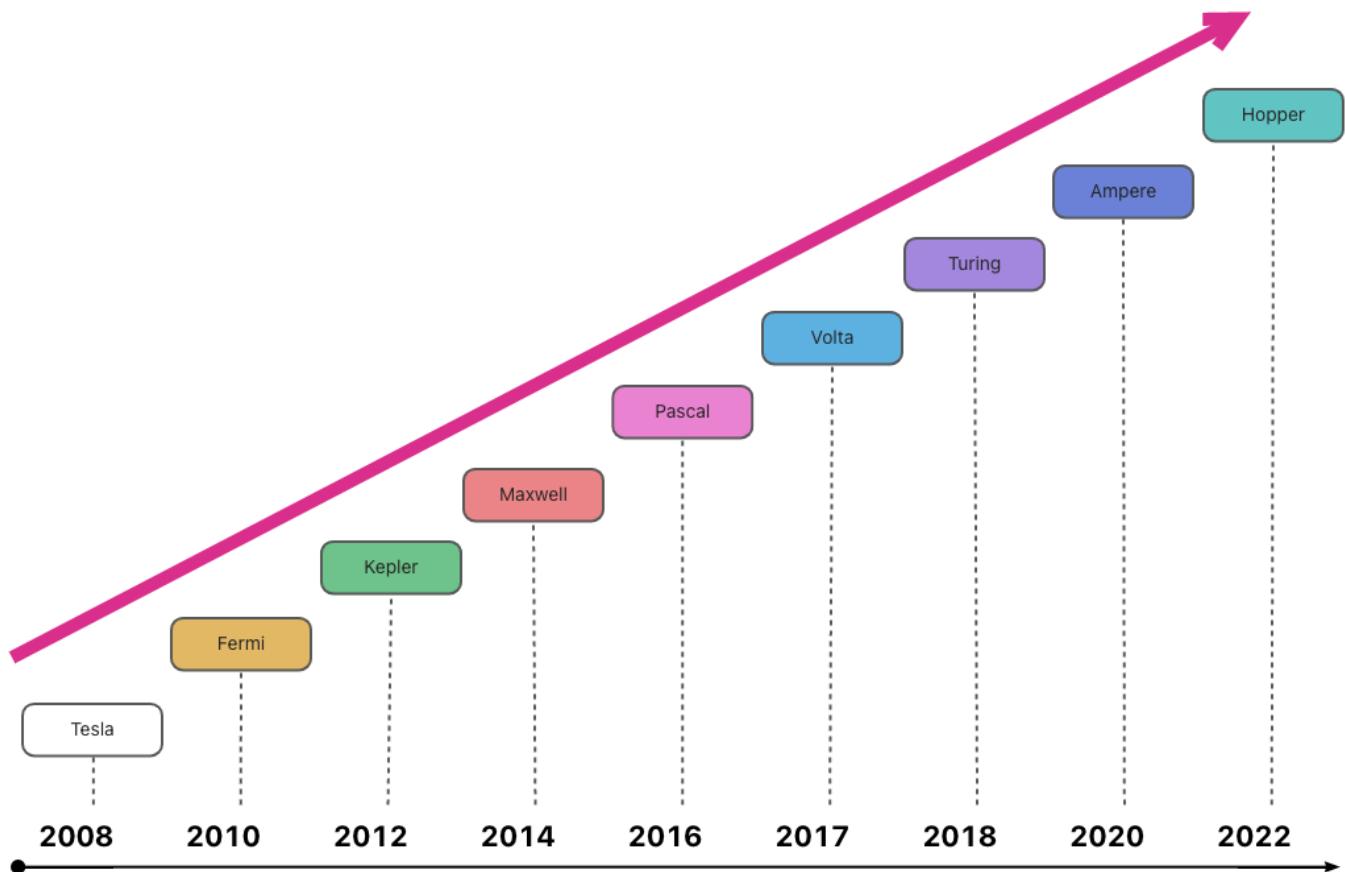
2010 Fermi	Fermi是第一个完整的GPU计算架构。 •首款可支持与共享存储结合纯cache层次的GPU架构，支持ECC的GPU架构。 •512个accelerator cores即所谓CUDA cores (包含ALU和FPU)，16个SM，每个SM包含32个CUDA core
2012 Kepler	Kepler相较于Fermi更快，效率更高，性能更好。 •15个SM，192个单精度CUDA cores，64个双精度单元。 •Kepler图形架构在极大提升游戏性能的同时，又在很大程度上降低了能耗。Kepler基于28纳米制造工艺
2014 Maxwell	紧随 Kepler 之后，Maxwell 是 NVIDIA 的第十代 GPU 架构。这一架构是下一代游戏体验的引擎，可解决视觉计算领域中最复杂的光照和图形难题。
2016 Pascal	适用于大数据工作负载的采用 HBM2 的 CoWoS 技术 •16 纳米 FINFET 工艺，在深度学习方面，由 Pascal 支持的系统的神经网络训练性能提高了12倍。
2017 Volta	•640 个 TENSOR 内核，巨大的性能飞跃；Volta 配备 640 个 Tensor 内核，可提供每秒超过 100 万亿次 (TFLOPS) 的深度学习性能。 •它将 CUDA 内核和 Tensor 内核搭配使用，在 GPU 中提供人工智能超级计算机的性能。
2018 Turing	•用于 AI 加速的 TENSOR CORE，用于实时光线追踪的 RT CORE •利用多达 4608 个 CUDA 核心及软件开发套件 (SDK) 创建复杂的模拟 •Turing 架构能够借助增强的图形管线和全新可编程着色技术显著提高光栅性能
2020 Ampere	基于Ampere GPU架构，基于TSMC 7nm制程；NVIDIA第一个统一了数据分析，训练和推理的弹性多实例GPU。 基于TF32的第三代张量核，NVLink3，结构稀疏性等特性

英伟达的GPU架构演进从2008年到2022年的14年间，进行了9次大的微架构更新，架构代号均以科学家名字来命名，以下是各阶段的架构参数及对表。

架构代号	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Hopper
中文代号	费米	开普勒	麦克斯韦	帕斯卡	伏特	图灵	安培	赫柏
时间	2010	2012	2014	2016	2017	2018	2020	2022

核心参数	16 个 SM,	15个	16个	Pascal架构	80个SM,	TU102核心	A100有	H100 132
	每个SM包括	SMx, 每	SMM, 每	构有	每个SM里	72个	108 SMs	SM
	32 Cuda Cores, 共计	个SMX包	个SM包括	GP100、GP102	32个FP64	SM, SM	每个SM	每个SM
	512 Cuda Cores	括 192个单精度+Cuda cores;	4个处理块, 每个处理块包括32个CUDA内核+8个LD/ST Unit+8个SFU	GP100有60个SM	64个INT32	全新设计, 每个SM里	64 个FP32	128个FP32
				每个SM包括	64个FP32	64个INT32	64 个INT32	64 个INT32
				8个Tensor core	64个FP32	32 个FP64	64个FP64	4个Tensor core
					8个Tensor core	4个Tensor core		
特点\优势	首个完整 GPU计算架构, 支持与共享存储结合纯Cache 层次的GPU 架构, 支持 ECC的GPU 架构	游戏性能大幅提升首次支持 GPU Direct 技术	相比Kpler 的每组SM 单元192个 GPU 减少到了每组128 个, 但是每个SMM 单元拥有更多的逻辑控制电路	NVLink一代, 双向互联带宽P100有56 个SM HBM	Nvlink 2.0 Core 1.0 RT Core 学习和AI 运算	Tensor Core 3.0 RT Core 1.0 2.0	Tensor Core 3.0 RT Core Nvlink 3.0 结构稀疏性 MIG 2.0 MIG 1.0	Tensor Core 4.0 Nvlink 4.0 矩阵 性
纳米制程	40/28nm	28nm	28nm	16nm	12nm	12nm	7nm	4nm
	30亿晶体管	71亿晶体管	80亿晶体管	153亿晶体管	211亿晶体管	186亿晶体管	283亿晶体管	800 亿个晶体管

代表型号	Quadro	K80	M5000	P100	V100	T4	A100、	H100
	7000	K40M	M4000	GTX 1080	TiTan V	2080TI	A30	
				P6000		RTX 5000	3090	

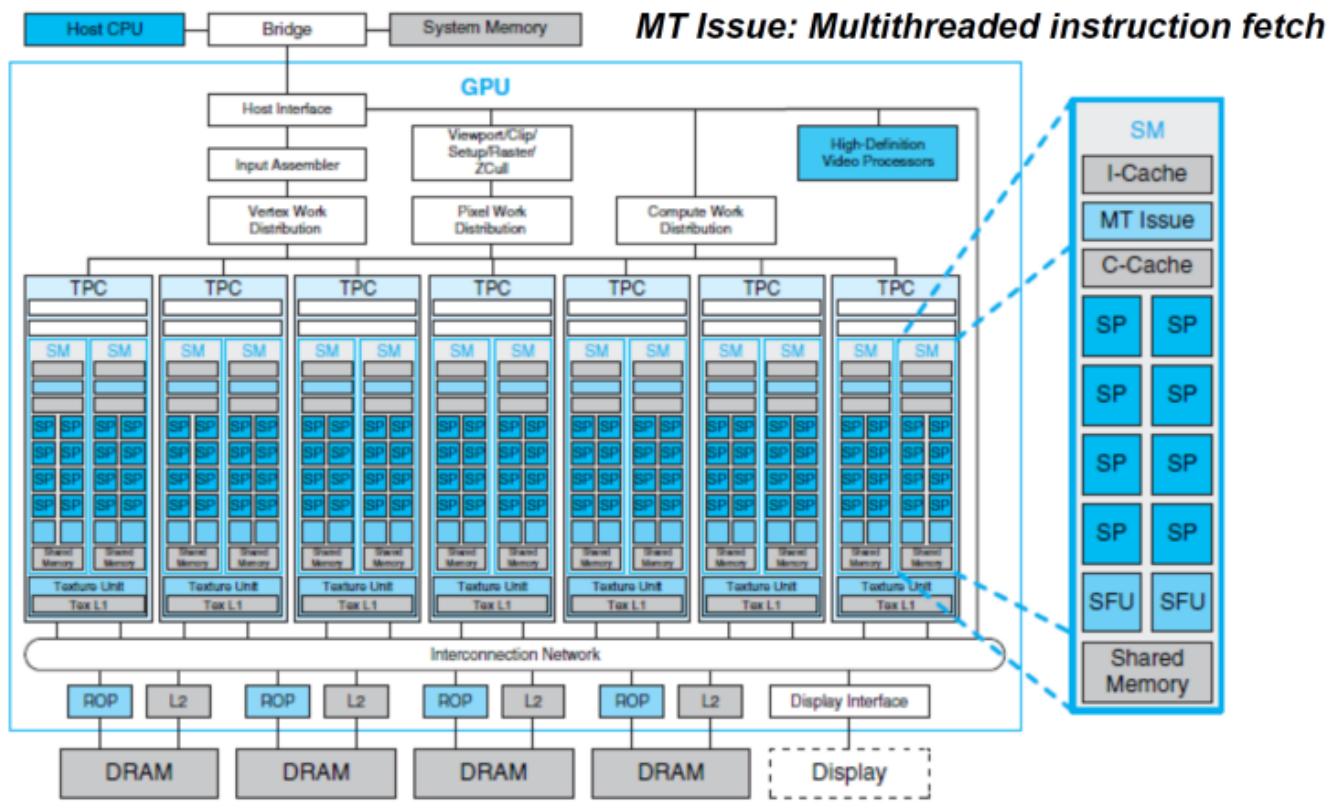


下面我们将对这9代架构分别进行简单的介绍，以方便理解每代架构的特点。

## 2、各架构特点

### 2.1、NVidia Tesla架构

# NVIDIA Tesla Architecture



**TPCs:** Texture/Processor Clusters  
**SMs:** Stream Multiprocessors

**SPs:** Streaming Processors  
**SFU:** Special Function Unit  
(4 floating-point multipliers)

Tesla微观架构图

Tesla微观架构总览图如上。下面将阐述它的特性和概念：

- 拥有7组TPC (Texture/Processor Cluster, 纹理处理簇)
- 每个TPC有两组SM (Stream Multiprocessor, 流多处理器)
- 每个SM包含：
  - 6个SP (Streaming Processor, 流处理器)
  - 2个SFU (Special Function Unit, 特殊函数单元)
  - L1缓存、MT Issue (多线程指令获取) 、C-Cache (常量缓存) 、共享内存
- 除了TPC核心单元，还有与显存、CPU、系统内存交互的各种部件。

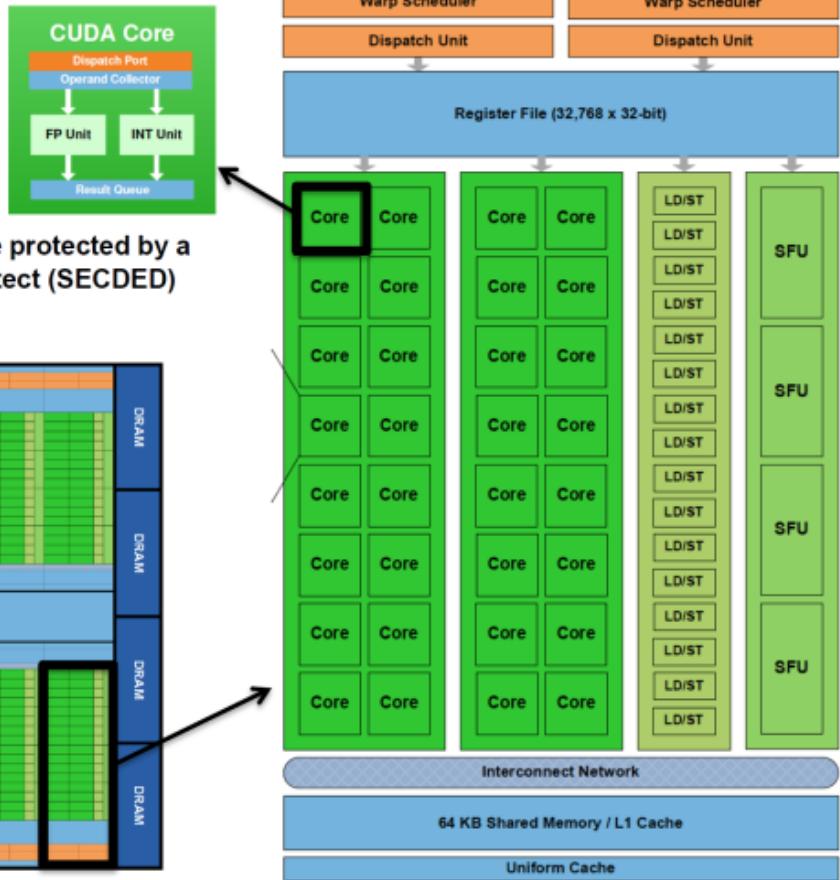
## 2.2、NVidia Fermi架构

# NVIDIA Fermi Architecture

Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).

More cores per multiprocessors and faster arithmetic operations

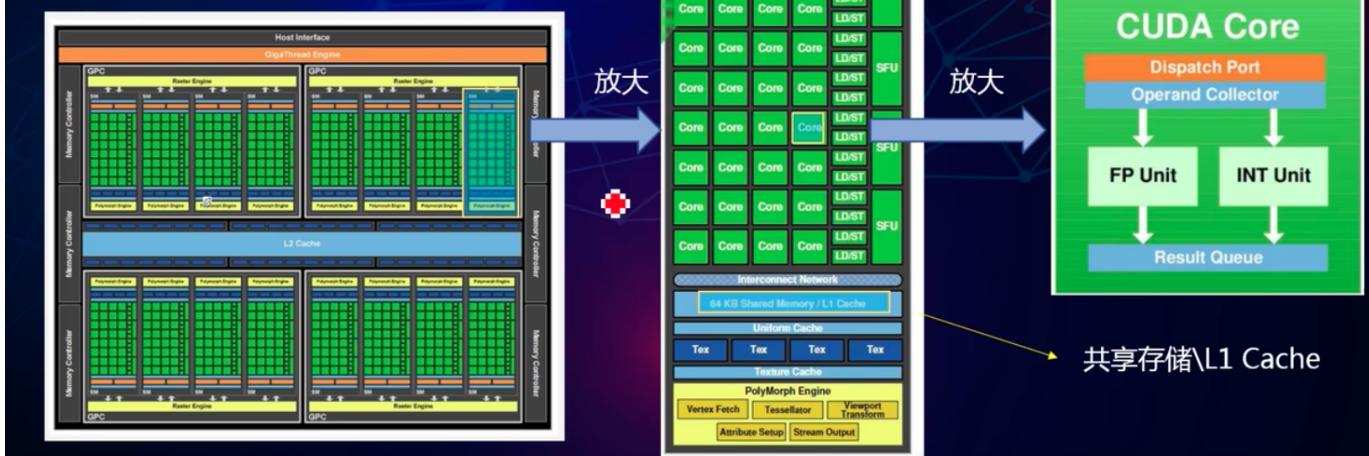
**Memory Protection Support:** Memory are protected by a Single - Error Correct Double - Error Detect (SECDED) ECC code



Fermi微观架构图

## Fermi架构的GPU

- 下图中可以看到，Fermi架构的GPU最大可支持16个SMs，每个SM带32个Cuda Cores，一共512个Cuda Cores。这些数量不是固定的，和具体的架构和型号相关。
- 整个GPU有多个GPC(图形处理集群)，单个GPC包含一个光栅引擎(Raster Engine)，四个SM(流式多处理器)，GPC可以被认为是一个独立的GPU。所有从Fermi开始的NVIDIA GPU，都有GPC。



Fermi架构如上图，它的特性如下：

- 拥有16个SM
- 每个SM：
  - 2个Warp (线程束)
  - 两组共32个Core
  - 16组加载存储单元 (LD/ST)
  - 4个特殊函数单元 (SFU)
- 每个Warp：
  - 16个Core
  - Warp编排器 (Warp Scheduler)
  - 分发单元 (Dispatch Unit)
- 每个Core：
  - 1个FPU (浮点数单元)
  - 1个ALU (逻辑运算单元)

## 2.3、NVidia Kepler架构

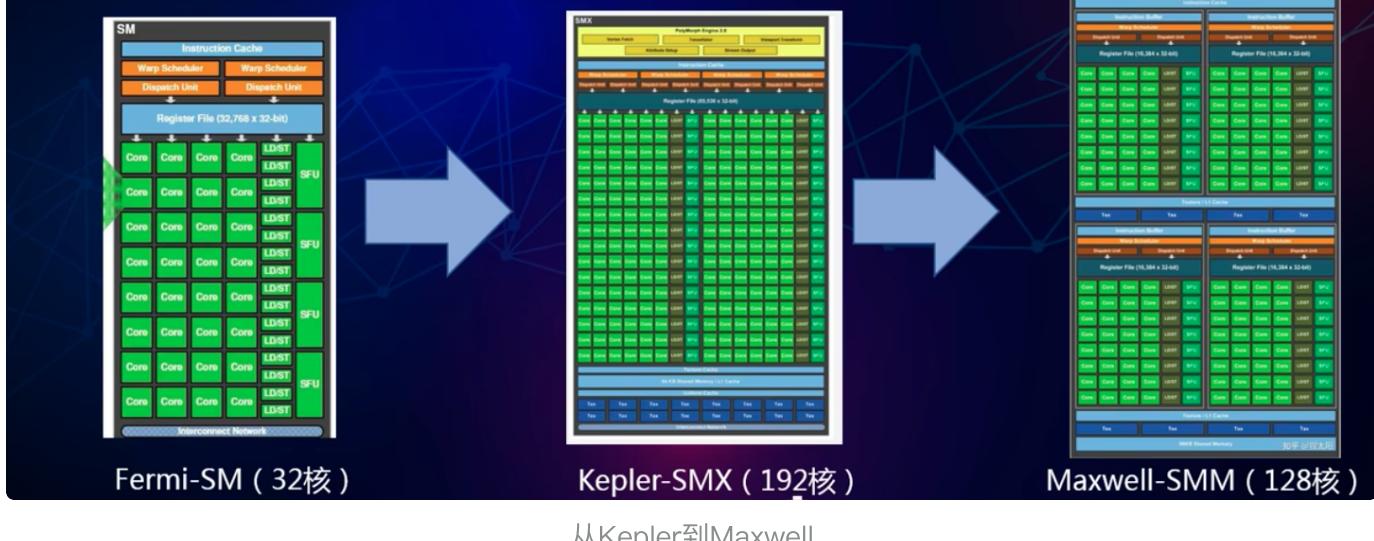
## SMX



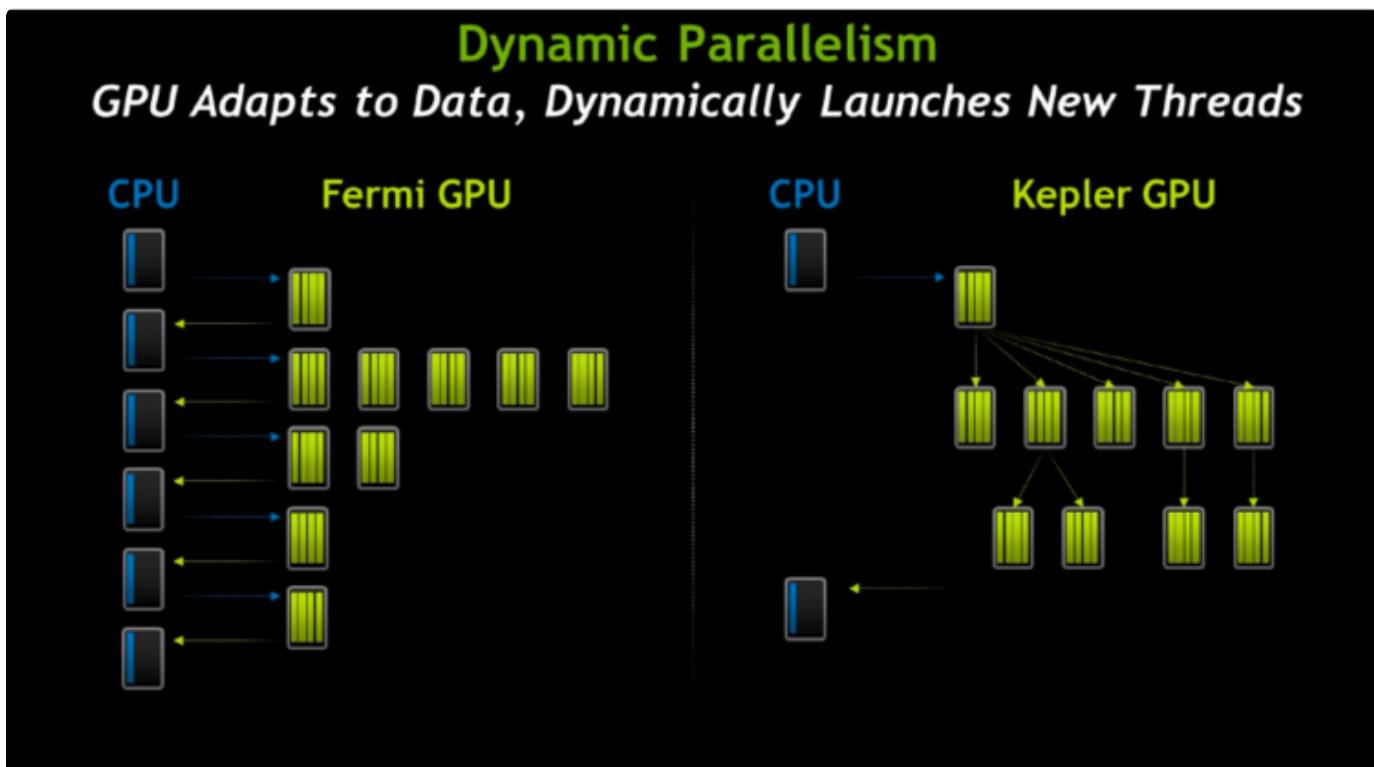
Kepler微观架构图

## Kepler和Maxwell架构

- 开普勒与上一代的SM整体结构基本一致的，只是在SM你增加了更多的运算单元，其他部分变动不大。
- Maxwell相比于Kepler的CUDA核心数减少，但是每个SMM单元拥有更多的逻辑控制电路，便于精准控制。



Kepler除了在硬件有了提升，有了更多处理单元之外，还将SM升级到了SMX。SMX是改进的架构，支持动态创建渲染线程（下图），以降低延迟。



动态渲染线程

## 2.4、Nvidia Maxwell架构



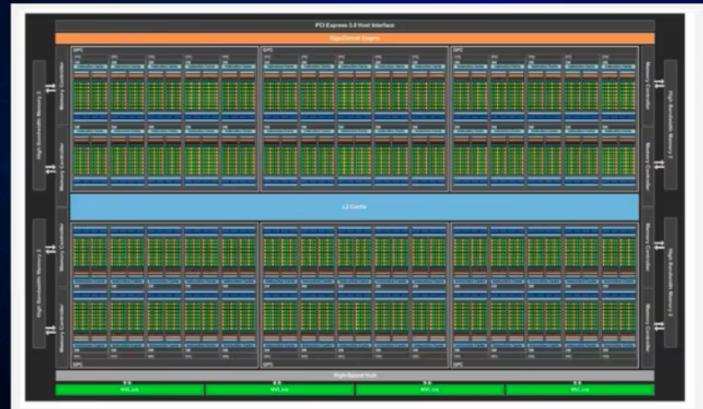
Maxwell微观架构图

采用了Maxwell的GM204，拥有4个GPC，每个GPC有4个SM，对比Tesla架构来说，在处理单元上有了很大的提升。

## 2.5、NVidia Pascal架构

## Pascal架构的GPU

- Pascal架构会有不同的核心，为GP100、GP102两种大核心：
- GP100：3840个CUDA核心，60组SM单元、GP102：3584个CUDA核心，28组SM单元；单个SM只有64个FP32 Cuda Cores，相比 Maxwell 的128和Kepler的192，这个数量要少很多，并且64个Cuda Cores分为了两个区块。



- CUDA内核总数从Maxwell时代的每组SM单元128个减少到了每组64个，又重新增加了DP双精度运算单元。
- 制程工艺升级到了16nm，性能大幅提升，功耗持平。

## 2.6、Nvidia Volta架构

### 四、Volta (伏特) 微架构

lt\_server技术分享 bilibili

## Volta架构的GPU

- 和Pascal类似，直接拆了4个区块，每个区块多配了一个L0指令缓存：
- 单个区块还多了两个名为Tensor Core的单元。
- 原有的CUDA Core被拆成了FP32 Cuda Core和INT32 Cuda Core，这意味着可以同时执行FP32和INT32的操作

### 最大的升级点是：Tensor Core

- Tensor核心是专为执行张量或矩阵运算而设计的专用执行单元，而这些运算是深度学习所采用的核心计算函数，它能够大幅加速深度学习神经网络训练和推理运算核心的矩阵计算。每个TensorCore只做如下操作： $D = A \times B + C$

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$



## 2.7、NVidia Turing架构



采纳了Turing架构的TU102 GPU

## 图灵架构的GPU

- 2018年NVIDIA发布了Turing架构，整体和Volta变化不大。
- 比较重要的是增加了一个RT Core，全名是Ray Tracing Core。光线追踪核心，主要面向于游戏或者仿真用的比如2080Ti光追游戏应用。
- Turing里的Tensor Core增加了对INT8/INT4/Binary的支持。但是去掉了FP64的支持。

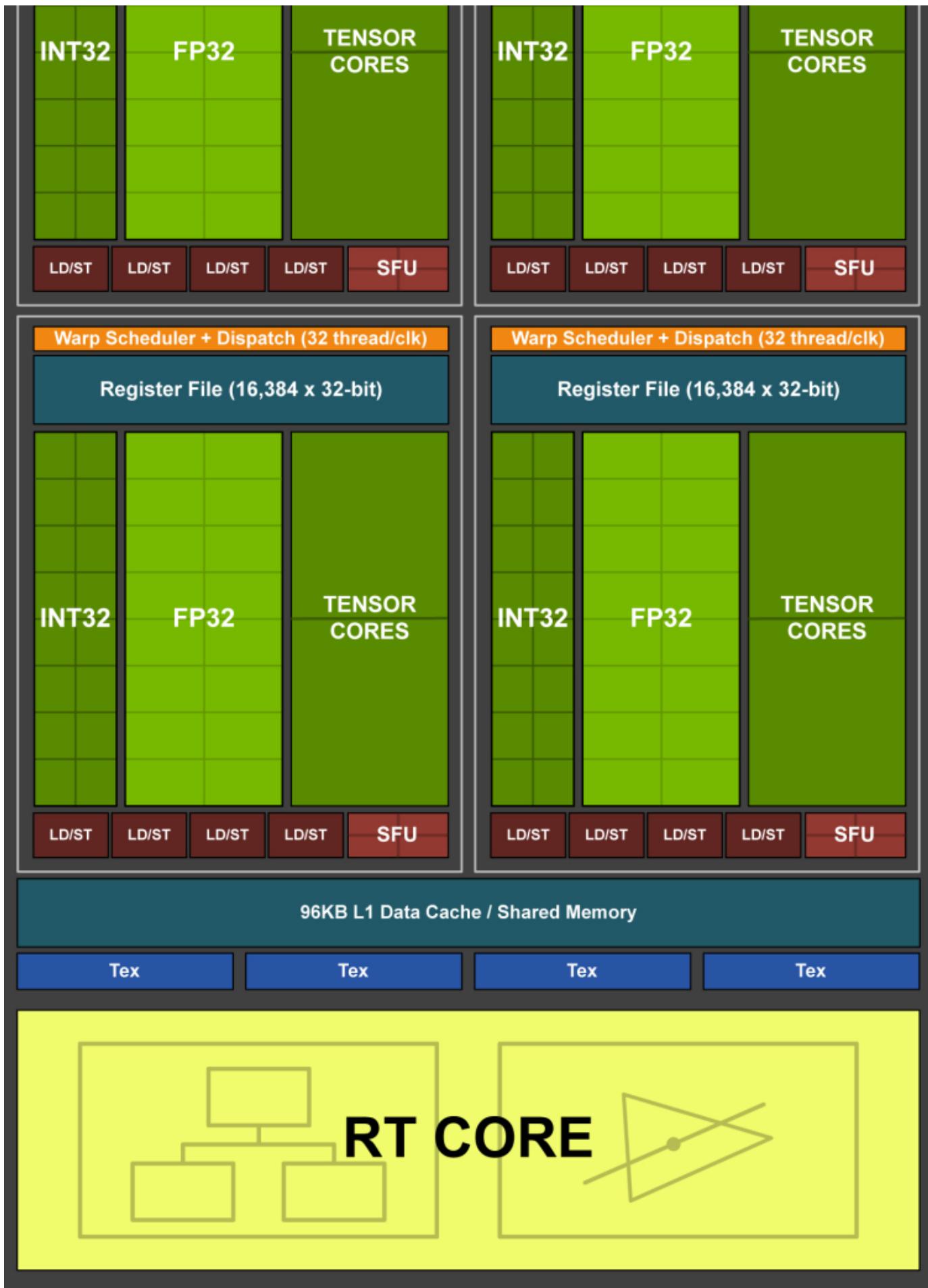


它的特点如下：

- 6 GPC (图形处理簇)
- 36 TPC (纹理处理簇)
- 72 SM (流多处理器)
- 每个GPC有6个TPC，每个TPC有2个SM
- 4,608 CUDA核
- 72 RT核
- 576 Tensor核
- 288 纹理单元
- 12x32位 GDDR6内存控制器 (共384位)

## SM

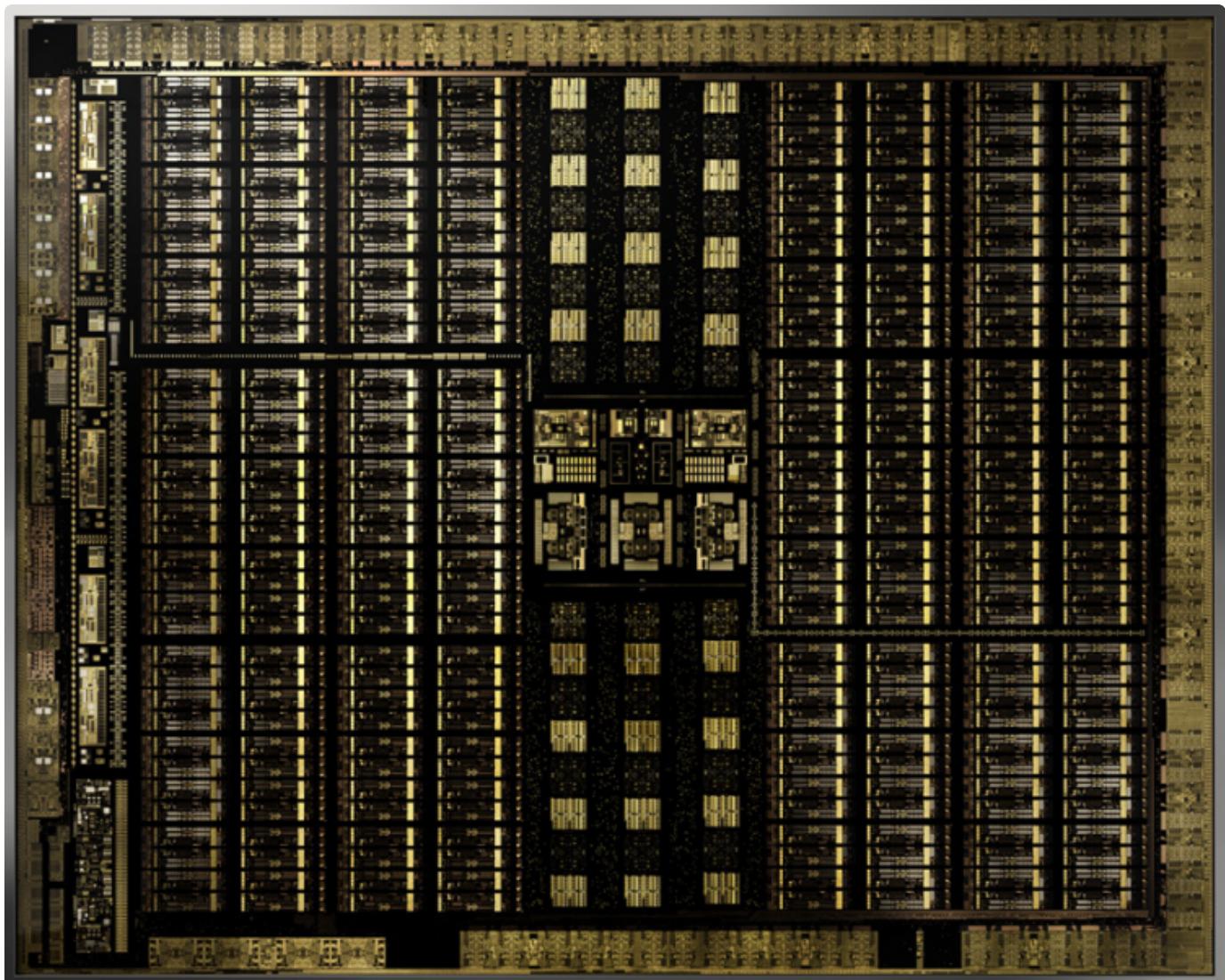




### 单个SM的结构图

每个SM包含：

- 64 CUDA核
- 8 Tensor核
- 256 KB寄存器文件



TU102 GPU芯片实物图

## 2.8、Nvidia Ampere架构

## Ampere架构的GPU

- 2020年NVIDIA发布了Ampere架构，又细分**GA100, GA102, GA104, GA106**。
- 除了Volta中的FP16以及在Turing中的INT8/INT4/Binary，这个版本加入了TF32, BF16, FP64的支持。
- 二代Tensor Core、多实例GPU(MIG)、第三代NVIDIA NVLink、结构化稀疏等新技术。



## 2.9、Nvidia Hopper架构

### Hopper架构的GPU

- 集成超过800亿个晶体管(台积电4nm工艺)
- Transformer Engine
- 第二代MIG：多实例GPU(Multi-Instance GPU)
- NVIDIA机密计算(Confidential Computing)
- 第四代NVLink
- 全新DPX指令

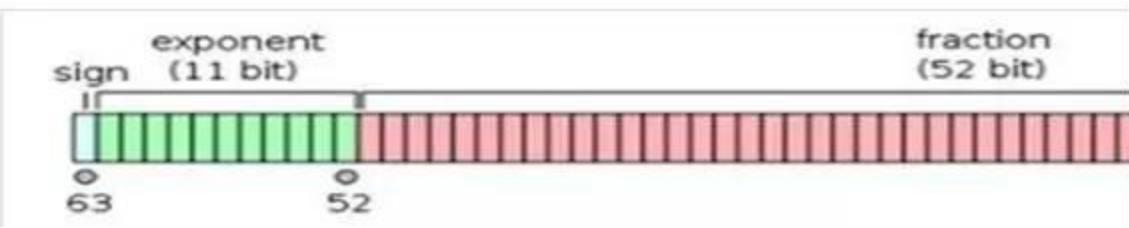


## 3、GPU核心参数的含义

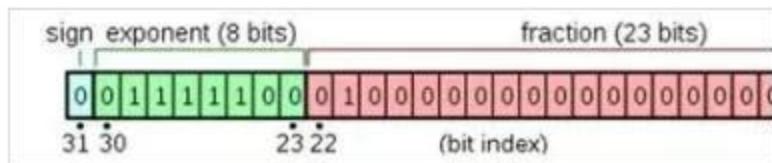
- 显存容量
  - 显存容量决定着显存临时存储数据的多少，大显存能减少读取数据的次数，降低延迟；
  - 其主要功能就是暂时储存GPU要处理的数据和处理完毕的数据。显存容量大小决定了GPU

能够加载的数据量大小。(在显存已经可以满足客户业务的情况下，提升显存不会对业务性能带来大的提升。在深度学习、机器学习的训练场景，显存的大小决定了一次能够加载训练数据的量，在大规模训练时，显存会显得比较重要。

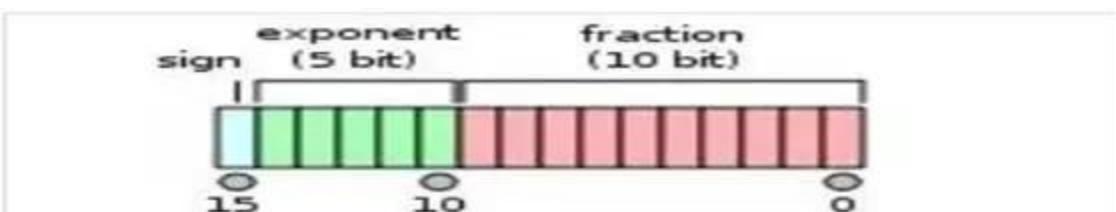
- 显存带宽
  - 指显示芯片与显存之间的数据传输速率，单位是字节/秒，显卡的显存是由一块块的显存芯片构成的，显存总位宽同样也是由显存颗粒的位宽组成，显存带宽 = 显存频率×显存位宽/8。
- 显存频率
  - 一定程度上反应着该显存的速度，以MHz(兆赫兹)为单位,显存频率随着显存的类型、性能的不同而不同。显存频率和位宽决定显存带宽。
- 显存位宽
  - 指显示芯片与显存之间的数据传输速率，它以字节/秒为单位。显存带宽是决定显卡性能和速度最重要的因素之一。
  - 显存在一个时钟周期内所能传送数据的位数,位数越大则瞬间所能传输的数据量越大,这是显存的重要参数之一
- CUDA Core
  - 英伟达 GPU 的参数中，最常看到的核心类型。Nvidia 通常用最小的运算单元表示自己的运算能力，CUDA Core 指的是一个执行基础运算的处理元件。我们所说的 CUDA Core 数量，通常对应的是 FP32 计算单元的数量。
  - CUDA核心数量决定了GPU并行处理的能力，在深度学习、机器学习等并行计算类业务下，CUDA核心多意味着性能好一些。
- Tensor Core
  - 张量核心，核心特别大，用于机器学习加速，它可以把整个矩阵都载入寄存器中批量运算，实现十几倍的效率提升，从 Volta 架构发布以来，奠定了英伟达在 AI 训练的领军地位，每次升级都有新支持的数据类型。
- RT Cores
  - 光线追踪核心，正常数据中心级的 GPU 核心是没有 RTCore 的，主要是消费级显卡才为光线追踪运算添加了RT Cores，考虑到芯片的空间有限，每个 SM 里面只有1 个光追核心（为此还砍掉了大部分的 FP64 ）。可以极大地提升了游戏渲染效率。
- FP64
  - 双精度，通常应用于HPC相关领域。
  - 双精度浮点数采用8个字节也就是64位二进制来表达一个数字，1位符号，11位指数，52位小数，有效位数为16位。



- FP32
  - 单精度，训练场景的数据格式。
  - 单精度的浮点数中采用4个字节也就是32位二进制来表达一个数字，1位符号，8位指数，23位小数，有效位数为7位。



- TF32
  - 从A100开始NVIDIA提出的新数据格式，比FP32精度低，比FP16精度高，主要用于深度学习训练，理论上比FP32+FP16混合精度效果更优。
- BF16
  - Intel x86、ARM采用的，可用于训练和推理。
- FP16
  - 主要用于推理。
  - 半精度浮点数采用2个字节也就是16位二进制来表达一个数字，1位符号、5位指数、10位小数，有效位数为3位。



- INT8
  - 主要用于推理。
- SM (Streaming Multiprocessor)
  - 流式多处理器，GPU的基本单元，每个GPU都由一组SM构成，SM中最重要找结构就是计算核心core。

- TPC (Texture Processor Cluster)

- 纹理处理集群

## 4、CUDA core VS Tensor Core

区别项	CUDA Core	Tensor Core
出现时间	2010年 费米架构	2017年 伏特架构
出现背景	科学计算迅速发展，为了使用GPU的高算力，科学家需要将科学计算任务适配成图形图像任务	AI迅速发展，对矩阵乘法的算力需求不断增大，有厂商提出TPU概念
设计目的	将高并发浮点计算能力暴露给科学计算领域，加速科学计算，占领科学计算市场	定制性争强矩阵计算能力（AI算力），融入CUDA生态，抢占人工智能计算所需要的算力高地
计算任务类型	浮点加、乘、乘加运算	矩阵乘法运算
计算精度	IEEE-754 标准的float精度	强调低精度:fp16/fp8/int8/int4/b1
SM上的装配数目	*128	*4
调度粒度/执行粒度	warp-level/thread-level	warp-level/warp-level

## 5、英伟达GPU参数速查表

架构	系列	型号	CUD A核	Tensor Core	RT Core	显存	PCIE 版本	制 程	功耗
安培	Tesla	A100 PCIe	6912	432	-	40GB	PCIE 4.0	7nm m	250W
安培	Tesla	A100 PCIe 80 GB	6912	432	-	80GB	PCIE 4.0	7nm m	250W
安培	Tesla	A100 SXM4 40 GB	6912	432	-	40GB	PCIE 4.0	7nm m	400W

安培	Tesla	A100 SXM4 80 GB	6912	432	-	80GB	PCIE 4.0	7n m	400W
安培	Tesla	A10 PCIe	9216	288	72	24GB	PCIE 4.0	7n m	150W
安培	Tesla	A16 PCIe	1280	40x4 x4	10x4	16 GB x4	PCIE 4.0	7n m	250W
安培	Tesla	A30 PCIe	358	224 4	-	24 GB	PCIE 4.0	7n m	165W
安培	Tesla	A40 PCIe	1075	336 2	84	48 GB	PCIE 4.0	7n m	300W
安培	Quadro	A2	1280	40	10	16 GB	PCIE 4.0	7n m	60W
安培	Quadro	RTX A2000	332	104 8	26	6GB	PCIE 4.0	7n m	70W
安培	Quadro	RTX A2000 12 GB	332	104 8	26	12GB	PCIE 4.0	7n m	70W
安培	Quadro	RTX A4000	6144	192	48	16 GB	PCIE 4.0	7n m	140W
安培	Quadro	RTX A4500	7168	224	56	20 GB	PCIE 4.0	7n m	200W
安培	Quadro	RTX A5000	8192	256	64	24 GB	PCIE 4.0	7n m	230W
安培	Quadro	RTX A5500	1024	320 0	80	24 GB	PCIE 4.0	7n m	230W
安培	Quadro	RTX A6000	1075	336 2	84	48 GB	PCIE 4.0	7n m	300W

安培	GeForce	GeForce RTX 3050	4	230	72	18	4 GB	PCIE	7n	90W
安培	GeForce	GeForce RTX 3050	8	256	80	20	8GB	PCIE	7n	130W
安培	GeForce	GeForce RTX 3060	4	358	112	28	12GB	PCIE	7n	170W
安培	GeForce	GeForce RTX 3060	4	486	152	38	8GB	PCIE	7n	200W
安培	GeForce	GeForce RTX 3070	8	588	184	46	8 GB	PCIE	7n	220W
安培	GeForce	GeForce RTX 3070	Ti	6144	192	48	8 GB	PCIE	7n	290W
安培	GeForce	GeForce RTX 3070	Ti 16 GB	6144	192	48	16 GB	PCIE	7n	290W
安培	GeForce	GeForce RTX 3080	12 GB	8704	272	68	10GB	PCIE	7n	320W
安培	GeForce	GeForce RTX 3080	0	896	280	70	12 GB	PCIE	7n	350W
安培	GeForce	GeForce RTX 3080	Ti	1024	320	80	12 GB	PCIE	7n	350W
安培	GeForce	GeForce RTX 3080	Ti 20 GB	1024	320	80	20 GB	PCIE	7n	350W
安培	GeForce	GeForce RTX 3090	6	1049	328	82	24 GB	PCIE	7n	350W
安培	GeForce	GeForce RTX 3090	2	1075	336	84	24 GB	PCIE	7n	450W

图灵	Tesla	Tesla T4	2560	320	40	16 GB	PCIe 3.0	12 nm	70W
图灵	GeForce	GeForce RTX 2060	1920	240	30	6 GB	PCIe 3.0	12 nm	160W
图灵	GeForce	GeForce RTX 2060 12 GB	2176	272	34	12 GB	PCIe 3.0	12 nm	184W
图灵	GeForce	GeForce RTX 2060 TU104	1920	240	30	16 GB	PCIe 3.0	12 nm	160W
图灵	GeForce	GeForce RTX 2070	2304	288	36	8 GB	PCIe 3.0	12 nm	175W
图灵	GeForce	GeForce RTX 2070 SUPER	2560	320	40	8 GB	PCIe 3.0	12 nm	215W
图灵	GeForce	GeForce RTX 2080	2944	368	46	8 GB	PCIe 3.0	12 nm	215W
图灵	GeForce	GeForce RTX 2080 SUPER	3072	384	48	8 GB	PCIe 3.0	12 nm	250W
图灵	GeForce	GeForce RTX 2080 Ti	4352	544	68	11 GB	PCIe 3.0	12 nm	250W

图灵	GeForce	TITAN RTX	4608	576	72	24 GB	PCIe 3.0	12 nm	280W
图灵	GeForce	GeForce GTX 1630	512	–	–	4 GB	PCIe 3.0	12 nm	75W
图灵	GeForce	GeForce GTX 1650	896	–	–	4 GB	PCIe 3.0	12 nm	75W
图灵	GeForce	GeForce GTX 1650 SUPER	1280	–	–	4 GB	PCIe 3.0	12 nm	100W
图灵	GeForce	GeForce GTX 1660	1408	–	–	6 GB	PCIe 3.0	12 nm	120W
图灵	GeForce	GeForce GTX 1660 SUPER	1408	–	–	6 GB	PCIe 3.0	12 nm	125W
图灵	GeForce	GeForce GTX 1660 Ti	1536	–	–	6 GB	PCIe 3.0	12 nm	120W
图灵	Quadro	Quadro RTX 4000	2304	288	36	8 GB	PCIe 3.0	12 nm	160W
图灵	Quadro	Quadro RTX 5000	3072	384	48	16 GB	PCIe 3.0	12 nm	230W

图灵	Quadro	Quadro RTX 6000	460 8	576	72	24 GB	PCIe 3.0	12 nm	260W
图灵	Quadro	Quadro RTX 6000 Passive	460 8	576	72	24 GB	PCIe 3.0	12 nm	260W
图灵	Quadro	Quadro RTX 8000	460 8	576	72	48 GB	PCIe 3.0	12 nm	260W
图灵	Quadro	Quadro RTX 8000 Passive	460 8	576	72	48 GB	PCIe 3.0	12 nm	260W
图灵	Quadro	T1000	896	–	–	4 GB	PCIe 3.0	12 nm	50W
图灵	Quadro	T1000 8 GB	896	–	–	8 GB	PCIe 3.0	12 nm	50W
图灵	Quadro	T400	384	–	–	2 GB	PCIe 3.0	12 nm	30W
图灵	Quadro	T400 4 GB	384	–	–	4 GB	PCIe 3.0	12 nm	30W
图灵	Quadro	T600	640	–	–	4 GB	PCIe 3.0	12 nm	40W

伏特	Tesla	Tesla V100 PCIe 16 GB	5120	640	–	16 GB	PCIe 3.0	12 nm	300W
伏特	Tesla	Tesla V100 PCIe 32 GB	5120	640	–	32 GB	PCIe 3.0	12 nm	250W
伏特	Tesla	Tesla V100 SXM2 16 GB	5120	640	–	16 GB	PCIe 3.0	12 nm	250W
伏特	Tesla	Tesla V100 SXM2 32 GB	5120	640	–	32 GB	PCIe 3.0	12 nm	250W
伏特	Tesla	Tesla V100 SXM3 32 GB	5120	640	–	32 GB	PCIe 3.0	12 nm	250W
伏特	Tesla	Tesla V100S PCIe 32 GB	5120	640	–	32 GB	PCIe 3.0	12 nm	250W
伏特	GeForce	TITAN V	5120	640	–	12 GB	PCIe 3.0	12 nm	250W
帕斯卡	Tesla	Tesla P10	3840	–	–	24 GB	PCIe 3.0	16 nm	250W
帕斯卡	Tesla	Tesla P100 PCIe 12 GB	3584	–	–	12 GB	PCIe 3.0	16 nm	250W

帕斯卡	Tesla	Tesla P100 DGXS	358 4	–	–	16 GB 3.0	PCIe 3.0	16 n m	300W
帕斯卡	Tesla	Tesla P100 PCIe 16 GB	358 4	–	–	16 GB 3.0	PCIe 3.0	16 n m	250W
帕斯卡	Tesla	Tesla P100 SXM2	358 4	–	–	16 GB 3.0	PCIe 3.0	16 n m	300W
帕斯卡	Tesla	Tesla P4	256 0	–	–	8 GB 3.0	PCIe 3.0	16 n m	75W
帕斯卡	Tesla	Tesla P40	384 0	–	–	24 GB 3.0	PCIe 3.0	16 n m	250W
帕斯卡	Quadro	Quadro P1000	640/ 512	–	–	4 GB 3.0	PCIe 3.0	14 n m	47W
帕斯卡	Quadro	Quadro P2000	1024	–	–	5 GB 3.0	PCIe 3.0	14 n m	75W
帕斯卡	Quadro	Quadro P2200	1280	–	–	5 GB 3.0	PCIe 3.0	14 n m	75W
帕斯卡	Quadro	Quadro P400	256	–	–	2 GB 3.0	PCIe 3.0	14 n m	30W

帕斯卡	Quadro	Quadro P4000	1792	–	–	8 GB	PCIe 3.0	14 nm	105W
帕斯卡	Quadro	Quadro P5000	2560	–	–	16 GB	PCIe 3.0	14 nm	180W
帕斯卡	Quadro	Quadro P600	384	–	–	2 GB	PCIe 3.0	14 nm	40W
帕斯卡	Quadro	Quadro P6000	3840	–	–	24 GB	PCIe 3.0	14 nm	250W
帕斯卡	Quadro	Quadro P620	512	–	–	2 GB	PCIe 3.0	14 nm	40W
帕斯卡	GeForce	GeForce GT 1010	256	–	–	2 GB	PCIe 3.0	14 nm	30W
帕斯卡	GeForce	GeForce GT 1030	384	–	–	2 GB	PCIe 3.0	14 nm	30W
帕斯卡	GeForce	GeForce GTX 1050	640	–	–	2 GB	PCIe 3.0	14 nm	75W
帕斯卡	GeForce	GeForce GTX 1050 Ti	768	–	–	4 GB	PCIe 3.0	14 nm	75W

帕斯卡	GeForce	GeForce GTX 1060 3 GB	1152	–	–	3 GB	PCIe 3.0	16 nm	120W
帕斯卡	GeForce	GeForce GTX 1060 5 GB	1280	–	–	5 GB	PCIe 3.0	16 nm	120W
帕斯卡	GeForce	GeForce GTX 1060 6 GB	1280	–	–	6 GB	PCIe 3.0	16 nm	120W
帕斯卡	GeForce	GeForce GTX 1070	1920	–	–	8 GB	PCIe 3.0	16 nm	150W
帕斯卡	GeForce	GeForce GTX 1070 Ti	2432	–	–	8 GB	PCIe 3.0	16 nm	180W
帕斯卡	GeForce	GeForce GTX 1080	2560	–	–	8 GB	PCIe 3.0	16 nm	180W
帕斯卡	GeForce	GeForce GTX 1080 Ti	3584	–	–	11 GB	PCIe 3.0	16 nm	250W
帕斯卡	GeForce	GeForce GTX 1080 Ti 10 GB	3200	–	–	10 GB	PCIe 3.0	16 nm	250W
帕斯卡	GeForce	TITAN X Pascal	3584	–	–	12 GB	PCIe 3.0	16 nm	250W

帕斯卡	GeForce	TITAN Xp	384	0		12 GB	PCIe 3.0	16 nm	250W
-----	---------	----------	-----	---	--	-------	----------	-------	------

## 四、如何理解GPU的底层架构

我们需要结合CPU的特点，与GPU进行对比，以便加强认识，减少理解的难度。

中心处理单元（Central Processing Unit、CPU）一词诞生于 1955 年，已经诞生 70 多年的 CPU 在今天已经是很成熟的技术了，不过它虽然能够很好地处理通用的计算任务，但是因为核心数量的限制在图形领域却远远不如图形处理单元（Graphics Processing Unit、GPU），复杂的图形渲染、全局光照等问题仍然需要 GPU 来解决，而大数据、机器学习和人工智能等技术的发展也推动着 GPU 的演进。

## 1、CPU

更高、更快和更强是人类永恒的追求，在科技上的进步也不例外，CPU 的主要演进方向其实只有一个：消耗最少的能源实现最快的计算速度，无数工程师的工作都是为了实现这个看起来简单的目的。然而在 CPU 已经逐渐成熟的今天，想要提高它的性能需要花费极大的努力，下面简单描述了历史上引入了哪些技术来提高 CPU 的性能。

### 1.1、制程

当讨论 CPU 的发展时，制程（Fabrication Process）是绕不开的关键字，相信不了解计算机的人也都听说过 Intel 处理器 10nm、7nm 的制程，而目前各个 CPU 制造厂商也都有各自的路线图来实现更小的制程，例如台积电准备在 2022 和 2023 年分别实现 3nm 和 2nm 的制造工艺。

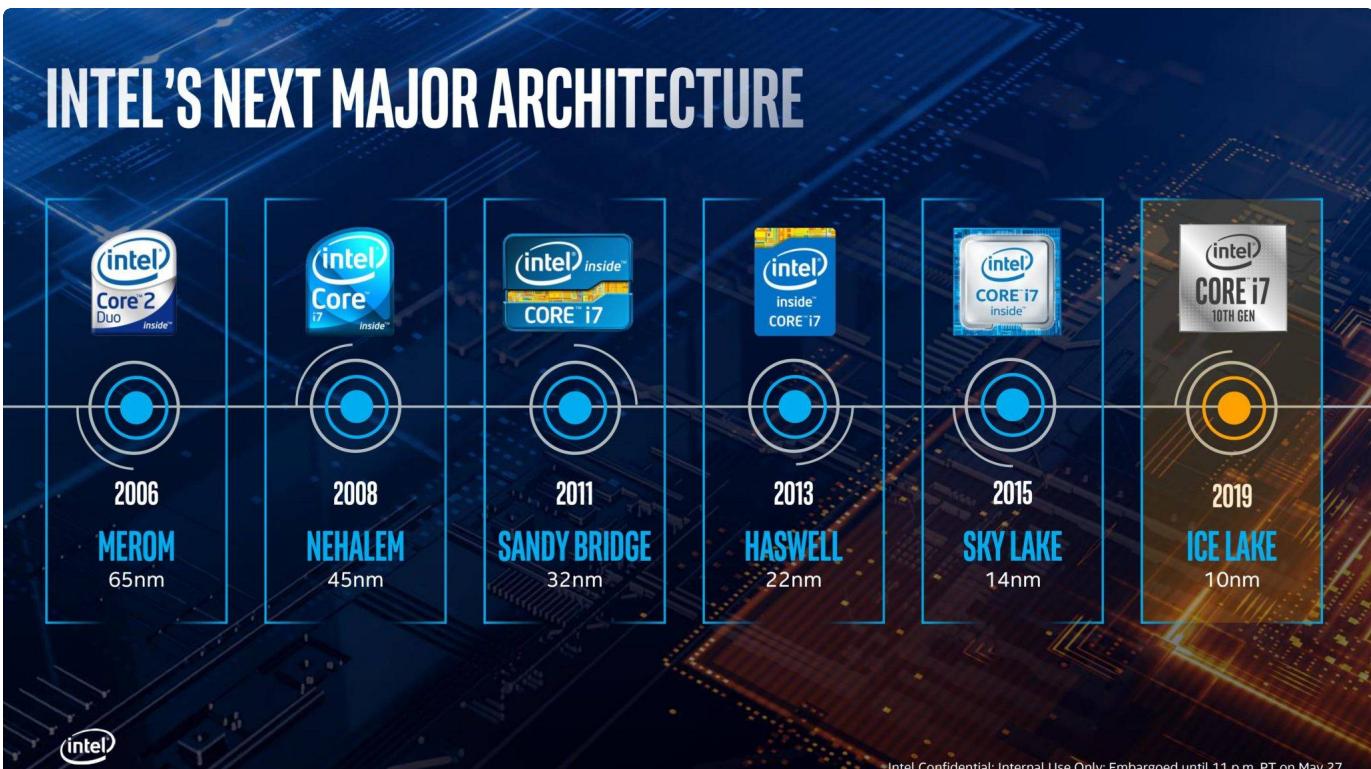


图 2 – Intel CPU 制程

在大多数人眼中，仿佛 CPU 的制程越少就越先进，性能也会越好，但是制程并不是衡量 CPU 性能的标准，最起码制程的演进不会直接提高 CPU 的性能。工艺制程的每次提升，都可以让我们在单位面积内容纳更多的晶体管（Transistor），只有越多的晶体管才意味着越强的性能。

越小的晶体管在开关时消耗的能量越少，既然晶体管需要一些时间充电和放电，那么消耗的能量也就越少，速度也越快，而这也解释了为什么增加 CPU 的电压可以提高它的运行速度。除此之外，更小的晶体管间隔使得信号的传输变得更快，这也能够加快 CPU 的处理速度。

## 1.2、缓存

缓存也是 CPU 的重要组成部分，它能够减少 CPU 访问内存所需要的时间，相信很多人都看过如下所示的表格，可以看到从 CPU 的一级缓存中读取数据大约是主存的 200 倍，哪怕是二级缓存也有将近 15 倍的提升：

Work	Latency
L1 cache reference	0.5 ns
Branch mispredict	5 ns

L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 1K bytes over 1 Gbps network	10,000 ns
Read 4K randomly from SSD*	150,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Read 1 MB sequentially from SSD*	1,000,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

表 1 – 2012 年延迟数字对比

今天的 CPU 一般都包含 L1、L2 和 L3 三级缓存，CPU 访问这些缓存的速度仅次于访问寄存器，虽然缓存的速度很快，但是因为高性能需要保证尽可能靠近 CPU，所以它的成本异常昂贵。Intel 等 CPU 厂商也会通过增加 CPU 缓存的方式提高性能，更大的 CPU 缓存意味着更高的缓存命中率，也意味着更快的速度。

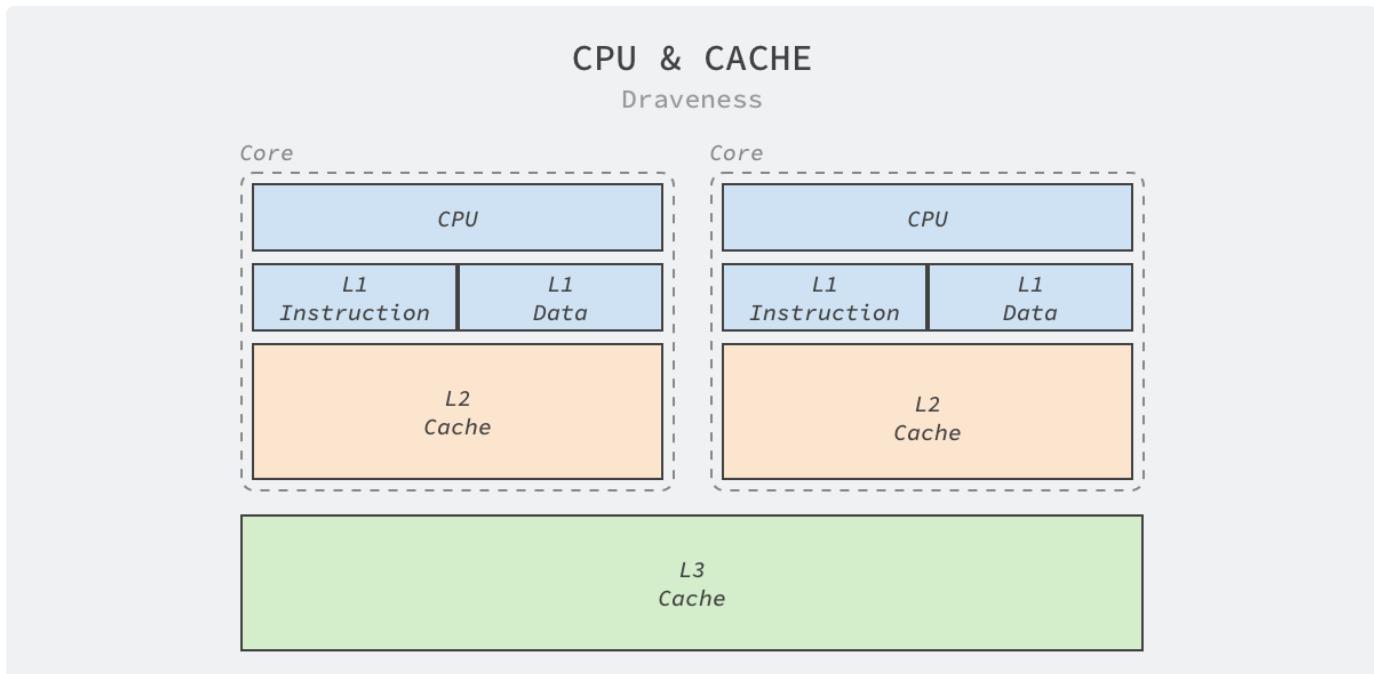


图 3 – CPU 缓存

Intel 的处理器就在过去几十年的时间中不断增加 L1、L2 和 L3 的缓存大小、将 L1 和 L2 缓存集成在 CPU 中以提高访问速度并在 L1 缓存中区分数据缓存和指令缓存以提高缓存的命中率。

今天的 Core i9 处理器每个核心都有 64 KB 的 L1 缓存和 256 KB 的 L2 缓存，所有的 CPU 还会共享 16 MB 的 L3 缓存<sup>7</sup>。

### 1.3、并行计算

多线程编程在今天几乎是工程师的必修课了，主机上越来越多的 CPU 核心让工程师不得不去思考如何才能通过多线程尽可能利用硬件的潜力，很多人可能都认为 CPU 会按照编写的程序串行执行命令，但是真正的现实往往比这复杂得多，早在很多年前嵌入式工程师就开始尝试在单个 CPU 上并行执行指令。

从软件工程师的角度，我们确实可以认为每一条汇编指令都是原子操作，而原子操作意味着该操作要么处于未执行的状态，要么处于已执行的状态，而数据库事务、日志以及并发控制都建立在原子操作上。不过如果再次放大指令的执行过程，我们会发现指令执行的过程并不是原子的：



图 4 – 指令执行的步骤

不同机器架构执行指令的过程会有所差别，上面是经典的精简指令集架构（RISC）中命令执行需要经过的 5 个步骤，其中包括获取指令、解码指令、执行、访问内存以及写回寄存器。

超标量处理器是可以实现指令级别并行的 CPU，它通过向处理器上的其他执行单元派发指令在一个时钟周期内同时执行多条指令<sup>8</sup>，这里的执行单元是 CPU 内的资源，例如算术逻辑单元、浮点数单元等<sup>9</sup>。

超标量设计意味着处理器会在一个时钟周期发出多条指令，该技术往往都与指令流水线一起使用<sup>10</sup>，流水线会将执行拆分成多个步骤，而处理器的不同部分会分别负责这些步骤的处理，例如：因为指令的获取和解码由不同的执行单元处理，所以它们可以并行执行。

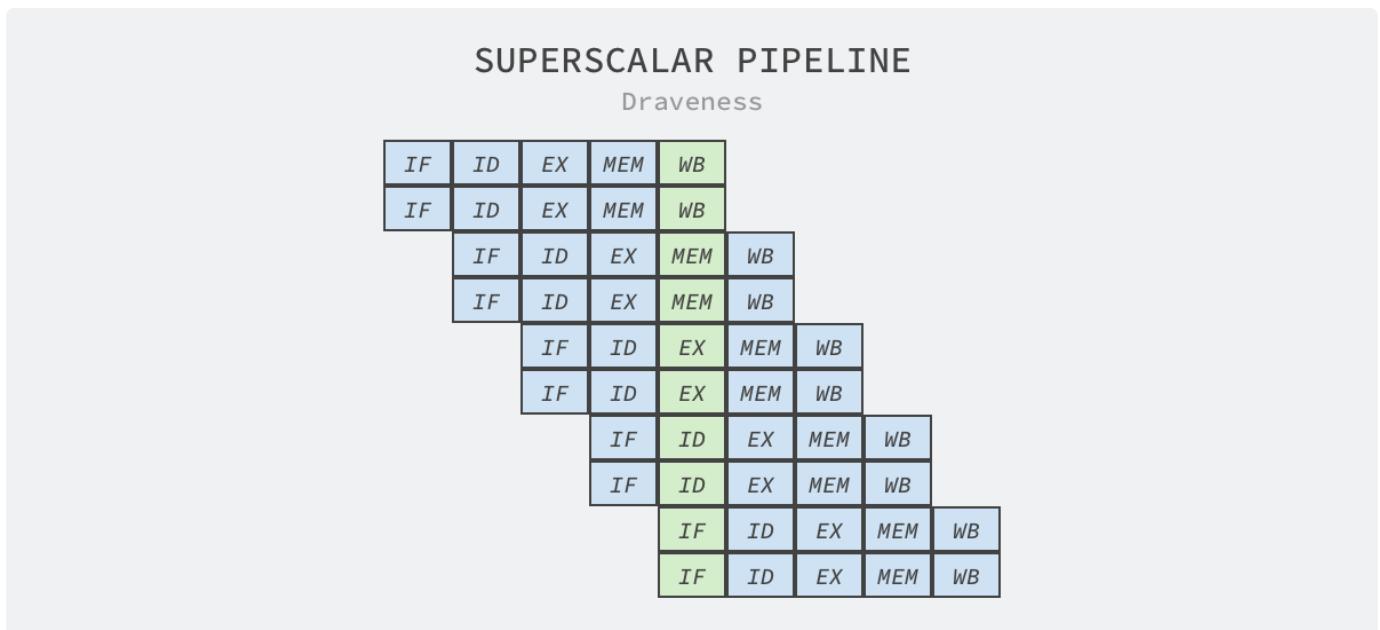


图 5 – 超标量和流水线

除了超标量和流水线技术之外，嵌入式工程师们还引入了乱序执行以及分支预测等更加复杂的技  
术，其中乱序执行也被称作动态执行，因为 CPU 执行指令时需要先将数据加载到寄存器中，所以分析  
CPU 的寄存器操作确定哪些指令可以乱序执行。

## OUT-OF-ORDER EXECUTION

Draveness

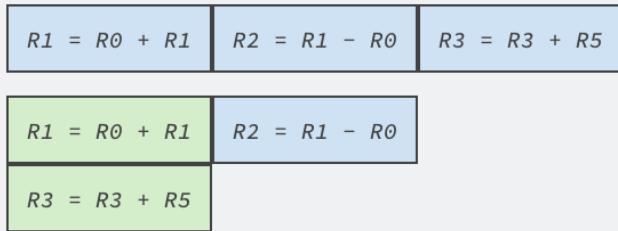


图 6 – 乱序执行

如上图所示，其中包含  $R1 = R0 + R1$ 、 $R2 = R1 - R0$  和  $R3 = R3 + R5$  三条指令，其中第三条指令使用的两个寄存器与前两条无关，所以该指令可以与前两条指令并行执行，也就能减少这段代码执行所需要的时间。

因为分支条件是程序中的常见逻辑，当在 CPU 的执行中引入流水线和乱序执行之后，如果遇到条件分支仍然需要等待分支确定才继续执行后面的代码，那么处理器可能会浪费很多时钟周期等待条件的确定。

在计算机架构中，分支预测器是用来在分支确定前预判的数字电路，在遇到条件跳转指令时，它会预测条件的执行结果并选择分支执行：

- 如果预判正确，可以节约等待所需要的时钟周期，提高 CPU 的利用率；
- 如果预判失败，需要丢弃预判执行的全部或者部分结果，重新执行正确的分支；

因为预判失败需要付出较大的代价，一般在 10 ~ 20 个时钟周期之间，所以如何提高分支预测器的准确率成为了比较重要的课题，常见的实现包括静态分支预测、动态分支预测和随机分支预测等。

上面的这些指令级并行仅仅存在于实现细节中，CPU 的使用者在外界观察时仍然会得到串行执行的观察结果，所以工程师可以认为 CPU 是能够串行执行指令的黑箱。想要充分利用多个 CPU 的资源，仍然需要工程师理解多线程模型并掌握操作系统中一些并发控制机制。

单核的超标量处理器一般被分类为单指令单数据流（Single Instruction stream, Single Data stream、SISD）处理器，而如果处理器支持向量操作，就被分为单指令多数据流（Single Instruction stream, Multiple Data streams、SIMD）处理器，而 CPU 厂商会引入 SIMD 指令来提高 CPU 的处理能力。

## 1.4、片内布局

前端总线是 Intel 在 1990 年在芯片中使用的通信接口，AMD 在 CPU 中也引入了类似的接口，它们的作用都是在 CPU 和内存控制器中心（也被称作北桥）之间传递数据。前端总线在刚设计时不仅灵活，而且成本很低，但是这种设计很难支持芯片中越来越多的 CPU。

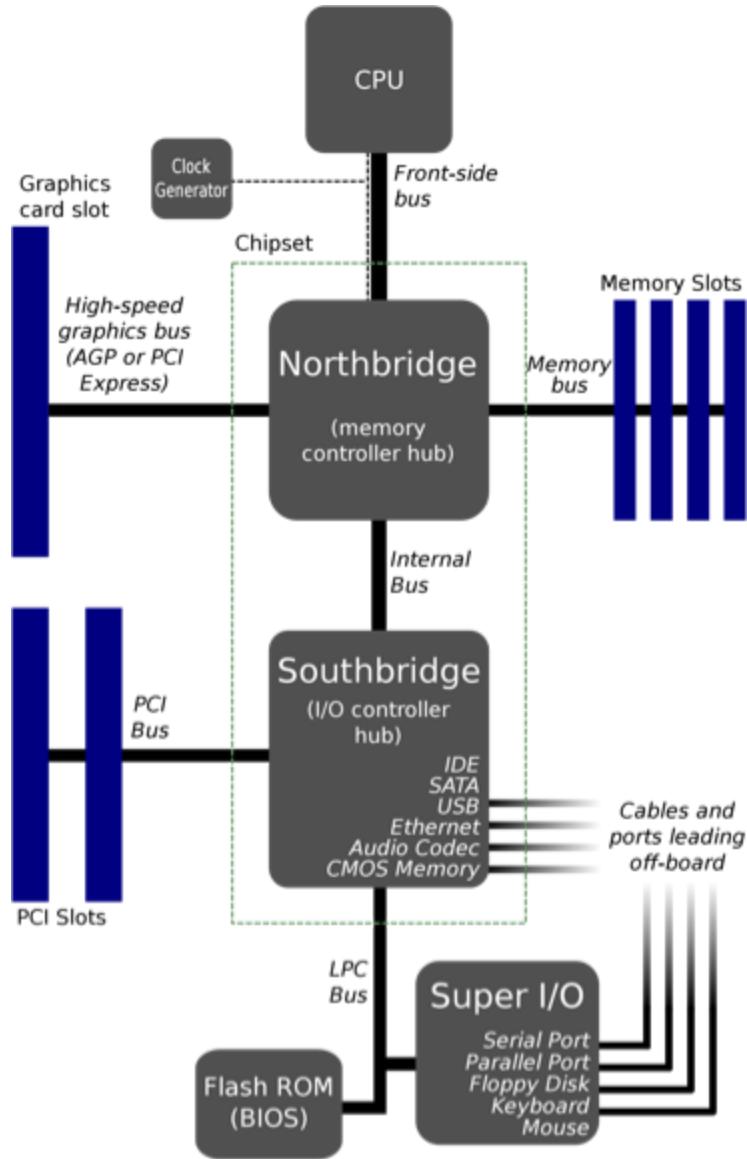


图 6 – 常见芯片布局

如果 CPU 不能从主存中快速获取指令和数据，那么它会花费大量的时间等待读写主存中的数据，所以越高端的处理器越需要高带宽和低延迟，而速度较慢的前端总线无法满足这样的需求。Intel 和 AMD 分别引入了点对点连接的 HyperTransport 和 QuickPath Interconnect (QPI) 机制解决这个问题，上图中的南桥被新的传输机制取代了，CPU 通过集成在内部的内存控制访问内存，通过 QPI 连接其他 CPU 以及 I/O 控制器。

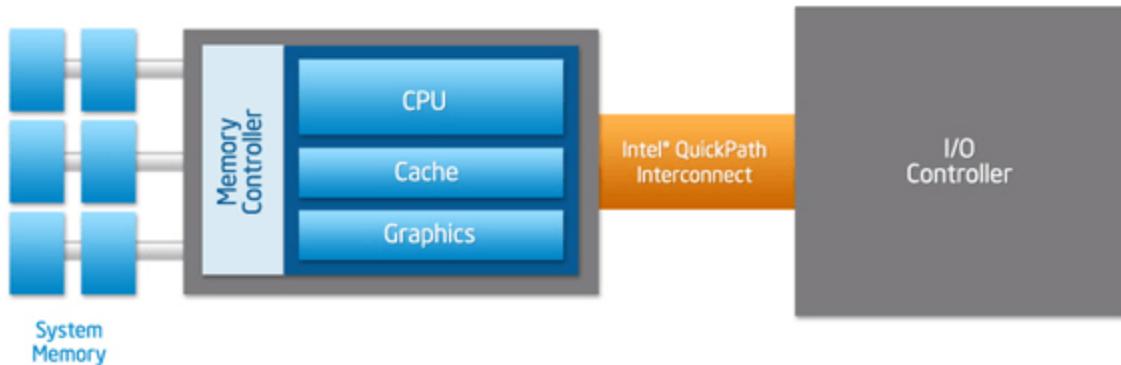


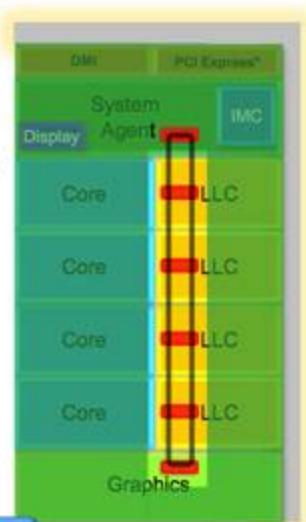
图 7 – Intel QPI

使用 QPI 让 CPU 直接连接其他组件确实可以提高效率，但是随着 CPU 核心数量的增加，这种连接的方式限制了核心的数量，所以 Intel 在 Sandy Bridge 微架构中引入了如下所示的环形总线（Ring Bus）。

## Scalable Ring On-die Interconnect

- **Ring-based** interconnect between Cores, Graphics, Last Level Cache (LLC) and System Agent domain
- Composed of **4 rings**
  - 32 Byte *Data* ring, *Request* ring, *Acknowledge* ring and *Snoop* ring
  - Fully pipelined at **core frequency/voltage**: bandwidth, latency and power scale with cores
- Massive ring **wire routing** runs over the LLC with no area impact
- Access on ring always picks the **shortest path** – minimize latency
- **Distributed arbitration**, sophisticated ring protocol to handle coherency, ordering, and core interface
- **Scalable to servers** with large number of processors

**High Bandwidth, Low Latency, Modular**



IDF2010

图 8 – 环形总线

Sandy Bridge 在架构中引入了片内的 GPU 和视频解码器，这些组件也需要与 CPU 共享 L3 缓存，如果所有的组件都与 L3 缓存直接连接，那么片内会出现大量的连接，而这是芯片工程师不能接受的。片内环形总线连接了 CPU、GPU、L3 缓存、PCIe 控制器、DMI 和内存等部分，其中包含四个功能各异的环：数据、请求、确认和监听<sup>13</sup>，这种设计减少了不同组件内部的连接同时也具有较好的可扩展性。

然而随着 CPU 核心数量的继续增加，环形的连接会不断变大，这会增加环的大小进而影响整个环上组件之间的访问延迟，导致该设计遇到瓶颈。Intel 由此引入了一种新的网格微架构（Mesh Interconnect Architecture）。

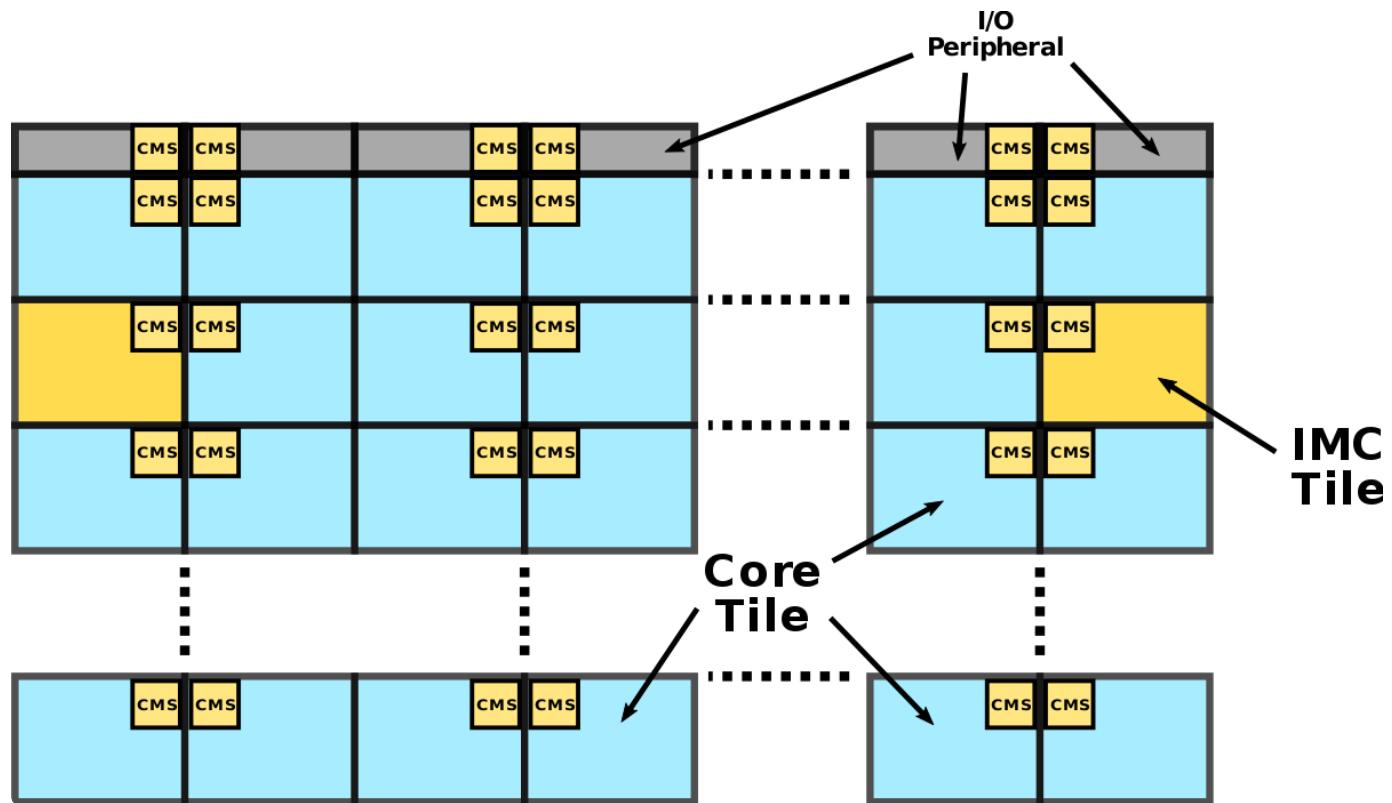


图 9 – 网格架构

如上所示，Intel 的 Mesh 架构是一个二维的 CPU 阵列，网络中有两种不同的组件，一种是上图中蓝色的 CPU 核心，另一种是上图中黄色的集成内存控制器，这些组件不会直接相连，相邻的模块会通过聚合网格站（Converged Mesh Stop、CMS）连接，这与我们今天看到的服务网格非常相似。

当不同组件需要传输数据时，数据包会由 CMS 负责传输，先纵向路由后水平路由，数据到达目标组件后，CMS 会将数据传给 CPU 或者集成的内存控制器。

## 2、GPU

图形处理单元 (Graphics Processing Unit、GPU) 是在缓冲区中快速操作和修改内存的专用电路，因为可以加速图片的创建和渲染，所以在嵌入式系统、移动设备、个人电脑以及工作站等设备上应用都很广泛。然而随着机器学习和大数据的发展，很多公司都会使用 GPU 加速训练任务的执行，这也是今天数据中心中比较常见的用例。

大多数的 CPU 不仅期望在尽可能短的时间内更快地完成任务以降低系统的延迟，还需要在不同任务之间快速切换保证实时性，正是因为这样的需求，CPU 往往都会串行地执行任务。GPU 的设计与 CPU 完全不同，它期望提高系统的吞吐量，在同一时间竭尽全力处理更多的任务，而设计理念上的差异最终反映到了 CPU 和 GPU 的核心数量上<sup>16</sup>：

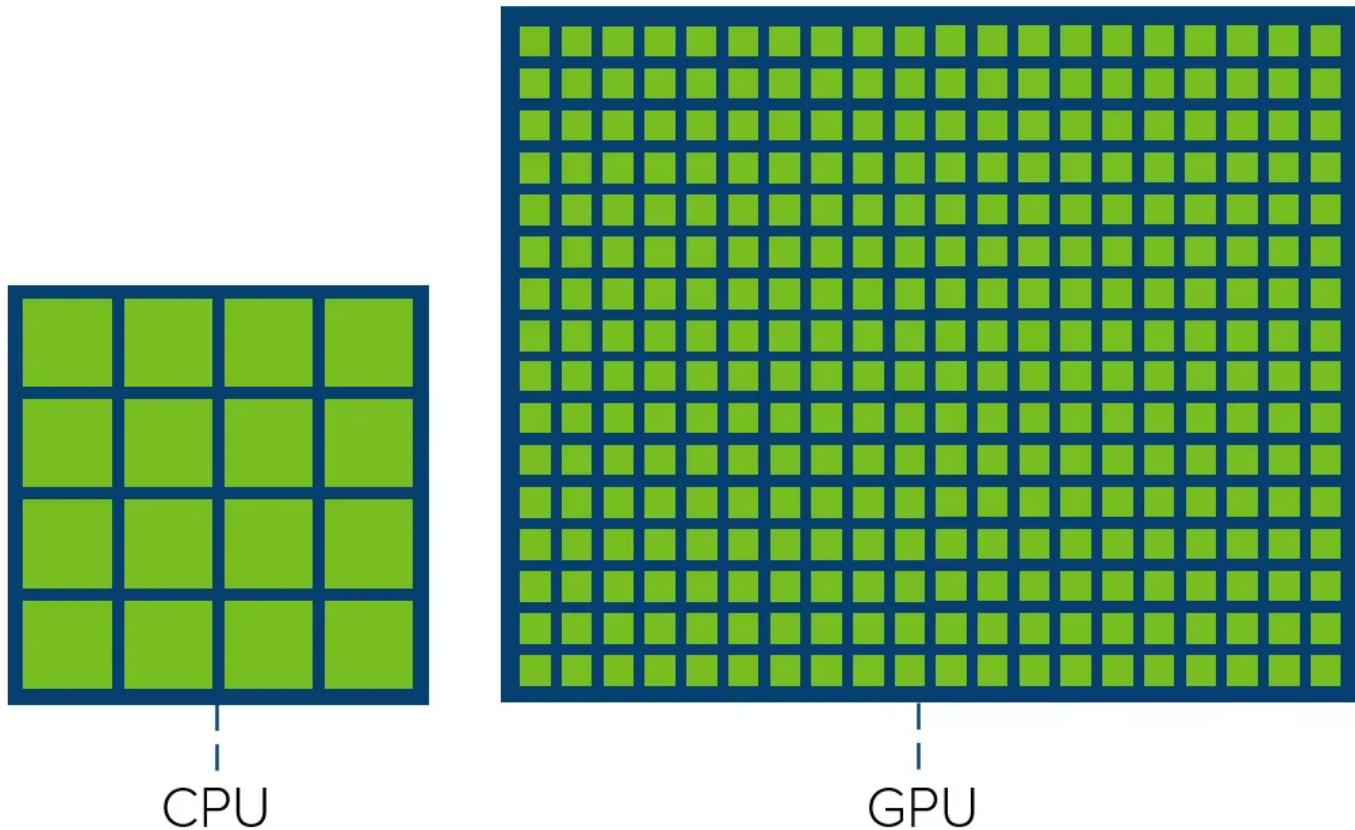
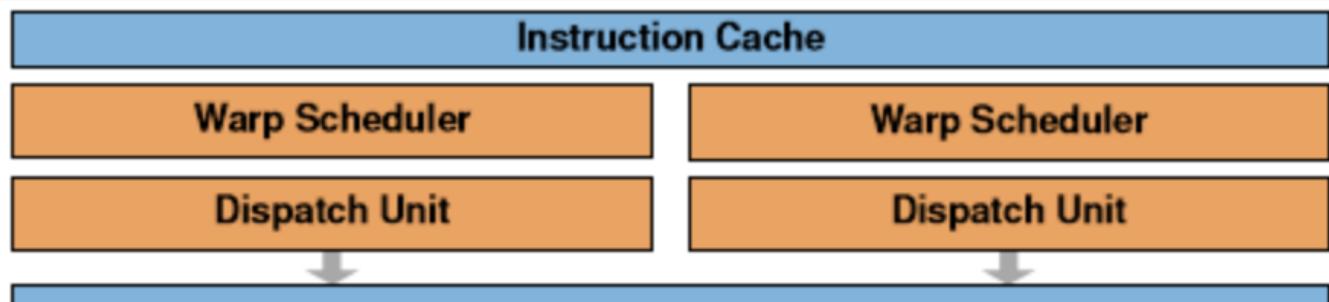
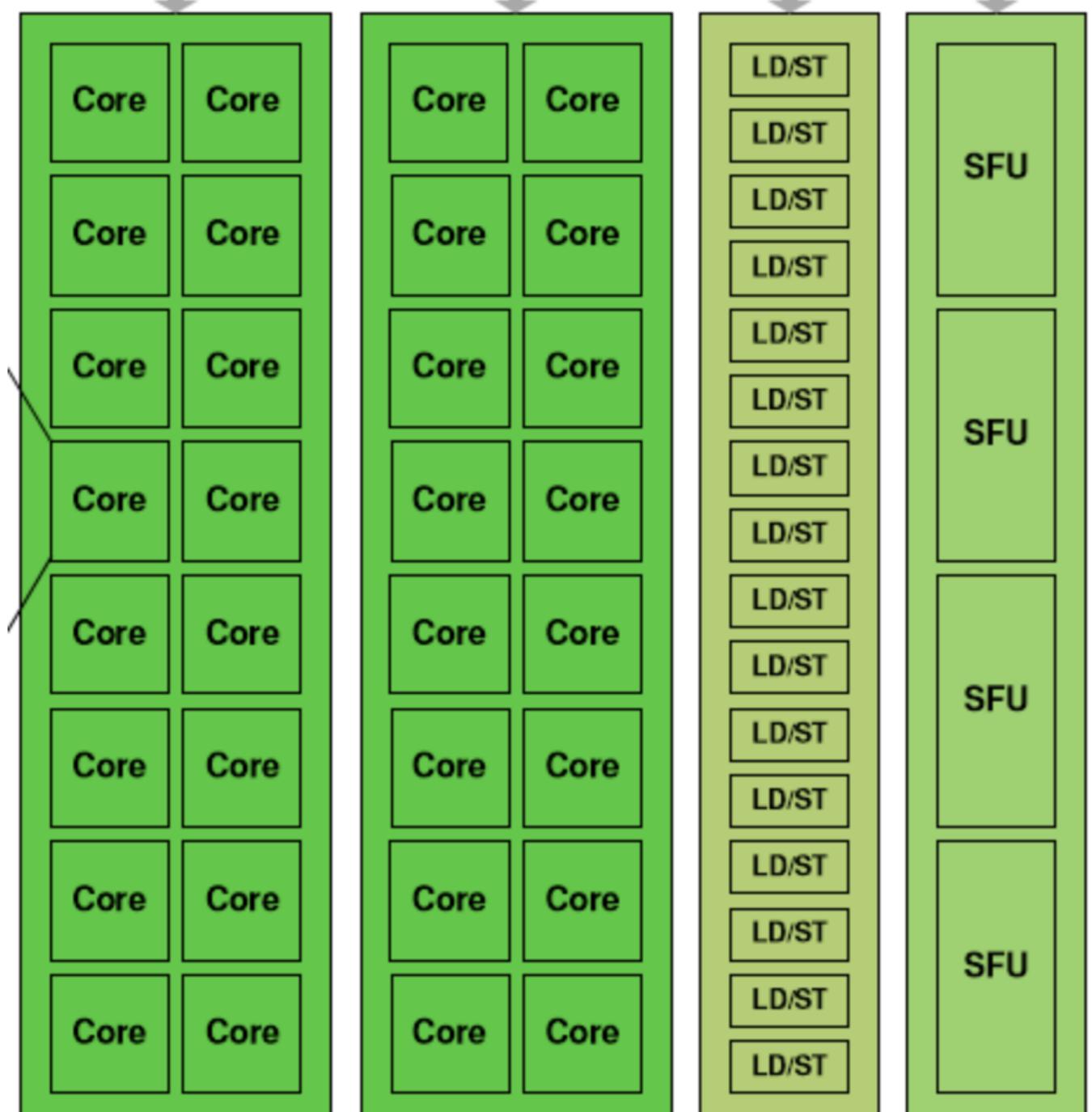


图 10 – CPU 和 GPU 的核心

虽然 GPU 在过去几十年的时间有着很大的发展，但是不同 GPU 的架构大同小异，我们在这里简单介绍下面的流式多处理器中不同组件的作用：



## Register File (4096 x 32-bit)



64 KB Shared Memory / L1 Cache

Uniform Cache

## 图 11 – 流式多处理器

流式多处理器（Streaming Multiprocessor、SM）是 GPU 的基本单元，每个 GPU 都由一组 SM 构成，SM 中最重要的结构就是计算核心 Core，上图中的 SM 包含以下组成部分：

- 线程调度器（Warp Scheduler）
  - 线程束（Warp）是最基本的单元，每个线程束中包含 32 个并行的线程，它们使用不同的数据执行相同的命令，调度器会负责这些线程的调度；
- 访问存储单元（Load/Store Queues）
  - 在核心和内存之间快速传输数据；
- 核心（Core）
  - GPU 最基本的处理单元，也被称作流处理器（Streaming Processor），每个核心都可以负责整数和单精度浮点数的计算；

除了上述这些组件之外，SM 中还包含特殊函数的计算单元（Special Functions Unit、SPU）以及用于存储和缓存数据的寄存器文件（Register File）、共享内存（Shared Memory）、一级缓存和通用缓存。

## 2.1、水平扩容

与 CPU 一样，增加架构中的核心数目是提高 GPU 性能和吞吐量最简单粗暴的手段。Fermi 架构是 Nvidia 早期图形处理器的微架构，在如下所示的架构中，共包含 16 个流式多处理器，512 个 CUDA 核心以及 3,000,000,000 个晶体管：



图 12 – Nvidia Fermi 架构

除了 512 个 CUDA 核心之外，上述架构中还包含 256 个用于传输数据的访问存储单元和 64 个特殊函数单元。如果我们把 2010 年发布的 Fermi 架构和 2020 年发布的 Ampere 做一个简单的对比，就可以发现两者核心数量的巨大差别：

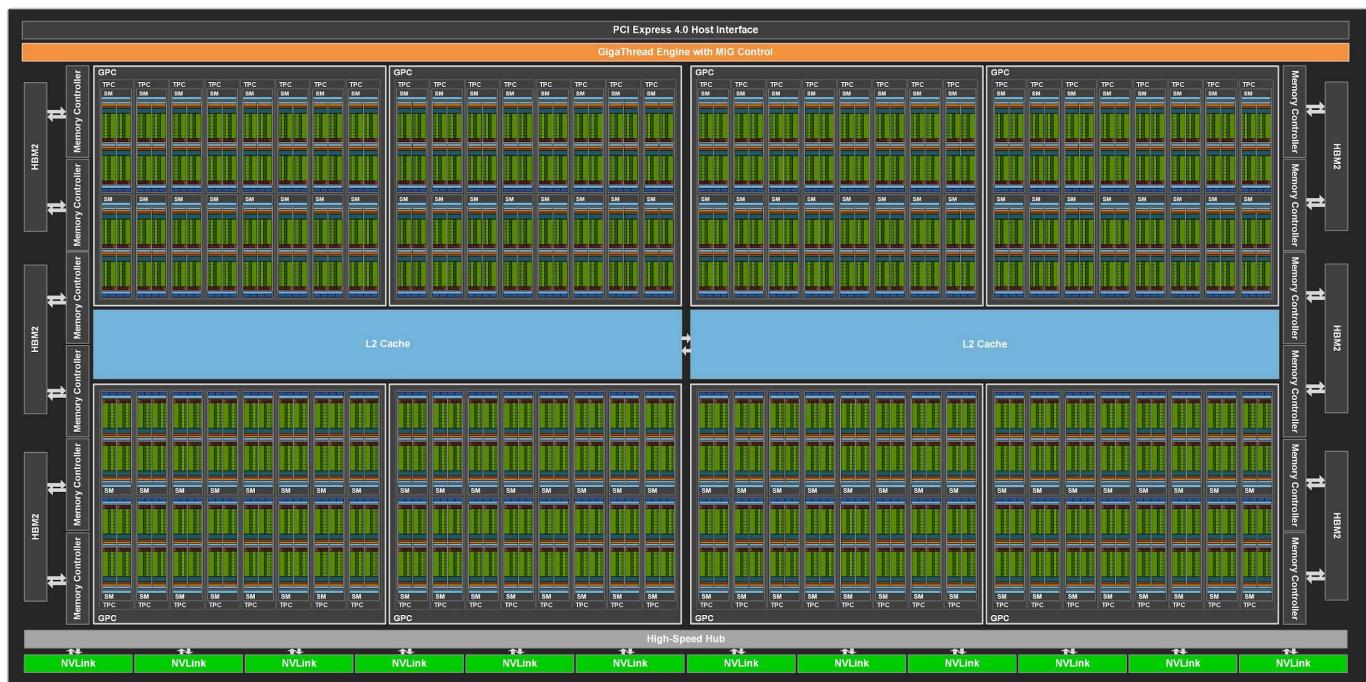


图 13 – Nvidia Ampere 架构

Ampere 架构中的流式多处理器增加到了 128 个，而每个处理器中的核心数也增加到了 64 个，整张显卡上一共包含 8,192 个 CUDA 核心，是 Fermi 架构中核心数量的 16 倍。为了提高系统的吞吐量，新的 GPU 架构不只拥有了更多的核心数量，它还需要更大的寄存器、内存、缓存以及带宽满足计算和传输的需求。

## 2.2、专用核心

最初的 GPU 仅仅是为了更快地创建和渲染图片，它们广泛存在于个人主机上承担着图像渲染的任务，但是随着机器学习等技术的发展，GPU 中出现了更多种类的专用核心来支撑特定的场景，我们在这里介绍两种 GPU 中存在的专用核心：张量核心（Tensor Core）和光线追踪核心（Ray-Tracing Core）：



图 14 – 专用核心

与个人电脑上的 GPU 不同，数据中心中的 GPU 往往都会用来执行高性能计算和 AI 模型的训练任务。正是因为社区有了类似的需求，Nvidia 才会在 GPU 中加入张量核心（Tensor Core）专门处理相关的任务。

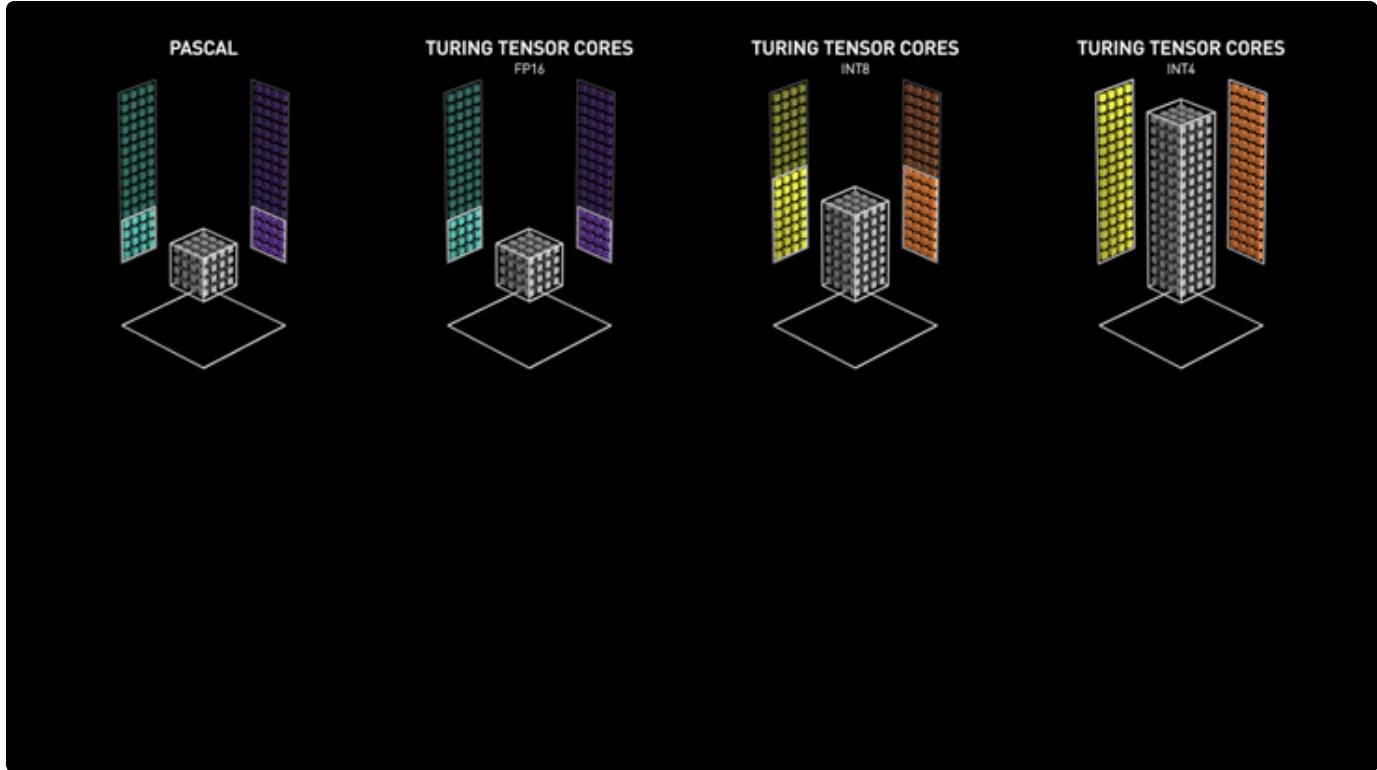


图 15 – 张量核心

张量核心与普通的 CUDA 核心其实有很大的区别，CUDA 核心在每个时钟周期都可以准确的执行一次整数或者浮点数的运算，时钟的速度和核心的数量都会影响整体性能。张量核心通过牺牲一定的精度可以在每个时钟计算执行一次  $4 \times 4$  的矩阵运算，它的引入使得游戏中的实时深度学习任务成为了可能，能够加速度图像的生成和渲染<sup>19</sup>。

计算机图形领域的圣杯是实时的全局光照，实现更好的光线追踪可以帮助我们在屏幕上渲染更加真实的图像，然而全局光照需要 GPU 进行大量的计算，而实时的全局光照更是对性能有着非常高的要求。传统的 GPU 架构并不擅长光线追踪等任务，所以 Nvidia 在 Turing 架构中首次引入了光线追踪核心 (Ray-Tracing Core、RT Core) 。

# TURING RAY TRACING WITH RT CORES

Hardware Acceleration Replaces Software Emulation

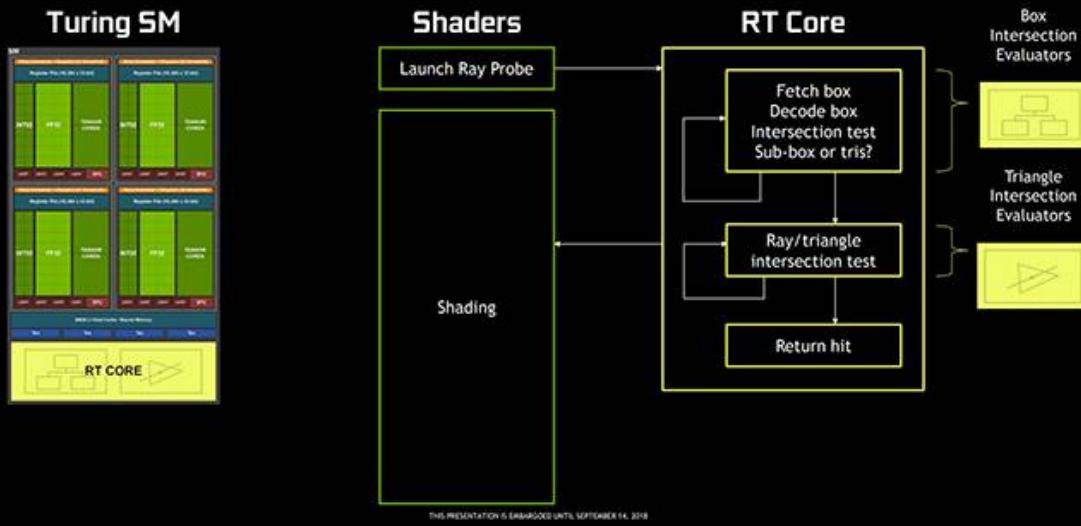


图 16 – 光线追踪核心

Nvidia 的光线追踪核心实际上是为了追踪光线设计的特殊电路，光线追踪中比较常见的算法就是 Bounding Volume Hierarchy (BVH) 遍历和光线三角形相交测试，使用流式多处理器计算该算法每条光线都会花费上千条指令20，而光线追踪核心可以加速这一过程。

## 2.3、多租户

今天 GPU 的性能已经非常强大，但是无论使用数据中心提供的 GPU 实例，还是自己搭建服务器运行计算任务都很昂贵，然而 GPU 算力的拆分在目前仍然是一个比较复杂的问题，运行简单的训练任务可能占用整块 GPU，在这种情况下每提升一点 GPU 的利用率都可以降低一些成本。

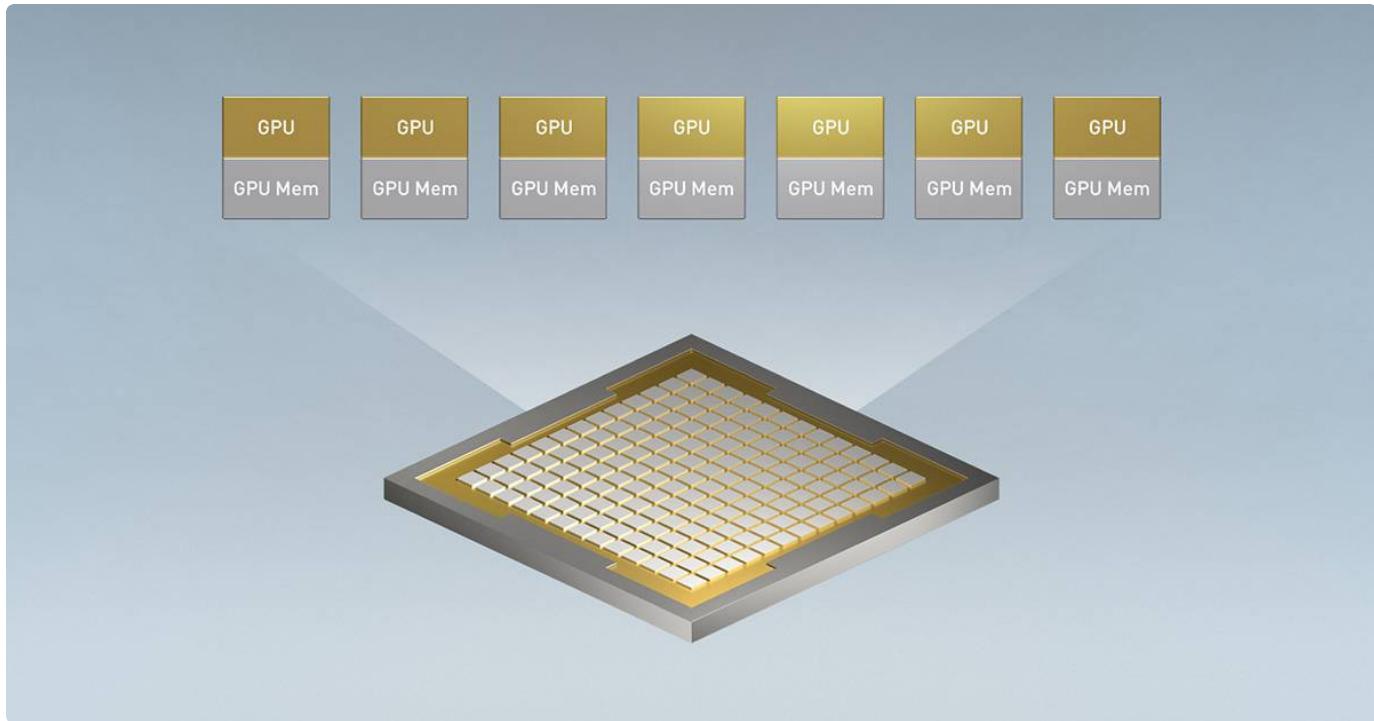


图 17 – 多实例 GPU

Nvidia 最新的 Ampere 架构支持多实例 GPU（Multi-Instance GPU、MIG）技术，它能够水平切分 GPU 资源<sup>21</sup>。每个 A100 GPU 都可以被拆分成 7 个 GPU 实例，每个实例都有隔离的内存、缓存和计算核心，这不仅可以满足数据中心分割 GPU 资源的需要，还能在同一张显卡上并行运行不同的训练任务。

### 3、与CPU的对比及GPU的底层架构

经常听到有人说GPU比CPU要强大，特别是ChatGPT面市，人工智能风靡全球后，既然GPU这么的强大，为什么不能取代CPU呢？

先说答案，答案就是CPU工作方式和GPU的工作方式截然不同，下面的两张图有助于帮助我们理解CPU和GPU的工作方式的不同。



左图：CPU architecture 右图： GPU architecture.

上图有几个重点的元素，也是我们下文重点要阐述的概念，绿色代表的是computational units(可计算单元) 或者称之为 cores(核心)， 橙色代表memories（内存） ， 黄色代表的是control units（控制单元）。

因此想要理解GPU的底层核心构成，就必须明确这几个元素的作用，下文会逐一讲解每个元素的作用。

### 3.1、Computational units(cores)

总的来看，我们可以这样说：CPU的Computational units是“大”而“少”的，然而GPU的Computational units是“小”而“多”的，这里的大小是指的计算能力，多少指的是设备中的数量。通过观察上图，显然可以看出，绿色的部分，CPU“大少”，GPU“小多”的特点。

CPU的cores 比GPU的cores要更加聪明(smarter)，这也是所谓“大”的特点。

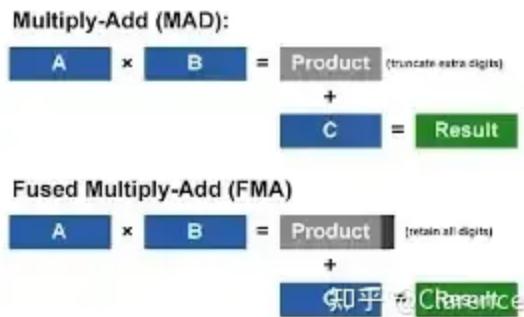
在过去的很长时间里，CPU的core计算能力增长是得益于主频时钟最大的频率增长。相反，GPU不仅没有主频时钟的提升，而且还经历过主频下降的情况，因为GPU需要适应嵌入式应用环境，在这个环境下对功耗的要求是比较高的，不能容忍超高主频的存在。例如英伟达的Jetson NANO,安装在室内导航机器人身上，就是一个很好的嵌入式环境应用示例，安装在机器人身上，就意味着使用电池供电，GPU的功耗不可以过高。(see,[Indoor Mapping and Navigation Robot Build with ROS and Nvidia Jetson Nano](#)):

CPU比GPU聪明，很大一个原因就是CPU拥有"out-of-order executions"（乱序执行）功能。出于优化的目的，CPU可以用不同于输入指令的顺序执行指令，当遇到分支的时候，它可以预测在不久的将来哪一个指令最有可能被执行到（multiple branch prediction 多重分支预测）。通过这种方式，它可

以预先准备好操作数，并且提前执行他们（speculative execution 预测执行），通过上述的几种方式节省了程序运行时间。

显然现代CPU拥有如此多的提升性能的机制，这是比GPU聪明的地方。

相比之下，GPU的core不能做任何类似out-of-order executions那样复杂的事情，总的来说，GPU的core只能做一些最简单的浮点运算，例如 multiply-add(MAD)或者 fused multiply-add(FMA)指令。



通过上图可以看出MAD指令实际是计算 $A \times B + C$ 的值。

实际上，现代GPU结构，CORE不仅仅可以结算FMA这样简单的运算，还可以执行更加复杂的运算操作，例如tensor张量(tensor core)或者光线追踪(ray tracing core)相关的操作。

## CUDA TENSOR CORE PROGRAMMING

16x16x16 Warp Matrix Multiply and Accumulate (WMMA)

```
wmma::mma_sync(Dmat, Amat, Bmat, Cmat);
```

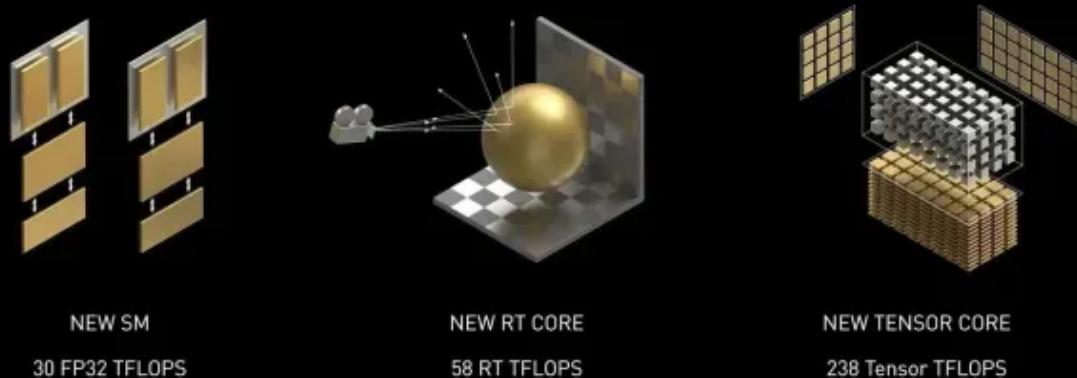
$$D = \left( \begin{array}{c|c|c} \text{FP16 or FP32} & \text{FP16} & \text{FP16 or FP32} \\ \hline \end{array} \right) \left( \begin{array}{c|c} \text{FP16} & \text{FP16} \\ \hline \end{array} \right) + \left( \begin{array}{c|c} \text{FP16 or FP32} & \text{FP16 or FP32} \\ \hline \end{array} \right)$$

$$D = AB + C$$

知乎 @Clarence  
13 NVIDIA

张量计算tensor core

## NVIDIA AMPERE ARCHITECTURE – 2<sup>ND</sup> GENERATION RTX



知乎 @Clarence

### 光线追踪

张量核心(tensor cores)的目的在于服务张量操作在一些人工智能运算场合，光纤追踪 (ray tracing) 旨在服务超现实主义 (hyper-realistic) 实时渲染的场合。

上文说到，GPU Core最开始只是支持一些简单的浮点运算FMA,后来经过发展又增加了一些复杂运算的机制tensor core以及ray trace，但是总体来说GPU的计算灵活性还是比不上CPU的核心。

值得一提的是，GPU的编程方式是SIMD(Single Instruction Multiple Data)意味着所有Core的计算操作完全是在相同的时间内进行的，但是输入的数据有所不同。显然，GPU的优势不在于核心的处理能力，而是在于他可以大规模并行处理数据。



知乎 @Clarence

赛艇运动中，所有人同时齐心划船

GPU中每个核心的作用有点像罗马帆船上的桨手：鼓手打着节拍（时钟），桨手跟着节拍一同滑动帆船。

SIMD编程模型允许加速运行非常多的应用，对图像进行缩放就是一个很好的例子。在这个例子中，每个core对应图像的一个像素点，这样就可以并行的处理每一个像素点的缩放操作，如果这个工作给到CPU来做，需要N的时间才可以做完，但是给到GPU只需要一个时钟周期就可以完成，当然，这样做的前提是有足够的core来覆盖所有的图像像素点。这个问题有个显著的特点，就是对一张图像进行缩放操作，各个像素点之间的信息是相互独立的，因此可以独立的放在不同的core中进行并行运算。我们认为不同的core操作的信息相互独立，是符合SIMD的模型的，使用SIMD来解决这样的问题非常方便。

但是，也不是所有的问题都是符合SIMD模型的，尤其在异步问题中，在这样的问题中，不同的core之间要相互交互信息，计算的结构不规则，负载不均衡，这样的问题交给GPU来处理就会比较复杂。

## 3.2、Memory

这里，先说一下GPU和CPU内存方面的差别。

CPU的memory系统一般是基于DRAM的，在桌面PC中，一般来说是8G，在服务器中能达到数百(256) Gbyte。

CPU内存系统中有个重要的概念就是cache，是用来减少CPU访问DRAM的时间。cache是一片小，但是访问速度更快，更加靠近处理器核心的内存段，用来储存DRAM中的数据副本。cache一般有一个分级，通常分为三个级别L1, L2, L3 cache，cache离核心越近就越小访问越快，例如 L1可以是64KB L2就是256KB L3是4MB。



左图：CPU architecture 右图： GPU architecture.

从上图可以看到GPU中有一大片橙色的内存，名称为DRAM，这一块被称为全局内存或者GMEM。

GMEM的内存大小要比CPU的DRAM小的多，在最便宜的显卡中一般只有几个G的大小，在最好的显卡中GMEM可以达到24G。

GMEM的尺寸大小是科学计算使用中的主要限制。十年前，显卡的容量最多也就只有512M,但是，现在已经完全克服了这个问题。

关于cache，从上图(右图)中不难推断，左上角的小橙色块就是GPU的cache段。然而GPU的缓存机制和CPU是存在一定的差异的，稍后将会证明这一点。

### 3.3、理解GPU的底层结构

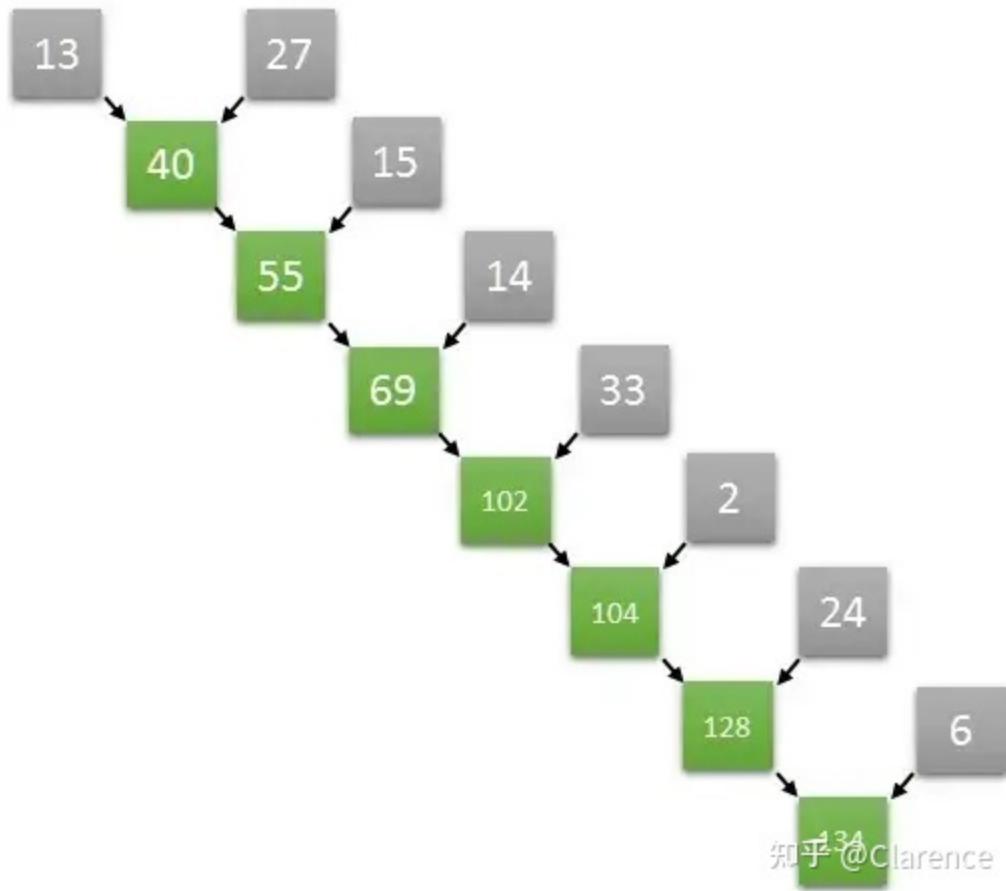
为了充分理解GPU的架构，让我们在返回来看下下图，一个显卡中绝大多数都是计算核心core组成海洋。



左图：CPU architecture 右图： GPU architecture.

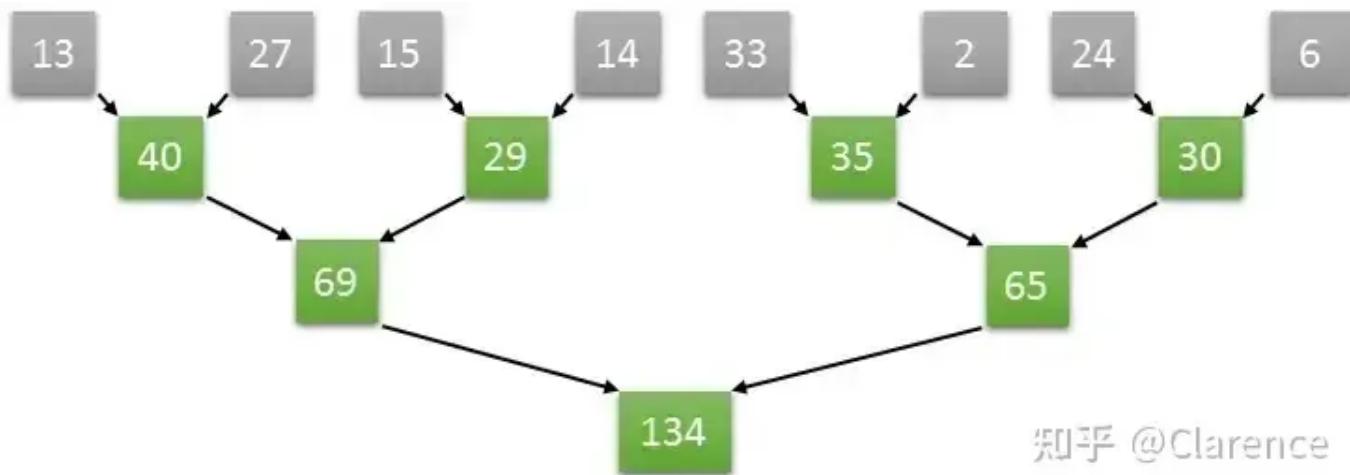
在图像缩放的例子中，core与core之间不需要任何协作，因为他们的任务是完全独立的，然而，GPU解决的问题不一定这么简单，举个例子。

假设我们需要对一个数组里的数进行求和，这样的运算属于reductuin family类型，因为这样的运算试图将一个序列“reduce”简化为一个数。计算数组的元素总和的操作看起来是顺序的，我们只需要获取第一个元素，求和到第二个元素中，获取结果，再将结果求和到第三个元素，以此类推。



Sequential reduction

一些看起来本质是顺序的运算，其实可以再并行算法中转化。假设一个长度为8的数组，在第一步中完全可以并行执行两个元素和两个元素的求和，从而同时获得四个元素，两两相加的结果，以此类推，通过并行的方式加速数组求和的运算速度。具体的操作如下图所示，



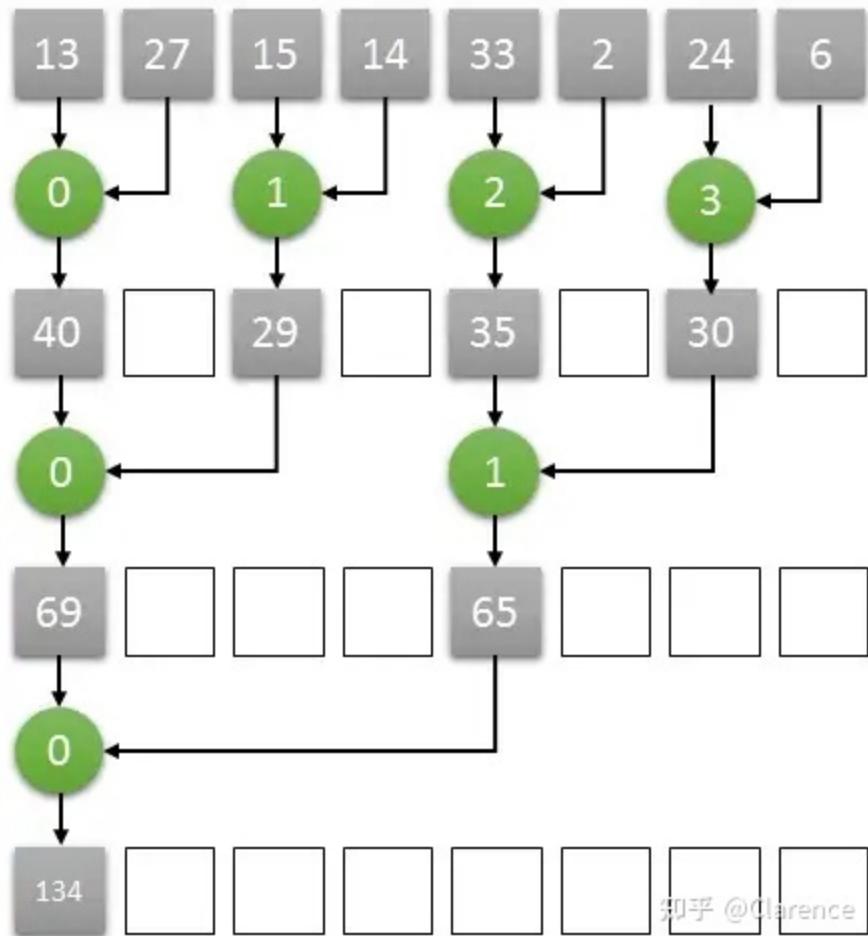
Parallel reduction

如上图计算方式，如果是长度为8的数组两两并行求和计算，那么只需要三次就可以计算出结果。如果是顺序计算需要8次。如果按照两两并行相加的算法，N个数字相加，那么仅需要 $\log_2(N)$  次就可以

完成计算。

从GPU的角度来讲，只需要四个core就可以完成长度为8的数组求和算法，我们将四个core编号为0, 1, 2, 3。

那么第一个时钟下，两两相加的结果通过0号core计算，放入了0号core可以访问到的内存中，另外两两对分别由1号2号3号core来计算，第二个时钟继续按照之前的算法计算，只需要0号和1号两个core即可完成，以此类推，最终的结果将在第三个时钟由0号core计算完成，并储存在0号core可以访问到的内存中。这样实际三次就能完成长度为8的数组求和计算。



Parallel reduction with a GPU

如果GPU想要完成上述的推理计算过程，显然，多个core之间要可以共享一段内存空间以此来完成数据之间的交互，需要多个core可以在共享的内存空间中完成读/写的操作。我们希望每个Cores都有交互数据的能力，但是不幸的是，一个GPU里面可以包含数以千计的core，如果使得这些core都可以访问共享的内存段是非常困难和昂贵的。出于成本的考虑，折中的解决方案是将各类GPU的core分类为多个组，形成多个流处理器(Streaming Multiprocessors )或者简称为SMs。

### 3.4、最终的GPU结构

这里我们以Turing架构为例。



The Turing architecture

上图的绿色部分意味着Core计算单元，绿色的块就是上文谈到的Streaming Multiprocessors，理解为Core的集合。黄色的部分名为RT COREs画的离SMs非常近。单个SM的图灵架构如下图所示：





The Turing SM

在SM的图灵结构中，绿色的部分CORE相关的，我们进一步区分了不同类型的CORE。主要分为INT32,FP32, TENSOR CORES。

- FP32 Cores, 执行单精度浮点运算, 在TU102卡中, 每个SM由64个FP32核, TU120由72个SMs因此, FP32 Core的数量是  $72 * 64$ 。
- FP64 Cores. 实际上每个SM都包含了2个64位浮点计算核心FP64 Cores, 用来计算双精度浮点运算, 虽然上图没有画出, 但是实际是存在的。
- Integer Cores, 这些core执行一些对整数的操作, 例如地址计算, 可以和浮点运算同时执行指令。在前几代GPU中, 执行这些整型操作指令都会使得浮点运算的管道停止工作。TU102总共由4608个Integer Cores, 每个SM有64个SM。
- Tensor Cores, 张量core是FP16单元的变种, 认为是半精度单元, 致力于张量积算加速常见的深度学习操作。图灵张量Core还可以执行INT8和INT4精度的操作, 用于可以接受量化而且不需要FP16精度的应用场景, 在TU102中, 我们每个SM有8个张量Cores, 一共有 $8 * 72$ 个Tensor Cores。

在大致描述了GPU的执行部分之后, 让我们回到上文提出的问题, 各个核心之间如何完成彼此的协作?

在四个SM块的底部有一个96KB的L1 Cache, 用浅蓝色标注的。这个cache段是允许各个Core都可以访问的段, 在L1 Cache中每个SM都有一块专用的共享内存。作为芯片上的L1 cache他的大小是有限的, 但它非常快, 肯定比访问GMEM快得多。

实际上L1 CACHE拥有两个功能:

- 用于SM上Core之间相互共享内存。
- 普通的cache功能。

当Core需要协同工作, 并且彼此交换结果的时候, 编译器编译后的指令会将部分结果储存在共享内存中, 以便于不同的core获取到对应数据。

当用做普通cache功能的时候, 当core需要访问GMEM数据的时候, 首先会在L1中查找, 如果没找到, 则回去L2 cache中寻找, 如果L2 cache也没有, 则会从GMEM中获取数据, L1访问最快 L2 以及 GMEM递减。缓存中的数据将会持续存在, 除非出现新的数据做替换。

从这个角度来看, 如果Core需要从GMEM中多次访问数据, 那么编程者应该将这块数据放入功能内存中, 以加快他们的获取速度。

其实可以将共享内存理解为一段受控制的cache, 事实上L1 cache和共享内存是同一块电路中实现的。编程者有权决定L1 的内存多少是用作cache多少是用作共享内存。

最后, 也是比较重要的是, 可以储存各个core的计算中间结果, 用于各个核心之间共享的内存段不仅仅可以是共享内存L1, 也可以是寄存器, 寄存器是离core最近的内存段, 但是也非常小。

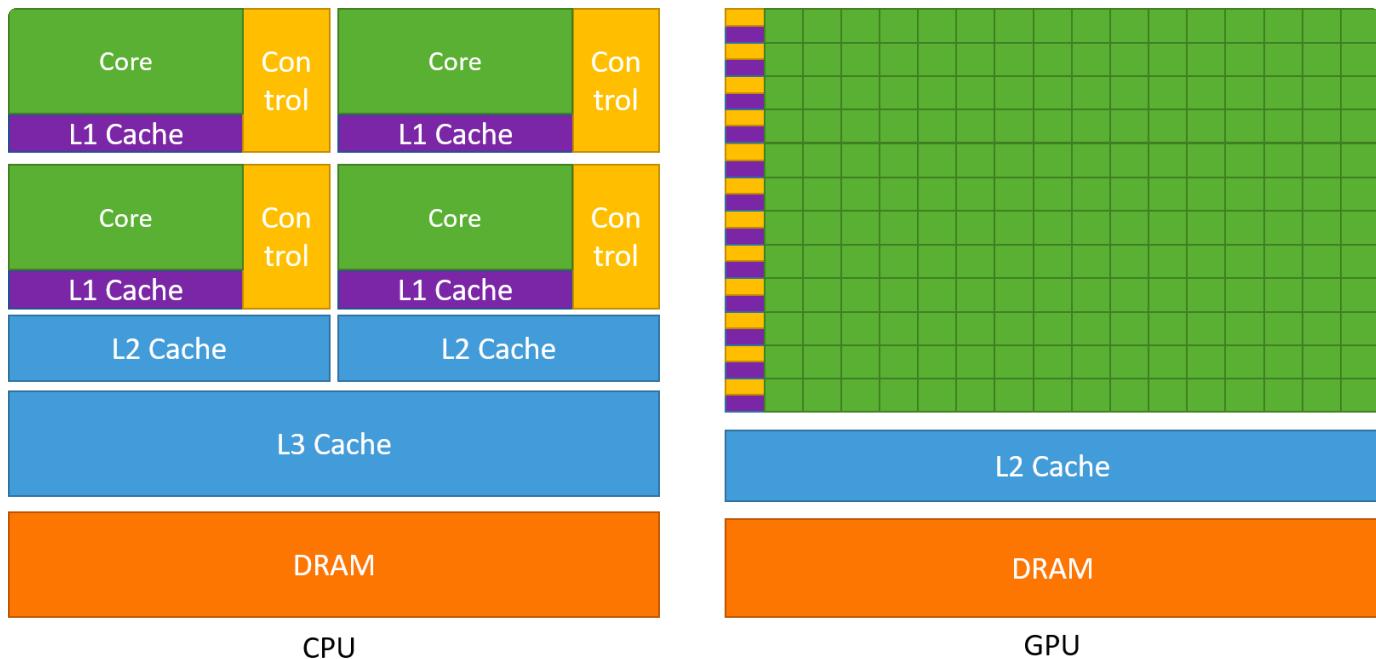
最底层的思想是每个线程都可以拥有一个寄存器来储存中间结果，每个寄存器只能由相同的一个线程来访问，或者由相同的warp或者组的线程访问。

## 五、为什么深度学习需要使用GPU

相比cpu, gpu有以上特点：

- 多核心
  - GPU有很多核心，多达上千个，甚至更多。
- 高带宽
  - gpu内存带宽更高，速度快就贵，所以显存容量一般不大。因为 CPU 首先得取得数据，才能进行运算，很多时候，限制我们程序运行速度的并非是 CPU 核的处理速度，而是数据访问的速度。
- 简单控制流
  - cpu 控制流很强，alu 只占cpu的一小部分。gpu 则要少用控制语句。现代 CPU 里的晶体管变得越来越多，越来越复杂，其实已经不是用来实现“计算”这个核心功能，而是拿来实现处理乱序执行、进行分支预测，以及高速缓存。
  - GPU 专门用于高度并行计算，因此设计时更多的晶体管用于数据处理，而不是数据缓存和流量控制。
  - GPU 只有 取指令、指令译码、ALU 以及执行这些计算需要的寄存器和缓存。PS：将更多晶体管用于数据处理，例如浮点计算，有利于高度并行计算。
- 编程
  - cpu 是各种编程语言，编译器成熟。
  - GPU 一开始是没有“可编程”能力的，程序员们只能够通过配置来设计需要用到的图形渲染效果（图形加速卡）。在游戏领域，3D 人物的建模都是用一个个小三角形拼接上的，而不是以像素的形式，对多个小三角形的操作，能使人物做出多种多样的动作，而 GPU 在此处就是用来计算三角形平移，旋转之后的位置。为了提高游戏的分辨率，程序会将每个小三角形细分为更小的三角形，每个小三角形包含两个属性，它的位置和它的纹理。
  - 在游戏领域应用的 GPU 与科学计算领域的 GPU 使用的不同是，当通过 CUDA 调用 GPU 来进行科学计算的时候，计算结果需要返回给 CPU，但是如果用 GPU 用作玩游戏的话，GPU 的计算结果直接输出到显示器上，也就不需要再返回到 CPU。

- 深度学习的模型训练，指的是利用数据通过计算梯度下降的方式迭代地去优化神经网络的参数，最终输出网络模型的过程。在这个过程中，通常在迭代计算的环节，会借助 GPU 进行计算的加速。



## 1、GPU 架构

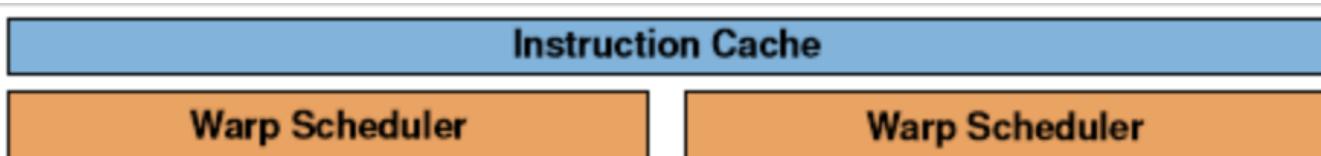


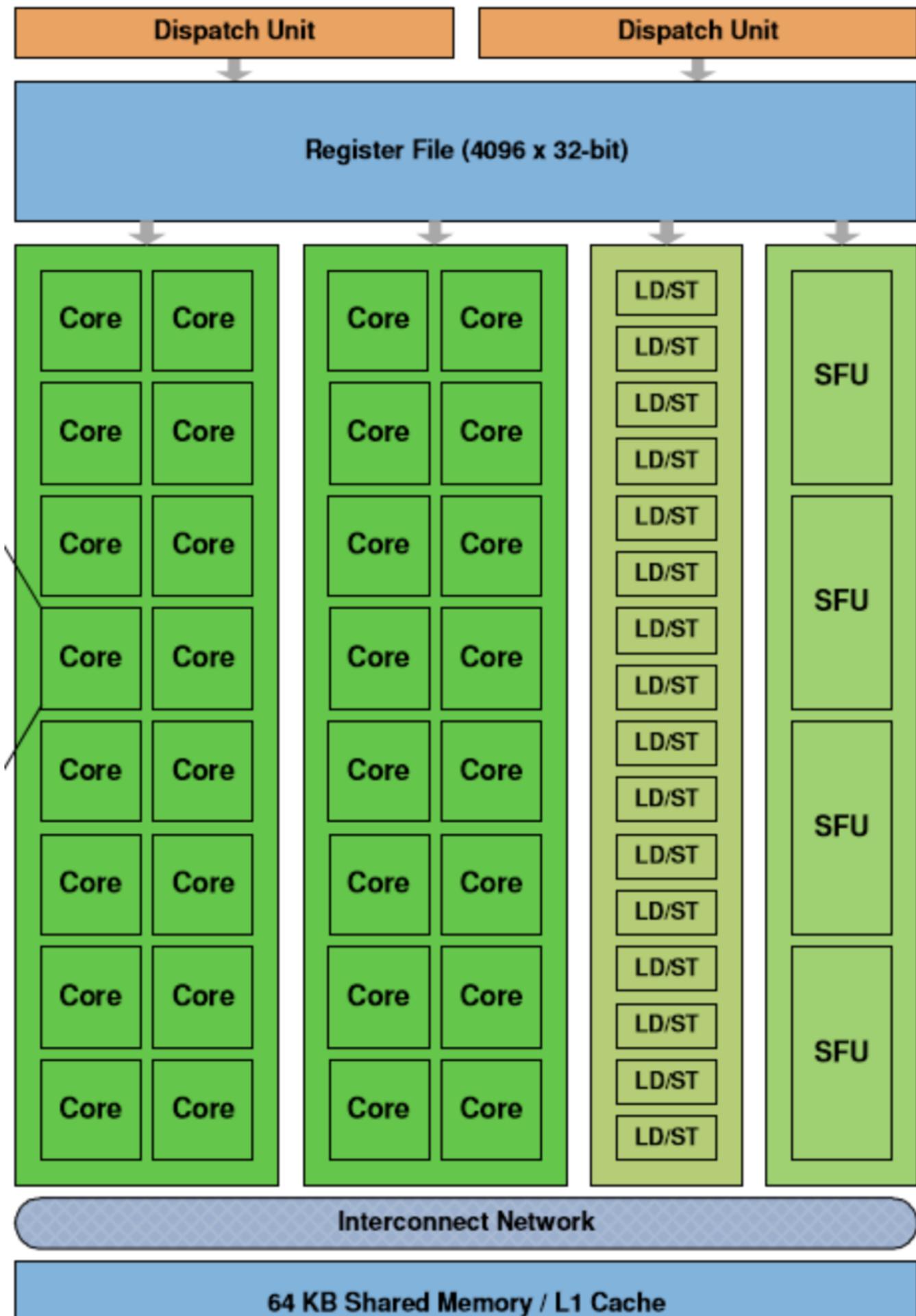
GPU的core不能做任何类似out-of-order executions那样复杂的事情，总的来说，GPU的core只能做一些最简单的浮点运算,例如 multiply-add(MAD)或者 fused multiply-add(FMA)指令，后来经过发展又增加了一些复杂运算，例如tensor张量(tensor core)或者光线追踪(ray tracing core)相关的操作。

GPU的编程方式是SIMD(Single Instruction Multiple Data)意味着所有Core的计算操作完全是在相同的时间内进行的，但是输入的数据有所不同。如果这个工作给到CPU来做，需要N的时间才可以做完，但是给到GPU只需要一个时钟周期就可以完成。

**多个core之间通讯：**在图像缩放的例子中，core与core之间不需要任何协作，因为他们的任务是完全独立的。然而，GPU解决的问题不一定这么简单，假设一个长度为8的数组，在第一步中完全可以并行执行两个元素和两个元素的求和，从而同时获得四个元素，两两相加的结果，以此类推，通过并行的方式加速数组求和的运算速度。如果是长度为8的数组两两并行求和计算，那么只需要三次就可以计算出结果。如果是顺序计算需要8次。如果GPU想要完成上述的推理计算过程，显然，**多个core之间要可以共享一段内存空间以此来完成数据之间的交互**，需要多个core可以在共享的内存空间中完成读/写的操作。我们希望每个Cores都有交互数据的能力，但是不幸的是，一个GPU里面可以包含数以千计的core，如果使得这些core都可以访问共享的内存段是非常困难和昂贵的。出于成本的考虑，折中的解决方案是将各类GPU的core分类为多个组，形成多个流处理器(Streaming Multiprocessors )或者简称为SMs。

SM块的底部有一个96KB的L1 Cache。L1 CACHE拥有两个功能，一个是用于SM上Core之间相互共享内存（寄存器 也可以），另一个则是普通的cache功能。存在全局的内存GMEM，但是访问较慢，Cores当需要访问GMEM的时候会首先访问L1,L2如果都miss了，那么才会花费大代价到GMEM中寻找数据。





## Uniform Cache

流式多处理器（Streaming Multiprocessor、SM）是 GPU 的基本单元，每个 GPU 都由一组 SM 构成，SM 中最重要的结构就是计算核心 Core。

- 线程调度器（Warp Scheduler）
  - 线程束（Warp）是最基本的单元，每个线程束中包含 32 个并行的线程，它们使用不同的数据执行相同的命令，调度器会负责这些线程的调度；
- 访问存储单元（Load/Store Queues）
  - 在核心和内存之间快速传输数据；
- 核心（Core）
  - GPU 最基本的处理单元，也被称作流处理器（Streaming Processor），每个核心都可以负责整数和单精度浮点数的计算；
- 特殊函数的计算单元（Special Functions Unit、SPU）
- 存储和缓存数据的寄存器文件（Register File）
- 共享内存（Shared Memory）

与个人电脑上的 GPU 不同，数据中心中的 GPU 往往都会用来执行高性能计算和 AI 模型的训练任务。

正是因为社区有了类似的需求，Nvidia 才会在 GPU 中加入张量（标量是0阶张量，向量是一阶张量，矩阵是二阶张量）核心（Tensor Core）专门处理相关的任务。

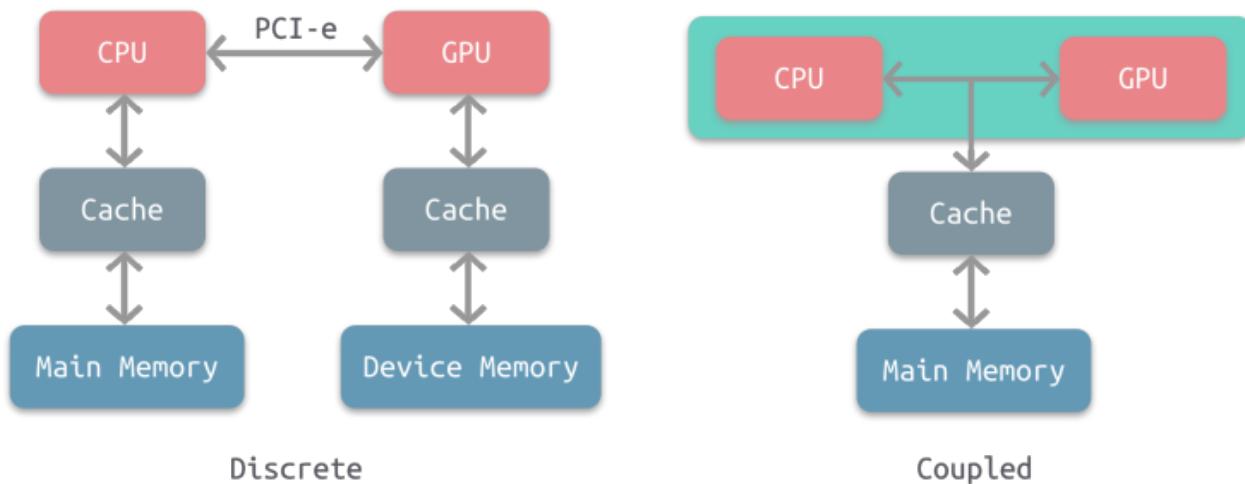
张量核心与普通的 CUDA 核心其实有很大的区别，CUDA 核心在每个时钟周期都可以准确的执行一次整数或者浮点数的运算，时钟的速度和核心的数量都会影响整体性能。

张量核心通过牺牲一定的精度可以在每个时钟计算执行一次  $4 \times 4$  的矩阵运算。PS：就像ALU 只需要加法器就行了（乘法指令转换为多个加法指令），但为了提高性能，直接做了一个乘法器和加法器并存。

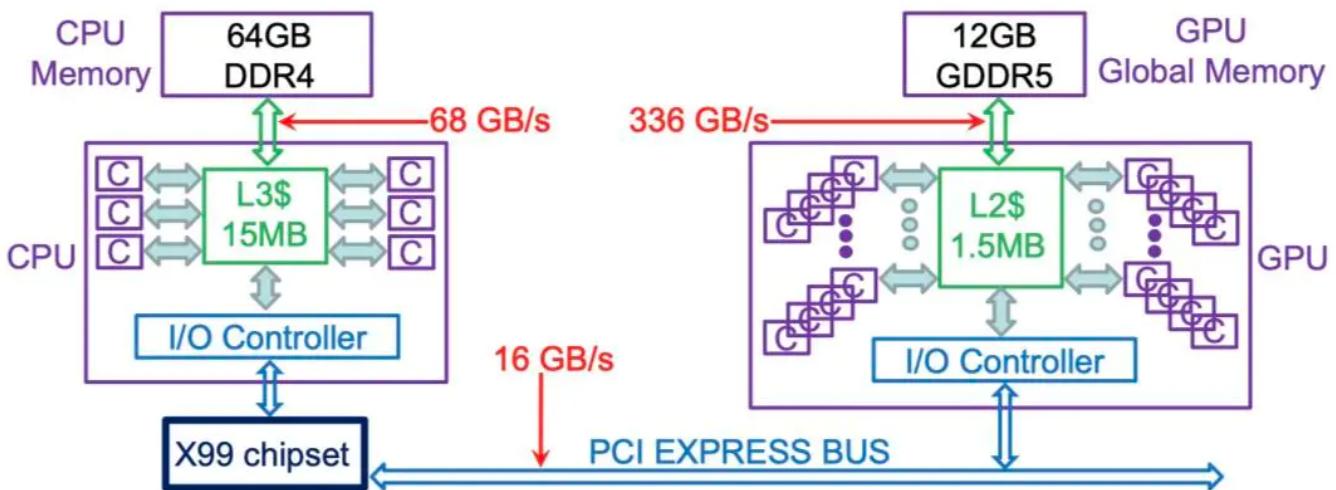
## 2、CPU 与 GPU 协作

GPU 无法自己独立工作，其工作任务还是由 CPU 进行触发的。整体的工作流程可以看做是 CPU 将需要执行的计算任务异步的交给 GPU，GPU 拿到任务后，会将 Kernel 调度到相应的 SM 上，而 SM

内部的线程则会按照任务的描述进行执行。



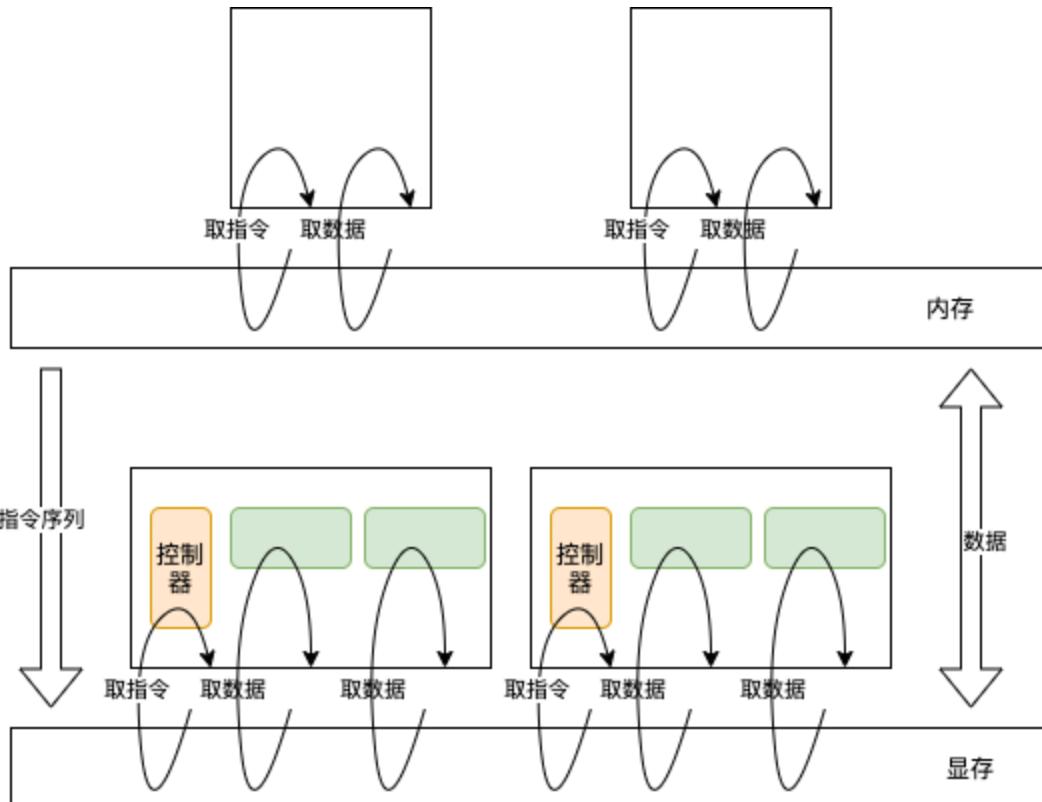
大多数采用的还是分离式结构，AMD 的 APU 采用耦合式结构，目前主要使用在游戏主机中，如 PS4。



- 锁页
  - GPU 可以直接访问 CPU 的内存。出于某些显而易见的原因，CPU 和 GPU 最擅长访问自己的内存，但 GPU 可以通过 DMA 来访问 CPU 中的锁页内存。锁页是操作系统常用的操作，可以使硬件外设直接访问内存，从而避免过多的复制操作。”被锁定“的页面被 OS 标记为不可被 OS 换出的，所以设备驱动程序在给这些外设编程时，可以使用页面的物理地址直接访问内存。PS：部分内存的使用权暂时移交给设备。
- 命令缓冲区
  - CPU 通过 CUDA 驱动写入指令，GPU 从缓冲区读取命令并控制其执行，
- CPU 与 GPU 同步

- cpu 如何跟踪GPU 的进度

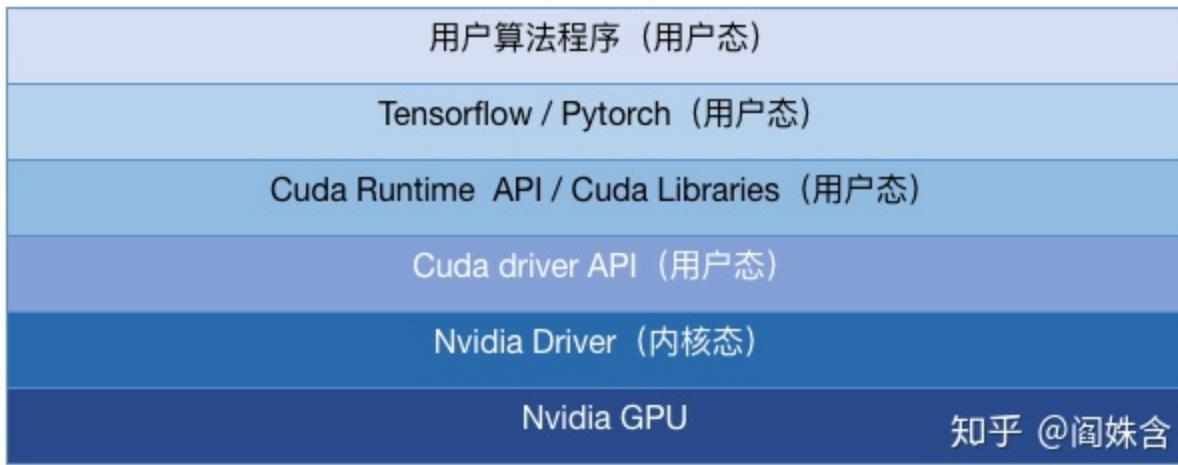
对于一般的外设来说，驱动程序提供几个api接口，约定好输入和输出的内存地址，向输入地址写数据，调接口，等中断，从输出地址拿数据。输出数据地址 `command_operation`(输入数据地址)。gpu 是可以编程的，变成了输出数据地址 `command_operation`(指令序列,输入数据地址)。



系统的三个要素: CPU, 内存, 设备。CPU 虚拟化由 VT-x/SVM 解决, 内存虚拟化由 EPT/NPT 解决, 设备虚拟化呢? 它的情况要复杂的多, 不管是 VirtIO, 还是 VT-d, 都不能彻底解决设备虚拟化的问题。除了这种完整的系统虚拟化, 还有一种也往往被称作「虚拟化」的方式: 从 OS 级别, 把一系列的 library 和 process 捆绑在一个环境中, 但所有的环境共享同一个 OS Kernel。

不考虑嵌入式平台的话, 那么, GPU 首先是一个 PCIe 设备。GPU 的虚拟化, 还是要首先从 PCIe 设备虚拟化角度来考虑。一个 PCIe 设备, 有什么资源? 有什么能力?

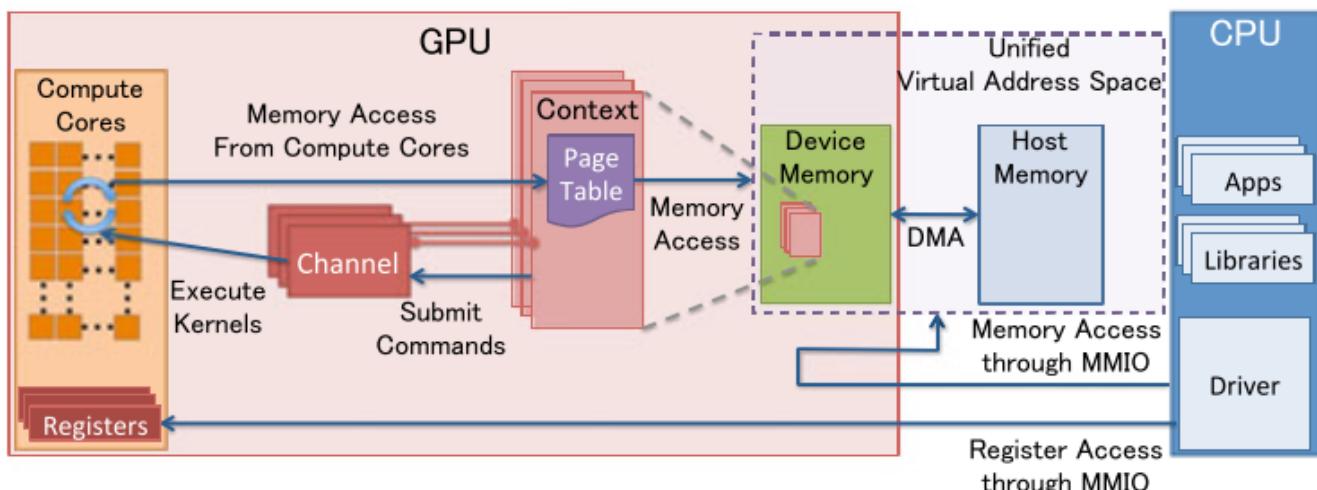
- 2 种资源:
  - 配置空间
  - MMIO(Memory-Mapped I/O)
- 2 种能力
  - 中断能力
  - DMA 能力



一个典型的 GPU 设备的工作流程是：

- 应用层调用 GPU 支持的某个 API，如 OpenGL 或 CUDA
- OpenGL 或 CUDA 库，通过 UMD (User Mode Driver)，提交 workload 到 KMD (Kernel Mode Driver)
- Kernel Mode Driver 写 CSR MMIO，把它提交给 GPU 硬件
- GPU 硬件开始工作... 完成后，DMA 到内存，发出中断给 CPU
- CPU 找到中断处理程序 —— Kernel Mode Driver 此前向 OS Kernel 注册过的 —— 调用它
- 中断处理程序找到是哪个 workload 被执行完毕了，... 最终驱动唤醒相关的应用

本质上 GPU 还是一个外设，有驱动程序（分为用户态和内核态）和 API，用户程序 ==> API ==> CPU ==> 驱动程序 ==> GPU ==> 中断 ==> CPU.



warp (gpu的一个单位) 是典型的单指令多线程 (SIMT, SIMD单指令多数据的升级) 的实现，也就是32个线程同时执行的指令是一模一样的，只是线程数据不一样，这样的好处就是一个warp只需要一个

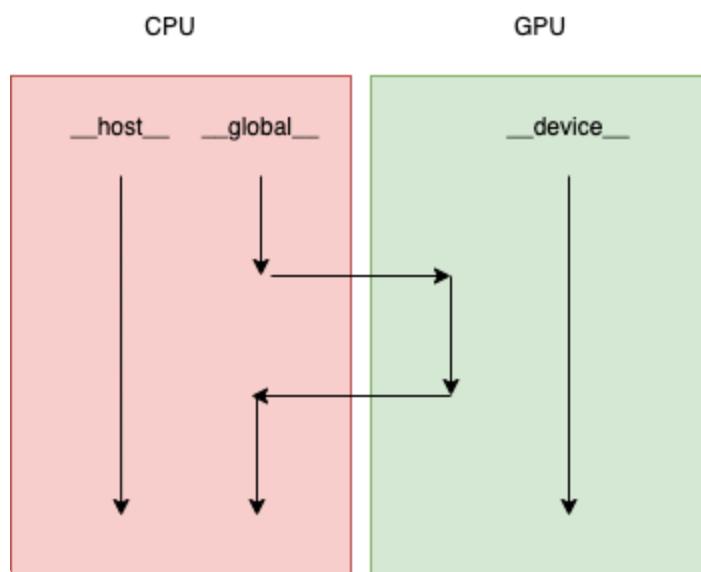
套逻辑对指令进行解码和执行就可以了，芯片可以做的更小更快，之所以可以这么做是由于GPU需要处理的任务是天然并行的。

### 3、CUDA——GPU 编程

2006年，NVIDIA公司发布了CUDA，CUDA是建立在NVIDIA的CPUs上的一个通用并行计算平台和编程模型。CUDA编程模型是一个异构模型，需要CPU和GPU协同工作。

CUDA的架构中引入了主机端（host, cpu）和设备（device, gpu）的概念，我们用host指代CPU及其内存，而用device指代GPU及其内存，CUDA程序中既包含host程序，又包含device程序，它们分别在CPU和GPU上运行。

CUDA的Kernel函数既可以运行在主机端，也可以运行在设备端。同时主机端与设备端之间可以进行数据拷贝。



device 函数和global函数因为需要在GPU上运行，因此不能调用常见的一些 C/C++ 函数（因为这些函数没有对应的 GPU 实现）。

限定符	执行	调用	备注
global	设备端执行	可以从主机调用也可以从某些特定设备调用	异步操作，host 将并行计算任务发射到GPU的任务调用单之后，不会等待kernel执行完就执行下一步
device	设备端执行	设备端调用	

host	主机端执行 行	主机调用	
------	------------	------	--

典型的CUDA程序的执行流程如下：

- 分配host内存，并进行数据初始化；
- 分配device内存，并从host将数据拷贝到device上；
- 调用CUDA的核函数在device上完成指定的运算；
- 将device上的运算结果拷贝到host上；
- 释放device和host上分配的内存。

矩阵加法示例

```

1 // __global__ 表示在device上执行，从host中调用
2 // 两个向量加法kernel, grid和block均为一维
3 __global__ void add(float* x, float * y, float* z, int n){
4     // 获取全局索引
5     int index = threadIdx.x + blockIdx.x * blockDim.x;
6     // 步长
7     int stride = blockDim.x * gridDim.x;
8     for (int i = index; i < n; i += stride){
9         z[i] = x[i] + y[i];
10    }
11 }
12 int main(){
13     int N = 1 << 20;
14     int nBytes = N * sizeof(float);
15     // 申请host内存
16     float *x, *y, *z;
17     x = (float*)malloc(nBytes);
18     y = (float*)malloc(nBytes);
19     z = (float*)malloc(nBytes);
20     // 初始化数据
21     for (int i = 0; i < N; ++i){
22         x[i] = 10.0;
23         y[i] = 20.0;
24     }
25     // 申请device内存
26     float *d_x, *d_y, *d_z;
27     cudaMalloc((void**)&d_x, nBytes);
28     cudaMalloc((void**)&d_y, nBytes);
29     cudaMalloc((void**)&d_z, nBytes);
30     // 将host数据拷贝到device
31     cudaMemcpy((void*)d_x, (void*)x, nBytes, cudaMemcpyHostToDevice);
32     cudaMemcpy((void*)d_y, (void*)y, nBytes, cudaMemcpyHostToDevice);
33     // 定义kernel的执行配置
34     dim3 blockSize(256);
35     dim3 gridSize((N + blockSize.x - 1) / blockSize.x);
36     // 执行kernel
37     add << < gridSize, blockSize >> >(d_x, d_y, d_z, N);
38     // 将device得到的结果拷贝到host
39     cudaMemcpy((void*)z, (void*)d_z, nBytes, cudaMemcpyDeviceToHost);
40     // 检查执行结果
41     float maxError = 0.0;
42     for (int i = 0; i < N; i++)
43         maxError = fmax(maxError, fabs(z[i] - 30.0));
44     std::cout << "最大误差: " << maxError << std::endl;
45     // 释放device内存

```

```
46     cudaFree(d_x);
47     cudaFree(d_y);
48     cudaFree(d_z);
49     // 释放host内存
50     free(x);
51     free(y);
52     free(z);
53     return 0;
54 }
```

如何在 CPU 之上调用 GPU 操作？可以通过调用 `__global__` 方法来在 GPU 之上执行并行操作。

## 4、GPU/CUDA/驱动和机器学习训练框架的关系

NGC: Caffe/TensorFlow/Torch/CNTK/Theano/MxNet...

TensorRT for Inference

GPU Accelerated SDK

cuDNN/cuBLAS/cuSparse/cuFFT/NCCL

CUDA ToolKits

GPU Driver

OS ( Specialized Optimized for DGX-1 )

GPU Computing Servers ( DGX-1/P40/... )



显卡是硬件，硬件需要驱动，否则不能调用其计算资源。CUDA 又是什么？

- 在2007年之前，GPU由CPU操作，CPU把一些图形图像的计算任务交给GPU执行。程序员不需要与GPU打交道。随着GPU计算能力的发展，越来越多的计算场景由GPU完成效果会更好。但现有的程序无法直接自由控制GPU的处理器。当然程序员也可以直接写代码与显卡驱动对接，从而直接控制GPU的处理器，但这样代码恐怕写起来要让人疯掉。nvidia当然会有动力提供一套软件接口来简化操作GPU的处理器。nvidia把这一套软件定义为CUDA。
- 多核 CPU 和众核 GPU 的出现意味着主流处理器芯片现在是并行系统。挑战在于开发能够透明地扩展可并行的应用软件，来利用不断增加的处理器内核数量。CUDA 并行编程模型旨在克服这一挑战，同时为熟悉 C 等标准编程语言的程序员保持较低的学习曲线。

gpu 和 cuda 和 gpu driver 之间的关系：比如 TX3090需要Compute Capability在8.6以上的cuda，而满足这个要求的cuda又只有11.0以上的版本。而cuda11版本又需要版本号>450的显卡驱动。

```
Plain Text |  
1 显卡  
2     ==> Compute Capability    查看显卡支持的Compute Capability, https://developer.nvidia.com/cuda-gpus  
3     ==> cuda  
4             ==> GPU driver    查看cuda对驱动的要求 (https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html)  
5             ==> tf/pytorch version
```

Compute Capability的数值和GPU的计算速度无关，但是和GPU可执行的任务种类有关。

The compute capability of a device is represented by a version number, also sometimes called its “SM version”. This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.