CPU 和 GPU - 异构计算的演进与发展

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。而在计算领域中，CPU（Central Processing Unit）和 GPU（Graphics Processing Unit）分别是通用的和特定的方案，前者可以提供最基本的计算能力解决几乎所有问题，而后者在图形计算和机器学习等领域内表现优异。

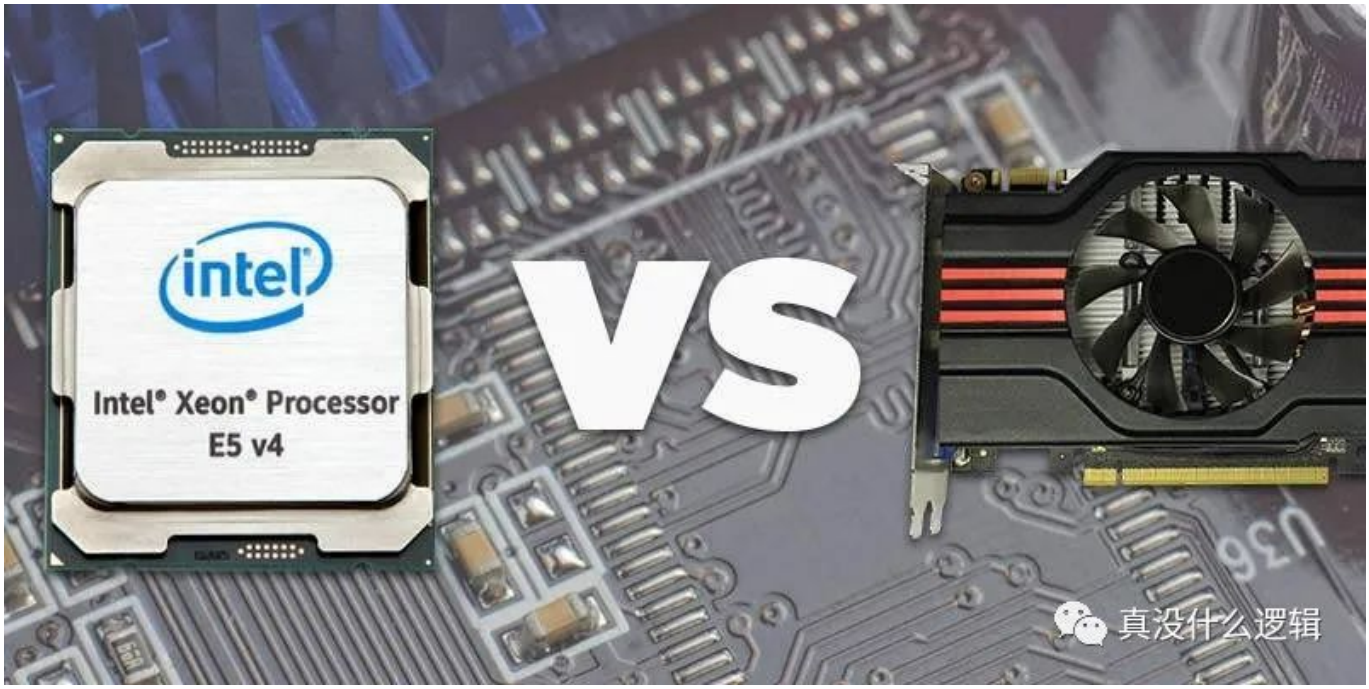

图 1 - CPU 和 GPU

异构计算是指系统同时使用多种处理器或者核心，这些系统通过增加不同的协处理器（Coprocessors）提高整体的性能或者资源的利用率[^1]，这些协处理器可以负责处理系统中特定的任务，例如用来渲染图形的 GPU 以及用来挖矿的 ASIC 集成电路。

中心处理单元（Central Processing Unit、CPU）[^2]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）[^3]，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。

今天的软件工程师，尤其是数据中心和云计算的工程师因为异构计算的发展面对着更加复杂的场景，我们在这篇文章中主要谈一谈 CPU 和 GPU 的演进过程，重新回顾一下在过去几十年的时间里，工程师为它们增加了哪些有趣的功能。

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，我们在这一节简单展示历史上引入了哪些技术来提高 CPU 的性能。

制程

当我们讨论 CPU 的发展时，制程（Fabrication Process）[^4]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。

[^4]: Wikipedia: Semiconductor device fabrication https://en.wikipedia.org/wiki/Semiconductor_device_fabrication

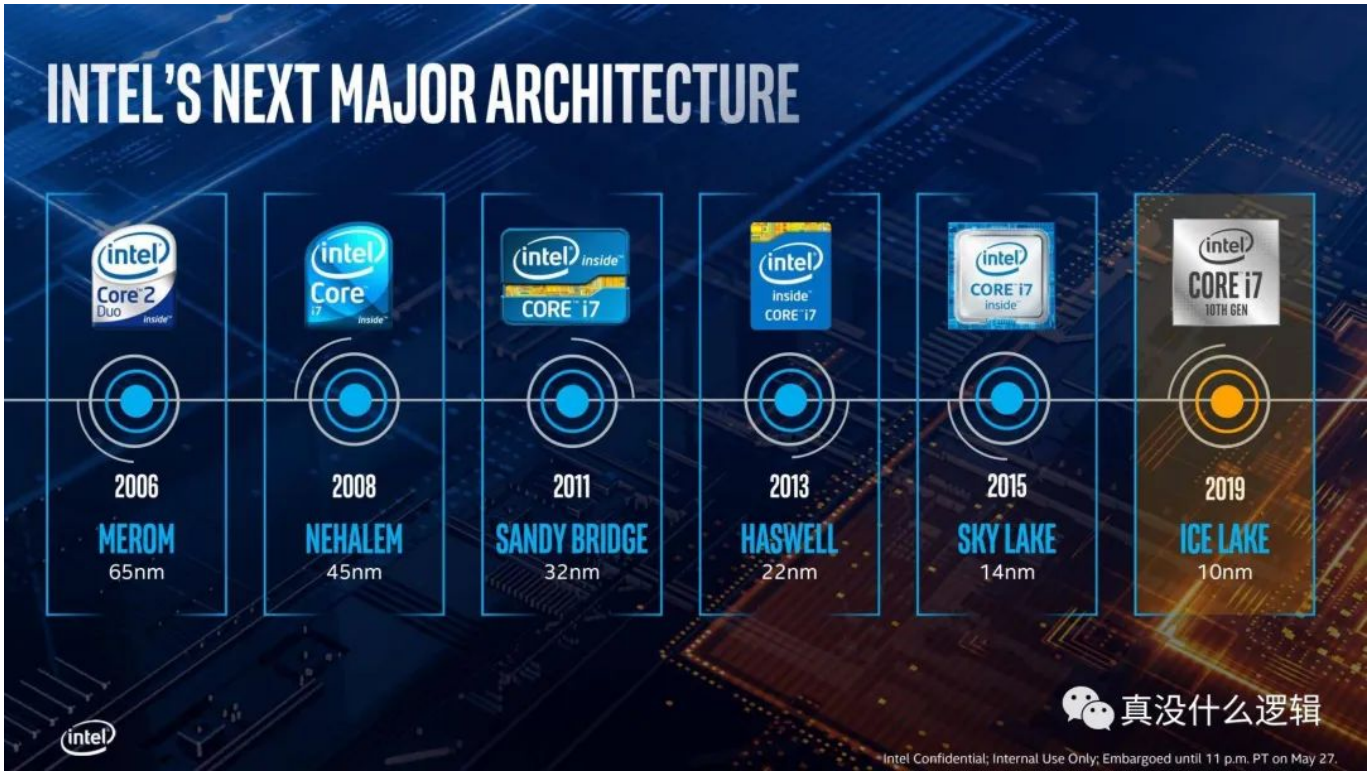

图 2 - Intel CPU 制程

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[^5]。

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，相信很多开发者都看过如下所示的表格，我们可以看到从 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 30 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[^6]

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。

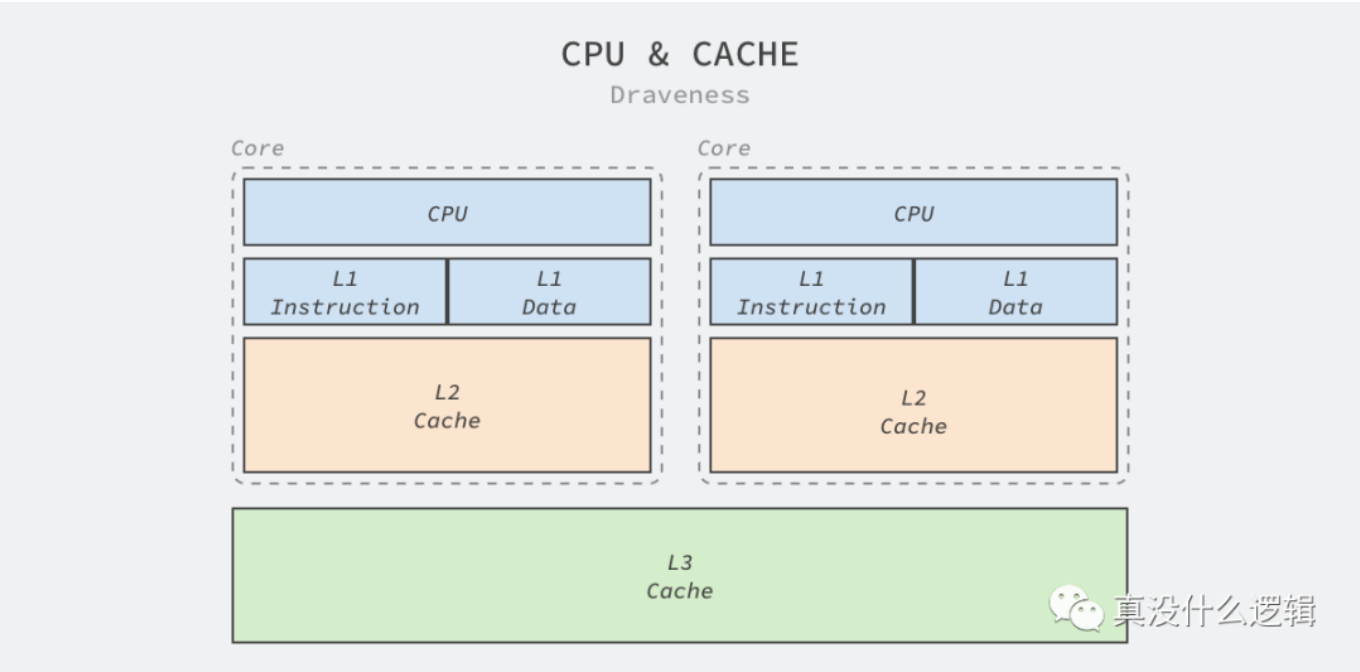

图 3 - CPU 缓存

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[^7]。

并行计算

多线程编程在今天几乎已经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：

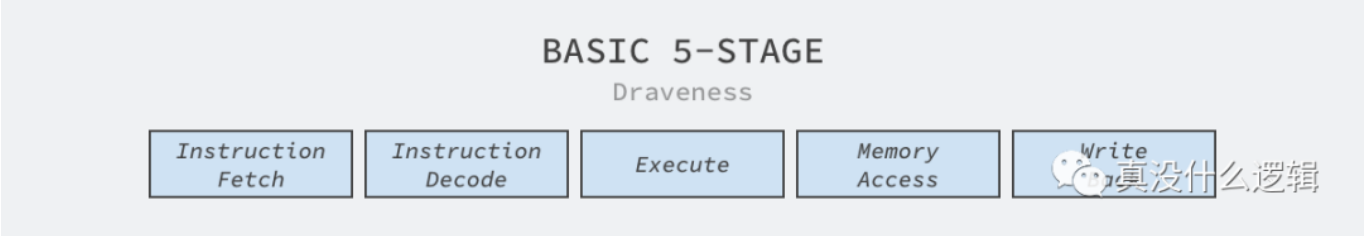

图 4 - 指令执行的步骤

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[^8]，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[^9]。

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[^10]，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。

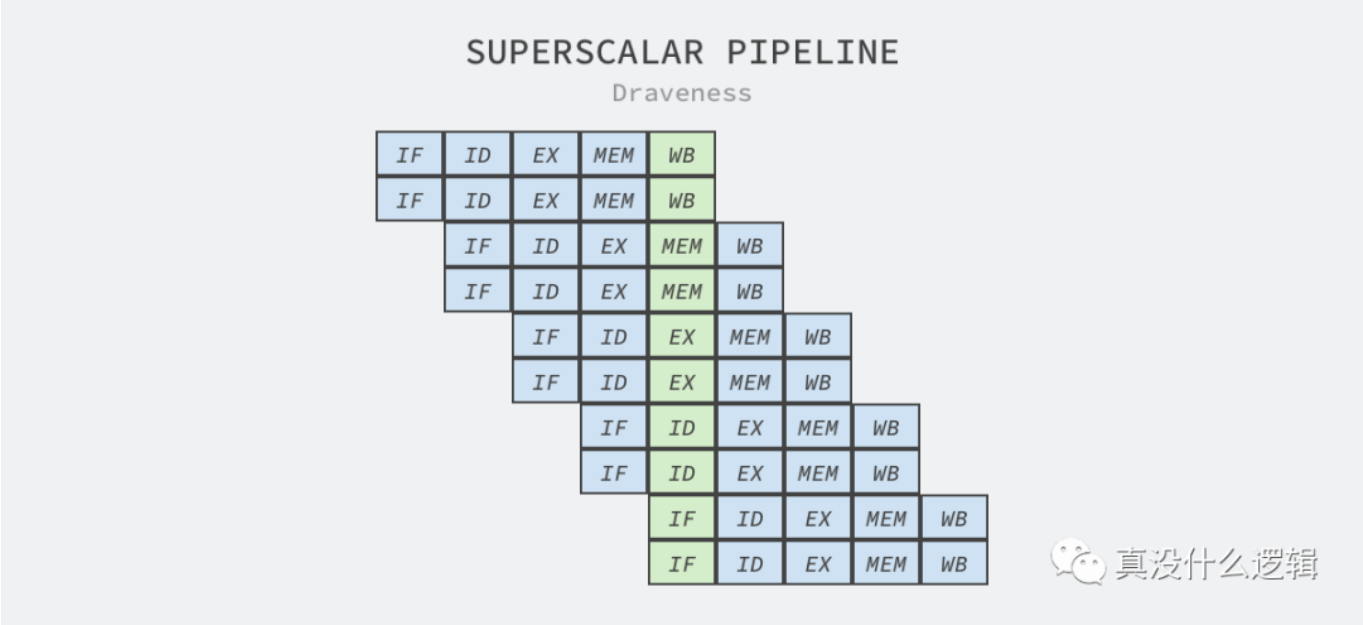

图 5 - 超标量和流水线

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，所以我们分析 CPU
的寄存器操作确定哪些指令可以乱序执行。

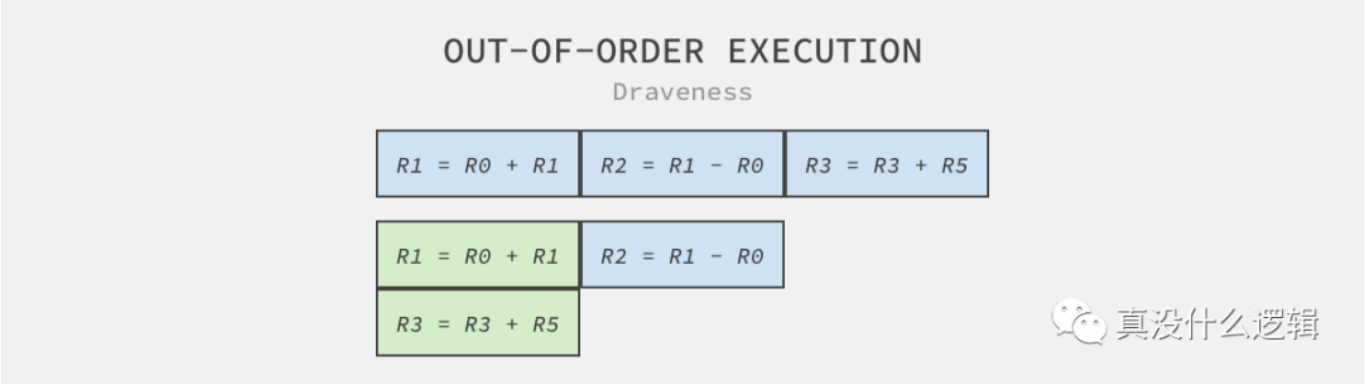

图 6 - 乱序执行

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，当我们在 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[^11]：

- 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。

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：

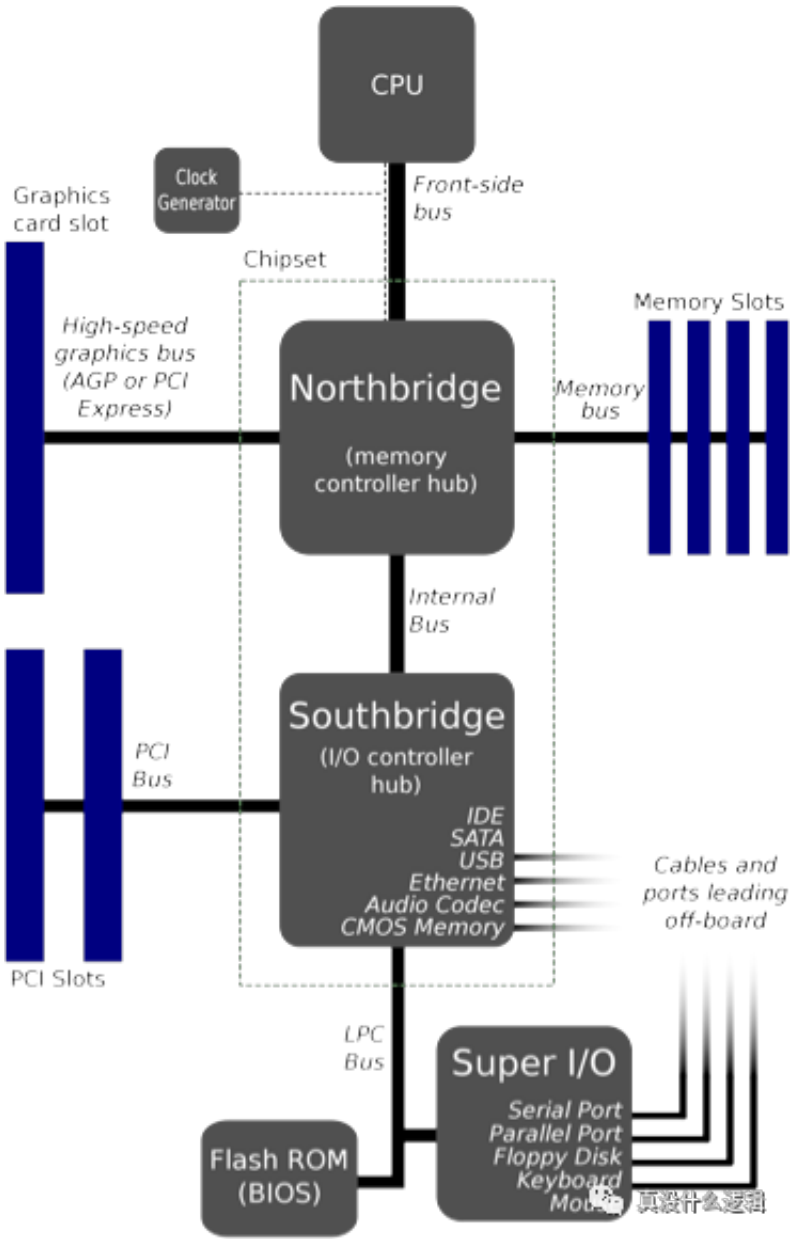

图 6 - 常见芯片布局

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事件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。

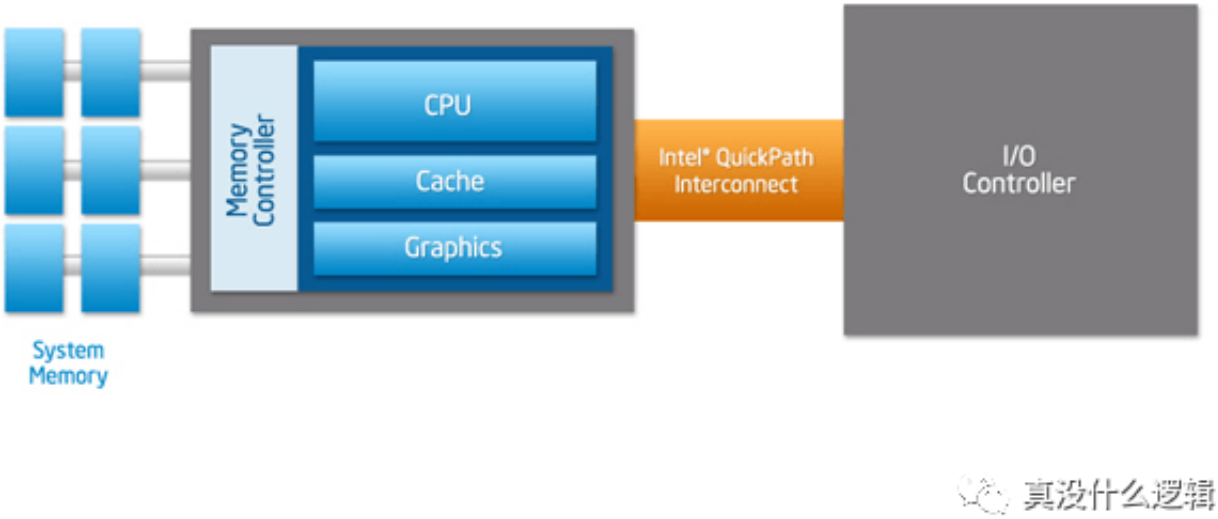

图 7 - Intel QPI

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）[^12]：

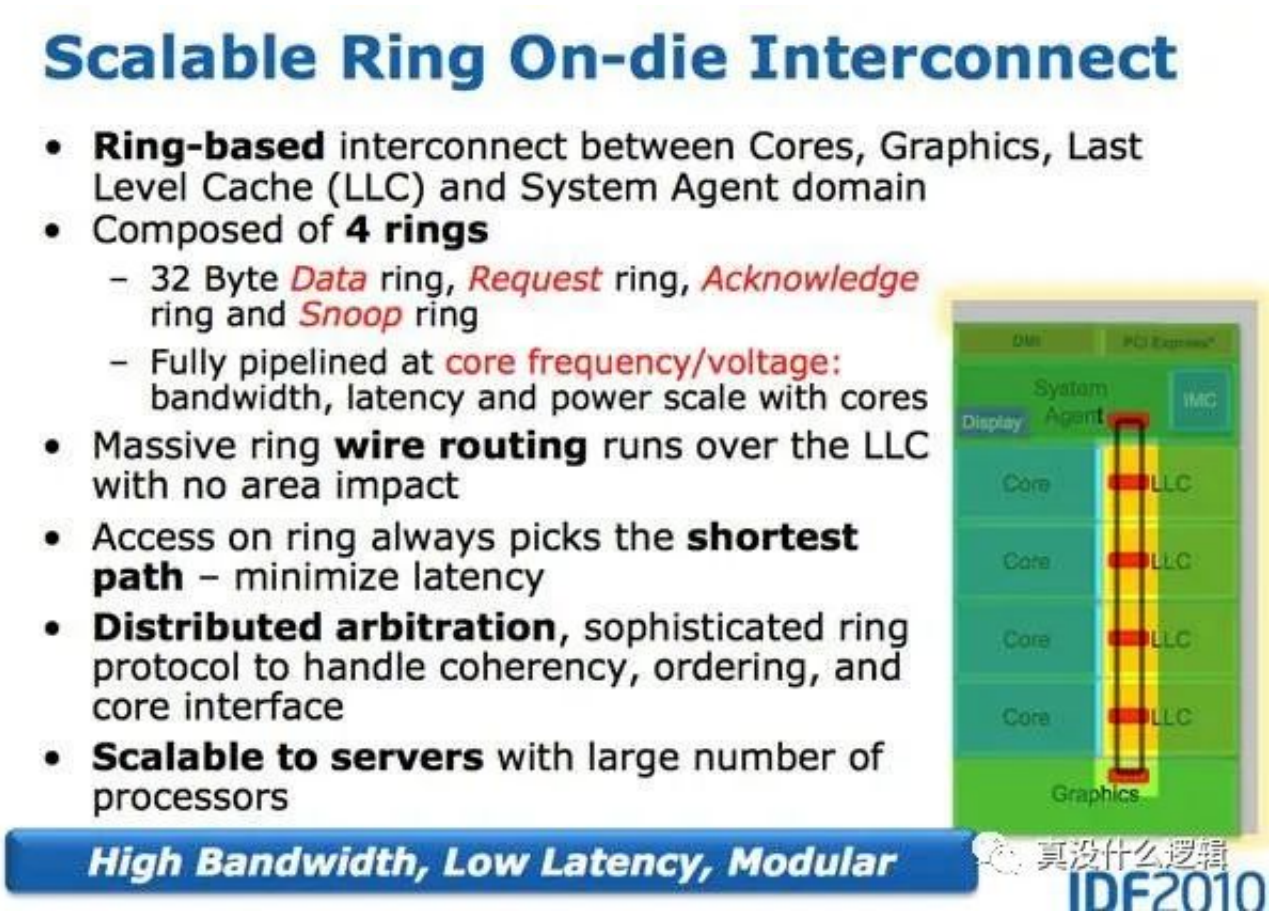

图 8 - 环形总线

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[^13]，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）[^14]：

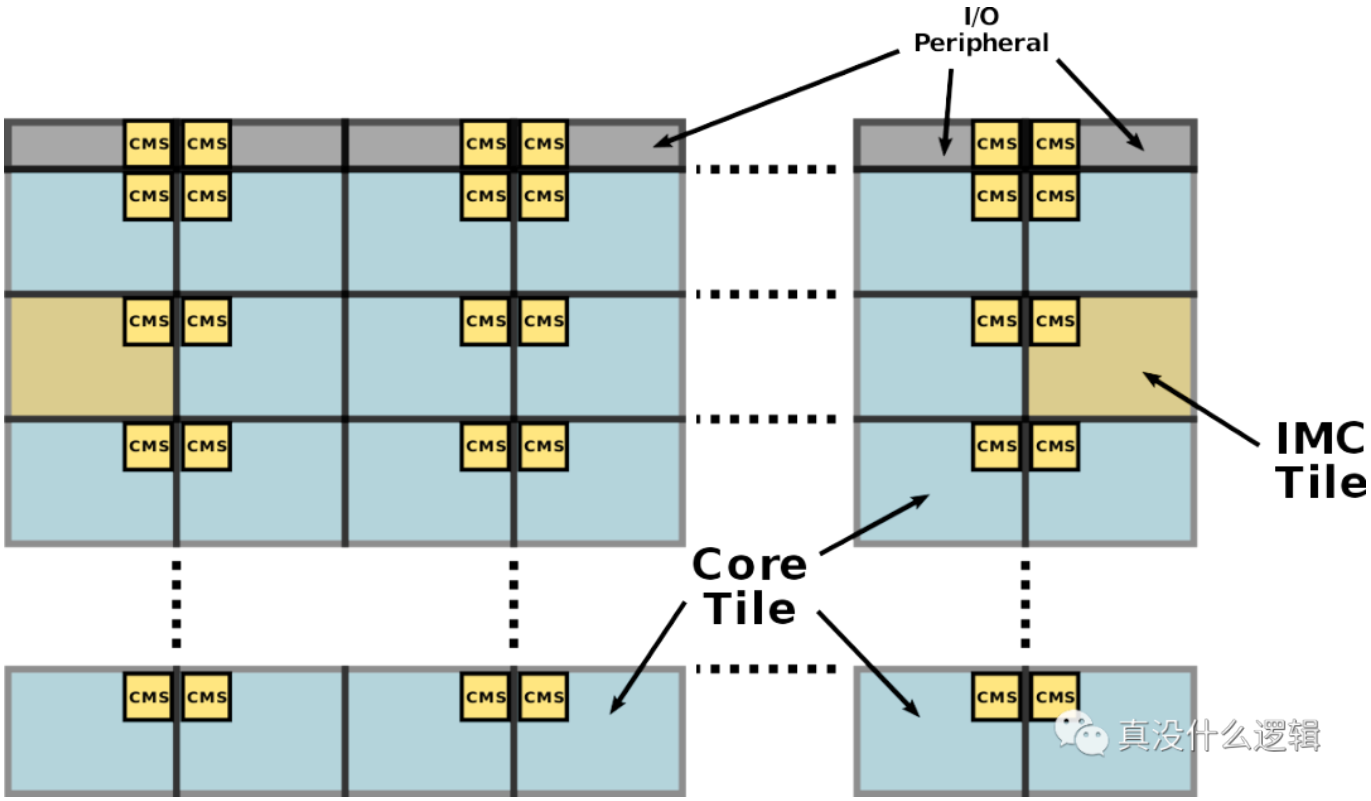

图 9 - 网格架构

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。

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[^15]。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[^16]：

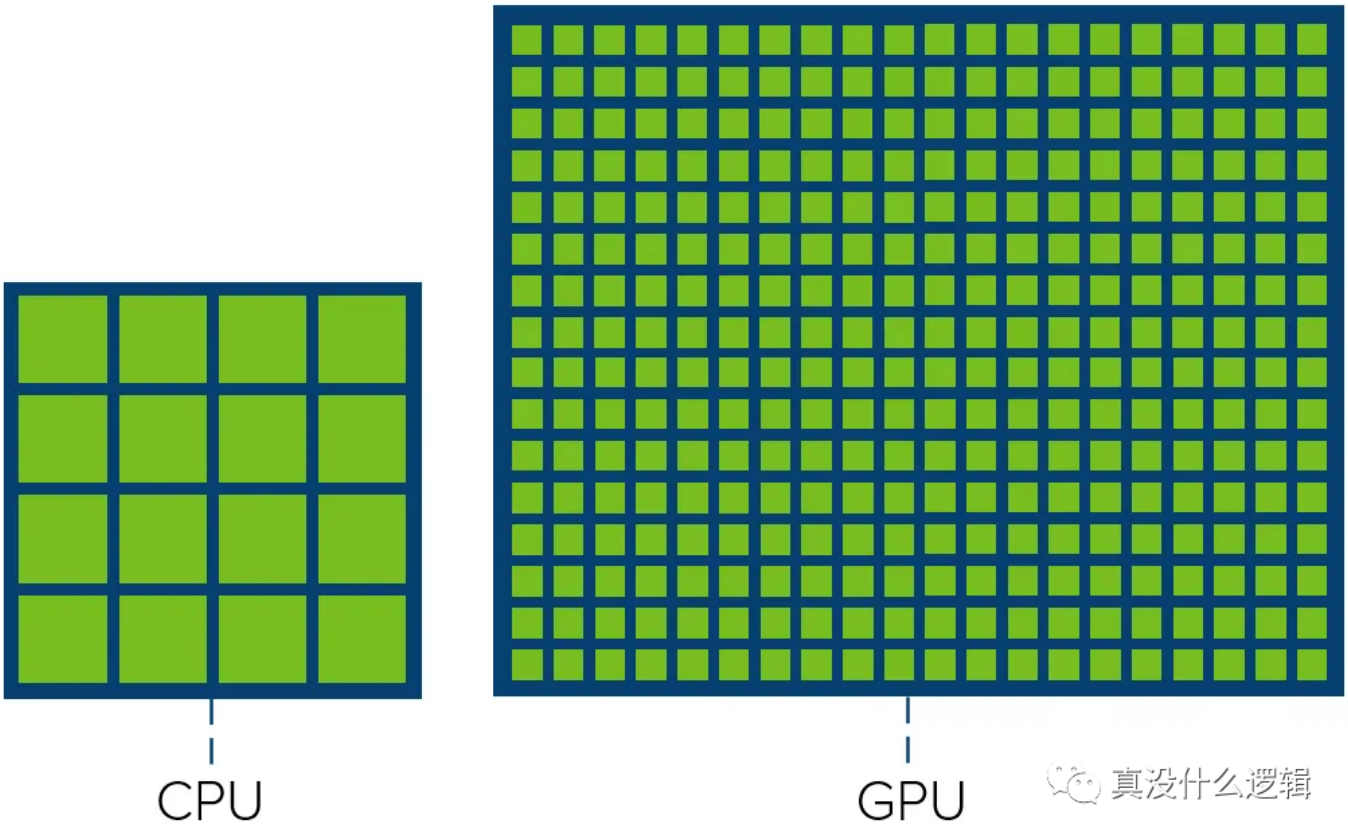

图 10 - CPU 和 GPU 的核心

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：

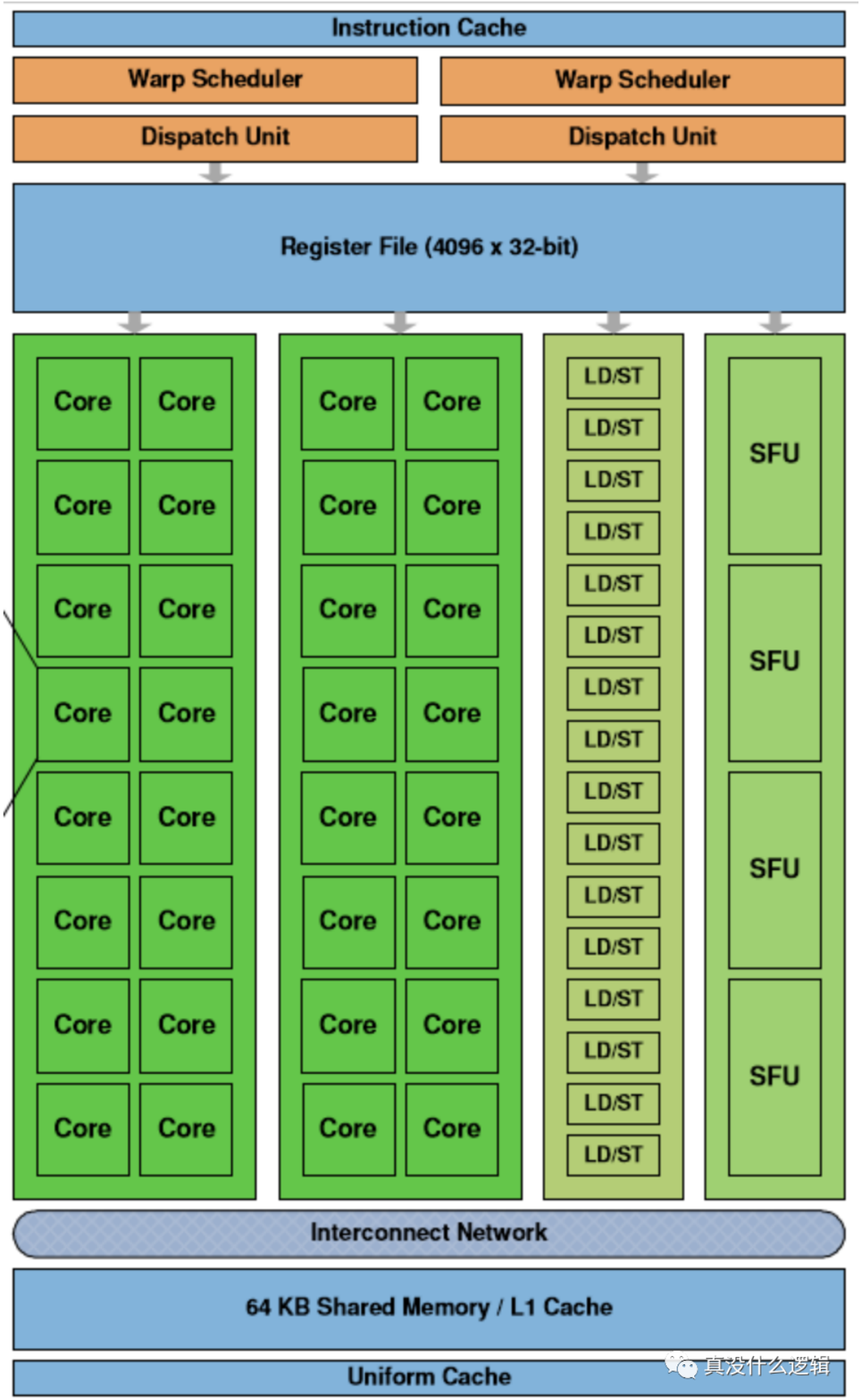
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (4096 x 32-bit)
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache

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。

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[^17] 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 12 - 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。

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 13 - 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）[^19]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 x 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[^20]。

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）。

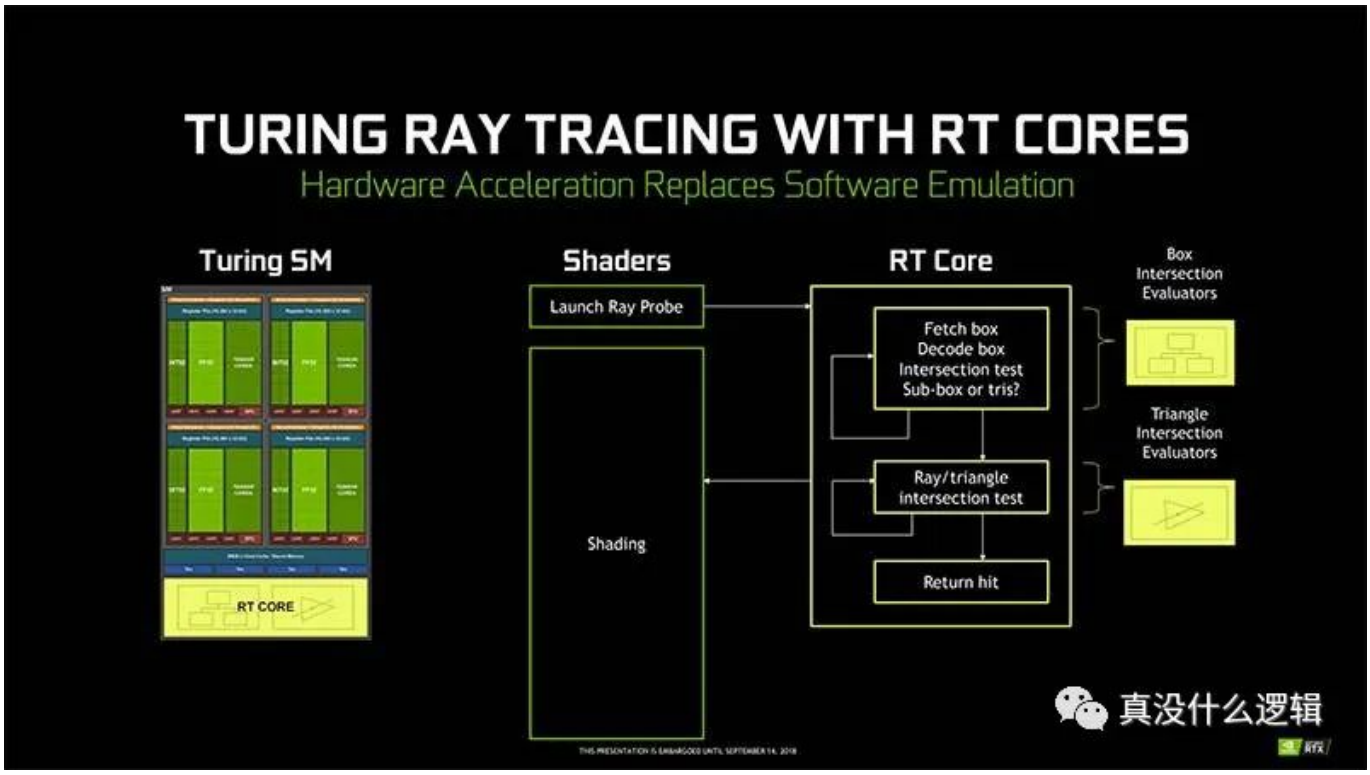

图 15 - 光线追踪核心

Nvidia 的光线追踪核心实际上是为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[^21]，而光线追踪核心可以加速这一过程。

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。

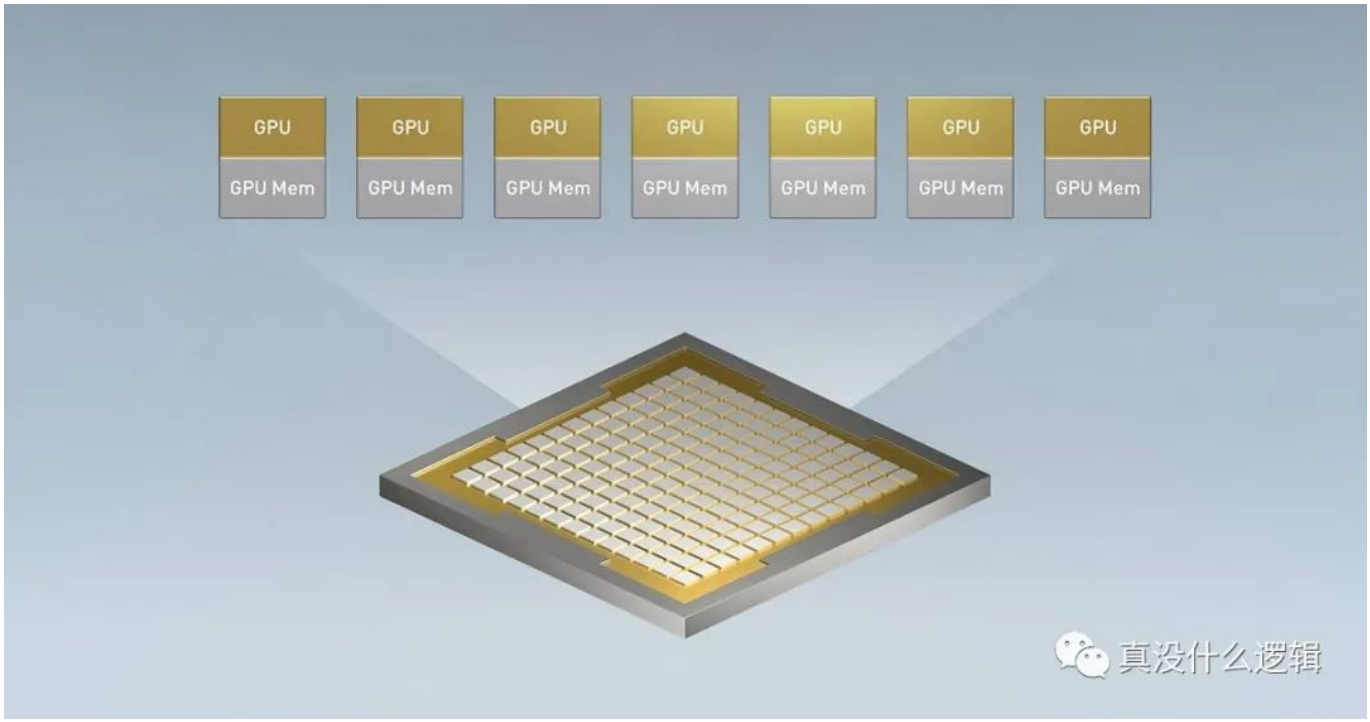

图 16 - 多实例 GPU

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[^18]。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。

总结

从 CPU 和 GPU 的演进过程我们可以看到，所有的计算单元都受益于更精细的制作工艺，我们尝试在相同的面积内放入更多的晶体管并增加更多的计算单元、使用更大的缓存，当这种『简单粗暴』的方式因为物理上的瓶颈逐渐变得困难时，我们开始为特定领域设计专门的计算单元。

文中没有提到的 ASIC 和 FPGA 是更加特殊的电路，在图像渲染领域之外，我们可以通过设计适用于特定领域的 ASIC 和 FPGA 电路提高某一项任务的性能，OSDI '20 的最佳论文 hXDP: Efficient Software Packet Processing on FPGA NICs[^23] 就研究了如何使用可编程的 FPGA 更高效地处理数据包的转发，而在未来越来越多的任务会使用专门的硬件。

推荐阅读

- An Introduction to Modern GPU Arhitecture http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf
- Wikiwand: Tick–tock model https://www.wikiwand.com/en/Tick–tock_model