# 深度学习物理层参考
工欲善其事必先利器

安军刚

2016/11/13

这个世界上唯一不变的是一直在变，所以量变会达到质变！

目录



## 一、GPU 科学计算认识

2012 年 Imagenet 比赛点燃 深度学习计算比较密集,一般使用多核思想路线。如果用 GPU 进行深度学习，那么实现大规模的神经网络成为比较实际的道路，之前很多人都在 matlab 的 nntool,2007 年开始流行的 Computer Unified Device Architecture 简称 CUDA 完成用 code 控制 GPU 完成并行计算，CUDA 编译相关硬件架构。

## 二、GPU 选择基本原则

①数理量小就要频率高
②数据量大选择显存核心 core 数量多
③有双精度计算就选开普勒架构
特斯拉 M40 专门为深度学习设计的一个卡，一般企业级研究就用。百度深度学习研究部门就用这个卡，M40 可以达到 7Tflops。

三、主要参考 GPU 参数：处理器核心数（core）、工作频率、显卡位宽、单卡/双卡/矩阵卡
一般情况下不是所有的核心都在运行计算，这样计算就是你的代码没有优化好，不是硬件资源问题。

四、GPU 的架构：我个人所了解的几部分是：Tesla，Fermi，Kepler，Maxwell，Pascal。

五、芯片型号：GT200、Gk210、GM104、GF104 常见。

六、显卡系列：GeForce、Quadro、Tesla、Tegra、NAS、NVIDIA GRID;

## 七、Geforce 显卡型号包括（G/GS　GT　GTS　GTX）

所谓的架构就是硬件的设计方式包括（流处理器核心数目；是否有 L1 和 L2 缓存；是否有双精度计算单元，这种排列组合的思想为了不同用途使用和更好实现并行计算）

芯片是对这种流水线设计的物理实现，直观体现设计思想。

例如在 GT200 中的 T 这个字母代表是一种架构，他们有 100 代芯片和 200 代的芯片。

在寄存器等方面多（GK210 和 GK110）。Tesla K80 拥有两块 GK210 芯片。要明白特斯拉架构不是特斯拉系列芯片，GeForce 是家庭娱乐级，quadro 是图行工作站，精确的线条图不是以像素数据的形式存在，而是以矢量数据定义的形式存在，如此才能保证图形计算的精确性模型一般是以精确的数学方程进行描述，在显示的时候则转换成相应的几何数据送入 GPU 进行显示的处理　geforce 是重视 directx 性能绝大多数专业制图软件不会调用这种显卡进行硬件加速。

quadro 是重视 openGL 性能的专业制图卡，相应的也会配备更大的显存，专业作图软件也会调用来进行渲染加速。

在专业级图形产品中使用与消费级相同的 GPU，Tesla 在容量较小的高性能计算市场上获得了规模经济效益。这就是所有定制高性能计算处理器与系统专业供应商被市场淘汰的主要原因，未来唯一可行的技术是基于大众市场的技术，例如 GPU。

## 八、GPU 与 CPU 计算的不同

Tesla 基于 NVIDIA CUDA，该技术最显著的特点就是能够利用 GPU 的并行计算能力，在大规模、高带宽计算中有着极大的优势。 但是，面对串行计算密集型任务，Tesla 是否有解决办法呢?GPU 及其内部的 CUDA 架构是专为并行计算而设计的。串行计算是一种有很大区别的架构，这种架构的设计目的是为了解决不同的问题。CPU 执行指令的方式就是一个接着另一个地执行。CPU 中有许多能够加速串行计算的技术。高速缓存、无次序执行、超标量技术、分支预测……均为抽取指令的技术或一系列指令的串行级并行机制。CPU 对片上高速缓存的设计与容量的依赖也非常大。如果程序大小与 CPU 高速缓存容量不匹配，那么该程序在 CPU 上的运行速度将会很慢。GPU 内部的并行计算架构围绕两个基本概念而设计。首先，程序中的数据可分成许多个部分，而为数众多的核群可以并行地处理这些数据。第二个架构方面的设想是，数据将不与高速缓存匹配。例如在图形计算或石油天然气数据处理上，数据量可能会达到兆字节甚至是太字节，用高速缓存来容纳如此巨大的数据量几乎是不切实际的。考虑到这两点设想，GPU 被设计为能够使用数以千计的线程，所有线程均并行地执行，能够访问巨大容量的本地存储器。在最新的 Tesla 产品中，每颗 GPU 均配备 4GB 存储器，可容纳待执行的数据。同时针对反复使用的数据，还设有较小的片上存储空间，GPU 所配备的巨大容量存储器等同于 CPU 内部的高速缓存，只是容量大了许多倍而已。Tesla 在高性能计算领域记目前 GPU 系统在全球高性能计算机 TOP500 排行榜中最好的成绩是第 29 位——东京大学的 Tsubame 超级计算机。 在您看来，到 2010 年之前，Tsubame 是否有机会跻身世界十强超级计算机? 还有哪些超级计算机有希望跻身世界十强或五百强?我不能代表东京工业大学超级计算领军人物的意见。但东京工业大学全球科学资讯和计算中心总监 Satoshi Matsuoka

博士曾公开表示，他打算使用 GPU 在 2010 年打造出一台荣登世界最快榜单的计算机。还有更多使用 GPU 打造的超级计算机。美国国家超级运算应用中心(NCSA)以及法国原子能委员会(CEA)是两家著名的超级计算中心，他们将跻身下一届世界五百强榜单。当前，NVIDIA GPU 的双精度性能仅相当于单精度的 8%。 到 2010 年之前，你认为这种性能可以有多大的改善？NVIDIA 会采取怎样的技术手段来确保这种性能提升呢?当前 10 系列 GPU 是首批拥有双精度的 NVIDIA 处理器。过去这种性能曾作为 GPU 的一个模块添加在 GPU 当中。而在这一代产品中，我们为每组八个单精度处理器加入了一个双精度单元。随着快速发展，未来的 GPU 将拥有更多双精度单元。由于 GPU 的性能一般每年都会翻一番，未来双精度性能将至少比当前的速度快 5 倍。中国的超级计算机用户在获取软件时，要么购买商业软件，要么自己开发，或者在商业软件方案的基础上进行二次开发。CUDA 与独立软件供应商之间有一定的合作，那么，你们有没有成熟的软件应用程序推荐给用户使用？另外，CUDA 怎样帮助那些想要自己开发软件的用户?我们与打算发布超级计算软件的独立软件供应商都进行了积极的开发工作。分子动力学领域的《NAMD/VMD》以及《GROMACS》是为群集 GPU 发布的两个应用程序例子。在美国的超级计算展会上，我们还展示了许多用于石油天然气领域地震处理、量子化学以及 Ansys 有限元设计的应用程序。对于想要自己设计和开发应用程序的开发人员，用于 CUDA 架构的并行计算开发 C 语言编译器可从 NVIDIA 网站上免费下载。Portland Group 的 Fortran 等其它编译器也正在陆续推出。有很多来自 NVIDIA 以及其它来源的程序库，这些库使应用程序更易于开发。针对这些不懂 C 语言或 Fortran 的开发人员，Accelereyes 以及 Wolfram(Mathematica)等公司还提供了 GPU 加速版的软件。因此你可以看到，利用 GPU 计算优势的方式有许多种。 今年 NVIDIA 推出了 Tesla 个人高性能计算机，并会与惠普、CRAY 公司联合发布一系列的小型高性能计算系统。 这是否意味着 NVIDIA 认为高性能计算将从大型计算机发展到台式机以及桌边型计算机上来?这对传统超大型计算机会产生怎样的影响?想要对科学技术产生最大的影响，高性能计算就必须发展到科学家们的桌面上。每一名研究人员、科学家以及工程师都应该拥有自己的超级计算机，这些计算机应该具备足够的实用性能来满足他们的工作需要。想象一下如果这些才华横溢的人们能够更快地解决问题，那么科学发展的节奏将会变得怎样。凭借这些基于 GPU 的工作站以及拥有兼容处理器的超级计算机，现在的技术计算达到了前所未有的全新水平。在接下来的几年里，GPU 将越来越多地被大型计算所采用。GPU 拥有超高的计算密度和显存带宽，足以支持这种计算性能的增长。GPU 将成为超级计算机中极其重要的动力源泉。GPU 每年的出货量数以百万计，在超级计算领域中，它现在已经成为高性能、低能耗并且是人们买得起的并行处理器。

## 九、Tesla 和 Geforce

1：T 内部和外部 RAM 都有 ECC 保护，而 G 无保护，ECC 的保护，大大降低了内存错误的发生几率；

2：T 全部通过 0 错误测试，保证计算结果的正确性，而 G 仅针对图形应用的测试，使用 T 计算得出结果是准确的，可信的。而使用 G 计算出的结果往往出现错误，经常同一个程序跑几遍，得出的结果却不一样，因此对于严谨的科学计算来说，G 的这种表现是致命的；3：T 是 Nvidia 原厂生产，

G是第三方厂商，相比质量更好；区别一：矢量点线硬件抗锯齿功能　区别二：OpenGL硬件逻辑操作　区别三：重叠图形处理　区别四：硬件加速的图形剖切　区别五：动态显存管理和UMA 区别六：专业软件认证和优化 区别七：寿命周期和技术支持芯片组平台不同：在计算机系统中，处理器的性能并不能决定一切。主板芯片组的性能同样决定着整个系统的性能。在上文中提到，工作站一般采用性能更为强劲的服务器处理器，而要充分发挥处理器的性能，需要适当的芯片组作为搭配。应用于工作站领域的芯片组一般支持双路处理器，相对于普通PC主板芯片组具有更高的前端总线，支持更大容量的内存并且支持多通道内存技术，这样可以提供更大数据吞吐量。同时，芯片组的性能也关系着专业图形的能力。

3、内存技术和容量不同：由于工作站需要长时间工作，对于系统的稳定性要求非常高。而内存如果出现错误，产生的后果是非常严重的。所以在工作站上一般应用了ECC技术，ECC被称作错误检测和纠正，可以检测1位或者4位数据错误，并且进行纠正。这样能有效避免随机出现的内存软错误，保证系统的高度稳定性。除了ECC技术之外，现在工作站内存也应用了全缓冲技术，可以串行的进行数据传输，提高了数据传输速度，并且显著提高了数据传输带宽。而桌面PC系统是不应用ECC技术和全缓冲内存技术的。在内存容量支持上来说，工作站可以支持比PC大的多的内存，比如HP去年年底发布的HP xw8600工作站，最大内存可以支持32GB。

4、存储系统不同：目前普通PC上应用的硬盘接口一般为IDE或者SATA接口，硬盘转速一般为7200转；而工作站上更多的采用SCSI或者SAS接口硬盘，转速一般可以达到10000转或者15000转，并且可以组建磁盘阵列如RAID0/1模式。这样来看磁盘系统性能的优势相对于PC是非常明显的，可以提供非常高的数据存取速度，并且可以实现数据的冗余容错。随着硬件的快速发展，计算机系统中处理器、内存的速度得到了大幅的提升，最终系统的瓶颈就落在了存储系统上，工作站上高速存储接口的应用和磁盘转速的提升，对工作站整体性能的提升是不言而喻的。

5、显示系统不同：作为主要承担图形处理任务的图形工作站，显示系统至关重要。在工作站上安装的专业显卡和民用的普通显卡具有很大区别。首先，专业显卡与普通显卡的侧重点不同。普通显卡的主要目的是很好地运行各种OpenGL和Direct3D游戏，在全屏幕下表现渲染好的三维图形，其主要追求游戏运行的流畅度，而不是精度和模型的复杂度。专业显卡的主要目的是高精度地显示各种复杂的、大规模的三维模型，而且需要显示复杂的线框模型，在一些DCC和CAD/CAM应用中，几何和光线处理的要求也很高。不同的功能侧重使得专业显卡和普通显卡的硬件设计完全不同。其次，普通显卡的驱动程序虽然也支持OpenGL，但其仅支持游戏软件常用的OpenGL子集。专业显卡由于需要运行专业应用软件，因此它必须支持OpenGL全集。这种差别使得许多高档普通显卡在运行专业软件时性能并不高。

6、操作系统不同：在普通PC应用中，为了追求更为广泛的软件兼容性，即使硬件平台为64位，用户也一般会安装32位操作系统。而在工作站应用领域则

不同，工作站需要更大的内存，而 32 位系统能够识别的内存容量仅为 3GB 左右。而 64 位操作系统可以支持 128GB 的内存，并且 64 位操作系统可以同时处理更多的数据，允许工程师创建更大、更复杂的模型，同样，通过 64 位计算，数字内容创作者（包括三维动画设计人员、数字艺术家和游戏开发人员）可以大大减少以数字方式呈现三维模型所用的时间。

7、软件驱动支持不同：对于图形工作站来说，其驱动一般经过独立软件设计商 ISV 的认证，使显卡等硬件可以更稳定更流畅的运行专业应用软件。并且，工作站整体硬件经过测试，可以完全兼容 CAD/CAM/DCC 等设计软件，比如 HP 给出了其工作站完全兼容的软件列表，能够保证这些软件稳定的运行。

8、特斯拉的 K 型号主要是双精度浮点计算和支持 ECC 内存。深度学习在此作用不突出，且没有输出接口它不是图形渲染显示主要是数据计算。所以 M 型号来专门做深度学习。

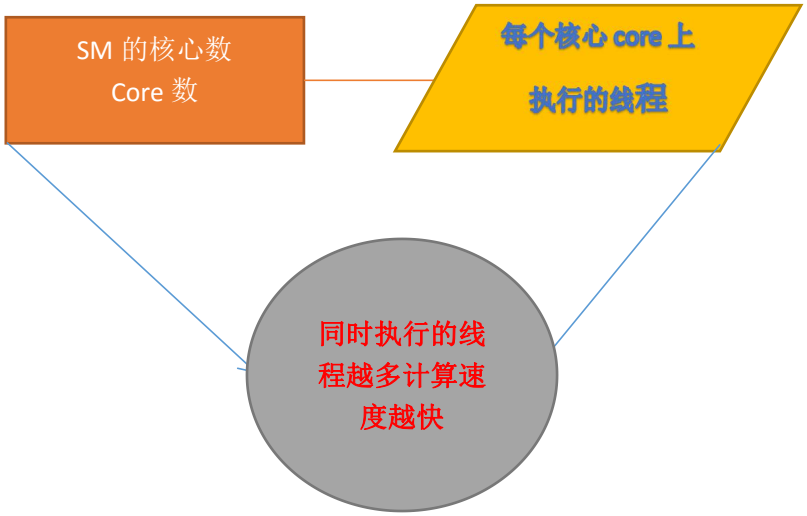
在 Fermi 架构开始就添加了 L1 和 L2 缓存硬件。包括（host interface 主机接口、copy engine 复制引擎、streaming Multiprocessors 流处理器簇、graphics processing clusters 图形处理簇、内存）

9、主机接口主要功能读取程序指令并行分配到对应的硬件单元，

10、复制引擎主要是 CPU 和 GPU 之间的复制传递，这个复制过程是可以与核函数的计算同步进行，现在深度学习计算不是计算速度快慢而是数据读入的速率，

11、数据读入是如何使用复制引擎问题（这也是一个值得研究的课题）

12、流处理器簇 SM 是 GPU 最核心的部分，（包括核心部件 warp 调度器、寄存器、core 核心数目、共享内存、它的设计个数决定 gpu 计算能力）



通常情况下 core 不会全部工作，开普勒架构中 SMX 和麦克斯韦架构的 SMM 都是流处理器簇只是不同命名。掌握 SM 结构和合理分配能大幅度提升计算。

流处理器簇的结构特定

GPU 设计为了同时处理简单任务，不像 CPU 可以处理复杂的进程，GPU 处理的问题能够分解成单元模块独立计算，在代码层就是很多个线程同时执行相同的代码，所以才有对应的大量相同处理单元 streamprocess，在这些单元上进行整形和浮点型计算，
开普勒架构中有 Gk110 芯片的 SM 结构所以擅长双精度计算。通常一个 SM 有多高 core 流处理器运算单元，要想提高计算速度还要考虑缓存，数据块在磁盘存储需要 CPU 调用到内存（高速缓存器，易失性存储器频率决定速度，内存大小决定数据吞吐量，两者乘积才是总性能），GK110 的一个 SM 架构有 192 个 core，因此一次可以执行 192 个线程，实现算法不会深入到 core 层次，SFU 这个特殊函数单元用来计算 log/sin/exp 等等，DL（load）和 Store（ST）决定执行线程所需的全部内存和局部内存。

做算法加速的话还是选台式机的比较好。性价比最高的我觉得是 GTX 980ti，从参数或者一些用户测评来看，性能并没有输给 TITAN X 多少，但价格却便宜不少。从图 1 可以看出，价位差不多的显卡都会有自己擅长的地方，根据自己的需求选择即可。要处理的数据量比较小就选择频率高的，要处理的数据量大就选显存大 core 数比较多的，有 double 的精度要求就最好选择 kepler 架构的。Tesla 的 M40 是专门为深度学习制作的，如果只有深度学习的训练，这张卡虽然贵，企业或者机构购买还是比较合适的(百度的深度学习研究院就用的这一款)，相对于 K40 单精度浮点运算性能是 4.29Tflops，M40 可以达到 7Tflops。QUADRO 系列比较少被人提起，它的 M6000 价格比 K80 还贵，性能参数上也并没有好多少。

在挑选的时候要注意的几个参数是处理器核心(core)、工作频率、显存位宽、单卡 or 双卡。有的人觉得位宽最重要，也有人觉得核心数量最重要，我觉得对深度学习计算而言处理器核心数和显存大小比较重要。这些参数越多越高是好，但是程序相应的也要写好，如果无法让所有的 core 都工作，资源就被浪费了。而且在购入显卡的时候，如果一台主机插多张显卡，要注意电源的选择。

开普勒构架中的 SMX 单元说起，与费米构架中 SM 单元不同，SMX 单元当中包含了巨大数量的 CUDA Core 核心，达到了夸张的 192 个，是原有费米的 SM 单元 CUDA Core 数量的 6 倍!完整的 GK104 核心共拥有 1536 个 CUDA Core,是 GF110 的 3 倍！而这之前，NVIDIA 对于 SM 当中的 CUDA Core 数量提升只能用保守来形容了，在 G80 时代为 16 个，GT200 之后增加到了 24 个，到了 GF100 时代才增加到 32 个，即便是算上中端产品 GF114 的 SM 单元架构，也最多不过达到了 48 个而已。NVIDIA 的此次举动绝对是非常大胆的一个突破。

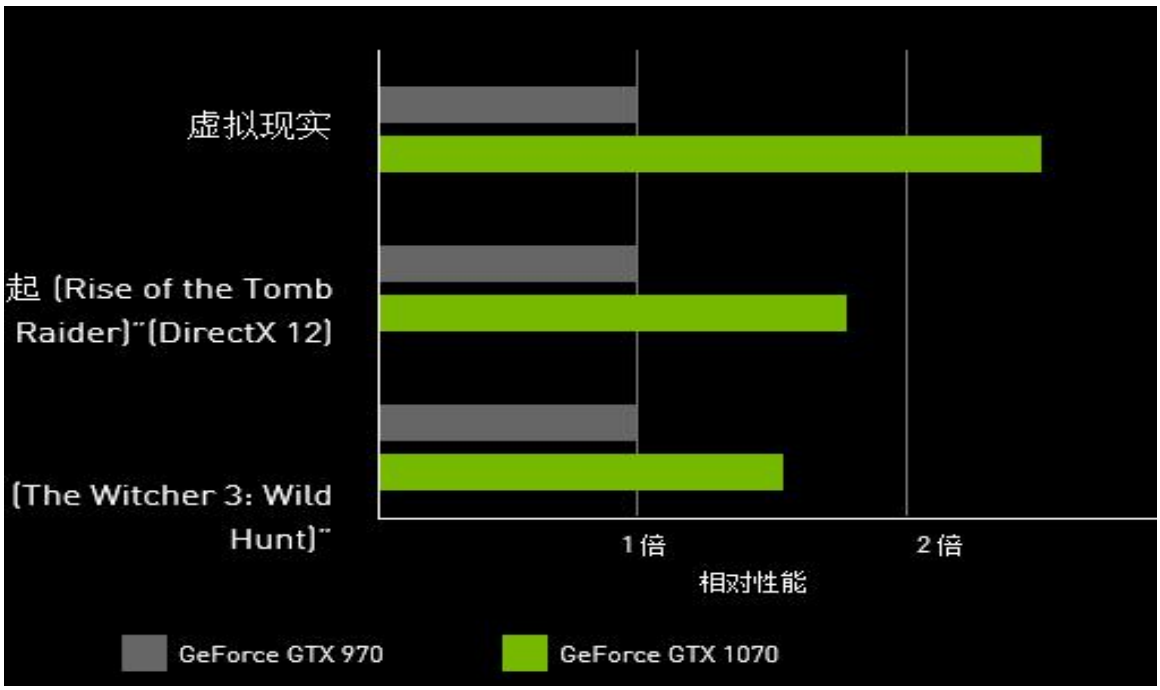
除了 CUDA Core 数量的激增外，对于 SMX 内部结构 NVIDIA 必然也要进行适当的调整，否则以原有的线程调度机制必然无法满足如此多 CUDA Core 的调度

需求。因此，NVIDIA 为每个 SMX 当中配备了四组 Warp Scheduler（Warp 调度器）以及八个 Dispatch Unit（分派单元），大大加强了 CUDA Core 的任务派发能力。

除此以外，Instrucition Cache（指令高速缓存）、Register File（寄存器文件）、64KB Shared Memory/L1 Cache（64KB 高速缓存）、Uniform Cache（统一高速缓存）等并没有太大变化。

十 GPU 关键技术、从以下几个方面我们对比 GPU 参数比对

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3D vision | | | | | | | |
| 4K | | | | | | | |
| Battery Boost | | | | | | | |
| CUDA | | | | | | | |
| DirectX12 | | | | | | | |
| DSR | | | | | | | |
| G-SYNC | | | | | | | |
| GPU boost2.0 | | | | | | | |
| MFAA | | | | | | | |
| Optimus | | | | | | | |
| PhysX | | | | | | | |
| SLI | | | | | | | |
| Surround | | | | | | | |
| VXGI | | | | | | | |
| 自适应垂直同步 | | | | | | | |
| 虚拟现实 | GeForce GTX 1080 | GeForce GTX 1070 | GeForce GTX TITAN X | GeForce GTX 980 Ti | GeForce GTX 980 | GeForce GTX 970 | GeForce GTX 960 |




芯片厂商 NVIDIA
显卡芯片 GeForce GTX 680
显示芯片系列 NVIDIA GTX 600 系列
制作工艺 28 纳米
核心代号 GK104

| | |
|---|---|
| 显存规格 | • 显存类型 GDDR5<br>• 显存容量 2048MB<br>• 显存位宽 256bit<br>• 显存速度 0.5ns<br>• 最大分辨率 2560×1600 |
| 显卡散热 | • 散热方式 涡轮风扇+散热片 |
| 显卡接口 | • 接口类型 PCI Express 3.0 16X<br>• I/O 接口 HDMI 接口/双 DVI 接口/DisplayPort 接口<br>• 电源接口 6pin+6pin |
| 物理特性 | 流处理单元 1536 个<br>• 晶体管数量 35 亿个 |

上面这两块显卡看似很普通实际上在价格和性能都很好。

| 项目 \ 显卡 | 680M | 965M | 965M OC | OC前领先幅度 | OC后领先幅度 | OC前后提升 |
|---|---|---|---|---|---|---|
| 3DMARK11 P模式 | 7574 | 7085 | 9025 | -6.46% | 19.16% | 27.38% |
| 3DMARK11 X模式 | 2343 | 2138 | 2704 | -8.75% | 15.41% | 26.47% |
| FireStrike 1080P | 5273 | 5325 | 6735 | 0.99% | 27.73% | 26.48% |
| FireStrike 4K | 1177 | 1175 | 1515 | -0.17% | 28.72% | 28.94% |
| 满载温度 | 87 | 59 | 82 | 32.18% | 5.75% | -38.98% |

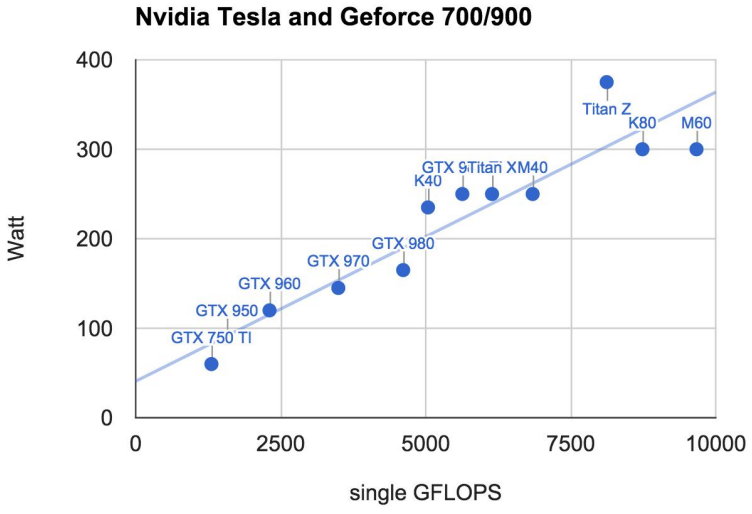
相关指标与 深度学习与机器学习中算法和编码有关的 arry 数组，Mat 矩阵，向量，Log/sin 特殊函数。字节比特率，计算单元，内存，缓存，功耗，时钟频率，

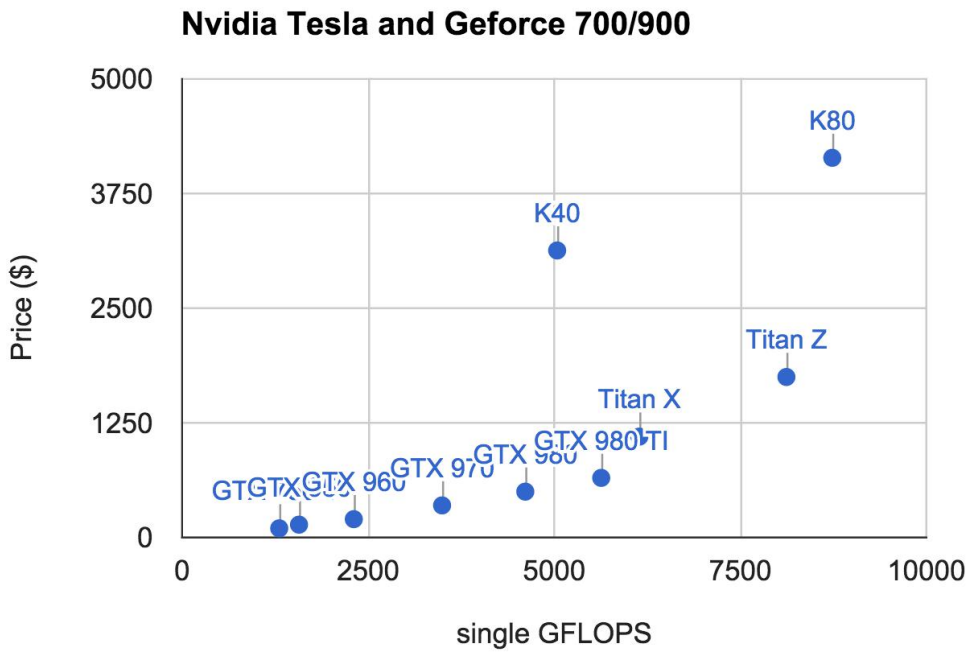
我们买 GPU 的目的是用来做科学计算，例如针对深度学习。另外两个 GPU 主要用途——游戏，挖矿——则不在此文讨论范围。推荐直接上大厂整体 GPU 集群解决方案，可省去大量力气。

买什么样的 GPU

对于很多科学计算而言，性能主要决定于 GPU 的浮点运算能力。特别是对深度学习任务来说，主要是单精度浮点运算（未来可能默认用更低的精度，例如 16bit）。而其他因素，例如 CPU 速度，内存大小，通讯带宽，对目前大多数深度学习任务而言，只要过了某个合理的阈值不够成性能瓶颈就行。

对于 GPU 而言，简单来看有三个重要参数，浮点运算能力，价格与功耗。下面两个图比较了 Nvidia Tesla，Geforce 700 和 900 系列各卡的这三个参数（前两个参考了 wikipedia，）。






整体来看浮点运算和功耗成正比。在价格上，如不考虑土豪用 Tesla 系列话，N 厂在消费级别卡上是一分钱一分货。目前的购买建议是优先考虑 Titan X，但如果不是特别需要其 12GB 内存的话,则考虑浮点运算性价比更好的 980 TI(6GB 内存）。但如果嫌 980 TI 功耗比过高的话，则考虑 980. 除非有特别的理由，不推荐 980 之下的卡了。因为达到同样的计算能力，使用更便宜的卡会增加机器数量，可能导致其他配套成本（例如 CPU，电源，主板）和维护成本的增加。（下面将会有血与泪的教训）。

GPU 更新换代快，不宜大规模采购超出现在需求的机器。例如 Nvidia 下一代号称 3D 内存可能会带来新的加速，通常认为性价比更高的 AMD 也将会推出新的编译器来更好支持科学计算。

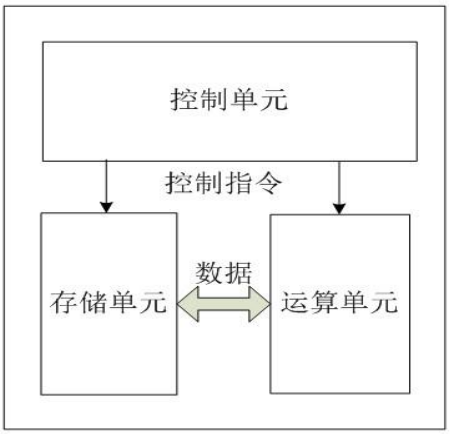
选好了 GPU 后便可以配套其他了，与正常装机无异。但有两点需要额外考虑。一是供电。一块 GPU 很可能有 200w，4 卡则 800w 了，加上 CPU 内存，电源至少要个 1kw 才行。二是散热。GPU 发热巨大，将多块 GPU 并排放会导致很严重的散热问题。此外，如果大量安装 GPU 机器，机房供电和散热也会是需要考虑。

折腾手记

## 第二章

### 一、CPU 组成及其计算工作原理：

中央处理单元(Central Processing Unit)的缩写
CPU 主要由运算器、控制器、寄存器组和内部总线等构成
芯片：是指将电子逻辑门电路用激录到硅片上，从而构成各种各样的芯片，当今集成度最高、功能最强大的应该 CPU 芯片了。 2、CPU：是指所有时期，各种电子元件构成的计算机中央处理器的统称
我们都知道 CPU 的根本任务就是执行指令，对计算机来说最终都是一串由“0”和“1”组成的序列。CPU 从逻辑上可以划分成 3 个模块，分别是控制单元、运算单元和存储单元，这三部分由 CPU 内部总线连接起来




控制单元:控制单元是整个 CPU 的指挥控制中心,由指令寄存器 IR(Instruction Register)、指令译码器 ID(Instruction Decoder)和操作控制器 OC(Operation Controller)等，对协调整个电脑有序工作极为重要。它根据用户预先编好的程序，依次从存储器中取出各条指令，放在指令寄存器 IR 中，通过指令译码(分析)确定应该进行什么操作，然后通过操作控制器 OC，按确定的时序，向相应的部件发出微操作控制信号。操作控制器 OC 中主要包括节拍脉冲发生器、控制矩阵、时钟脉冲发生器、复位电路和启停电路等控制逻辑。
运算单元：是运算器的核心。可以执行算术运算(包括加减乘数等基本运算及其附加运算)和逻辑运算(包括移位、逻辑测试或两个值比较)。相对控制单元而言，运算器接受控制单元的命令而进行动作，即运算单元所进行的全部操作都是由控制单元发出的控制信号来指挥的，所以它是执行部件。
存储单元：包括 CPU 片内缓存和寄存器组，是 CPU 中暂时存放数据的地方，里面保存着那些等待处理的数据，或已经处理过的数据，CPU 访问寄存器所用的时间要比访问内存的时间短。采用寄存器，可以减少 CPU 访问内存的次数，从而提高了 CPU 的工作速度。但因为受到芯片面积和集成度所限，寄存器组的容量不可能很大。寄存器组可分为专用寄存器和通用寄存器。专用寄存器的作用是固定的，分别寄存相应的数据。而通用寄存器用途广泛并可由程序员规定其用途，通用寄存器的数目因微处理器而异。这个是我们以后要介绍这个重点，这里先提一下。

## 二、CPU 的工作原理概括如下：

总的来说，CPU 从内存中一条一条地取出指令和相应的数据，按指令操作码的规定，对数据进行运算处理，直到程序执行完毕为止。

上图中我没有画总线，只是用逻辑方式对其进行呈现。原因早期 Intel 的微处理器，诸如 8085，8086/8088CPU，普遍采用了地址总线和数据总线复用技术，即将部分(或全部)地址总线与数据总线共用 CPU 的一些引脚。例如 8086 外部地址总线有 20 根，数据总线复用了地址总线的前 16 根引脚。复用的数据总线和地址总线虽然可以少 CPU 的引脚数，但却引入了控制逻辑及操作序列上的复杂性。所以，自 80286 开始，Intel 的 CPU 才采用分开的地址总线和数据总线。

不管是复用还是分开，对我们理解 CPU 的运行原理没啥影响，上图没画总线的目的就是怕有些人太过于追求细节，一头扎下去，浮不起来，不能从宏观上藐视敌人。

OK，总结一下，CPU 的运行原理就是：控制单元在时序脉冲的作用下，将指令计数器里所指向的指令地址(这个地址是在内存里的)送到地址总线上去，然后 CPU 将这个地址里的指令读到指令寄存器进行译码。对于执行指令过程中所需要用到的数据，会将数据地址也送到地址总线，然后 CPU 把数据读到 CPU 的内部存储单元(就是内部寄存器)暂存起来，最后命令运算单元对数据进行处理加工。周而复始，一直这样执行下去，天荒地老，海枯枝烂，直到停电。

如果你对这段话还是觉得比较晕乎，那么就看我们老师是怎么讲的：

1、取指令：CPU 的控制器从内存读取一条指令并放入指令寄存器。指令的格式一般是这个样子滴：

操作码就是汇编语言里的 mov,add,jmp 等符号码；操作数地址说明该指令需要的操作数所在的地方，是在内存里还是在 CPU 的内部寄存器里。

2、指令译码：指令寄存器中的指令经过译码，决定该指令应进行何种操作(就是指令里的操作码)、操作数在哪里(操作数的地址)。

3、 执行指令，分两个阶段“取操作数”和“进行运算”。

4、 修改指令计数器，决定下一条指令的地址。

CPU 从存储器或高速缓冲存储器中取指令，放入指令寄存器，并对指令译码。它把指令分解成一系列的微操作，然后发出各种控制命令，执行微操作系列，从而完成一条指令的执行。指令是计算机规定执行操作的类型和操作数的基本命令。指令是由一个字节或者多个字节组成，其中包括操作码字段、一个或多个有关操作数地址的字段以及一些表征机器状态的状态字以及特征码。有的指令中也直接包含操作数本身。

## 三、计算流程：

第一阶段，提取：从存储器或高速缓冲存储器中检索指令(为数值或一系列数值)。由程序计数器(Program Counter)指定存储器的位置，程序计数器保存供

识别目前程序位置的数值。换言之，程序计数器记录了CPU在目前程序里的踪迹。提取指令之后，程序计数器根据指令长度增加存储器单元。指令的提取必须常常从相对较慢的存储器寻找，因此导致CPU等候指令的送入。这个问题主要被论及在现代处理器的快取和管线化架构。

第二阶段解码：CPU根据存储器提取到的指令来决定其执行行为。在解码阶段，指令被拆解为有意义的片断。根据CPU的指令集架构(ISA)定义将数值解译为指令。一部分的指令数值为运算码(Opcode)，其指示要进行哪些运算。其它的数值通常供给指令必要的信息，诸如一个加法(Addition)运算的运算目标。这样的运算目标也许提供一个常数值(即立即值)，或是一个空间的定址值：暂存器或存储器位址，以定址模式决定。在旧的设计中，CPU里的指令解码部分是无法改变的硬件设备。不过在众多抽象且复杂的CPU和指令集架构中，一个微程序时常用来帮助转换指令为各种形态的讯号。这些微程序在已成品的CPU中往往可以重写，方便变更解码指令。

第三阶段执行：在提取和解码阶段之后，接着进入执行阶段。该阶段中，连接到各种能够进行所需运算的CPU部件。例如，要求一个加法运算，算数逻辑单元(ALU, Arithmetic Logic Unit)将会连接到一组输入和一组输出。输入提供了要相加的数值，而输出将含有总和的结果。ALU内含电路系统，易于输出端完成简单的普通运算和逻辑运算(比如加法和位元运算)。如果加法运算产生一个对该CPU处理而言过大的结果，在标志暂存器里，运算溢出(Arithmetic Overflow)标志可能会被设置。

第四阶段：最终阶段，写回，以一定格式将执行阶段的结果简单的写回。运算结果经常被写进CPU内部的暂存器，以供随后指令快速存取。在其它案例中，运算结果可能写进速度较慢，但容量较大且较便宜的主记忆体中。某些类型的指令会操作程序计数器，而不直接产生结果。这些一般称作 ldquo;跳转(Jumps)，并在程式中带来循环行为、条件性执行(透过条件跳转)和函式。许多指令也会改变标志暂存器的状态位元。

这些标志可用来影响程式行为，缘由于它们时常显出各种运算结果。例如，以一个比较指令判断两个值的大小，根据比较结果在标志暂存器上设置一个数值。这个标志可藉由随后的跳转指令来决定程式动向。在执行指令并写回结果之后，程序计数器的值会递增，反覆整个过程，下一个指令周期正常的提取下一个顺序指令。如果完成的是跳转指令，程序计数器将会修改成跳转到的指令

位址，且程序继续正常执行。许多复杂的 CPU 可以一次提取多个指令、解码，并且同时执行。这个部分一般涉及经典 RISC 管线，那些实际上是在众多使用

运算逻辑部件

运算逻辑部件，可以执行定点或浮点的算术运算操作、移位操作以及逻辑操作，也可执行地址的运算和转换。

寄存器部件

寄存器部件，包括通用寄存器、专用寄存器和控制寄存器。通用寄存器又可分定点数和浮点数两类，它们用来保存指令中的寄存器操作数和操作结果。通用寄存器是中央处理器的重要组成部分，大多数指令都要访问到通用寄存器。通用寄存器的宽度决定计算机内部的数据通路宽度，其端口数目往往可影响内部操作的并行性。专用寄存器是为了执行一些特殊操作所需用的寄存器。控制寄存器通常用来指示机器执行的状态，或者保持某些指针，有处理状态寄存器、地址转换目录的基地址寄存器、特权状态寄存器、条件码寄存器、处理异常事故寄存器以及检错寄存器等。有的时候，中央处理器中还有一些缓存，用来暂时存放一些数据指令，缓存越大，说明 CPU 的运算速度越快，目前市场上的中高端中央处理器都有 2M 左右的二级缓存，高端中央处理器有 4M 左右的二级缓存（CPU 这个芯片结构特别复杂，尤其发展到现在几种芯片架构。一般情况下应该驱动硬件方面的工程和芯片架构师懂，但是要把英特尔芯片流水线都完全掌握绝对不可能，这个公司核心技术没有几个人能完全掌握不然它不会是教主一直带领所以全世界的 FPGA）。

## 四、CPU 结构分析与性能优化（这部分资料参考计算机系统结构）

流水线技术

1. 流水线的基本概念及分类
2. DLX 的基本流水线
3. 流水线性能分析（时空图、吞吐率、加速比、效率、消除流水线瓶颈段的方法）
4. 流水线中的相关及解决方法（结构相关、数据相关、控制相关、定向技术、指令调度、分支预测、延迟分支等）
5. MIPS R4000 流水线计算机简介
6. 向量处理机（向量处理方式、向量处理机的概念、提高向量处理机性能的主要技术、向量处理机的性能评价）

指令级并行

1. 指令级并行的概念（循环展开，相关）

2. 指令的动态调度（动态调度的原理、记分牌技术、Tomasulo 算法）
3. 控制相关的动态解决技术（分支预测缓冲、分支目标缓冲、基于硬件的前瞻执行）
4. 多指令流出技术（静态超标量、动态多指令流出、超长指令字）

多处理机

1. 并行计算机系统结构的分类
2. 通信模型和存储器的结构模型
3. 对称式共享存储器系统结构（多处理机 Cache 一致性、实现一致性的基本方案、监听协议及其实现）
5. 互连网络（互连网络的性能参数、静态连接网络、动态连接网络）
6. 同步（基本硬件原语、用一致性实现锁、同步性能问题、大规模机器的同步）

ISA 一个处理器支持的指令和指令的字节级编码称为它的指令集体系结构

虽然每个厂商制造的处理器性能和复杂性不断提高，但是不同型号在 ISA 级别上都保持着兼容。因此，ISA 在编译器编写者和处理器设计人员之间提供了一个概念抽象层。

这个概念抽象层即 ISA 模型：CPU 允许的指令集编码，且顺序地执行指令，也就是先取出一条指令，等到她执行完毕，再开始下一条。然而，现代处理器的实际工作方式可能跟 ISA 隐含的计算模型大相径庭。通过同时处理多条指令的不同部分，处理器可以获得较高的性能。但其必须对外表现出符合 ISA 模型的执行结果。

在计算机科学中，用巧妙的方法在提高性能的同时，又保持一个更简单、更抽象模型的功能，这种思想是众所周知的（抽象）。

CPU 硬件简介

大多数现代电路设计都是用信号线上的高电压和低电压来表示不同的位值。

要实现一个数字系统需要三个主要的组成部分：

①计算对位进行操作的函数的组合逻辑(ALU)

②存储位的存储器元素(寄存器)

③控制存储器元素更新的时钟信号

逻辑门是数字电路的基本计算元素，它们产生的输出，等于它们输入位值的某个布尔函数。

将很多逻辑门组合成一个网，就能构建计算块，称为组合电路。(相当于一个表达式)

算术/逻辑单元(ALU)是一种很重要的组合电路，这个电路有三个输入：两个数据输入及一个控制输入。根据控制输入的设置，电路会对数据输入执行不同的算术或逻辑操作。

存储器和时钟

任何信息。它们只是简单地响应输入信号，产生等于输入的某个函数的输出。

为了产生时序电路，也就是有状态并且在这个状态上进行计算的系统，我们必须引入按位存储信息的设备。

存储设备都是由同一个时钟控制，时钟是一个周期性信号，决定了什么时候要把新值加载到设备中。

大多数时候，寄存器都保持在稳定状态(用 x 表示)，产生的输出等于它的当前状态。信号沿着寄存器前面的组合逻辑传播，这时，产生了一个新的寄存器输入(用 y 表示)，但只要时钟是低电位的，寄存器的输出就仍然保持不变。当时钟变成高电位的时候，输入信号才加载到寄存器中，成为下一个状态 y，直至下一个时钟的上升沿。

寄存器是作为电路不同部分中的组合逻辑之间的屏障。每当每个时钟到达上升沿时，值才会从寄存器的输入传送到输出。

寄存器文件(通用寄存器组成的逻辑块) 有两个读端口，还有一个写端口。电路可以读两个程序寄存器的值，同时更新第三个寄存器的状态。每个端口都有一个地址输入，表明选择哪个程序寄存器。

虽然寄存器文件不是组合电路，因为它有内部存储。不过，从寄存器文件读数据就好像它是一个以地址为输入、数据为输出的一个组合逻辑块。

指令编码

指令集的一个重要性质就是字节编码必须有唯一的解释。任意一个字节序列要么是一个唯一的指令序列的编码，要么就不是一个合法的字节序列。因为每条指令的第一个字节有唯一的代码和功能组合，给定这个字节，我们就可以决定所有其他附加字节的长度和含义。

每条指令需要 1——6 个字节不等，这取决于需要哪些字段。每条指令的第一个字节表明指令的类型：高 4 位是代码部分(例：6 为整数类操作指令)，低 4 位是功能部分(例：1 为整数类中的减法指令) 61 合起来即为 sub 指令。

处理一条指令的序列：

取指(fetch)

取值阶段从存储器读取指令字节，放到指令存储器(CPU 中)中，地址为程序计数器(PC)的值。

它按顺序的方式计算当前指令的下一条指令的地址(即 PC 的值加上已取出指令的长度)

译码(decode)

ALU 从寄存器文件(通用寄存器的集合)读入最多两个操作数。（即一次最多读取两个寄存器中的内容）

执行(execute)

在执行阶段会根据指令的类型，将算数/逻辑单元(ALU)用于不同的目的。对其他指令，它会作为一个加法器来计算增加或减少栈指针，或者计算有效地址，或者只是简单地加 0，将一个输入传递到输出。

条件码寄存器(CC)有三个条件位。ALU 负责计算条件码新值。当执行一条跳转指令时，会根据条件码和跳转类型来计算分支信号 cnd。

访存(memory)

访存阶段，数据存储器(CPU 中)读出或写入一个存储器字。指令和数据存储器访问的是相同的存储器位置，但是用于不同的目的。

写回(write back)

写回阶段最多可以写两个结果到寄存器文件。寄存器文件有两个写端口。端口 E 用来写 ALU 计算出来的值，而端口 M 用来写从数据存储器中读出的值。

更新 PC(PC update)

根据指令代码和分支标志，从前几步得出的信号值中，选出下一个 PC 的值。

我们以SEQ(sequential 顺序的)处理器为例讲解CPU的基本原理。每个时钟周期上，SEQ执行处理一条完整指令所需的所有步骤。不过这需要一个很长的时钟周期时间，因此时钟周期频率会低到不可接受。

SEQ的时序

组合逻辑不需要任何时序或控制——只要输入变化了，值就通过逻辑门网络传播。

我们也将读随机访问存储器(寄存器文件、指令存储器和数据存储器)看成和组合逻辑一样的操作。（写随机访问存储器需要等待高电平）

由于指令存储器只用来读指令，因此我们可以将这个单元看成是组合逻辑。（内存向指令存储器中写指令是CPU外部的事件 不属于CPU内的时序）

每个时钟周期，程序计数器都会装载新的指令地址。

只有在执行整数运算指令时，才会装载条件码寄存器。

只有在执行mov、push、call指令时，才会写数据存储器。

要控制处理器中活动的时序，只需要寄存器和存储器的时钟控制。

因为指令运行计算的结果，写入寄存器或存储器中。

我们可以把取指、译码、执行等过程看做是组合逻辑的处理过程（因为它们不涉及写入寄存器）。把写回看做是另一个过程。

则整个过程可简化为下图所示：

【举例详解】

有如下指令：

```
0x000 :  irmovl  $0x100, %ebx
0x006 :  irmovl  $0x200, %edx
0x00c :  addl   %edx, %ebx
0x00e :  je  dest
0x013 :  rmmovl  %ebx, 0(%edx)
0x019 :  dest: halt
```

在我们的SEQ处理器中，一个时钟周期(即两次高电平时间的时间间隔)执行一条指令。

时钟周期3开始时(点1处)，一个高电平打入，地址0x00c载入程序计数器PC中。这样，与PC相连的MCU(主存控制单元)就在内存中把地址0x00c处的addl指令提取出来，加载到指令存储器中。（从内存中读取数据很慢，这个过程会很久，所以我们的时钟周期要很长，才能做到一个时钟周期执行一条指令）同时，PC的值加上addl指令的长度，得出新PC值，新PC值通过总线传播，等待下次高电平时写入PC。

组合逻辑指令存储器中的输入一变化，值(addl 指令)就通过逻辑门网络传播。故，瞬间读出了寄存器文件中%edx、%ebx 的值（因为读寄存器文件不需要高电平触发）

读出的%edx、%ebx 的值瞬间流动到组合逻辑 ALU 中，ALU 根据之前传播的 addl 指令，知道此为加法指令，瞬间计算出这两个值的结果 valE。valE 通过总线传播瞬间到达寄存器文件，但是此时还不能向寄存器文件写入，必须等待下次的高电平。

故此时，寄存器文件和存储器中保存的还都是上条指令的结果值。（点 1、2 处）

时钟周期 4 开始时(点 3 处)，一个高电平打入，上周期产生的新 PC 值写入程序计数器，上周期计算得到的 addl 的结果 valE 值写入寄存器文件中的%ebx 中。

因为地址 0x00e 载入了程序计数器中，故会取出并执行跳转指令 je。因为条件码 ZF 为 0，所以不会选择分支。在这个周期末尾(点 4)，程序计数器已经产生了新值 0x013。但是直到下个周期开始之前，寄存器和存储器中的状态还是保持着 addl 指令设置的值

【如此例所示，用时钟来控制状态元素的更新，以及值通过组合逻辑来传播，足够控制我们 SEQ 实现中每条指令执行的计算了。每次时钟由低变高时，处理器开始执行一条新指令。】

【读操作沿着这些单元传播，就好像它们是组合逻辑，而写操作是由时钟控制的。】

【注意】（个人理解）

早期的没有流水线的 CPU 可不是一个周期执行一条指令，我们的 SEQ 处理器只不过是为了讲解 CPU 的时序而特意做成的一个周期执行一条指令，这样做，使得一个时钟周期的时间特别长（因为我们要等主存把指令加载到指令寄存器，有的指令还要等待数据寄存器把数据写入到主存）。

若按照执行时间最长的指令的执行时间作为时钟周期，则时钟的粒度太大，因为不同的指令需要的执行时间不同，时钟粒度太大会导致有些指令早早执行完毕，但 CPU 还得闲着，等待本周期结束。（我们的六步划分是针对所有指令整体而言的，很多指令只经历其中几步）

故早期的 CPU 设计者,把由一个大组合逻辑完成的执行,分割成几个阶段,由几个小组合逻辑完成。中间插入寄存器保存中间结果，就像后面讲的流水线机制那样，只是指令顺序进入，一条指令运行完，下条指令才开始进入。

这样做的好处是，由于各种指令涉及的阶段不同，有的指令经历较少的阶段就完成了，有的指令要经历较多的阶段，运行一条指令所需的时间不同了，小于等于最耗时指令的时间。（而我们设计的 SEQ 处理器每条指令都要经历同样的大组合逻辑，时钟周期只能定为最耗时指令的时间）

[类比内存管理的分页机制，提高内存利用率。]

1978 年的 Intel 8086，需要多个(通常是 3~10 个)时钟周期来执行一条指令。比较先进的处理器可以保持每个时钟 2~4 条指令的执行速率。其实每条指令从开始到结束需要长的多的时间，大约 20 个或者更多的周期，但是处理器使用了非常多的聪明技巧来同时处理多达 100 条的指令。

流水线原理

我们通过将执行每条指令所需的步骤组织成一个统一的流程，就可以用很少量的各种硬件单元以及一个时钟来控制计算的顺序，从而实现整个处理器。不过这样一来，控制逻辑就必须要在这些单元之间路由信号，并根据指令类型和分支条件产生适当的控制信号。（CPU 内有三种总线：控制总线、地址总线、数据总线）

SEQ 处理器不能充分利用硬件单元，因为每个单元只在整个时钟周期的一部分时间内才被使用。我们会看到引入流水线能获得更好的性能。

在流水线化的系统中，待执行的任务被划分成了若干独立的阶段。

例如在汽车清洗中，这些阶段包括喷水、打肥皂、擦洗、上蜡和烘干。通常都会允许多个顾客同时经过系统，而不是要等到一个用户完成了所有从头至尾的过程才让下一个开始。

当前面一辆汽车从喷水阶段进入擦洗阶段时，下一辆就可以进入喷水阶段了。通常，汽车必须以相同的速度通过这个系统，以避免撞车。

流水线化的一个重要特性就是增加了系统的吞吐量，也就是单位时间内服务的顾客总数，不过它也会轻微地增加延迟，也就是服务一个用户所需要的时间。（例如一个只需要喷水的汽车，在非流水线的系统，它喷完水就可以走了。而在流水线化的系统，不管你是什么需求，都要走完整个流程的时间）

（我们之前的设计是一条指令执行完，下条指令才能进入 CPU，（所不同的是时钟周期的粒度）。流水线化是允许多条指令在 CPU 中，每条指令在 CPU 中的时间是一样的，哪怕你一个周期就执行完了，你也得等剩下的阶段结束，使后面的指令被延迟了。

虽然流水线化，所有指令在 CPU 中待的时间都一样（且都按最耗时指令算的），但它们的时间是重叠的。假设一条指令在 CPU 中待 6ms，那么 12ms

能处理 7 条指令，而非流水线，虽然一条指令最多执行 6ms，但它们的时间是相加的，12ms 可能只执行 3 条。12=6+2+4）

【例如】

一个简单地非流水化的硬件系统：

它是由一些执行计算的逻辑以及一个保存计算结果的寄存器组成的。时钟信号控制在每个特定的时间间隔加载寄存器。

（CD 播放器中的译码器就是这样的一个系统。输入信号是从 CD 表面读出的位，逻辑电路对这些位进行译码，产生音频信号。图中的计算块是用组合逻辑来实现的，意味着信号会穿过一系列逻辑门，在一定时间的延迟之后，输出就成为了输入的某个函数）

在这个例子中，我们假设组合逻辑需要 300ps，而加载寄存器需要 20ps。这个实现中，在开始下一条指令之前必须完成前一个。执行一条指令需要 320ps，即每秒钟系统吞吐量 3.12GIPS。

流水化的硬件系统

假设将系统执行的计算分成三个阶段（A、B 和 C），每个阶段需要 100ps，如图所示、然后在各个阶段之间放上流水线寄存器，这样每条指令都会按照三步经过这个系统，从头到尾需要三个时钟周期。

（流水线寄存器的作用：作为电路不同部分中的组合逻辑之间的屏障。保存每步组合逻辑的运算结果。这是为了分割流水而插入的寄存器。）

流水线，在稳定状态下，三个阶段应该都是活动的，每个时钟周期，一条指令离开系统，一条新的进入。

这样，我们一个阶段的时间，相当于运行了一条指令，在这个系统中，我们将时钟周期设为 100+20=120ps，得到的吞吐量大约为 8.33GIPS。（这是在）

因为处理一条指令需要 3 个时钟周期，所以这条流水线的延迟就是 3*120=360ps。非流水运行一条完整指令需要 320ps。

（从宏观整体上看，一个时钟周期运行了一条指令（这条指令是由多条指令的各阶段拼合的），而从单条指令的执行看，需要 3 个时钟周期执行一条完整指令。）

我们将系统吞吐量提高到原来的 8.33/3.12=2.67 倍，代价是增加一些硬件（流水线寄存器），以及延迟的少量增加（360/320=1.12）。延迟变大是由于增加的流水线寄存器的时间开销。

时钟周期的时间就是流水线分割的一个阶段的时间，这样，从宏观上看，是一个时钟周期执行一条指令。

【注意】

如果时钟运行得太快，就会有灾难性的后果。值可能会来不及通过组合逻辑，并且当时钟上升时，寄存器的输入还不是合法的值。（即时钟周期比流水线一个阶段的时间短）

而我们减缓时钟不会影响流水线的行为。信号传播到流水线寄存器的输入，但是直到时钟上升时才会改变寄存器的状态。（即时钟周期比流水线一个阶段的时间长）

故，我们通过改变倍频器的值来提高时钟频率的超频手段，其提高是有限的。

流水线的局限性

1、不一致的划分

之前的是一个理想的流水线化的系统，每个阶段需要的时间都相同。而实际系统通过各阶段的延迟一般是不同的。且运行时钟的速率是由最慢阶段的延迟限制的。（即系统吞吐量受最慢阶段的速度所限制）

2、流水线过深，收益反而下降

例如，我们把计算分成 6 个阶段，每个阶段需要 50ps。在每对阶段之间插入流水线寄存器就得到了一个六阶段流水线。

这个系统的最小时钟周期为 50+20=70ps，吞吐量为 14.29GIPS。性能比 3 阶段流水提高了 14.29/8.33=1.71 倍。由于通过流水线寄存器的延迟，吞吐量并没有加倍。这个延迟成了流水线吞吐量的一个制约因素。

为了提高时钟频率，现代处理器采用了很深的（15 或更多的阶段）流水线。

分支预测

流水线化设计的目的就是每个时钟周期都发射一条新指令，要做到这一点，我们必须在取出当前指令之后，马上确定下一条指令的位置。

但如果取出的指令是条件分支指令，要到几个周期后，也就是指令通过执行阶段之后，我们才能知道是否要选择分支。类似的，如果取出的指令是 ret，要到指令通过访存阶段，才能确定返回地址。

对条件转移来说，我们既可以预测选择了分支，那么新 PC 值应为 valC，也可以预测没有选择分支，那么新 PC 值应为 valP。

对 ret 指令，可能的返回值几乎是无限的，因为返回地址位于栈顶的字，其内容可以是任意的。在设计中，我们不会试图对返回地址做任何预测。只是简单地暂停处理新指令，直到 ret 指令通过写回阶段。

无论哪种情况，我们都必须以某种方式来处理预测错误的情况，因为此时已经取出并部分执行了错误的指令。

（流水线惩罚待写）

流水线冒险

使用流水线技术，当相邻指令间存在相关时会导致出现问题。

这些相关有：

1、数据相关：下一条指令会用到这一条指令计算出的结果

2、控制相关：一条指令要确定下一条指令的位置，例如在执行跳转、调用或返回指令时。

这些相关可能会导致流水线产生计算错误，称为冒险。

用暂停来避免数据冒险

暂停(stalling)是避免冒险的一种常用技术。让一条指令停顿在译码阶段，直到产生它的源操作数的指令通过了写回阶段，这样我们的处理器就能避免数据冒险。

暂停技术就是让一组指令阻塞在它们所处的阶段，而允许其他指令继续通过流水线。

【例】

```
irmovl  $10, %edx
irmovl  $3, %eax
addl   %edx, %eax
halt
```

当对 addl 指令译码之后，暂停控制逻辑发现了对两个源寄存器的数据冒险。（其发现前面的执行、访存或写回阶段中至少有一条指令会更新寄存器%edx 或%eax 我们 addl 下一阶段就要取%eax 和%edx 的值，但却不能保证其是更新过的值）

暂停控制逻辑就在执行阶段中插入一个气泡，并在下个周期重复对 addl 的译码。

它再次发现对两个源寄存器的冒险，就在执行阶段中插入一个气泡，并在下个周期重复对 addl 的译码。

实际上，机器是动态地插入 3 条 nop 指令。（插到执行阶段，而不是从取指开始）

```
irmovl  $10, %edx
irmovl  $3, %eax
bubble
bubble
bubble
addl   %edx, %eax
halt
```

（这个过程就像，排队的时候前面的人前进了一步，但这时有另一个人插在了你前面的空缺中，你的位置保持不动，但前面的人都前进了一步。不断的有空缺，但不断地有人插入，你就一直在原地不动）

当确定前面的指令已经更新过了我们要的两个寄存器的值，则 addl 开始前行。

但是这样的解决方案得到的性能并不好，一条指令更新一个寄存器，紧跟其后的指令就使用被更新的寄存器，像这样的情况不胜枚举。这会导致流水线暂停长达三个周期，严重降低了整体的吞吐量。

用转发来避免数据冒险

在译码阶段从寄存器文件中读入源操作数，但是对这些源寄存器的写有可能要在写回阶段才能进行。与其暂停直到写完成，不如简单地将要写的值传到流水线寄存器 E 作为源操作数。

（即，我们不必等到 irmovl $10, %edx 和 irmovl $3, %eax 完成对寄存器的写更新之后再继续 addl，而是在 addl 译码阶段发现需要%edx、%eax 值，译码逻辑不从寄存器文件中去读，而是用前面阶段未写入寄存器的值。）

这种将结果直接从一个流水线阶段传到较早阶段的技术称为数据转发。

在周期 4 中，译码阶段逻辑发现有在访存阶段中对寄存器%edx 未进行的写，还发现在执行阶段中正在计算寄存器%eax 的新值。它用这些值，而不是从寄存器文件中读出的值，作为 valA 和 valB 的值。

加载/使用数据冒险

有一类数据冒险不能单纯用转发来解决，因为存储器读(访存阶段)在流水线发生的比较晚。

例：
```
mrmovl  0(%edx), %eax
addl  %ebx, %eax
halt
```

指令 mrmovl 读取存储器 0(%edx)处的值，发生在访存阶段，而此时指令 addl 已经在执行阶段了！其已经读取了%eax 的值了。即由于 mrmovl 指令获取的操作数值比较晚，来不及发送给后面需要用的指令了。

我们可以将暂停和转发结合起来，避免加载/使用数据冒险。（既然是来不及发送给后面的指令，那就让后面的指令暂停几个周期，再发送）

当 mrmovl 指令通过执行阶段时，流水线控制逻辑发现译码阶段中的指令(addl)需要从存储器中读出的结果。它会将译码阶段中的 addl 指令暂停

一个周期，导致执行阶段中插入一个气泡。 mrmovl 指令从存储器中读出的值可以从访存阶段转发到译码阶段中的 addl 指令。

这种用暂停来处理加载/使用冒险的方法称为加载互锁。加载互锁和转发技术结合起来足以处理所有可能类型的数据冒险。

异常处理

异常可以由程序执行从内部产生，也可以由某个外部信号从外部产生。

简单的三种内部异常：

1、halt 指令

2、非法指令

3、访问非法地址

（还有一些外部异常：网口收到新包、用户点击鼠标等）

在简化的 ISA 模型中，当处理器遇到异常时，会停止，设置适当的状态码，且应该是到异常指令之前的所有指令都已经完成，而其后的指令都不应该对程序员可见的状态产生任何影响。

在一个更完整的设计中，处理器会继续调用异常处理程序，这是操作系统的一部分。

★一般地，通过在流水线结构中加入异常处理逻辑，我们会在每个流水线寄存器中包括一个状态码 Stat。如果一条指令在其处理器中于某个阶段产生了一个异常，这个状态字段就被设置成指示异常的种类。异常状态和该指令的其他信息一起沿着流水线传播，直到它到达写回阶段。在此，流水线控制逻辑发现了异常，并停止执行。

异常事件不会对流水线中的指令流有任何影响，除了会禁止流水线中后面的指令更新程序员的可见状态（条件码寄存器和存储器），直到异常指令到达最后的流水线阶段。

因为指令到达写回阶段的顺序与它们在非流水化的处理器中执行的顺序相同，所以我们可以保证第一条遇到异常的指令会第一个到达写回阶段，此时程序执行会停止，流水线寄存器(W 写回)中的状态码会被记录为程序状态。

其他问题

多周期指令

我们之前设计的处理器指令集中的所有指令都包括一些简单的操作，例如数字加法。这些操作可以在执行阶段中一个周期内处理完。在一个更完整

的指令集中，还有整数乘法除法、以及浮点运算。在我们之前设计的流水化处理器中，浮点加法需要3、4个周期，整数除法需要32个周期。

实现多周期指令的一种简单方法就是简单地扩展执行阶段逻辑的功能，添加一些整数和浮点算数运算单元。一条指令在执行阶段中逗留它所需要的多个时钟周期，会导致取指和译码阶段暂停。这种方法实现起来很简单，但是得到的性能并不是太好。

通过采用独立于主流水线的特殊硬件功能单元来处理较为复杂的操作，可以得到更好的性能。通常，有一个功能单元来执行整数乘法和除法，还有一个来执行浮点操作(协处理器)。

当一条指令进入到译码阶段时，它可以被发射到特殊单元。在这个特殊单元执行该操作时，流水线会继续处理其他指令。通常，浮点单元本身也是流水线化的，因此多条指令可以在主流水线和各个单元中并行执行。

不同单元的操作必须同步，以避免出错。

如果在不同单元执行的各个指令之间有数据相关，控制逻辑可能需要暂停系统的某个部分，直到由系统其他部分处理的操作的结果完成。

使用各种形式的转发，将结果从系统的一部分传递到其他部分，这和前面的PIPE流水线各个阶段之间的转发一样。虽然与PIPE相比，整个设计变得更复杂，但还是可以使用暂停、转发、以及流水线控制等同样的技术，使整体行为与顺序的ISA模型相匹配。

与存储系统的接口

在我们之前的流水化CPU中，我们假设取指单元和数据存储器都可以在一个时钟周期内读或是写存储器中任意的位置。

但是，实际情况是，我们以存储器位置的虚拟地址来引用数据，这就要求在执行实际的读写操作之前，要将虚拟地址翻译成物理地址。显然，要在一个时钟周期内完成所有这些处理是不现实的。更糟糕的是，要访问的存储器的的值可能位于磁盘上，这会需要上百万个时钟周期才能把数据读入到处理器存储器中。

存储系统：

CPU的存储系统是由多种硬件存储器和管理虚拟存储器的操作系统软件共同组成的。

存储系统被组织成一个层次结构，较快但是较小的存储器保持着存储器的一个子集，而较慢但是较大的存储器作为它的后备。

最靠近处理器的一层是高速缓存(cache)存储器，它提供对最常使用的存储器位置的快速访问。一般有2个一层cache——一个用于读指令，一个用于读写数据。

还有另一种类型的高速缓存存储器，称为 TLB(Translation Look-aside Buffer 翻译后备缓冲器)，它提供了从虚拟地址到物理地址的快速翻译。将 TLB 和 cache 结合起来使用，在大多数时候，确实可能在一个时钟周期内读指令并读或是写数据。

缓存不命中：有些引用的位置不在高速缓存中，即出现高速缓存不命中。在最好的情况下，可以冲=从较高层的 cache 或处理器的主存中找到不命中的数据，这需要 3--20 个时钟周期。同时，流水线会简单地暂停，将指令保持在取值或访存阶段，直到高速缓存能够执行读或写操作。

缺页异常：当被引用的存储器位置实际上是在磁盘存储器上的，硬件会产生一个缺页异常信号。同其他异常一样，这个异常会导致处理器调用操作系统的异常处理程序代码。然后这段代码会发起一个从磁盘到主存的传送操作。

让硬件调用操作系统例程，然后操作系统例程又会将控制返回给硬件，这就使得硬件和系统软件在处理缺页时能协同工作。从处理器的角度来看，将用暂停来处理短时间的高速缓存不命中和用异常处理来处理长时间的缺页结合起来，能够顾及到存储器访问时由于存储器层次结构引起的所有不可预测性

## 第三章

选择要点综述与 TensorFlow 安装指导

CPU 对比参考表：

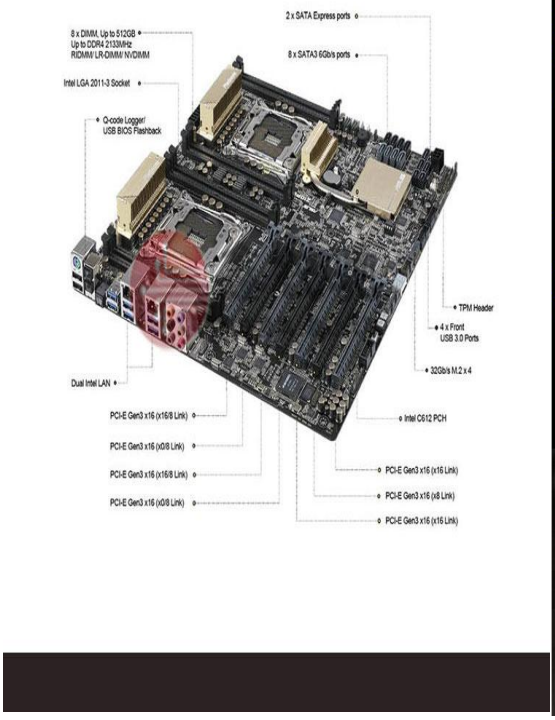
| 处理器型号 | i7-5960x | E5-2687wv4 | I7-3970x | E5-2698v4 |
|---|---|---|---|---|
| 内核数 | 8 | 12 | 6 | 20 |
| 线程数 | 16 | 24 | 12 | 40 |
| 最大睿频频率 | 3.5 | 3.5 | 4.0 | 3.6 |
| 基本频率 | 3.0 | 3.0 | 3.5 | 2.2 |
| 缓存 | 20MB | 30MB | 15MB | 50M |
| TDP | 140W | 160W | 150w | 135w |
| 最大内存 | 64G | 1.54TB | 64.45GB | 1.54TB |
| 最大内存通道 | 4 | 4 | 4 | 4 |
| 内存类型 | DDR4 1333/1600/2133 | DDR4 1600/1866/2133/2400 | DDR4 1066/1333/1600 | DDR41600/1866/2133/2400 |
| ECC 内存支持 | 否 | 是 | 否 | 是 |

固态硬盘对比

| 产品名称 | 顺序性读写 MB/秒 | 随机 4KB 读写 IOPS | 接口 |
|---|---|---|---|
| Inter 固态盘 pro6000p （专业） | 1800/560 | 155000/128000 | PCIe*3x4 |
| Inter 固态盘 600p | 1800/560 | 155000/128000 | PCIe*Gen3x4 |
| 三星 PM961 NVME | 1356/463.6 | 141405/114347 | PCIE 3.0 X4 NVME |
| 浦科特 M7VC | 560/530 | 98000/84000 | SATA 3.0 |

内存：海盗船或金士顿就 ok


主板：（服务器主板矩阵式/工作站/个人类型）这部分需要注意的就是问题：
CPU 针脚数目和支持的 CPU
GPU 的接口类型
电源型号
最大支持内存
PCI 接口数目
主板上的核心部件是南桥和北桥控制芯片匹配就 ok





华硕 Z10PE-D8 WS 双路

电源的电压一定在配置时要小心，GUP 比较耗能，所以一定要匹配，不然会降低硬件性能使用寿命。
冷却和散热（一般机房有空调所以配个好风扇就很好）
深度学习在 GPU 方面首选 GTX690，层次高一点用 GTX  Titan X（用来做卷积比较有优势）。
在学生做论文基础实验时用 GTX980 可以，但是大型卷积神经网络有点局限 50GB 训练集就吃不消啦！
（需要其他显卡一块价格相对低的辅助完成低价格高性能存储(一定要考虑你是否使用 cnDNN)像 GTX580 这样的卡不行，我个人参考当时资源用 GTX Titan 选择条件一定要支持 cnDNN 的 GPU，如果完全研究卷积方面那就 GTX580 性能提升 25%，矩阵显卡观念是对的。
CPU 在你用 GPU 上跑时需要处理：
①读取代码中的变量或读取代码中变量
②完成函数调用指令
③产生小批量数据向 core 上传输数据块

1）看看是不是自己用同一个库，如果用三个不同的库，测试实验结果显示 cpu 一个线程显示 100%；另外一个在浮动。这就发现问题，其实很多库用的同一个线程，所以 CPU 就看时钟频率，线程数和缓存。
2）运行多 GPU 或者同时跑多个程序的 MPI 框架，这时候才考虑 CPU 的核数，每个 GPU 跑各自的线程效果那就很棒。提升 35%以上左右
3）通常能让一个 GPU 跑两个线程对多数深度学习效果更好，原因在于一个库在单核上运行在数据块从代码中读取写入 core 时调用同步异布函数会用到第二个线程，C++语言本身就有个析构函数和同步异构特定。Inter 的 CPU 每一个核上运行多个线程特点，所以按照比例每个 GPU 对于一个 CPU 核就 ok
4）特殊情况特斯拉 K80 这个卡有两块芯片，很多说 K80 不如两个 1080，K80 两块 GK110，
5）GTX680 有一颗 GK110，1080 一块 GK04，所以你用双路 CPU 和 K80 跑肯定跑死 1080。

GPU 最初就是用于快速生成高分辨率计算机图像，由于它的计算效率使得其非常适合用于深度学习算法。原先需要用好几周才能分析出来的结果，现在几天时间就能完成

下面我具体折腾过的机器。按照时间顺序介绍：

| GPU | CPU | 内存 | 尺寸 | 单价 ($) |
|---|---|---|---|---|
| 2 x GTX 980 | 2 x E5-2680 v2 | 128G | 2U | 12k |
| 2 x GTX 970 | i5-3437 | 32G | mid tower | $1.3k |
| 4 x GTX 980 | E5-1650 | 64G | super tower | 6.5k |
| 4 x GTX 750 TI | i7-4790 | 16G | full tower | 1k |
| 4 x K40 | 2 x E5-2670 | 64G | 4U | free |
| 4 x Titan X | 2 x E5-2630 v3 | 128G | 1U | 11k |

# GTX 1080+Ubuntu16.04+CUDA8.0+cuDNN5.0+TensorFlow 安装指导

This is the third article in the series “Dive Into TensorFlow“, here is an index of all the articles in the series that have been published to date:

Part I: Getting Started with TensorFlow
Part II: Basic Concepts
Part III: Ubuntu16.04+GTX 1080+CUDA8.0+cuDNN5.0+TensorFlow (this article)

## Deep Learning PC with GTX 1080

Recently, I got a deep learning pc with gtx 1080, following is the details of the pc:

*CPU Intel Core i7-6800K 15M Broadwell-E 6-Core 3.4 GHz LGA 2011-v3 140W*
*Motherboard Asus X99-A/USB 3.1 ATX LGA2011-3*
*Video Card GIGABYTE GeForce GTX 1080 G1 Gaming GV-N1080G1 GAMING-8GD Video Card*
*Storage SAMSUNG SM951 256GB SSD + WD Blue 4TB Desktop Hard Disk Drive*
*Memory Kingston HyperX Fury 64GB (4 x 16G) DDR4 2400 RAM HX424C15FBK4/64*
*……*

The system is Ubuntu16.04 64-bit, after the system ready, first need to install Nvidia GTX 1080 driver 367.27:

```
sudo add-apt-repository ppa:graphics-drivers/ppa
```

Meet the warning:

> *Fresh drivers from upstream, currently shipping Nvidia.*
>
> *## Current Status*
>
> *We currently recommend: `nvidia-361`, Nvidia’s current long lived branch.*
> *For GeForce 8 and 9 series GPUs use `nvidia-340`*
> *For GeForce 6 and 7 series GPUs use `nvidia-304`*
>
> *## What we’re working on right now:*
>
> *- Normal driver updates*
> *- Investigating how to bring this goodness to distro on a cadence.*
>
> *## WARNINGS:*

*This PPA is currently in testing, you should be experienced with packaging before you dive in here. Give us a few days to sort out the kinks.*

Ignore it, and continue to install the GTX1080 driver:

```
sudo apt-get update
sudo apt-get install nvidia-367
sudo apt-get install mesa-common-dev
sudo apt-get install freeglut3-dev
```

Then restart the deep learning pc and load the NVIDIA drivers.

Download and install CUDA

We choose the new CUDA8, which support Nvidia GTX 1080:

*New in CUDA 8*

*Pascal Architecture Support*
*Out of box performance improvements on Tesla P100, supports GeForce GTX 1080*
*Simplify programming using Unified memory on Pascal including support for large datasets, concurrent data access and atomics**
*Optimize Unified Memory performance using new data migration APIs**
*Faster Deep Learning using optimized cuBLAS routines for native FP16 computation*
*Developer Tools*
*Quickly identify latent system-level bottlenecks using the new critical path analysis feature*
*Improve productivity with up to 2x faster NVCC compilation speed*
*Tune OpenACC applications and overall host code using new profiling extensions*
*Libraries*
*Accelerate graph analytics algorithms with nvGRAPH*
*New cuBLAS matrix multiply optimizations for matrices with sizes smaller than 512 and for batched operation*

Here is the CUDA 8 download link: https://developer.nvidia.com/cuda-release-candidate-download, which need you register or log into the Nvidia developer program to download. We choose the Ubuntu16.04 runfile install method:

Select Target platfrom

Operating System : Linux

Architecture: X 86_64

Distribution: Ubuntu

Version : 16.04

Installer Type: runfile[local]

After download the 1.4G cuda_8.0.27_linux.run file, you can run:

sudo sh cuda_8.0.27_linux.run --tmpdir=/opt/temp/

Here you can add the --tmpdir parameter if you meet the follow error:

> *Not enough space on parition mounted at /.*
> *Need 5091561472 bytes.*
>
> *Disk space check has failed. Installation cannot continue.*

After run the cuda_8.0.27_linux.run, you will meet some yes or no question, it is very important that you should answer “n” for this quesion:

> *Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 361.62?*

Very important, If you answer yes, the GTX 1080 367 driver will be overwritten.

> *Logging to /opt/temp//cuda_install_6583.log*
> *Using more to view the EULA.*
> *End User License Agreement*
> *——————————-*
>
> *Preface*
> *————*
>
> *The following contains specific license terms and conditions for four separate NVIDIA products. By accepting this agreement, you agree to comply with all the terms and conditions applicable to the specific product(s) included herein.*
>
> *Do you accept the previously read EULA?*
> *accept/decline/quit: accept*
>
> *Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 361.62?*
> *(y)es/(n)o/(q)uit: n*

```
Install the CUDA 8.0 Toolkit?
(y)es/(n)o/(q)uit: y

Enter Toolkit Location
[ default is /usr/local/cuda-8.0 ]:

Do you want to install a symbolic link at /usr/local/cuda?
(y)es/(n)o/(q)uit: y

Install the CUDA 8.0 Samples?
(y)es/(n)o/(q)uit: y

Enter CUDA Samples Location
[ default is /home/textminer ]:

Installing the CUDA Toolkit in /usr/local/cuda-8.0 ...
Installing the CUDA Samples in /home/textminer ...
Copying samples to /home/textminer/NVIDIA_CUDA-8.0_Samples
now...
Finished copying samples.

===========
= Summary =
===========

Driver: Not Selected
Toolkit: Installed in /usr/local/cuda-8.0
Samples: Installed in /home/textminer

Please make sure that
 - PATH includes /usr/local/cuda-8.0/bin
 - LD_LIBRARY_PATH includes /usr/local/cuda-8.0/lib64, or,
add /usr/local/cuda-8.0/lib64 to /etc/ld.so.conf and run
ldconfig as root

To uninstall the CUDA Toolkit, run the uninstall script in
/usr/local/cuda-8.0/bin

Please see CUDA_Installation_Guide_Linux.pdf in
/usr/local/cuda-8.0/doc/pdf for detailed information on
setting up CUDA.

***WARNING: Incomplete installation! This installation did
not install the CUDA Driver. A driver of version at least
361.00 is required for CUDA 8.0 functionality to work.
To install the driver using this installer, run the following
command, replacing with the name of this run file:
sudo .run -silent -driver
```

*Logfile is /opt/temp//cuda_install_6583.log*

Then set up the development environment by modifying the PATH and LD_LIBRARY_PATH variables, also add them to the end of .bashrc file:

```
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}
export
LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64${LD_LIBRARY_PATH:+:${LD_LI
BRARY_PATH}}
```

Finally we can test CUDA 8 in our Ubunt16.04 with GeForce GTX 1080:

nvidia-smi

textminer@textminer:~/NVIDIA_CUDA-8.0_Sample$-smi

| | | |
|---|---|---|
| NVIDIA-SM I367.27 | Driver Version 367.27 | |
| GPU Name Persistence-M<br>Fan Temp Perf Pwr: usage/cap | Bus-Id Disp.A<br>Memory-usage | Volatile Uncorr.ECC<br>GPU-util computer M. |
| 0 geForce GTX 1080 OFF<br>0% 41c p8 9w /200w | 0000:01:00.0 on<br>249MiB/8112MiB | N/A<br>0% Default |
| Processes:<br>GPU PID Type prpcess name | | GPU memory<br>Usage |
| 0 4169 G /usr/lib/xorg/xorg<br>0 4615 G compiz | | 156MiB<br>90MiB |
| | | |

```
cd 1_Utilities/deviceQuery
make
```

*"/usr/local/cuda-8.0" /bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode arch=compute_20,code=sm_20 -gencode*

*arch=compute_30,code=sm_30 -gencode*
*arch=compute_35,code=sm_35 -gencode*
*arch=compute_37,code=sm_37 -gencode*
*arch=compute_50,code=sm_50 -gencode*
*arch=compute_52,code=sm_52 -gencode*
*arch=compute_60,code=sm_60 -gencode*
*arch=compute_60,code=compute_60 -o deviceQuery.o -c*
*deviceQuery.cpp*
*"/usr/local/cuda-8.0" /bin/nvcc -ccbin g++ -m64 -gencode*
*arch=compute_20,code=sm_20 -gencode*
*arch=compute_30,code=sm_30 -gencode*
*arch=compute_35,code=sm_35 -gencode*
*arch=compute_37,code=sm_37 -gencode*
*arch=compute_50,code=sm_50 -gencode*
*arch=compute_52,code=sm_52 -gencode*
*arch=compute_60,code=sm_60 -gencode*
*arch=compute_60,code=compute_60 -o deviceQuery*
*deviceQuery.o*
*mkdir -p ../../bin/x86_64/linux/release*
*cp deviceQuery ../../bin/x86_64/linux/release*

Run ./deviceQuery:

*./deviceQuery Starting…*

*CUDA Device Query (Runtime API) version (CUDART static linking)*

*Detected 1 CUDA Capable device(s)*

*Device 0: "GeForce GTX 1080"*
*CUDA Driver Version / Runtime Version 8.0 / 8.0*
*CUDA Capability Major/Minor version number: 6.1*
*Total amount of global memory: 8112 MBytes (8506179584 bytes)*
*(20) Multiprocessors, (128) CUDA Cores/MP: 2560 CUDA Cores*
*GPU Max Clock rate: 1835 MHz (1.84 GHz)*
*Memory Clock rate: 5005 Mhz*
*Memory Bus Width: 256-bit*
*L2 Cache Size: 2097152 bytes*
*Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)*
*Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers*
*Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers*
*Total amount of constant memory: 65536 bytes*

*Total amount of shared memory per block: 49152 bytes*
*Total number of registers available per block: 65536*
*Warp size: 32*
*Maximum number of threads per multiprocessor: 2048*
*Maximum number of threads per block: 1024*
*Max dimension size of a thread block (x,y,z): (1024, 1024, 64)*
*Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)*
*Maximum memory pitch: 2147483647 bytes*
*Texture alignment: 512 bytes*
*Concurrent copy and kernel execution: Yes with 2 copy engine(s)*
*Run time limit on kernels: Yes*
*Integrated GPU sharing Host Memory: No*
*Support host page-locked memory mapping: Yes*
*Alignment requirement for Surfaces: Yes*
*Device has ECC support: Disabled*
*Device supports Unified Addressing (UVA): Yes*
*Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0*
*Compute Mode:*
*< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >*

*deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 1, Device0 = GeForce GTX 1080*
*Result = PASS*

Test nobody sample:

```
cd ../../5_Simulations/nbody/
make
```

Run:

```
./nbody -benchmark -numbodies=256000 -device=0
```

*> Windowed mode*
*> Simulation data stored in video memory*
*> Single precision floating point simulation*
*> 1 Devices used for simulation*
*gpuDeviceInit() CUDA Device [0]: “GeForce GTX 1080*
*> Compute 6.1 CUDA device: [GeForce GTX 1080]*
*number of bodies = 256000*
*256000 bodies, total time for 10 iterations: 2291.469 ms*

*= 286.000 billion interactions per second*
*= 5719.998 single-precision GFLOP/s at 20 flops per interaction*

Download and install cuDNN

About cuDNN:

> *The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. cuDNN is part of the NVIDIA Deep Learning SDK.*
>
> *Deep learning researchers and framework developers worldwide rely on cuDNN for high-performance GPU acceleration. It allows them to focus on training neural networks and developing software applications rather than spending time on low-level GPU performance tuning. cuDNN accelerates widely used deep learning frameworks, including Caffe, TensorFlow, Theano, Torch, and CNTK. See supported frameworks for more details.*

We choose cuDNN v5, which support CUDA8.0, the official download link is:https://developer.nvidia.com/rdp/cudnn-download

Install cuDNN is very simple:

tar -zxvf cudnn-8.0-linux-x64-v5.0-ga.tgz

> *cuda/include/cudnn.h*
> *cuda/lib64/libcudnn.so*
> *cuda/lib64/libcudnn.so.5*
> *cuda/lib64/libcudnn.so.5.0.5*
> *cuda/lib64/libcudnn_static.a*

```
sudo cp cuda/include/cudnn.h /usr/local/cuda/include/
sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64/
sudo chmod a+r /usr/local/cuda/include/cudnn.h
sudo chmod a+r /usr/local/cuda/lib64/libcudnn*
```

Install TensorFlow Python Dependency

The Python version is Python2.7:

```
sudo apt-get install python-pip
sudo apt-get install python-numpy swig python-dev python-wheel
```

Install Google Build Tool Bazel

Bazel is a build tool that builds code quickly and reliably. It is used to build the majority of Google's software, and thus it has been designed to handle build problems present in Google's development environment, including:

> *A massive, shared code repository, in which all software is built from source. Bazel has been built for speed, using both caching and parallelism to achieve this. Bazel is critical to Google's ability to continue to scale its software development practices as the company grows.*
>
> *An emphasis on automated testing and releases. Bazel has been built for correctness and reproducibility, meaning that a build performed on a continuous build machine or in a release pipeline will generate bitwise-identical outputs to those generated on a developer's machine.*
>
> *Language and platform diversity. Bazel's architecture is general enough to support many different programming languages within Google, and can be used to build both client and server software targeting multiple architectures from the same underlying codebase.*

Get the Bazel 0.3.0 install script from Bazel github:

```
wget
https://github.com/bazelbuild/bazel/releases/download/0.3.0/bazel-0.
3.0-installer-linux-x86_64.sh
```

Then run:

```
chmod +x bazel-0.3.0-installer-linux-x86_64.sh
./bazel-0.3.0-installer-linux-x86_64.sh --user
```

If you meet the error:

> *Java not found, please install the corresponding package*
> *See http://bazel.io/docs/install.html for more information*
> *on*

Install the Java JDK in Ubuntu 16.04 by apt-get:

```
sudo apt-get update
sudo apt-get install default-jre
sudo apt-get install default-jdk
```

Then run the following command again:

```
./bazel-0.3.0-installer-linux-x86_64.sh --user
```

*Bazel installer*
*———————*

*# Release 0.3.0 (2016-06-10)*

*Baseline: a9301fa*

*Cherry picks:*
*+ ff30a73: Turn --legacy_external_runfiles back on by default*
*+ aeee3b8: Fix delete[] warning on fsevents.cc*

*Incompatible changes:*

*- The --cwarn command line option is not supported anymore. Use*
*--copt instead.*

*New features:*

*- On OSX, --watchfs now uses FsEvents to be notified of changes*
*from the filesystem (previously, this flag had no effect on OS X).*
*- add support for the '-=', '*=', '/=', and '%=' operators to*
*skylark. Notably, we do not support '|=' because the semantics*
*of skylark sets are sufficiently different from python sets.*

*Important changes:*

*- Use singular form when appropriate in blaze's test result summary*
*message.*
*- Added supported for Android NDK revision 11*
*- --objc_generate_debug_symbols is now deprecated.*
*- swift_library now generates an Objective-C header for its @objc*
*interfaces.*

*- new_objc_provider can now set the USES_SWIFT flag.*
*- objc_framework now supports dynamic frameworks.*
*- Symlinks in zip files are now unzipped correctly by http_archive, download_and_extract, etc.*
*- swift_library is now able to import framework rules such as objc_framework.*
*- Adds "jre_deps" attribute to j2objc_library.*
*- Release apple_binary rule, for creating multi-architecture ("fat") objc/cc binaries and libraries, targeting ios platforms.*
*- Aspects documentation added.*
*- The -ues_isystem_for_includes command line option is not supported anymore.*
*- global function 'provider' is removed from .bzl files. Providers can only be accessed through fields in a 'target' object.*

*## Build informations*
*- [Build log](http://ci.bazel.io/job/Bazel/JAVA_VERSION=1.8,PLATFORM_NAME=linux-x86_64/595/)*
*- [Commit](https://github.com/bazelbuild/bazel/commit/e671d29)*
*Uncompressing……Extracting Bazel installation…*
*.*

*Bazel is now installed!*

*Make sure you have "/home/textminer/bin" in your path. You can also activate bash completion by adding the following line to your ~/.bashrc:*
*source /home/textminer/.bazel/bin/bazel-complete.bash*

*See http://bazel.io/docs/getting-started.html to start a new project!*

Then add following code in ~/.bashrc:

```
source /home/textminer/.bazel/bin/bazel-complete.bash
export PATH=$PATH:/home/textminer/.bazel/bin
```

and execute:

source ~/.bashrc

Now, Bazel installed successfully.

Build and install TensorFlow GPU Version by Source Code

Get the newest TensorFlow code from tensorflow github repository:

git clone https://github.com/tensorflow/tensorflow

then:

```
cd tensorflow
./configure
```

If you meet the error:

> *ERROR: It appears that the development version of libcurl is not available. Please install the libcurl3-dev package.*

You should install libcurl3-dev by apt-get:

sudo apt-get install libcurl3 libcurl3-dev

Then run configure again:

./configure

> *Please specify the location of python. [Default is /usr/bin/python]:*
> *Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] y*
> *Google Cloud Platform support will be enabled for TensorFlow*
> *Do you wish to build TensorFlow with GPU support? [y/N] y*
> *GPU support will be enabled for TensorFlow*
> *Please specify which gcc nvcc should use as the host compiler. [Default is /usr/bin/gcc]:*
> *Please specify the Cuda SDK version you want to use, e.g. 7.0. [Leave empty to use system default]:*
> *Please specify the location where CUDA toolkit is installed. Refer to README.md for more details. [Default is /usr/local/cuda]:*
> *Please specify the Cudnn version you want to use. [Leave empty to use system default]:*
> *Please specify the location where cuDNN library is installed. Refer to README.md for more details. [Default is*

*/usr/local/cuda]:*
*Please specify a list of comma-separated Cuda compute capabilities you want to build with.*
*You can find the compute capability of your device at: https://developer.nvidia.com/cuda-gpus.*
*Please note that each additional compute capability significantly increases your build time and binary size.*
*[Default is: "3.5,5.2"]:*
*Setting up Cuda include*
*Setting up Cuda lib64*
*Setting up Cuda bin*
*Setting up Cuda nvvm*
*Setting up CUPTI include*
*Setting up CUPTI lib64*
*Configuration finished*

Now the TensorFlow can be build by Bazel:

```
bazel build -c opt --config=cuda
//tensorflow/cc:tutorials_example_trainer
```

If you meet the error:

*configure: error: zlib not installed*
*Target //tensorflow/cc:tutorials_example_trainer failed to build*

Install zlib1g-dev:

```
sudo apt-get install zlib1g-dev
```

Then build tensorflow by bazel again, and wait a cup of coffee time:

*.....*
*Target //tensorflow/cc:tutorials_example_trainer up-to-date:*
*bazel-bin/tensorflow/cc/tutorials_example_trainer*
*INFO: Elapsed time: 897.845s, Critical Path: 533.72s*

Excute the tensorflow tutorials sample to call the GPU GTX 1080:

*bazel-bin/tensorflow/cc/tutorials_example_trainer --use_gpu*
*I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally*
*I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcudnn.so locally*
*I tensorflow/stream_executor/dso_loader.cc:108]*

*successfully opened CUDA library libcufft.so locally*
*I tensorflow/stream_executor/dso_loader.cc:108]*
*successfully opened CUDA library libcuda.so.1 locally*
*I tensorflow/stream_executor/dso_loader.cc:108]*
*successfully opened CUDA library libcurand.so locally*
*I tensorflow/core/common_runtime/gpu/gpu_init.cc:102]*
*Found device 0 with properties:*
*name: GeForce GTX 1080*
*major: 6 minor: 1 memoryClockRate (GHz) 1.835*
*pciBusID 0000:01:00.0*
*Total memory: 7.92GiB*
*Free memory: 7.65GiB*
*I tensorflow/core/common_runtime/gpu/gpu_init.cc:126] DMA: 0*
*I tensorflow/core/common_runtime/gpu/gpu_init.cc:136] 0: Y*
*I tensorflow/core/common_runtime/gpu/gpu_device.cc:838]*
*Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX 1080, pci bus id: 0000:01:00.0)*
*I tensorflow/core/common_runtime/gpu/gpu_device.cc:838]*
*Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX 1080, pci bus id: 0000:01:00.0)*
*000003/000006 lambda = 1.841570 x = [0.669396 0.742906] y = [3.493999 -0.669396]*
*000006/000007 lambda = 1.841570 x = [0.669396 0.742906] y = [3.493999 -0.669396]*
*000009/000006 lambda = 1.841570 x = [0.669396 0.742906] y = [3.493999 -0.669396]*
*000009/000004 lambda = 1.841570 x = [0.669396 0.742906] y = [3.493999 -0.669396]*
*000000/000005 lambda = 1.841570 x = [0.669396 0.742906] y = [3.493999 -0.669396]*
*000000/000004 lambda = 1.841570 x = [0.669396 0.742906] y = [3.493999 -0.669396]*
*……*

Everything is ok. Now test Tensorflow in ipyhton:

```
import tensorflow as tf
```

*ImportError: cannot import name pywrap_tensorflow*

Continue to build the Python TensorFlow wraper by Bazel:

```
bazel build -c opt --config=cuda
//tensorflow/tools/pip_package:build_pip_package
```

bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
sudo pip install /tmp/tensorflow_pkg/tensorflow-0.9.0-py2-none-any.whl

> *Requirement already satisfied (use --upgrade to upgrade): setuptools in /usr/lib/python2.7/dist-packages (from protobuf==3.0.0b2->tensorflow==0.9.0)*
> *Installing collected packages: six, funcsigs, pbr, mock, protobuf, tensorflow*
> *Successfully installed funcsigs-1.0.2 mock-2.0.0 pbr-1.10.0 protobuf-3.0.0b2 six-1.10.0 tensorflow-0.9.0*

Now test TensorFlow in iPython again:

Now, just enjoy TensorFlow in your Ubuntu 16.04 system with GeForce GTX 1080

```
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import tensorflow as tf
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library

In [2]: import numpy as np

In [3]: x_data = np.random.rand(100).astype(np.float32)

In [4]: y_data = x_data * 0.1 + 0.3

In [5]: W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))

In [6]: b = tf.Variable(tf.zeros([1]))

In [7]: y = W * x_data + b

In [8]: loss = tf.reduce_mean(tf.square(y - y_data))

In [9]: optimizer = tf.train.GradientDescentOptimizer(0.5)

In [10]: train = optimizer.minimize(loss)

In [11]: init = tf.initialize_all_variables()

In [12]: sess = tf.Session()
I tensorflow/core/common_runtime/gpu/gpu_init.cc:102] Found device 0 with proper
name: GeForce GTX 1080
major: 6 minor: 1 memoryClockRate (GHz) 1.835
pciBusID 0000:01:00.0
Total memory: 7.92GiB
Free memory: 7.65GiB
I tensorflow/core/common_runtime/gpu/gpu_init.cc:126] DMA: 0
I tensorflow/core/common_runtime/gpu/gpu_init.cc:136] 0:   Y
I tensorflow/core/common_runtime/gpu/gpu_device.cc:838] Creating TensorFlow devi

In [13]: sess.run(init)

In [14]: for step in range(201):
   ....:     sess.run(train)
   ....:     if step % 20 == 0:
   ....:         print(step, sess.run(W), sess.run(b))
   ....:
(0, array([-0.10331395], dtype=float32), array([ 0.62236434], dtype=float32))
(20, array([ 0.03067014], dtype=float32), array([ 0.3403711], dtype=float32))
(40, array([ 0.08353967], dtype=float32), array([ 0.30958495], dtype=float32))
(60, array([ 0.09609199], dtype=float32), array([ 0.30227566], dtype=float32))
(80, array([ 0.09907217], dtype=float32), array([ 0.3005403], dtype=float32))
(100, array([ 0.09977971], dtype=float32), array([ 0.30012828], dtype=float32))
(120, array([ 0.0999477], dtype=float32), array([ 0.30003047], dtype=float32))
(140, array([ 0.0999876], dtype=float32), array([ 0.30000722], dtype=float32))
(160, array([ 0.09999706], dtype=float32), array([ 0.30000171], dtype=float32))
(180, array([ 0.09999929], dtype=float32), ar...y([ 0.30000043], dtype=float32))
(200, array([ 0.09999985], dtype=float32), ar...([ 0.30000001], dtype=float32))
```