第3章 GPU硬件架构

本章将介绍GPU硬件架构，主要包括GPU架构、kernel函数的GPU映射、GPU存储体系和GPU计算能力，帮助读者对GPU硬件有一个整体认知。

3.1 GPU架构

到目前为止，NVIDIA主要有5类不同架构的GPU产品。

(1) Tesla架构。其硬件核心有G80、G92、GT200，产品有8800GTX、9800GTX、GTX280等。

(2) Fermi架构。其硬件核心有GF100、GF104，产品有Tesla M2050、GTX480等。

(3) Kepler架构。其硬件核心有GK104、GK110，产品有Tesla K10、K20、GTX680等，其中GK104核心只是一个过渡，不支持所有的Kepler架构功能。目前市面上能买到的基本上都是Kepler架构产品。

(4) Maxwell架构。其硬件核心有GM107、GM200、GM204，产品有GTX 750Ti、Tesla M40、GTX TITAN X、GTX980等，Maxwell产品Tesla M40针对深度学习进行了专门优化，比如提供了16位数据类型。

(5) Pascal架构。目前发布的核心有GP100，产品有Tesla P100 GPU。在2016年4月的GPU技术大会(GPU technology conference，GTC)上，NVIDIA发布了Pascal架构产品Tesla P100，该产品预计于同年6月份全面上市(截至本书定稿还未上市)。根据NVIDIA公布的数据，该产品拥有153亿个16纳米制造工艺的晶体管，计算性能达到双精度5.3Teraflop和单精度10.6Teraflop，若使用NVIDIA GPU BOOST技术可达到单精度21.2Teraflop。此外P100还采用NVLink技术，令双向通信带宽高达160GB/s；配置了CoWoS HBM2堆叠内存，显存带宽可达720GB/s。

不同GPU架构的设计理念、工艺水平等均不相同，相应的内部体系结构和性能也不一致。接下来本书将展开介绍这几类GPU架构。

为了更好地认知GPU体系结构，首先，介绍几个GPU体系结构相关

术语。

流处理器(streaming processor,SP):也称为core,是GPU运算的最基本计算单元。早期(tesla架构)只能进行单精度浮点运算和整数运算,符合IEEE 754-1985标准;后来(Fermi架构开始)采用IEEE 754-2008标准,增加了对双精度浮点运算的支持,并支持完整的32位整数运算。

渲染核(shader core):SP的另一个名称,又称为CUDA core,始于Fermi架构。

双精度浮点运算单元(DP):专用于双精度浮点运算的处理单元。

特殊功能单元(special function unit,SFU):用来执行超越函数指令,比如正弦、余弦、倒数和平方根等函数。每个SFU一次执行一个线程的一条指令需要一个时钟周期,因此Tesla架构(2个SFU,半warp执行)中的一个warp需要消耗8个时钟周期。SFU并不始终占用SM中的dispatch,即当SFU处于执行状态的时候,指令分发单元可以向其他的执行单元分发相应的指令。

流处理器(streaming multiprocessors,SM):GPU架构中的基本计算单元,也是GPU性能的源泉,由SP、DP、SFU等运算单元组成。这是一个典型的阵列机,其执行方式为SIMT(单指令多线程),区别于传统的SIMD,能够保证多线程的同时执行(向量化)。

SMX:Kepler架构中的SM。

SMM:Maxwell架构中的SM。

线程处理器簇(thread processing cluster,TPC):由SM和L1 cache组成,存在于Tesla架构中,后又出现在Pascal架构。G80架构包含2个SM和16KB L1 cache;GT200架构包含3个SM和24KB L1 cache。

图形处理器簇(graph processing cluster,GPC):类似于TPC,是介于整个GPU和SM间的硬件单元,Fermi架构开始有这个称谓。TPC和GPC还是有所区别的,TPC由SM控制器、SM和纹理缓存组成,而GPC由一个光栅单元、4个SM和SM控制器组成。GPC中SM数量是可扩展的。

注意:开始笔者认为Fermi架构开始用GPC取代了TPC,甚至认为两者是同一个概念的不同叫法。但在最新的Pascal架构中,同时出现了GPC和TPC,且GPC包含TPC,故两者并不等价。

流处理器阵列(scalable streaming processor array,SPA):所有处理核心和高速缓存的总和,包含所有的SM、TPC和GPC。与存储器系统共同组成GPU架构。

存储控制器(memory controller,MMC):顾名思义,控制存储访问的单元,合并访存。每个存储控制器可以支持一定位宽(比如64位)的数据合并访存。

光栅操作单元(raster operation processors,POP)。

存取单元(load/store unites,LD/ST)。

3.1.1 Tesla架构

Tesla架构的计算能力为1.X,不同计算能力版本间有些功能上的区别,比如1.0不支持原子操作,也不支持双精度运算;1.1添加了全局存储器的32位字原子操作函数;1.2开始支持共享存储原子操作功能,并添加了对全局存储64位字原子操作的支持;1.3

开始支持双精度运算。由于 Tesla 架构中 SP 不支持双精度运算，双精度运算是通过增加 DP 实现的。

Tesla 架构的 SM 如图 3.1 所示，由 8 个 SP 和 2 个 SFU 组成，在 GT200 a/b 中增加了 DP 支持双精度运算，同时，SM 中还包含寄存器、共享存储、常量存储等存储单元。其中，G80 核心拥有 8192 个 32 位寄存器单元，GT200 扩充了一倍，共 16 384 个。

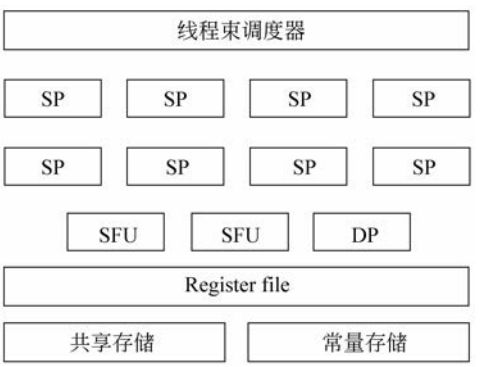

图 3.1　SM 1.X

2～3 个 SM 配合 L1 Cache 构成 TPC，Tesla 架构主要核心型号有 G80 和 GT200，G80 核心中 2 个 SM 和 16KB L1 Cache 组成 1 个 TPC，GT200 包含 3 个 SM 和 24KB L1 Cache，另外，每个 TPC 均由 1 个 SM 控制器进行统一控制（见图 3.2）。

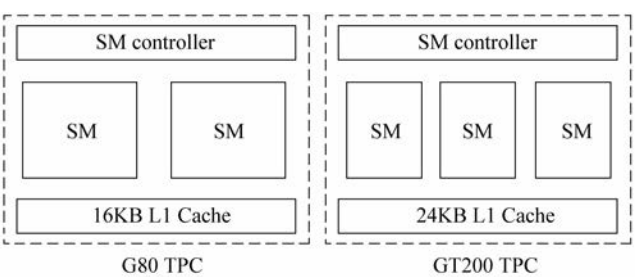

图 3.2　TPC

Tesla 架构从整体上主要由两个部分组成，即 SPA 和存储系统。其中 SPA 由所有的 TPC 构成，存储系统包含一系列可编程存储单元，将在 3.3 节详细介绍。这两部分通过片上高速交叉互联网络连接，具有良好的扩展性。当然，Tesla 架构中 G80 和 GT200 两种核心也存在差异，如图 3.3 和图 3.4 所示。

GPU 可扩展性主要通过增减 TPC 和内存控制器实现，NVIDIA 可以生产出满足不同消费者需求的产品，比如在基于 G80 核心的 GeForce 8800 GTX 中，包含 8 个 TPC，每个 TPC 有 2 个 SM；基于 GT200 的产品中，根据市场定位不同，NVIDIA 推出了诸多不同档次的产品，比如包含 10 个 TPC（30 个 SM）和 8 个存储控制器的 GTX285，包含 10 个 TPC 和 7 个存储控制器的 GTX275，和由 9 个 TPC 和 7 个存储控制器组成的 GTX260＋。

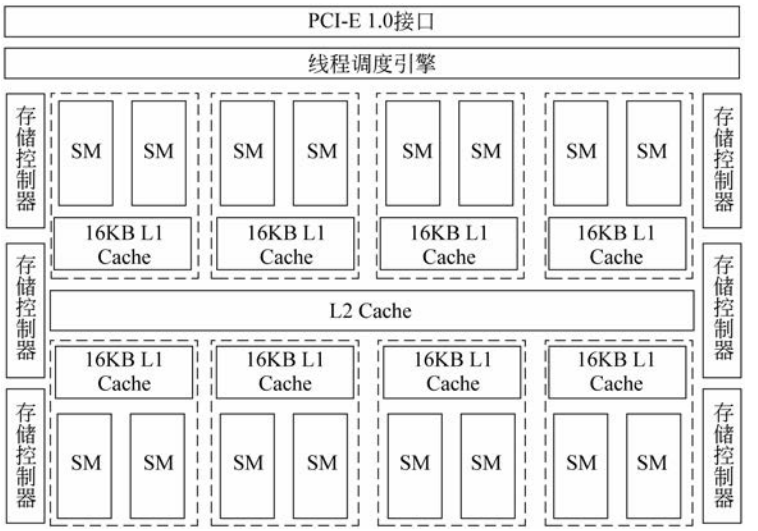

图 3.3 Tesla 架构 G80 核心

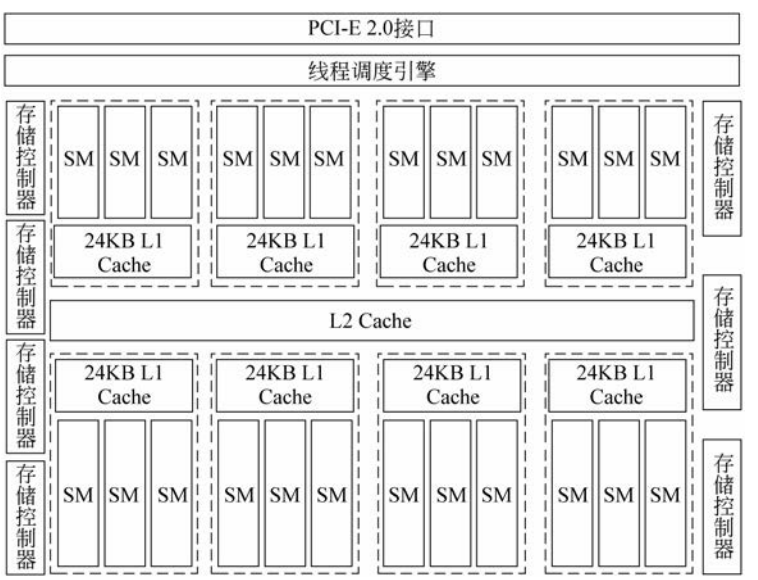

图 3.4 Tesla 架构 GT200 核心

3.1.2 Fermi 架构

Fermi 架构的计算能力是 2.X，主要包括 2.0 和 2.1。在 Fermi 架构开始，正式出现核心(Core)的概念，取代了原来的 SP，主要区别是 core 采用 IEEE 754-2008 标准，增加了对双精度运算和完整 32 位整数运算的支持。此时即使没有单独的 DP，GPU 也能进行双精度运算。另外，Fermi 架构对原子操作进行了巨大改进，使得该功能真正有了应用的可行性(此前的 Tesla 架构原子操作性能极差，不适合实际应用)。Fermi 还增加了 ECC 内存校验、多 kernel 函数并发执行、共享存储可配置(16KB 或 48KB)、双 warp 调度机制等

功能。

图 3.5 和图 3.6 分别展示了 SM 2.0 和 SM 2.1 的组成结构，SM 2.0 中包含 32 个 Core、16 个 LD/ST、4 个 SFU、2 个线程束调度器。SM 2.1 对 SM 2.0 再次进行了改进，将 Core 数量增加到 64 个，LD/ST 增加到 32 个，SFU 增加到 8 个。

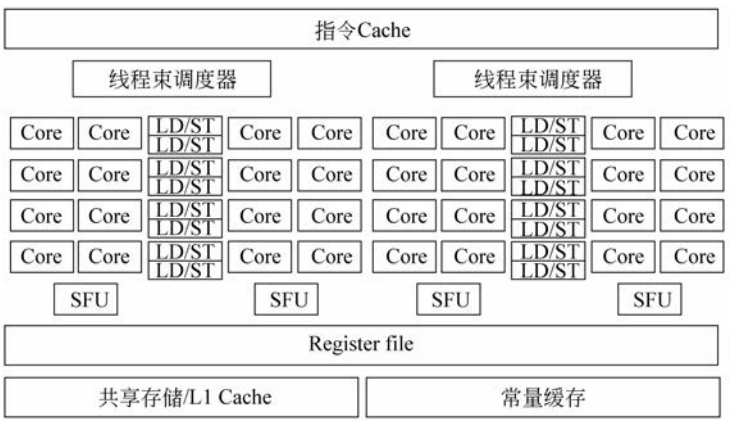

图 3.5　SM 2.0

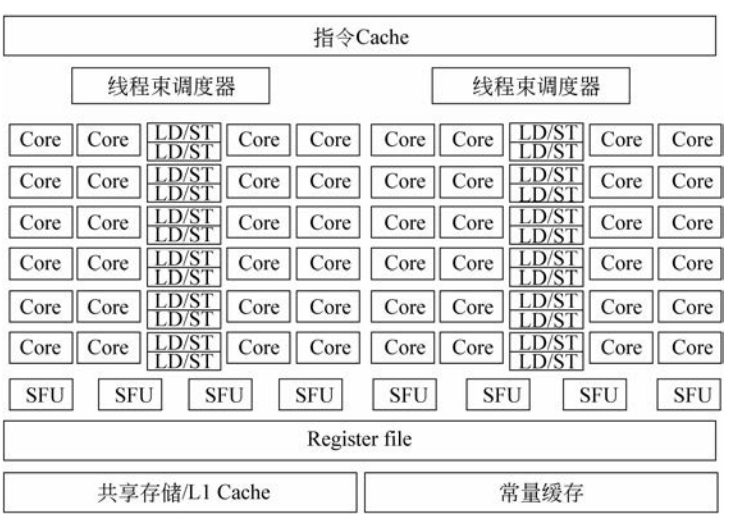

图 3.6　SM 2.1

Fermi 架构（见图 3.7）中，4 个 SM 和 1 个光栅单元组成 1 个 GPC，所有 GPC 构成 SPA。其他部分与 Tesla 类似，包括 L2 Cache、PCI-E 接口、内存控制器（MMC）、线程调度引擎等。每个内存控制器控制 1 个 64b（位）显存分区，即每个内存控制器支持每次 64b 数据合并访存，因此图 3.7 中 6 个内存控制器供支持 384b 显存数据合并访存。

3.1.3　Kepler 架构

本书所描述的 Kepler 架构是 GK110 及以后的 Kepler 架构，计算能力为 3.5 和 3.7。

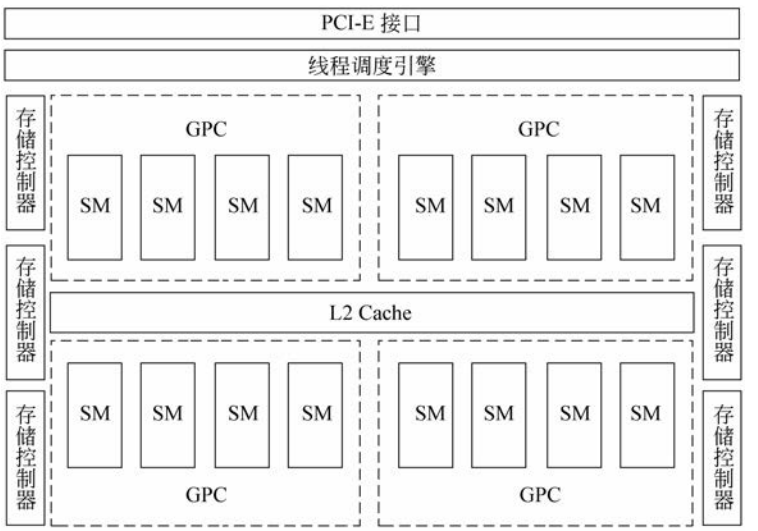

图 3.7 Fermi 架构

不包括 GK104(计算能力为 3.0)。GK110 开始有了重大创新,比如引入 SMX 替代了原来的 SM,增加了动态并行功能,引入 Hyper-Q 技术允许多个 CPU 线程同时在同一个 GPU 上启动 kernel 函数运算,引入 GPUDirect 技术利用远程直接存储访问(remote directly memory accessing,RDMA)支持 GPU 间的直接数据通信等。

SMX(见图 3.8)包含 192 个单精度 CUDA 核心、64 个双精度单元(DP)、32 个特殊功能单元(SFU)和 32 个加载存储单元(LD/ST),其中,单精度 CUDA 核采用 IEEE 754-2008 标准支持单精度核双精度运算,Kepler 架构额外增加 DP 来增强双精度运算能力(GK110 专为高性能计算设计)。SMX 以单个 warp(32 线程)为单位进行调度。SMX 还

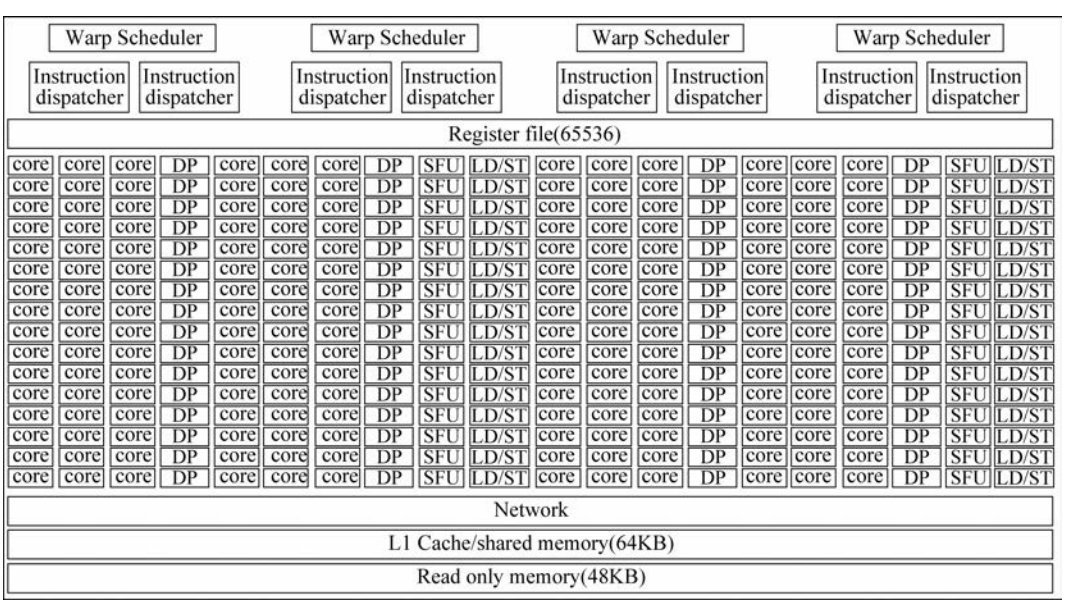

图 3.8 SMX

包含 4 个指令分发单元、8 个指令调度器、可配置共享存储和 L1 Cache 共 64KB(首次支持 16/32/48KB 三种配置),48KB 只读存储。

Kepler GK110 架构(见图 3.9)有 15 个 SMX 和 6 个 64 位内存控制器,部分产品有所不同(比如 Tesla K20c GPU 只有 13 个 SMX 和 5 个 MMC)。官方的 Kepler 架构白皮书中没有提及 GPC 的概念(笔者推测 Kepler 架构中不存在 GPC,因为 13 或 15 个 SMX 不能平均分配到 GPC 中)。其他组成部分类似,只是性能比之前架构有了大幅提升,比如支持 PCI-E 3.0 接口等。

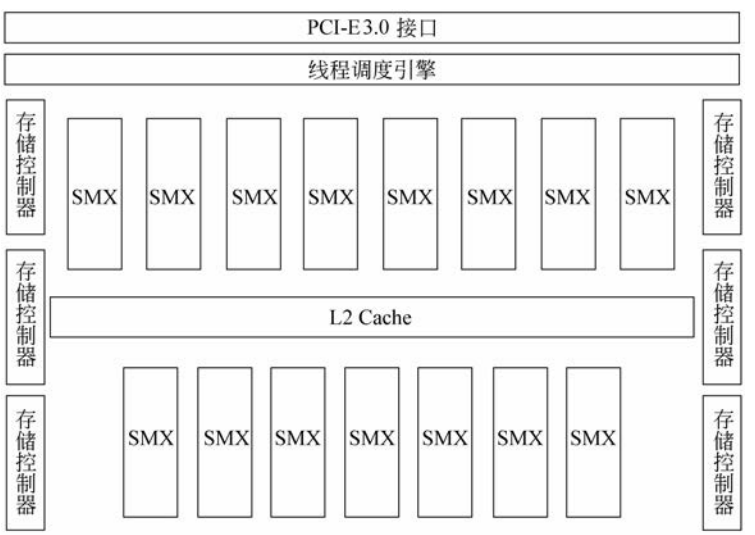

图 3.9　Kepler 架构

3.1.4　Maxwell 架构

Maxwell 架构对 SM 再次做了升级,称谓变为 SMM。SMM(见图 3.10)中包含 128 个 CUDA 核、32 个 SFU 和 32 个 LD/ST、4 个指令分发单元和 8 个调度器(每个调度器每个时钟能启动 1 条指令)。SMM 将 L1 Cache 从 Shared Memory 中抽离了出来,提供了专门的 Shared Memory,将 L1 Cache 与 Texture 相结合。在 GM204 中提供了 96KB 共享存储、PolyMorph Engine 3.0,而在 GM107 中仍提供 64KB 共享存储和 PolyMorph Engine 2.0。

图 3.11 展示了 Maxwell 架构最后一代产品 GTX980 上的 GM204 核心架构,由内存控制器、GPC、线程调度引擎和 PCI-E 接口共同组成了整个架构。其中 GPC 由 1 个光栅单元和 4 个 SMM 组成。

从 GTX750Ti 架构白皮书中了解到,其 GM107 核心仅有 1 个 GPC,包含 5 个 SMM。由此可知,GPU 架构具有非常优秀的多层次可扩展性,包括 TPC(GPC)层可扩展性、SM(SMX、SMM)层可扩展性和 SP(Core)层可扩展性。

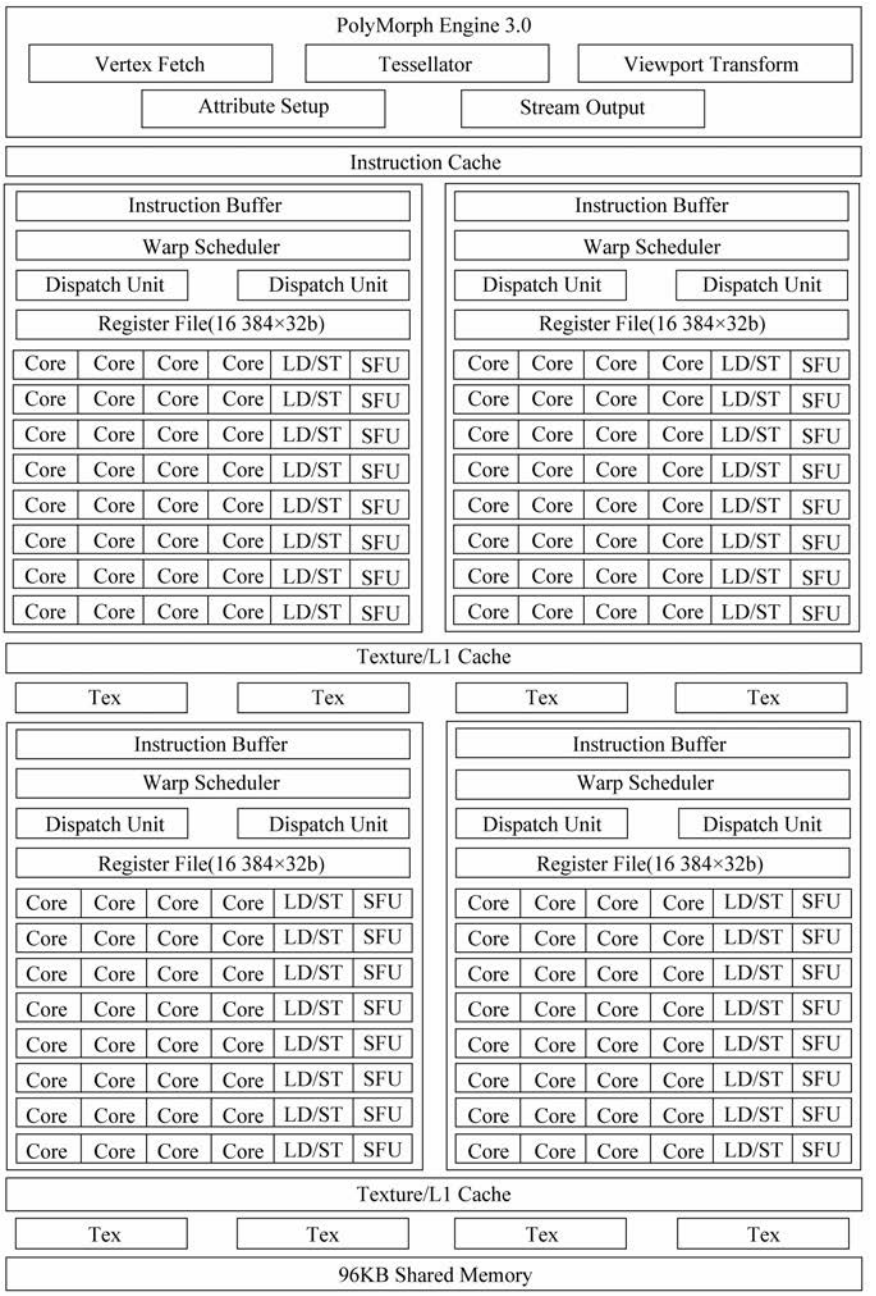

图 3.10 SMM

3.1.5 Pascal 架构

Pascal 架构其实是 NVIDIA 的第 6 代 GPU 架构，GP100 核心的计算能力为 6.0，但是真正发布的产品仅有 5 代。

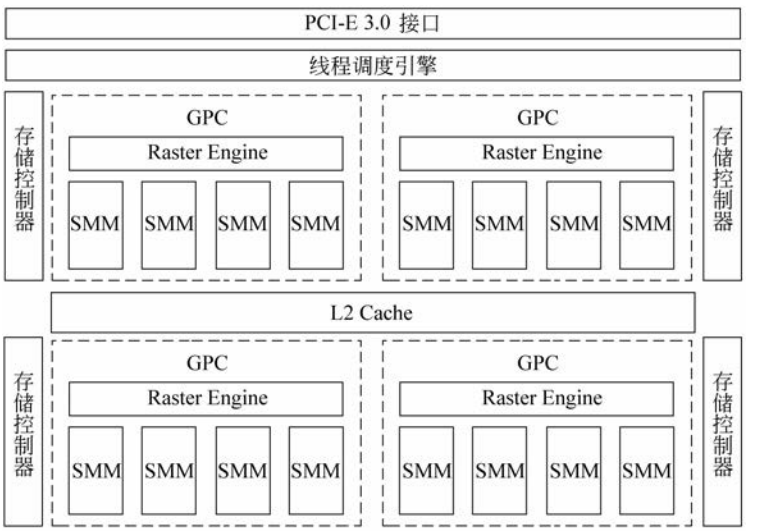

图 3.11　Maxwell 架构 GM204 核心

Pascal 架构(GP100)中的 SM 可分为两个处理块，每个处理块拥有 32 个单精度 CUDA 核、16 个双精度处理单元、8 个 LD/ST 单元、8 个 SFU 单元、1 个指令 buffer、1 个 warp 调度器、2 个指令分发单元、32 768 个 32 位寄存器(见图 3.12)。整个 SM 还拥有 4 个纹理单元和 64KB 共享存储。

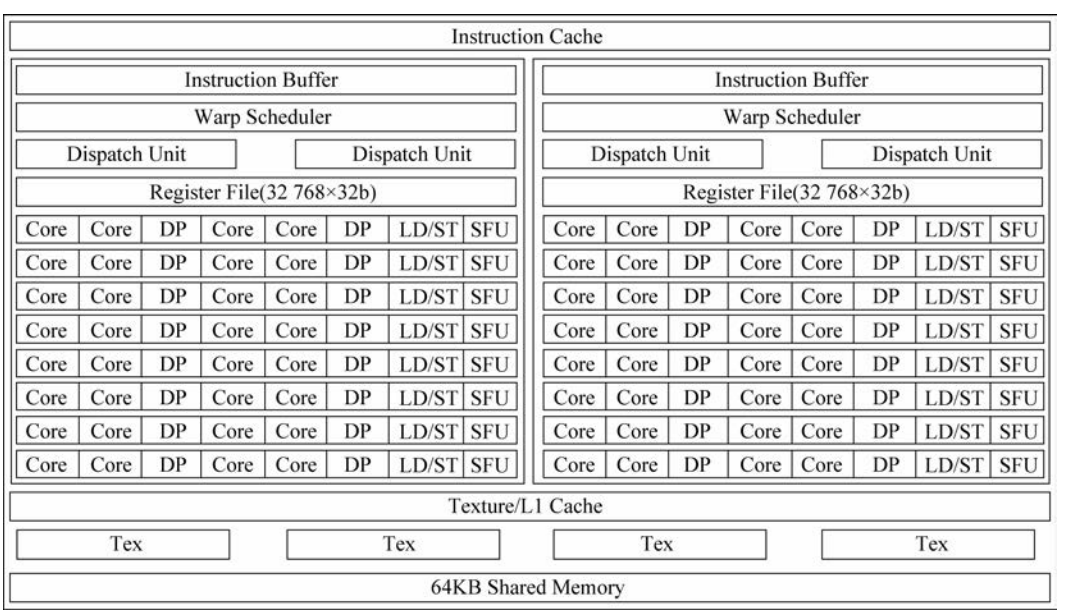

图 3.12　Pascal(GP100) SM

在 Pascal 架构中，2 个 SM 组成 1 个 TPC，5 个 TPC 组成 1 个 GPC(该架构中 TPC 和 GPC 同时出现，且是包含与被包含的关系)。1 个完整的 GP100 核心(见图 3.13)拥有

6 个 GPC、30 个 TPC、60 个 Pascal 架构 SM、8 个 512 位宽的存储控制器(共 4096 位)、3840 个单精度 CUDA 核、240 个纹理单元和 4 个 NVLink 单元。每个存储控制器管理 512KB 的 L2 Cache(共 4096KB)。两个存储控制器控制 1 个 HBM2(high bandwidth memory 2) DRAM stack。每个 NVLink 单元可以提供 40GB/s 的双向通信带宽。

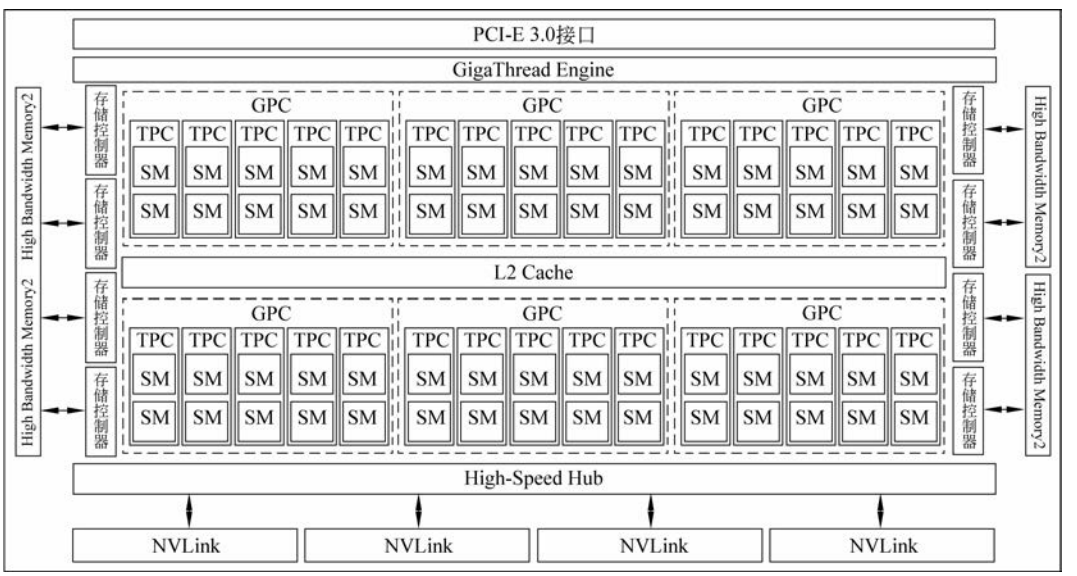

图 3.13 完整的 GP100 GPU

当然,根据市场定位的不同,不同的 GPU 产品的 SM 数量和 TPC 数量等均有所差异,比如 Tesla P100 GPU 采用了 GP100 核心,但并不完整,仅包含 28 个 TPC、56 个 SM、3584 个单精度 CUDA 核心。

相对于 Kepler 架构和 Maxwell 架构,Pascal 架构引入了不少新技术,提供了更加强大的性能。Pascal 架构的优势如下。

(1) 超高的浮点运算性能。Tesla P100 GPU 提供了 5.3Tflops 双精度运算性能、10.6Tflops 单精度运算性能和 21.2Tflops 半精度运算性能(半精度运算主要是针对当前极火的深度学习开发的),利用 GPU Boost 技术可达到单精度 21.2Tflops 运算性能。

(2) NVLink 技术。NVLink 技术能支持多 GPU 间或 CPU 与 GPU 间的通信,能提供相对 PCI-E 带宽 5 倍的通信速率,双向通信带宽可达 160GB/s。

NVLink 可以将多个 GPU 交叉连接,构建成网格以获得更好的通信效益。比如图 3.14 展示了一个由 8 个 P100 GPU 组成的立体网格。图中每个虚线表示的 NVLink 连接可以提供 40GB/s 的双向通信带宽。若是相互连接的 GPU 数量下降,那么多出来的 NVLink 单元可以与现有其他 GPU 连接,若两个 GPU 间连接了两个 NVLink 连接,那么其双向通信带宽为 80GB/s。若系统中只有两个 GPU 通过 NVLink 连接(4 条虚线),则双向通信带宽为 160GB/s。

当 NVLink 被用来进行 CPU 和 GPU 通信时,前提条件是 CPU 和 GPU 都支持 NVLink 技术。目前只有 Pascal 架构的 NVIDIA Tesla P100 GPU 和 IBM 公司的

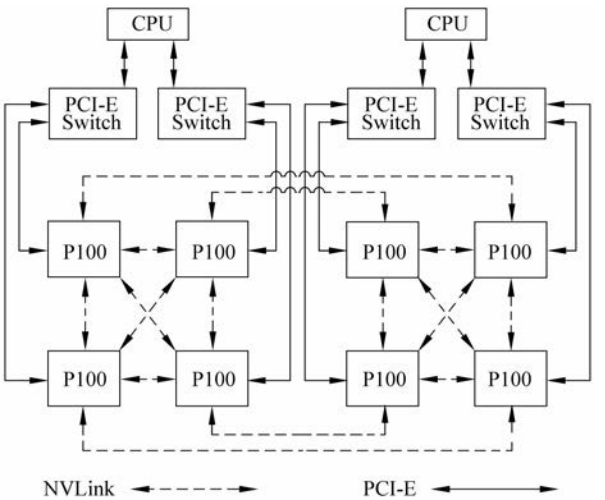

图 3.14　8 个 P100 组成的立体网格

Power8 CPU 支持 NVLink 技术。当 CPU 和 1 个 GPU 利用 NVLink 技术连接时，如图 3.15(a)所示，有 4 条 NVLink 连接，可提供 160GB/s 的双向通信带宽。而当 CPU 连接两个 GPU 时，如图 3.15(b)所示，每个 GPU 只能提供 2 条 NVLink 连接与 CPU 相连，GPU 间有两条 NVLink 连接，故双向通信带宽仅为 80GB/s。

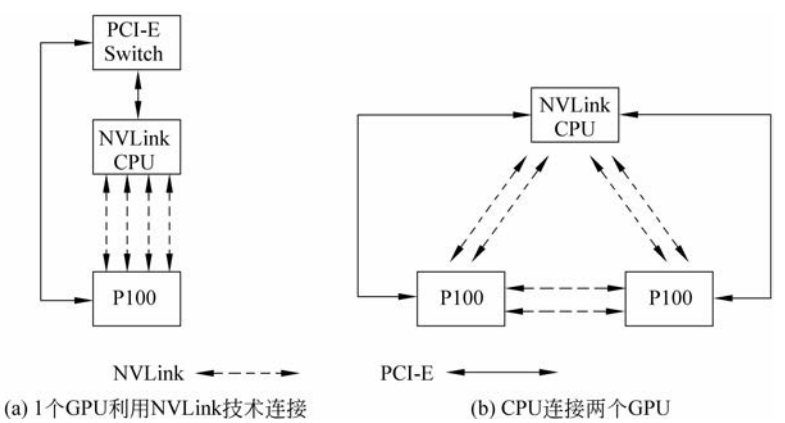

图 3.15　NVLink 连接 CPU 和 GPU

注意：也别把上述带宽数据想象得太过美好，双向通信带宽在 GPU 程序开发过程中应该极少能够用到，我们能够用到的可能仅仅是其一半。比如在本书实验平台上测得的页锁定存储的 PCI-E 带宽大概是 6GB/s，其双向通信带宽至少是 12GB/s，但极少有应用存在同时双向通信来使用这 12GB/s 的双向通信带宽，多数情况只能用到 6GB/s 带宽，甚至有时由于通信数据量过小导致真实的通信速率远低于 6GB/s。

(3) HBM2 堆叠内存。P100 首次在 GPU 中引入 HBM2 高速 GPU 存储架构，访存带宽同比增长 3 倍，最高可达 720GB/s。在 Pascal 架构中，一个存储控制器的访存位宽为 4096 位，而 Kepler 架构和 Maxwell 架构的 GDDR5 位宽普遍为 384 位。

(4) 统一存储空间。P100 首次在 GPU 中引入统一存储空间(在软件上,CUDA 6 开始就引入了统一虚拟存储,不过 Pascal 架构引入的统一存储空间做了重要的扩展和改进,详细内容请查阅 Pascal 架构白皮书),统一存储提供了 CPU 和 GPU 存储的统一地址空间(49 位寻址空间,可容纳 512TB 虚拟存储空间)。利用统一存储,程序员无须关心数据存储位置和设备间通信。

(5) 计算抢占。Pascal 架构允许计算任务在指令级被中断,将程序上下文交换到 DRAM,其他程序被调入和执行(此前的 Maxwell 和 Kepler 架构只能在 block 级进行任务切换)。计算抢占技术能避免长时间运行的应用独占系统或发生运行超时。这样,在与类似于交互式图像任务或交互式调试应用协同运行时,该长时运行的应用可以长时间运行来处理大规模数据或等待其他条件。另外,计算抢占可以打破 kernel 函数的运行时间限制,GPU 程序开发时将不必专门考虑 kernel 函数运行是否会超时。

(6) 原子操作扩展。Pascal 架构对原子操作进行了扩展,增加了 64 位数据的支持,包括 64 位长整型数据和双精度浮点型数据。

3.2 Kernel 的硬件映射

Kernel 函数是指在 GPU 上执行的函数。一般用__device__和__global__关键字声明,在 CPU 端调用时利用<<<>>>配置相应的线程(块)维度、共享存储容量和 CUDA 流等信息。

Kernel 函数映射到 GPU 上执行,对比普通函数,增加了以下几个概念,分别是 grid(线程格)、block(线程块)、thread(线程)。三者是包含关系,大量的 thread 组成一个 block,大量的 block 组成一个 grid(参见图 5.1)。在执行 kernel 函数时,一个 kernel 函数对应一个 grid(从 Fermi 架构开始,一个 GPU 支持同时多个 kernel 函数执行),grid 内 block 数量和 block 内 thread 数量在 kernel 函数启动时在<<<>>>内指定。声明方法如下:

```
dim3 blocks(bx,by,bz),threads(tx,ty,tz);
gpu_kernel<<<blocks,threads>>>(…);
```

Tesla K20c GPU 中 threads(tx,ty,tz)尺寸维度为(1024,1024,64),每个 block 最多 1024 个 thread;blocks(bx,by,bz)尺寸维度为(2147483647,65535,65535)。线程设计原则:每个 block 中的 thread 数量合适(一般为 256、512、1024),使得占用率最大化(占用率计算工具在 CUDA 安装目录的 tools 文件夹下,CUDA_Occupancy_Calculator. xls);block 数量越大越好(GPU 中 block 的切换开销极小,几乎可以忽略不计)。

在执行时,kernel 函数映射到 GPU,对应的 block 映射到 SM(SMX,SMM),thread 映射到 SP(CUDA core)。很明显,kernel 中设置的 block 数量和 thread 数量都远远超过 GPU 硬件中的 SMX 和 core 数量,GPU 通过频繁的线程切换实现硬件资源的分时使用,切换开销极小,几乎可以忽略。在实际的开发过程中,与 GPU 映射和执行相关的开销主

要有两个，分别是 kernel 函数启动开销、GPU 线程(块)切换开销。

从较细粒度的角度看 GPU 执行，GPU 执行采用单指令多线程(single instruction multiple thread，SIMT)模式，即每个 warp 执行同一条指令，比如 ID 为 0～31 的 thread 同时执行同一条指令。借用 1.5 节的比喻，该过程可以看成是战场上 32 个士兵(thread)为战阵的一排(warp)，同时挥刀砍向敌军，此时，如果敌方(数据)也是整齐排列(访存对齐)，那么效果最佳，否则自然会导致效果不好，条件是战阵中一排士兵只能做相同的动作(指令)。

需要说明的是，在 Tesla 架构中，SIMT 线程调度单位是半 warp，即 16 线程；从 Fermi 架构开始，SIMT 线程调度的单位改成了一个 warp，即 32 线程。

3.3　GPU 存储体系

GPU 存储体系是影响 GPU 程序性能的最重要的因素之一。GPU 设计了鲜明的层次式存储，使用好层次式存储是进行性能优化的关键。GPU 存储结构如图 3.16 所示。

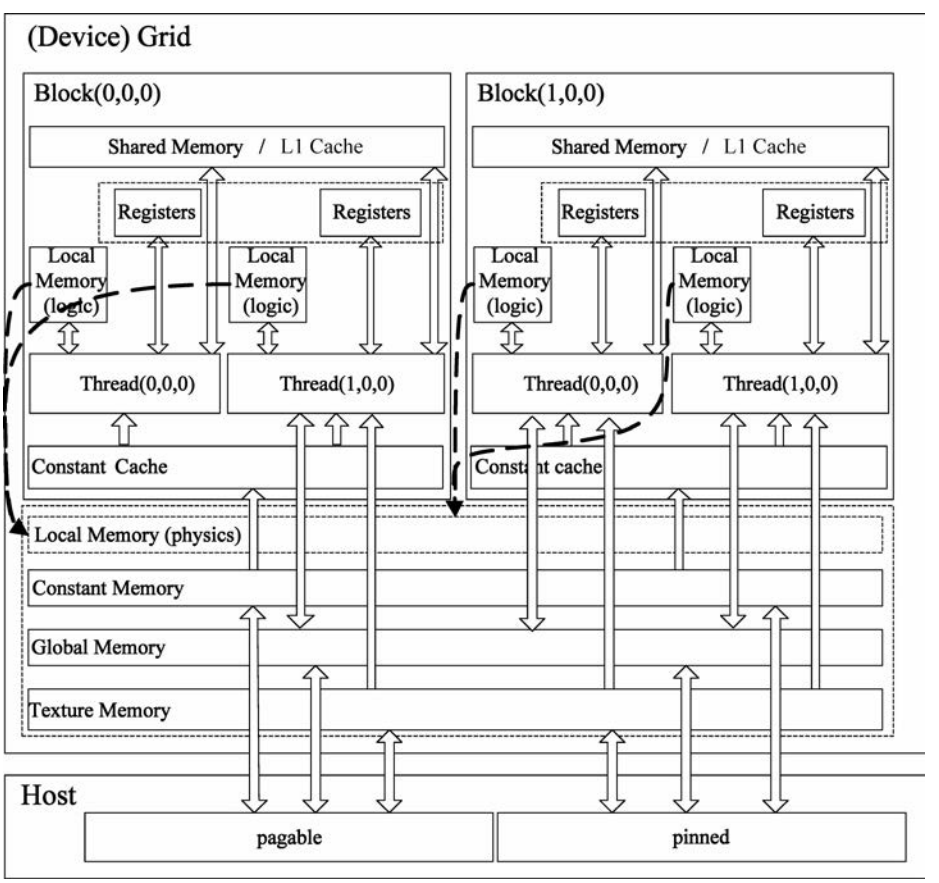

图 3.16　GPU 存储模型

(1) 寄存器和局部存储对线程私有,寄存器资源位于SMX上,数量有限,线程运行时动态获得;局部存储逻辑上等同寄存器,但其物理局部存储空间位于显存中,故其访存延迟与全局存储相当。

(2) 共享存储与L1 Cache使用相同的硬件资源(在Maxwell架构进行了分离,提供专门的共享存储,将L1 Cache与Texture结合),一般总大小为64KB,其中共享存储可通过编程控制,且对线程块内所有线程共享。

(3) 每个SMX上都有常量缓存,主要支持常量存储的缓存和广播功能,常量存储本身位于显存中,通过SMX上的常量缓存实现常量存储快速访问。

(4) 全局存储和纹理存储都位于显存中,区别是纹理存储提供了专门的纹理缓存通道。

(5) 主机端提供了页锁定内存和可分页内存两种形式的存储类型,可与设备端的常量存储、全局存储和纹理存储进行数据交换。各级存储根据离核距离不同访存延迟也不相同,基本使用原则是尽量使用离核近的、访存延迟小的存储单元,尽可能地避免使用全局存储等访存延迟高的存储单元。(具体的存储单元阐述和使用请阅读本书第16章。)

3.4 GPU计算能力

在GPU中,计算能力和运算性能是两个不同的概念,在直观上容易令人混淆。

运算性能包括整数运算性能、单精度浮点运算性能和双精度浮点运算性能等,表示GPU处理算术运算的能力,也是衡量GPU好坏的关键标准之一。

GPU计算能力是指GPU架构或GPU支持的功能,而与GPU的浮点运算性能无关(一些旧版本的高端卡性能往往超过新版本的中端卡或低端卡)。随着GPU架构的发展,GPU支持的功能也越来越多。从最初的计算能力1.0到目前最新的计算能力6.0,GPU经历了Tesla架构、Fermi架构、Kepler架构、Maxwell架构和Pascal架构,增加了大量实用功能的支持。

计算能力1.0:主要产品是8800Ultras和8000系列卡,GPU核心为G80,经典产品如8800GTX。笔者姑且将计算能力1.0称为基准版本。尽管此时还不支持比如原子操作、动态并行、多kernel同步执行等功能,但此时基本的GPU框架已经成型,其后所有的GPU架构发展都没有脱离此时的框架。一般情况下,使用NVCC进行程序编译时,无须特殊功能的不需要指定计算能力,此时默认采用的就是计算能力1.0的编译选项,经笔者测试,多数情况往往比采用更高计算能力编译得到的程序性能还好一点。

计算能力1.1:主要产品是9000系列卡,GPU核心为G92,经典产品有9800GTX。计算能力1.1在1.0的基础上增加了计算通信重叠功能,即数据传输和kernel函数可以重叠执行,设备函数中cudaGetDeviceProperties()函数的deviceOverlap属性反映了该项功能。此外,计算能力1.1还添加了全局存储器的32位字原子操作函数。

计算能力1.2:主要产品是GT200系列卡,GPU核心为GT200,经典产品GTX260和GTX280。计算能力1.2大幅度提升了GPU的浮点运算性能,主要表现在:①SM和SP数量增加,片上集成的计算核心数量增加;②SM中并发执行的线程束(warp)从原来

的 24 增加到了 32；③大大减少了在计算能力 1.0 和 1.1 中常见的对全局存储器的访问限制和共享存储器的存储体冲突(bank conflict)。此外，计算能力 1.2 开始支持共享存储原子操作功能，并添加了对全局存储 64 位字原子操作的支持。

计算能力 1.3：主要产品是 GT200 系列高端卡，GPU 核心为 GT200 a/b 修订版。相对于计算能力 1.2，计算能力 1.3 的主要改进是开始支持双精度运算，该功能的引入为 GPU 后来在高性能计算领域的卓越成绩做出了不可磨灭的贡献。

计算能力 2.0：计算能力 2.X 开始采用 Fermi 架构的 GPU 核心，计算能力 2.0 采用的是 GF100(SM 2.0)(详见 3.1.2 节)，主要产品是 GT400 和 GT500 系列，经典产品有 GTX480(入选 NVIDIA 提供的 CUDA 入门套包)、GTX580、M2050(天河 1A 超级计算机上装备)。计算能力 2.0 做了大量改进：①L1 Cache 引入 SM 中(此前 Tesla 架构中位于 SM 外 TPC 中，详见 3.1.1 节和 3.1.2 节)，并可选择 16KB 和 48KB 配置；②增加了 ECC (error correcting code)校验；③支持双复制(dual-copy)引擎；④扩充了共享存储容量；⑤共享存储片(体，bank)数从 16 增加到 32；⑥执行时开始以一个线程束(warp)为单位执行(此前是半 warp 执行)；⑦大幅改进了原子操作的性能，使其真正有了实用的可能性。

计算能力 2.1：采用 Fermi 架构的 GF104(SM 2.1)核心，主要产品有 GT400 和 GT500 系列中端卡、GT600 系列低端卡，比如 GTX460、GTX550 Ti、GT610 等。计算能力 2.1 产品的市场定位是中端游戏卡，而非高端计算卡，因此牺牲了其双精度运算来增加 CUDA 核心数量。计算能力 2.1 的主要特色包括：①SM 中 CUDA 核数量从 32 增加到 48；②SM 中 SFU 数量增加 1 倍，从 4 个增加到 8 个；③采用了双束调度器替代单束调度器(单束调度器需要 2 个时钟取出 2 条指令，而双束调度器 2 个时钟可取出 4 条指令执行)。

计算能力 3.0：计算能力 3.X 采用了 Kepler 架构，计算能力 3.0 的核心是 GK104，主要产品是 GT700 系列中高端卡、GT600 系列高端卡，比如 GTX770、GTX690、Tesla K10 等。

计算能力 3.5：NVIDIA 官方给出的 Kepler 架构白皮书所描述的 Kepler 架构就是指代计算能力 3.5 的 GPU 架构。计算能力 3.5 的核心是 GK110，主要产品有 GT700 系列高端卡、TITAN 系列，比如 GTX780、GTX TITAN、GTX TITAN Z、Tesla K20、Tesla K40 等。计算能力 3.5 主要有以下改进：①引入 SMX 替代了原来的 SM；②SMX 中 CUDA 核数量增加到 192；③增加了动态并行功能；④引入 Hyper-Q 技术允许多个 CPU 同时在同一个 GPU 上启动 kernel 函数运算；⑤引入 GPUDirect 技术利用 RDMA 支持 GPU 间的直接数据通信。

计算能力 3.7：核心为 GK210，主要产品 Tesla K80。主要特征有：①GPU Boost(超频)技术；②双倍的共享存储和寄存器；③Zero-power Idle(闲置时零功耗)。

计算能力 5.0：计算能力 5.X 采用了 Maxwell 架构，计算能力 5.0 采用了 GM107 核心，主要产品有 GTX750 Ti、GTX 960M 等。计算能力 5.0 的改进如下：①引入了 SMM 替代 SM；②SMM 中 CUDA 核数量降到 128；③提升了每 Watt 性能；④提供了 2048KB 的 L2 Cache(GK107 中仅有 256KB)；⑤L1 Cache 从共享存储抽离，与 Texture memory 结合。

计算能力5.2：采用GM204核心，主要产品有GT900系列高端卡，如GTX TITAN X、GTX980等。计算能力5.2做了如下改进：①SMM中PolyMorph Engine升级为3.0；②SMM中共享存储提升到了96KB。

计算能力6.0：采用GP100核心，当前产品有Tesla P100 GPU。计算能力6.0属于Pascal架构，引入了大量改进：①SM结构进行调整，单个SM中的CUDA核调整为64个，加入32个DP；②GPU的组成做了调整，GPC和TPC同时出现，且GPC包含TPC，TPC包含2个SM；③计算性能极大增强；④采用NVLink技术，支持GPU间和CPU与GPU间通信，双向带宽可达160GB/s；⑤采用CoWoS HBM2堆叠内存，访存带宽高达720GB/s；⑥统一存储空间，简化编程时的显式数据传输；⑦计算抢占，避免应用独占系统或运行超时，应用可以长期运行来处理大规模数据或等待其他条件；⑧扩展了原子操作，增加了对64位数据的支持，包括64位长整型数据和双精度浮点型数据。

此外，关于GPU计算能力，值得一提的是，在NVCC编译器中的--gpu-architecture(-arch)和--gpu-code(-code)选项都需要指定计算能力(关于这两个选项的区别请参见附录E)，此时可设置的计算能力有两大类：compute_35和sm_35。其中，compute_35指定虚拟编译计算能力，NVCC编译器利用该指定的计算能力编译CUDA C代码为PTX(parallel thread execution)中间代码；而sm_35指定GPU架构，NVCC编译器根据该项数据将PTX中间代码编译为二进制执行文件。

第4章 GPU软件体系

本章主要介绍GPU软件生态系统、CUDA Toolkit、CUDA环境安装配置等内容。

4.1 GPU软件生态系统

时至今日，GPU并行程序开发已经得到全面发展，形成了有机的软件生态系统。构成GPU软件生态系统的成分包括编译器、编程模型、数学函数库、性能分析工具、程序调试工具、代码实例(SDK)、管理软件、应用软件和完整的文档等。

编译器主要有NVIDIA CUDA Compiler(NVCC)、PGI CUDA Fortran Compiler、ptxas(PTX汇编工具)、cuobjdump(CUDA目标文件转储工具)。此外，还有些自动化并行编译器(源到源的转换工具)，比如，可将Fortran和C转换成CUDA C的PGI编译器和CAPS HMPP，可将C转换为CUDA C代码的Goose编译器，将Fortran转换为CUDA C的NOAA F2C编译器。

GPU编程模型主要有CUDA、OpenCL、OpenACC，其中CUDA已支持诸多编程语言，最常用的是CUDA C(NVCC)，其他的还包括CUDA Fortran(PGI、FLAGON支持、ArrayFire GPU library)、CUDA Python(PyCUDA、ArrayFire GPU library)、CUDA Java(jCUDA、JaCUDA)、CUDA .NET、CUDA C++(Thrust、CuPP、Libra、ArrayFire GPU library)、CUDA F#。

面向GPU的数学函数库种类相当丰富，比如线性代数库CUBLAS、稀疏矩阵运算CUSPARSE、快速傅里叶变换CUFFT、深度学习CUDNN、AMGX、NPP、FFmpeg、CHOLMOD、CULA Tools、MAGMA、IMSL Fortran数值库、CUSOLVER、ArrayFire、CURAND、Thrust、NVBIO、NVIDIA VIDEO CODEC SDK、HiPLAR、OpenCV、GPP(Geometry Performance Primatives)、Paralution、Triton Ocean SDK等。

性能分析工具主要有NVIDIA Visual Profiler。

程序调试工具有命令行的CUDA-GBD、界面版开发工具的Nsight。

代码实例有 NVIDIA 提供的 CUDA SDK Code samples。

管理软件包括 NVIDIA 系统管理工具 nvidia-smi。

另外，针对常用软件如 MATLAB、Mathematica、R、LabView 等提供了二次 GPU 开发支持，比如支持 MATLAB 二次开发的 MathWorks 和 GPUlib 等。

4.2 CUDA Toolkit

在 CUDA 5.0 之前的 CUDA Toolkit 包中不包含 GPU 驱动和 SDK，而 CUDA 5.0 开始三者集成到统一的 CUDA Toolkit 安装包中。

CUDA Toolkit 提供了 CUDA C 程序开发的基本软件环境，包括 NVCC 编译器、CUDA 驱动 API、CUDA 运行时 API、CUDA 库函数等(见图 4.1)。程序员可根据需求选择驱动 API、运行时 API 或 CUDA 库函数进行 CUDA C 编程，然后利用 NVCC 编译器编译即可得到可执行程序。其中，驱动 API 是 CUDA 编程时使用的最底层的 API，使用较为复杂，其函数命名为 cu*()；运行时 API 使用相对简单，其函数命名方式为 cuda*()。

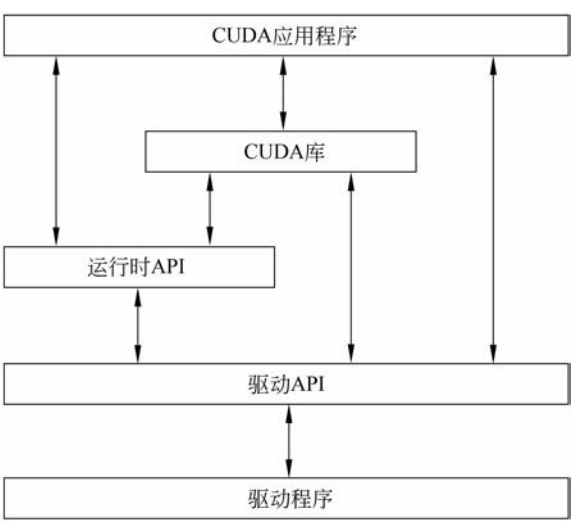

图 4.1 CUDA Toolkit 软件层次

此外，CUDA Toolkit 还提供了其他编译工具，如 ptxas 汇编工具、cuobjdump 等，还有 GPU 管理工具 nvidia-smi、性能分析工具 visual profiler、occupancy 计算工具(excel 文档)、丰富的文档资料等。

4.2.1 NVCC 编译器

NVCC(NVIDIA CUDA Compiler)是编译 CUDA 代码的编译器，其功能是将 CUDA C 源代码编译为一个可执行的 CUDA 应用程序。NVCC 包含编译、链接等多种功能。最简单的使用方法如下：

```
$nvcc abc.cu -o abc
```

除此之外，NVCC 还提供了多种编译开关，详细信息请阅读附录 E。这里仅介绍两个最简单、最实用的编译开关，-O3 是优化等级，等同 ICC、GCC 等编译器，只在编译 CPU 端代码时起作用；-arch 'compute_35' -code 'compute_35' 指定代码编译的计算能力，最好与 GPU 计算能力一致，35 即计算能力为 3.5，与本文代码性能测试的 Tesla K20c GPU 一致。完整指令如下，详细功能说明参考附录 B。如果需要编译驱动 API 程序，需要加入 -lcuda 编译选项，否则链接时会出错。

```
$nvcc abc.cu -o abc -O3 -arch 'compute_35' -code 'compute_35'
$nvcc abc.cu -o abc -O3 -arch 'compute_35' -code 'compute_35' -lcuda
```

注意：附录 E 提供了 NVCC 编译器的帮助菜单供读者参考，CUDA Toolkit 包中提供了 nvcc.pdf 文档。

4.2.2　cuobjdump

PTX(并行线程执行)代码是编译后的 GPU 代码的一种中间形式，可以再编译为原生的 GPU 微码。PTX 编译成 GPU 微码有离线和在线两种方式。离线编译可生成未来可在计算机上执行的目标程序。在线编译指程序在运行时将 PTX 代码在线编译成 GPU 微码。PTX 离线编译采用 ptxas 编译器，生成微码存储在 CUDA 二进制形式的 cubin 文件中。cubin 文件可以反汇编成 SASS 代码，查看 SASS 代码是 GPU 优化的高级手段之一。下面演示了编译和反汇编的相关指令，这里以 6.4 节的代码为例子进行演示。

```
[fangmq@cn18%yhstar SASS_test]$nvcc vectoradd_gpu_3.cu -cubin
[fangmq@cn18%yhstar SASS_test]$cuobjdump --dump-sass vectoradd_gpu_3.cubin
  code for sm_10
    Function : _Z16vector_add_gpu_3PfS_S_i
  .headerflags  @"EF_CUDA_SM10 EF_CUDA_PTX_SM(EF_CUDA_SM10)"
    /*0000*/  MOV.U16 R0H, g [0x1].U16;             /*0x0023c78010004205*/
    /*0008*/  I2I.U32.U16 R1, R0L;                  /*0x04000780a0000005*/
    /*0010*/  IMAD.U16 R0, g [0x6].U16, R0H, R1;    /*0x0020478060014c01*/
    /*0018*/  ISET.S32.C0 o[0x7f], g [0xa], R0, LE; /*0x6c20c7c83000d5fd*/
    /*0020*/  RET C0.NE;                            /*0x0000028030000003*/
    /*0028*/  MOV.U16 R1L, g [0x1].U16;             /*0x0023c78010004209*/
    /*0030*/  SHL R3, R0, 0x2;                      /*0xc41007803002000d*/
    /*0038*/  IMUL32.U16.U16 R5, g [0x4].U16, R1L;  /*0x41022814*/
    /*003c*/  IADD32 R4, g [0x4], R3;               /*0x2103e810*/
    /*0040*/  IADD32 R2, g [0x6], R3;               /*0x2103ec08*/
    /*0044*/  IADD32 R1, g [0x8], R3;               /*0x2103f004*/
    /*0048*/  SHL R3, R5, 0x2;                      /*0xc410078030020a0d*/
    /*0050*/  GLD.U32 R7, global14[R4];             /*0x80c00780d00e081d*/
```

```
/*0058*/  GLD.U32 R6, global14[R2];              /*0x80c00780d00e0419*/
/*0060*/  FADD32 R6, R7, R6;                     /*0xb0060e18*/
/*0064*/  IADD32 R0, R0, R5;                     /*0x20058000*/
/*0068*/  GST.U32 global14[R1], R6;              /*0xa0c00780d00e0219*/
/*0070*/  ISET.S32.C0 o[0x7f], g [0xa], R0, GT;/*0x6c2107c83000d5fd*/
/*0078*/  IADD32 R2, R3, R2;                     /*0x20028608*/
/*007c*/  IADD32 R4, R4, R3;                     /*0x20038810*/
/*0080*/  IADD R1, R3, R1;                       /*0x0400478020000605*/
/*0088*/  BRA C0.NE, 0x50;                       /*0x000002801000a003*/
/*0090*/  NOP;                                   /*0xe0000001f0000001*/
...
```

详情可参考 CUDA Toolkit 提供的 cuobjdump.pdf 文档。

4.3 CUDA 环境安装

4.3.1 Windows 7 安装 CUDA 4.2

本文为什么要在 Windows 环境下选择介绍 CUDA 4.2 的安装流程而非最新版本？首先 CUDA 5 开始已经做了较好的集成，可以直接安装，不需要进行手动配置；而 CUDA 5 之前的版本均需要手动配置。其次是 CUDA 4.2 版本稳定好用，笔者曾安装使用过 CUDA 5 和 CUDA 5.5，使用时曾多次出现 CUDA 环境不明原因的损坏、使用不方便（如编译时出一大堆无用信息）等现象，另外，高版本 CUDA 需要的 VS 版本也越来越高，而笔者认为 CUDA 4.2 已经能够提供几乎所有的 CUDA 编程功能，因此在此推荐使用。

1. 安装前需要准备的软件

（1）VS2008ProEdition90DayTrialCHSX1435983。

（2）VisualAssistX_10.7.1912_Soft711。

（3）310.90-notebook-win8-win7-winvista-32bit-international-whql（选择计算机对应的驱动）。

（4）cudatoolkit_4.2.9_win_32。

（5）gpucomputingsdk_4.2.9_win_32（cudatoolkit 和 skd 版本必须一致）。

（6）CUDA VS Wizard（32 位和 64 位有区别的），32 位：CUDA_VS_Wizard_W32.2.2.exe。

2. 软件安装（按照上面罗列的顺序逐步安装软件）

CUDA 环境配置：64 位请选择相应目录。

1）添加文件包含

依次打开“工具”→“选项”→“项目和解决方案”→“VC++ 目录”，添加包含文件、库文件和源文件。

包含文件：