第19章

通用图形处理单元（GPU）

  本章贡献人：

彼得·芝诺

布里奇波特大学博士研究生

19.1 CUDA基础

19.2 GPU与CPU

CPU与GPU体系结构的基本区别

性能与性能比较

19.3 GPU架构概述

  基线GPU架构

全芯片布局

流式多处理器体系结构细节

认识并按照存储类型编程的重要性

19.4英特尔GE8 GPU

19.5使用GPU作为协处理器的时机

19.6关键术语与习题

**学习目标**

学习本章后，你应该能够做到：

* 概述CUDA。
* 理解GPU和CPU之间的区别。
* 描述典型GPU体系结构的基本元素。
* 描述何时应该使用GPU作为协处理器。

**图形处理器单元**（GPU）是专门为优化三维（3D）图形渲染和视频处理而设计的。今天几乎所有的工作站、笔记本电脑、平板电脑和智能手机中，都会用到GPU[OWEN08]。GPU有很多尺寸。较大的单元在单个集成电路（IC）上有数百到数千个并行处理器核心。它们可以以单独的协处理器卡片的形式，运行在在工作站、游戏系统，甚至超级计算机[SLAV12]中，且通常基于PCIe。最小的GPU存在于嵌入式系统中，例如平板电脑和智能手机，其中GPU仅由不到10个核组成，并且通常与多个传统的核（常被称为同一硅片IC上的**中央处理单元（CPU）**）组合。

  在过去的几年中，GPU在大规模并行编程环境中有广泛的应用，如生物信息学、分子动力学、石油和天然气勘探、计算金融、信号和音频处理、统计建模、计算机视觉，医学影像学。这也就是**通用GPU计算（GPGPGPU）**的术语来源。将高度并行的应用程序迁移到GPU的主要原因，是因为对程序员友好的GPGPU语言的出现，例如NVIDIA的CUDA和Khronos Group的OpenCL，其对GPU架构进行了一些轻微的修改，促进了通用计算[SAND10]的发展。（从这里开始我们称作GPGPU体系结构），以及GPU的低成本和高性能。例如，只需要200美元，就可以为你的工作站购买具有960个并行处理器核心的GPU（例如，NVIDIA的GeForce GTX 660）。

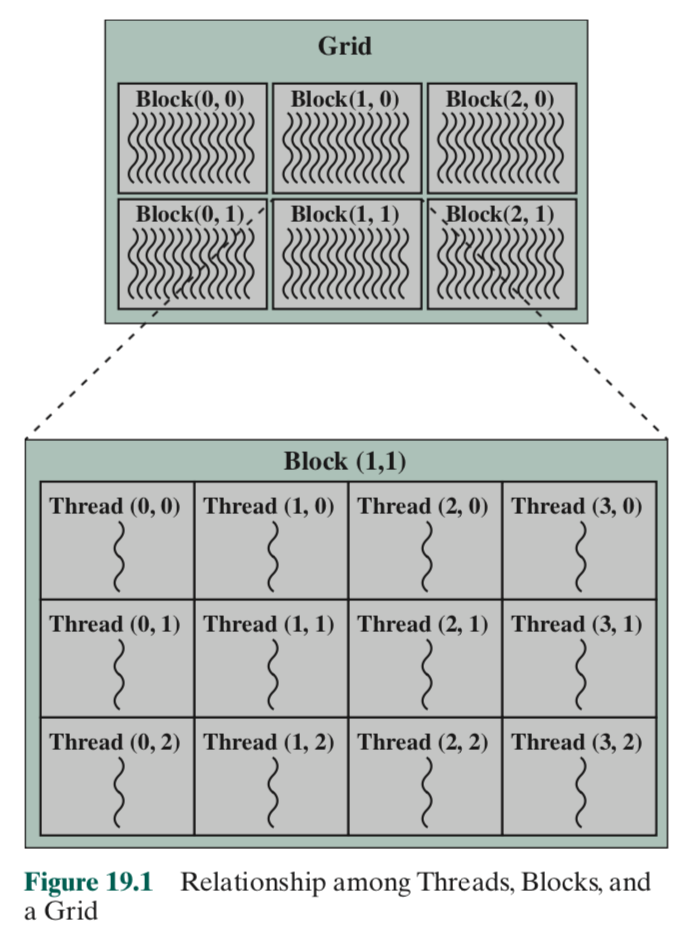
  本章首先概述CUDA模型，这对于理解GPU的设计和使用很重要。然后，本章将对比GPU和CPU。接下来是GPU体系结构的详细介绍。然后，本章会介绍英特尔的GPU。最后，本章将讨论何时使用GPU作为协处理器。

## 19.1 CUDA基础

**CUDA（统一计算设备架构Compute Uni. Device Architecture）**是由NVIDIA创建的并行计算平台和编程模型，并由它们出产的图形处理单元（GPU）实现。为了充分描述GPGPU体系结构，首先需要介绍几个CUDA软件术语和概念。这并不是对CUDA编程语言的全面介绍，要考虑到本章和本书的重点是计算机体系结构。然而，如果不首先介绍清楚CUDA的软件术语以及它的的编程框架，就很难描述明白GPGPU系统的硬件部分。这些概念将延续到GPU/GPGPU体系结构领域。

  CUDA C是一种基于C/C++的编程语言。CUDA程序可以分为三个主要部分：（1）在主机（CPU）上运行的代码；（2）在设备（GPU）上运行的代码；（3）与主机和设备之间的数据传输相关的代码。在主机上运行的代码是不能并行化运行，同时也不值得并行化运行的串行代码。要在GPU上运行的数据并行代码称为**内核（kernel）**，而**线程**是这个内核函数的单个实例。内核通常很少有或者没有分支语句。内核中的分支语句会导致线程在GPU硬件中串行执行。关于这一点的更多内容将在第19.3节中予以说明。

程序员定义调用内核函数时启动的线程数。定义的线程总数通常数以千计，用来最大化**GPU处理器核**（也称为**CUDA核**）的利用率，同时也用来最大化可用的加速。此外，程序员指定如何绑定这些线程。更具体地说，线程以**块**的形式统一绑定，每个内核启动的块数（也称为**线程块**）称为**网格**（参见图19.1）。表19.1给出了刚刚定义的CUDA术语的摘要。



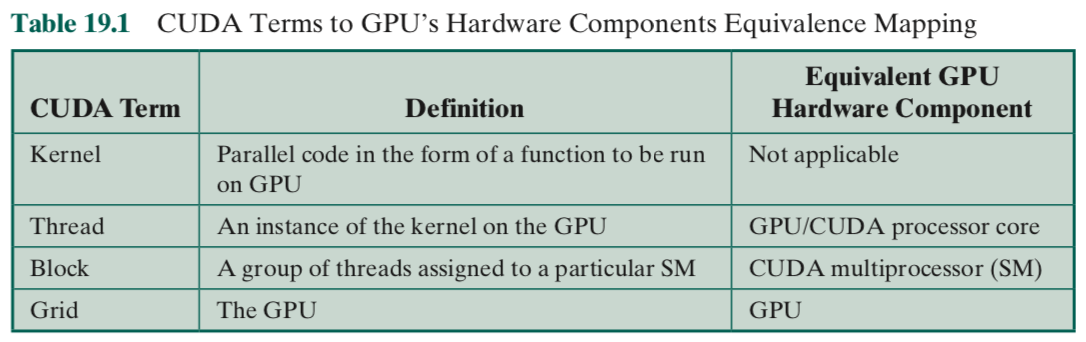


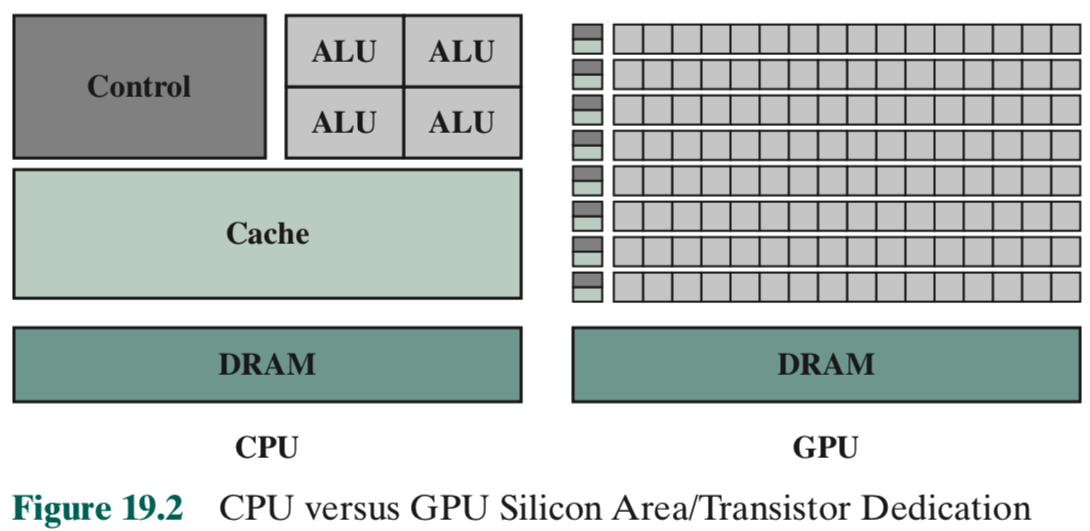
图19.1显示了二维线程块的二维**网格**。网格和块维度可以是一维、二维或三维。它们不需要具有相同的维度。例如，网格可以被设置为一维，线程块可以设置为三维。然而，正如我们将很快看到的，这种配置不能充分利用GPU处理器，因为一个块只分配给多个GPU**流多处理器（SMs）**中的一个。一个块从来没有在流多处理器之间分开。因此，除了一组GPU处理器核心外，所有进程都将空闲，而这一组流多处理器将承担全部处理负载。此外，存在一个流多处理器可以接受的最大线程数量。如果超过了这个数量，代码就不能编译了。因此，由程序员来使用GPU的规格数据，并尽可能均匀地分配负载。至少，启动的线程块的数量应该不小于GPU上的流多处理器的数量。然而，找到最理想配置可能是一个非常耗时而且令人畏缩的过程。

## 19.2 GPU与CPU

  本节将比较GPU和CPU的互补架构。因为GPU和CPU是相互正交优化的，所以与纯CPU方法相比，它们组合而成的异构GPGPU系统可以为某些应用提供优越的成本和性能增益。

### CPU与GPU体系结构的基本区别

因为GPU和CPU是为两种不同类型的应用程序而设计和优化的，所以它们的体系结构有显著的不同。这可以通过比较两种处理器技术的专用于高速缓存、控制逻辑和处理逻辑的相对模面积（晶体管数量）来看出（参见图19.2）。在CPU中，正如在第18章中所讨论的，控制逻辑和高速缓存存储器构成了CPU的大部分资源。这正符合需要尽可能快地处理顺序代码的体系结构的要求。另一方面，GPU使用大量并行的SIMD（单指令多数据）架构来执行主要数学操作。就本身而言，GPU不需要具备和CPU的控制逻辑一样复杂的能力（例如，无序执行、分支预测、数据冲突等）。它也不需要大量的高速缓冲存储器。GPU只是在大量数据上运行相同的代码线程并且能够通过管理比可用处理器核心



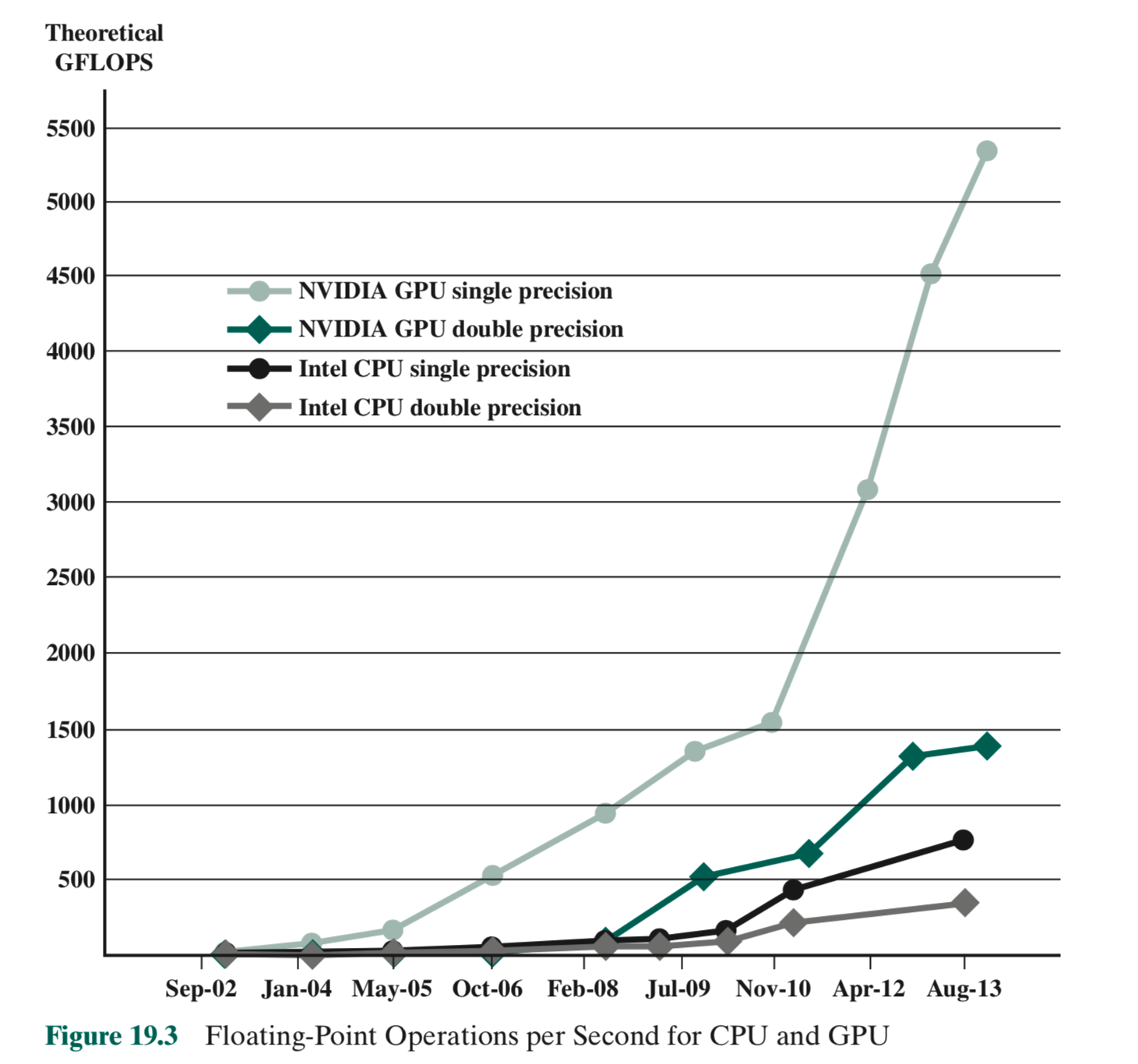
更多的线程的执行，来达到隐藏内存延迟的目的。

### 性能与性能比较

电子游戏市场已经刺激了对不断增长的实时图像显示的需求。这中需求可以由具有更强浮点运算能力的更大并行度的GPU处理器核心实现。因此，GPU被设计成可以最大限度地提高每秒中浮点运算数目（FLOPs）。此外，较新的NVIDIA体系结构，例如开普勒和麦克斯韦体系结构，已经致力于提高每瓦特比(FLOPs/瓦特)性能，与以前的GPU体系结构相比，其降低每个GPU处理器核心所需的功率。开普勒通过减少处理器内核的时钟，同时增加片上晶体管的数量（遵循摩尔定律），使得在费米结构上每瓦的性能净增益3倍。此外，Maxwell架构提高了执行效率。与多核CPU相比，GPU增加FLOP的速率是指数级的（参见图19.3[NVID14]），因此产生了很大的性能差距。两种不同的处理技术之间的每瓦性能差距也可以说明这一点。

## 19.3 GPU架构概述

GPU体系结构的发展历史可以分为三个主要阶段。首先是GPU体系结构发展的早期阶段（上世纪80年代早期到90年代后期），其中GPU由固定的、不可编程的、专门的处理模块（例如，顶点、光栅、着色器等）组成。此外，这一时期持续的技术进步，使得图形系统的尺寸和成本显著降低，继而在20世纪90年代中后期将图形处理器带入到PC。第二阶段则通过多次迭代修改，使得第一阶段的GPU体系结构，从固定的、专用的硬件流水线结构转变为完全可编程的处理器结构（大约在2000年初到2005年左右）。NVIDIA在2006年引入的综合的、最终修改的版本，促进了其新的GPGPU语言CUDA的使用。第三阶段从第二阶段未完成的部分开始发展，并解决GPU/GPGPU体系结构如何研制高效省钱的高并行度SIMD协处理器的问题，这种协处理器可以加速某些非图像相关程序的执行次数，该阶段同时解决如何让GPGPU语言（在这里我们指CUDA）映射到这种体系结构。本章将重点放在GPU发展的第三阶段上。



第一个带有GPGPU硬件支持的NVIDIA GPU是GeForce 8800 GTX。为了使GPU能够被程序员用到通用并行计算的应用程序，增加了真高速缓存层次结构和用户可寻址的共享内存。此外，可编程GPU处理器核阵列被平均地分为多个可拓展SM（流多处理器）。这种体系结构的好处是GPU处理器内核以及GPU的新版本或不同模型中的SM具有可拓展性，而不需要修改CUDA编程语言。

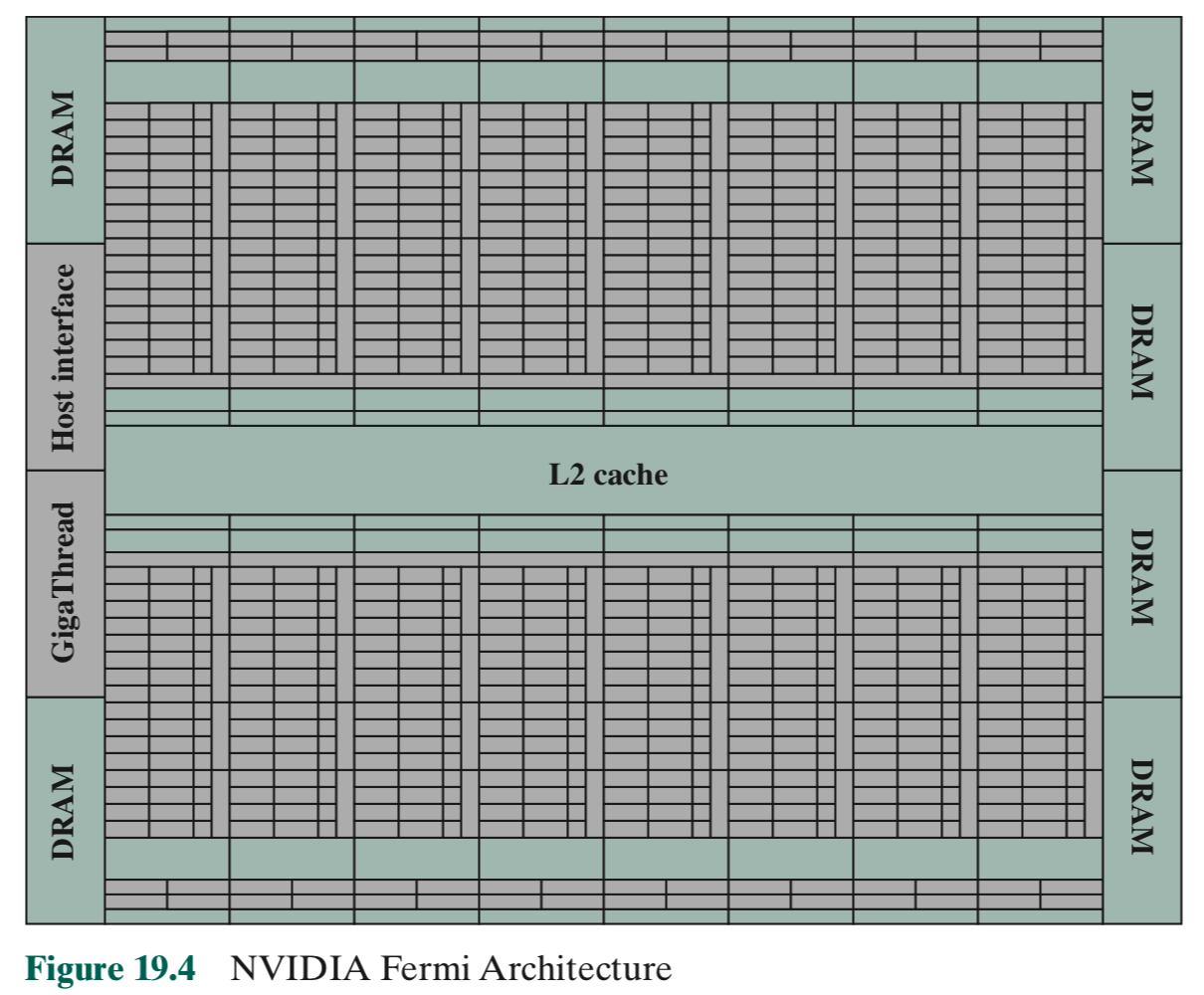
### 基线GPU体系结构

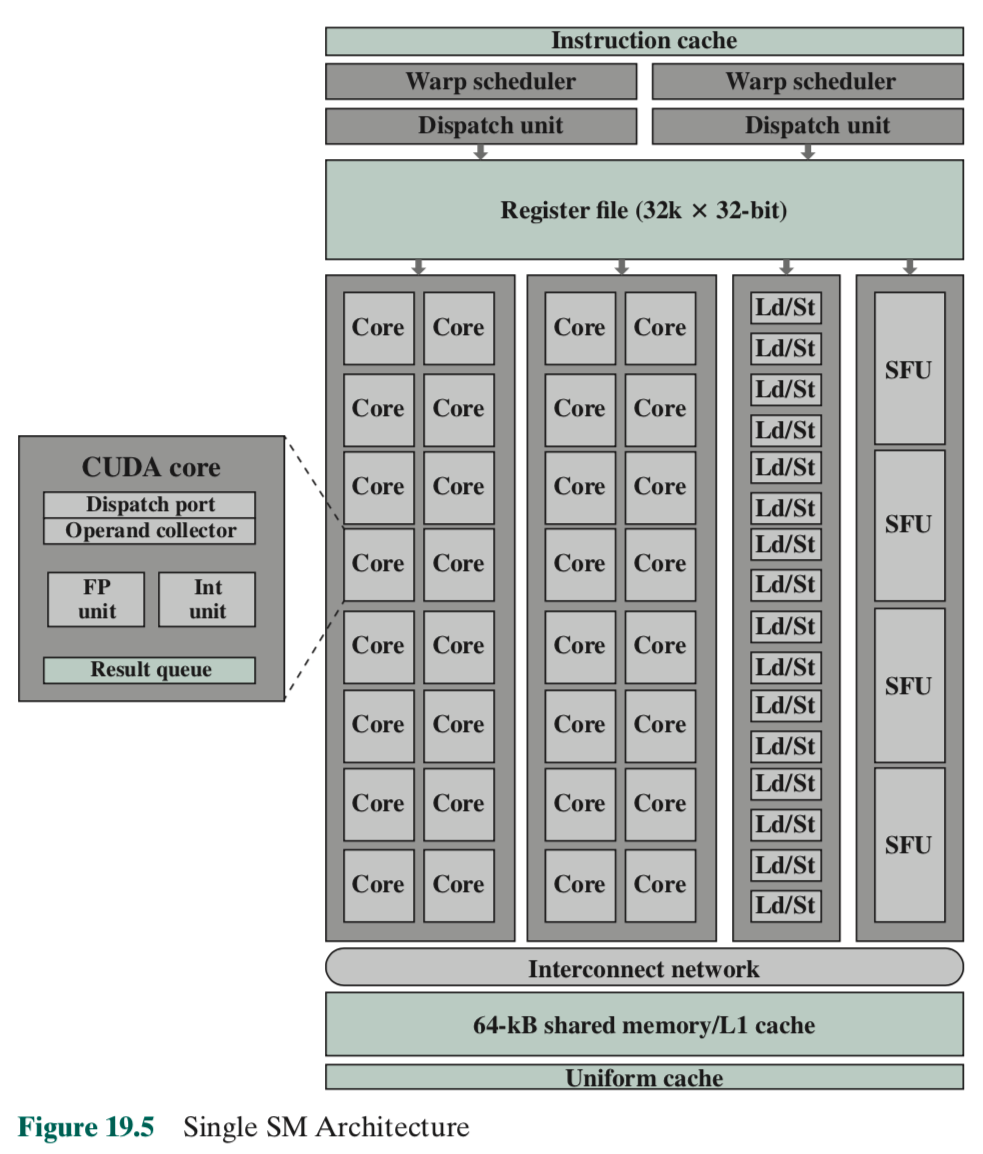
如前所述，NVIDIA已经通过GPU处理技术的几次迭代（即，Tesla、Fermi、Kepler和Maxwell）取得了进步，每一代GPU处理技术在其微结构上与前代相比，具有小或中等的差异。新一代GPU技术已经对SM的命名做了较小的修改，例如Kepler的SMX和Maxwell的SMM。这有助于表达出SM架构相对于前版本发生了相对显著的变化（这也有助于新产品的促销！）尽管如此，从CUDA编程的角度来看，所有的这些处理技术仍然具有相同的顶级架构。

  本章后续内容中，我们将使用NVIDIA的Fermi架构作为示例基线架构。之所以选择Fermi架构，是因为其相当有代表性的GPU架构以及较低的CUDA核心/SM数量，这简化了GPU硬件与CUDA软件之间的映射。这个示例体系结构由16个SM组成，其中每个SM包含一组32个CUDA核心。因此，费米GPU总共有16 SM ×32 CUDA核心/SM，即512个CUDA核心。

### 全芯片布局

图19.4显示了NVIDIA Fermi架构GPU的总体布局。如图所示，L2高速缓存位于16个SM（上面和下面的8个SM）的中心。每个SM由两个相邻的列、16行矩形(GPU处理器核心)、有16个load/store单元的一列和有4个特殊功能单元(SFU)的一列组成。对SM模块的更详细的说明如图19.5[NIVD09]所示。图19.4中SM的头部和底部的矩形是寄存器和L1/共享内存所在的位置。六个DRAM I/O接口中的每一个都具有64位的存储器接口（DRAM接口电路在最外层的左右两侧以深蓝色矩形的形式表示）。因此，总的来说，有一个384位的接口连接到GPU的GDDR5 （图像双数据率，一个专门为处





理图像而设计的DDR存储器）DRAM，其支持高达6GB的SM片外存储器（即，全局、常量、纹理和局部）。关于这些不同存储器类型的更多细节内容将在下一节讨论。另外，图19.4所示的是主机接口，它可以在GPU布局图的左侧找到。主机接口支持GPU和CPU之间的PCIe连接。最后，GigaThread全局调度器（橙色，位于主机接口旁边）则负责将线程块分发给所有SM的warp调度器（参见图19.5）。

### 流式多处理器体系结构细节

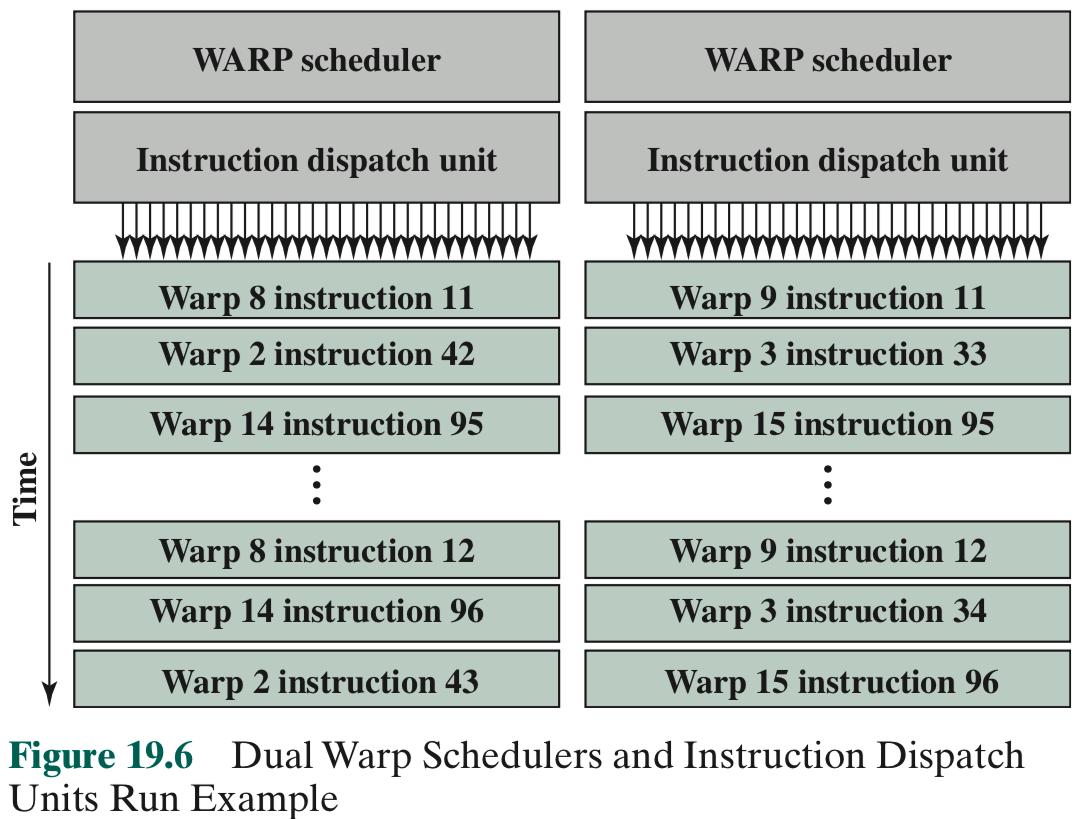
  图19.5的右侧将NVIDIA 费米架构分解为单个SM的基本组件。这些组件是：

* GPU处理器内核（总共32个CUDA内核）
* Warp调度器与调度端口
* 十六个load/store单元
* 四个SFU
* 32k×32位寄存器
* 共享内存和L1高速缓存（总共64 kB）

 双重WARP调度器 如前所述，GPU芯片上的GigaThread全局调度器单元将线程块分发给SM。然后双重warp调度器将其处理的每个线程块分解为**warp**，其中**warp**是一组32个线程，它们从相同的起始地址处开始，其线程ID是连续的。一旦一个warp被发出，每个线程将有它自己的指令地址计数器和寄存器集。从而支持SM中独立地分支和执行每个线程。

  要想让GPU利用最大化，需要其处理尽可能多的warp来保持CUDA核心最大化利用。如图19.6所示，当双重warp调度器和指令调度单元能够每两个时钟周期发出两个warp时（费米体系结构），就会达到最大的SM硬件利用率。正如下一步要解释的， SM不能达到最大处理速率的主要原因是结构冲突，而并不是片外存储器访问延迟。

16个CUDA核心（×2）、16个load/store单元和4个SFU（见图19.5）的每个分隔列都可以被分配一半warp（16个线程）来处理每个时钟周期中的两个warp调度器/调度单元中的每一个，因为组成列没有发生结构冲突。结构冲突的原因有SFU数量不够、双精度乘法和分支。然而，warp调度器有一个内置的记分板来跟踪可用于执行的warp以及结构冲突。这使得SM既能克服结构冲突，又能尽可能地帮助优化隐藏片外存储器的访问延迟。



因此，对于程序员来说，重要的是将线程块的大小设置为大于SM中CUDA核的总数，但小于每个块允许的最大线程数，并确保线程块大小（x和/或y维）是32（warp大小）的倍数，以实现SM最接近理想的利用率。

CUDA 核心 如CUDA基础部分所述，NVIDIA GPU处理器核心也称为CUDA核心（参见图19.5）。

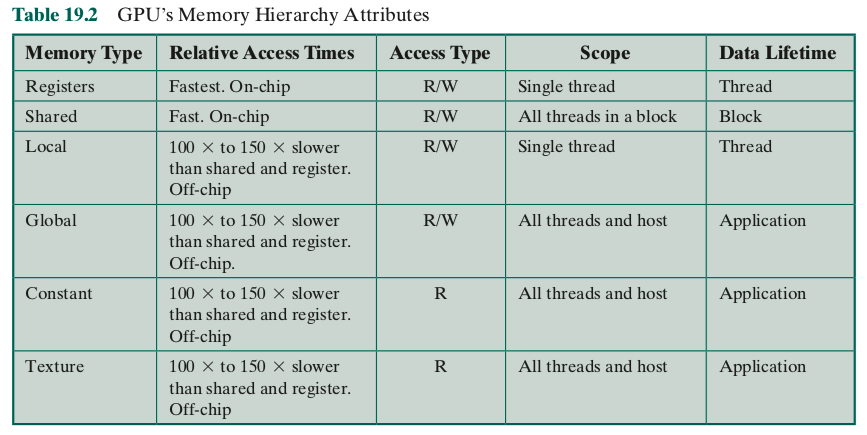
同样是在前面定义的，如图19.4所示，在Fermi架构中，总共有32个CUDA核心专用于每个SM。每个CUDA核心都有两个独立的流水线或数据路径：整数（INT）单元流水线和浮点（FP）单元流水线（参见图19.5）。这些数据路径中只有一个可以在单个时钟周期中使用。INT单元能够进行32位、64位和扩展精度的整数和逻辑/位操作。FP单元可以执行单精度FP操作，而双精度FP操作需要两个CUDA核。因此，只执行双精度FP操作的线程的运行时间将是单精度FP线程的两倍。在开普勒体系结构中，通过在每个SM中使用专用的双精度单元以及大部分单精度单元，可以解决双精度FP运算的性能影响。幸运的是，线程级FP的单精度、双精度操作对CUDA程序员来说是隐藏的。然而，程序员应该认识到，基于用到的GPU，使用两种精度类型之间可能产生的潜在性能影响。

  费米架构相比其之前的架构，对CUDA核心的FP单元进行了改进。它从IEEE 754-1985浮点算术标准升级到IEEE 754-2008标准。这是通过提高多路加法指令（MAD）与混合多路加法（FMA）指令的精度来实现的。FMA指令对于单精度和双精度算法都是有效的。在FMA指令的末尾，Fermi架构只执行单个轮询。这不仅提高了结果的精确度，而且还将执行FMA指令压缩为单处理器时钟周期。因此，32个单精度或16个双精度FMA操作可以在每个SM的单处理器时钟周期中发生。

特殊功能单元，每个SM有四个SFU。SFU在一个时钟周期内执行先验操作，如余弦、正弦、倒数和平方根。由于SM中只有4个SFU，warp中单个指令有32个并行线程，所以完成需要SFU的warp需要8个时钟周期。然而，CUDA处理器和load/store单元仍然可以同时使用。

load和store单元 SM有16个load和store单元，每一个单元会在每个时钟周期中为单个线程计算源地址和目的地址。这些地址将被用于那些线程希望写入数据或读取数据的高速缓存或DRAM。

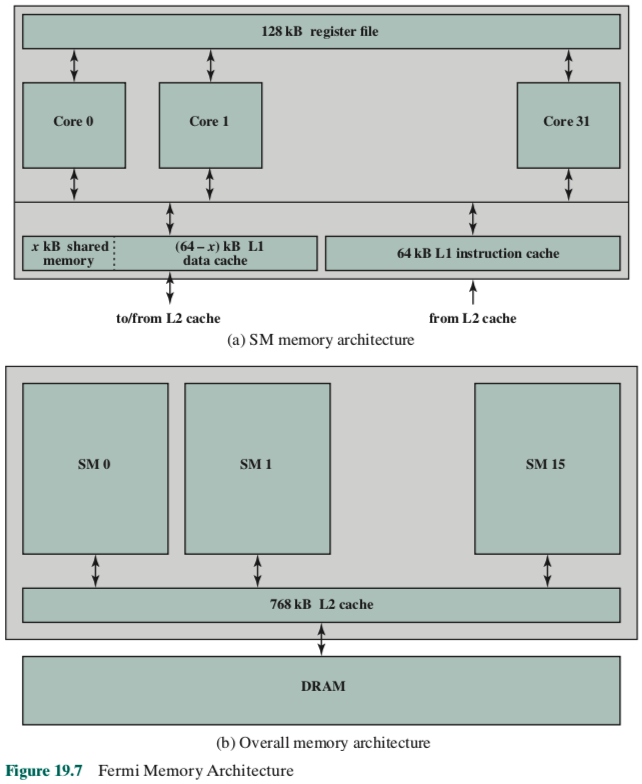
寄存器、共享内存和L1级高速缓存 如图19.5所示，每个SM都有自己的（片上）专用寄存器集和共享内存/L1高速缓存块。关于这些低延迟、芯片上的存储器的细节以及优点将在下面描述。



尽管Fermi架构中每个SM上有惊人的32k×32位寄存器，但是每个线程都具有由CUDA计算能力版本2.x定义并分配给它的最大可到达64×32位寄存器，该版本是每个SM所允许的最大活动warp数量以及每个SM的寄存器数量的函数。如表19.2所示，寄存器和共享内存具有最快的访问时间，只有几纳秒（ns）。如果存在任何临时寄存器溢出，数据在被发送到L2级高速缓存之前将首先被移动到L1级高速缓存，然后发送到具有长访问延迟的本地存储器（参见图19.7a）。使用L1高速缓存有助于防止发生数据读/写冒险。因此，分配给线程的寄存器中的数据的生命周期只有线程的生命周期那么长。

与当代多核微处理器（如CPU）相比，专用于SM的GPU处理器内核的可寻址、片上共享存储器具有独特的配置。这些当代的体系结构，正如第18章所述，图18.6所示，具有专用的片上L1级高速缓存和每个核上的一组寄存器。然而，它们通常没有片上可寻址的存储器。相反，专用的存储器管理硬件在没有程序员的控制的情况下，可以控制高速缓存和主存储器之间的数据移动。这与GPU体系结构相比有显著的不同（参见图19.5）。

正如本章开头所讨论的，共享内存被添加到GPU体系结构中，专用于帮助GPGPU应用程序的执行。通过消除对片外存储器的不必要的长延迟访问，可以优化共享存储器的使用，从而显著提高GPGPU应用程序的速度和性能。尽管共享内存对于每个SM来说体积很小（在最大配置时为48kB），但是它具有比全局内存要小的100×到150×的低访问延迟（参见表19.2）。因此，共享内存有三种加速并行任务处理的方法：（1）块的所有线程（例如，用于矩阵乘法的数据块）多次重复使用共享内存数据；（2）块的线程选择（基于特定ID）是为了将数据从全局存储器传输到共享存储



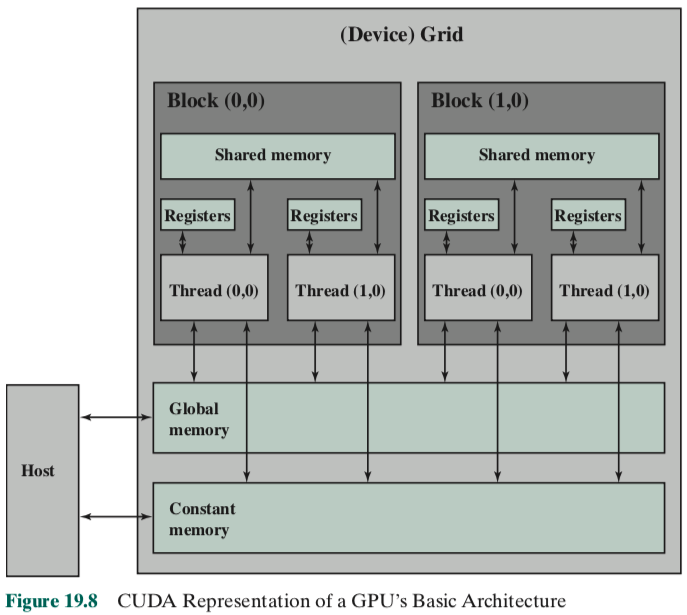
器，因此需要删除对相同内存位置的冗余读写；（3）如果可能，用户可以通过保证合并访问来优化对全局内存的数据访问。所有这些点也有利于减少芯片外存储器的带宽约束问题。SM共享内存中的数据的生命周期和执行中的线程块的生命周期一致。因此，一旦块上所有线程执行完成，SM共享存储器中的数据就不再有效。

  虽然共享内存的使用将给出最佳运行时间，但在某些应用程序中，在编程阶段的内存访问是未知的。这是有更多的可用L1级高速缓存（最大被设置为48kB）可以给出最佳结果的地方。此外，L1级高速缓存有助于帮助解决寄存器溢出问题，而不是直接进入本地（片外）DRAM存储器。两级高速缓存的层级结构——每个SM一个单个的L1级高速缓存，跨芯片的SM共享L2级高速缓存——提供了与传统多核微处理器相同的优点。

### 知晓并根据你的存储器类型来编程的重要性

对于程序员来说，理解各种GPU存储器的细微差别是很重要的，特别是每种存储器类型可用的大小、它们的相对访问时间和可访问性限制的差别，以便使用CUDA进行正确且有效的代码开发。从本章开头所介绍的CUDA Basics部分、刚刚介绍的SM层级存储器以及表19.2中列出的术语和参数中可以看出，GPGPU编程与针对CPU的编程相比需要一个非常不同的方法，其中，使用到的特定的数据存储硬件（文件I/O除外）对程序员是隐藏的。

  例如，使用GPU体系结构，分配给CUDA核的每个线程都有自己的寄存器集，使得一个线程不能访问另一个线程的寄存器，无论是否在同一个SM中。特定SM中的线程相互协作（通过数据共享）的唯一方式是通过共享内存（参见图19.8）。这通常需要程序员仅分配SM的某些线程，用以写入其共享内存的特定位置，从而防止写入冒险或周期浪费（例如，许多线程从全局内存中读取相同的



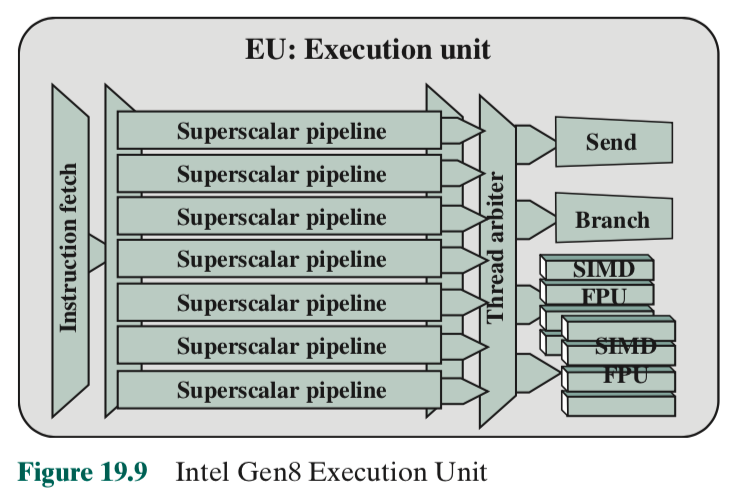
数据，并将其写入相同的共享内存地址中）。在允许特定SM的所有线程从刚刚写入的共享内存中读取之前，需要对该SM的所有线程进行同步以防止读后写（RAW）数据冲突1。

## 19.4英特尔GEN8 GPU

作为GPGPU架构的另一个示例，本节将概述Gen8处理器图像架构[INTE14，PEDD14]。

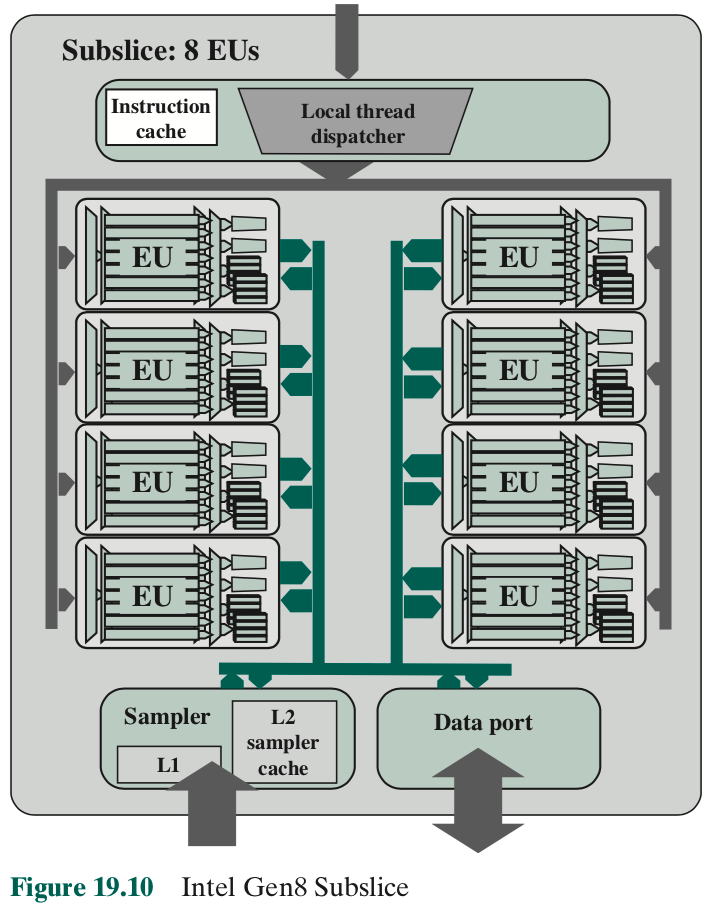
 Gen8体系结构的基本构建块是图19.9所示的执行单元（execution unit，简称EU）。EU是一个具有七个线程的同步多线程（simultaneous multithreading, 简称SMT）体系结构。回顾第17和18章，在SMT体系结构中，寄存器组被扩展，以便多个线程可以共享流水线资源的使用。EU有七个线程，并以超标量流水线架构实现。每个线程包括128个通用寄存器。在每个EU的内部，主要的计算单元是两个同时支持浮点和整数计算的SIMD浮点单元。每个SIMD FPU在每个周期内可以同时执行浮点数加法和乘法指令。还有专用于分支指令的分支单元和用于内存操作的发送单元。

每个寄存器存储32个字节，可作为32位数据元素的SIMD 8元素向量访问。因此，每个Gen8线程有 4 kB的通用寄存器文件（GRF），每个EU总共有28 kB的GRF。灵活的寻址模式允许寄存器一起寻址，来高效构建更宽广的寄存器，甚至可以组成跨域式矩形块数据结构2。每个线程的架构状态保存在一个单独的专用架构寄存器文件(ARF)中。



1请参阅第16章对RAW冒险的描述。

2 术语跨域式的（*strided*）指的是对地址的一系列存储器读写，每个操作与前面最后一个操作之间隔开一个叫做*stride length*的常量间隔。遍历数组时常常需要跨域式引用，并且（如果数据足够大，使得访问时间很长）可以通过反转双重循环或部分展开嵌套循环的外部循环来调整位置。

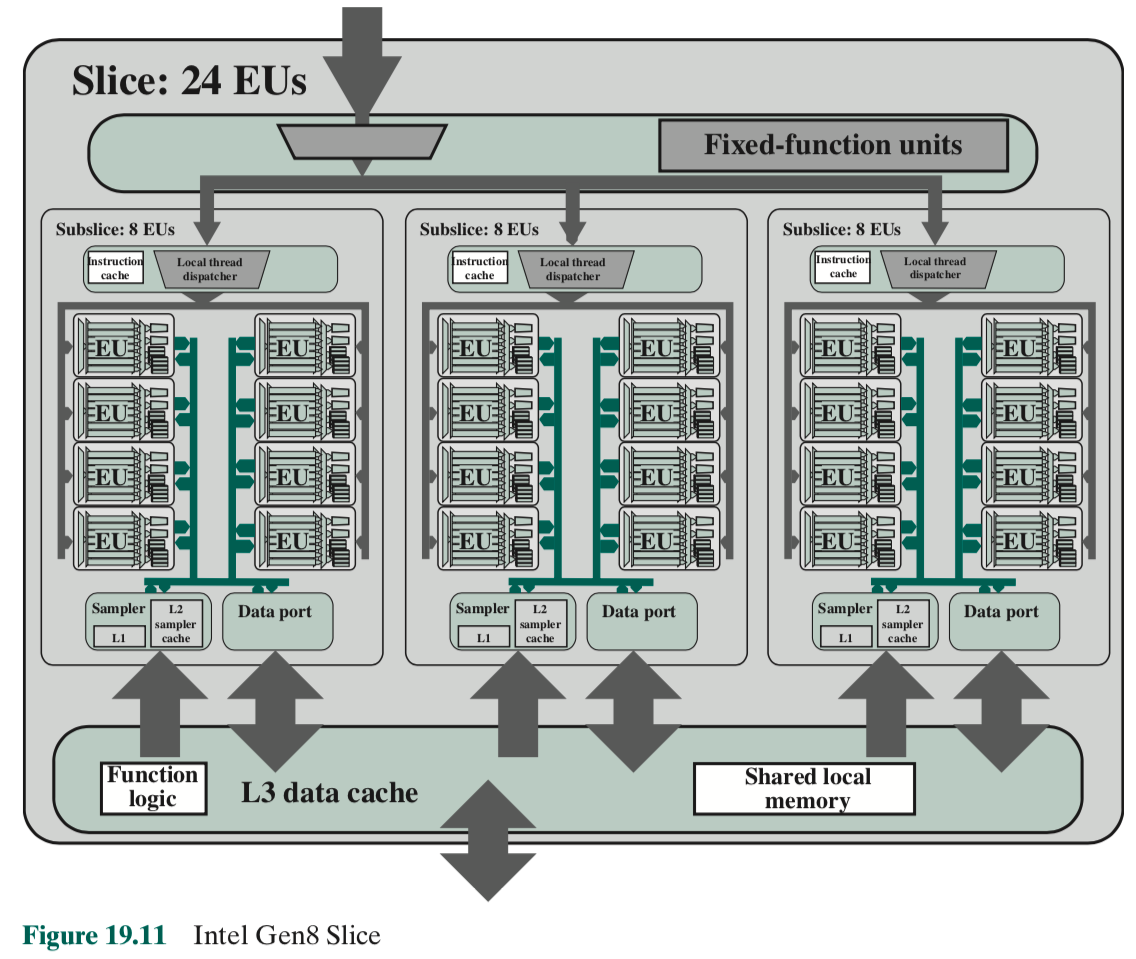


EU可以同时从不同的线程发布多达四个不同的指令。线程仲裁器将每个指令分派给四个功能单元中之一来执行。

  EU被组织成一个二次切片（如图19.10），其可以包含最多8个EU。每个二次切片包含它自己的本地线程调度器单元和它自己的支持指令高速缓存。因此，单个二次切片具有专用的硬件资源和总共56个同步线程的寄存器文件。

二次切片还包括一个称为采样器的单元，它具有自己的本地L1和L2级高速缓存。采样器用于纹理和图像表面的采样。采样器包括支持块压缩纹理格式的动态解压缩的逻辑。采样器还包括固定功能逻辑，可以完成图像（u，v）坐标和地址箝位模式（如镜像，包装，边界和钳位）的地址转换。采样器还支持多种采样滤波模式，如点、双线性、三线性和各向异性。数据端口提供有效的读/写操作，这些操作试图利用高速缓存行的大小来合并来自不同线程的读操作。

  为了创建产品变体，二次切片可以集群到称为切片的组中（图19.11）。目前，最多可以把三个二次切片组织成一个片，从而组织总共24个EU。除了二次切片以外，该切片还包括用于线程调度路由



的逻辑、其它用于优化图像数据处理的功能逻辑、一个共享的L3级高速缓存和一个较小的共享本地存储器结构。后者对于EU是可见的（可寻址存储器），并且可用于共享临时变量。

  为了提高性能，共享L3级数据高速缓存使用了一种称为**高速缓存库**的技术。为了实现高带宽，高速缓存被划分为大小相等的内存模块，被称为存储库，可以同时被访问。因此，由n个不同的存储器组中的n个地址构成的任何存储器读写请求可以同时得到服务，从而产生n倍于单个模块的带宽的总带宽。然而，如果内存请求的两个地址落在同一个内存库中，则会发生存储冲突，并且访问必须顺序执行。硬件将存储请求依照存储冲突按需要分割成独立无冲突的请求，通过一个等于独立内存请求数量的因子减少吞吐量。如果单独的内存请求的数量是n，则初始内存请求将导致n路存储冲突。因此，为了获得最大的性能，了解内存地址如何映射到内存库，以便调度内存请求，从而最小化存储冲突，这一点非常重要。

  最后，SoC产品架构师可以通过在一个SoC芯片上安放单个或多个切片来创建产品系列或系列中的特定产品。这些切片与其它前端逻辑结合起来管理命令的提交，同时管理支持3D渲染和媒体流水线的固定功能逻辑。此外，整个Gen8计算架构通过一个称为图形技术接口（GTI）的专用单元连接到SoC组件的其余部分。

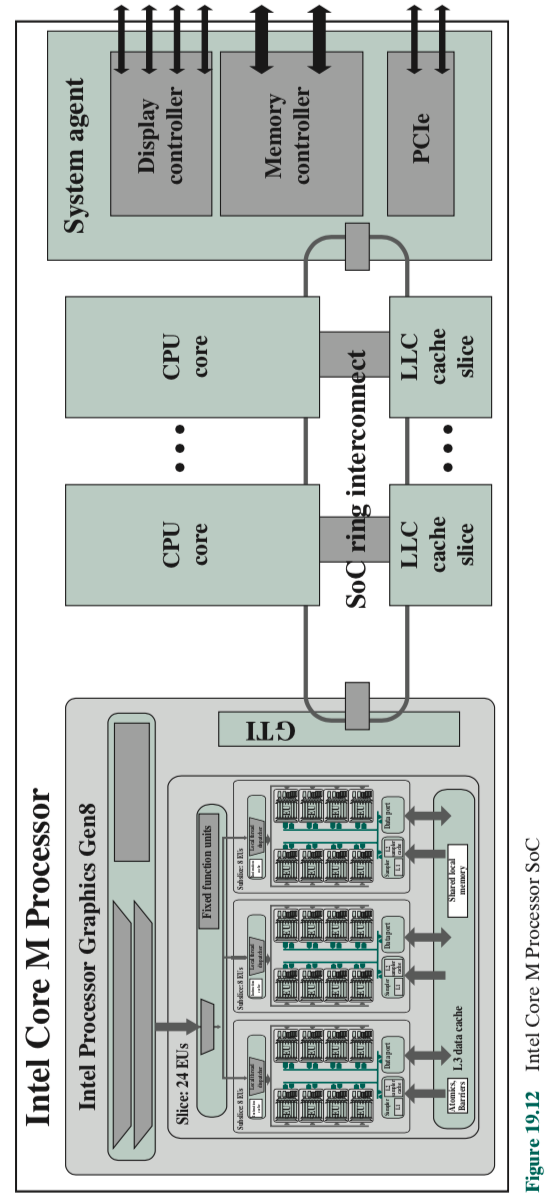
  这种SoC的一个例子是Intel HD Graphics 5300 Gen8的Intel Core M处理器（图19.12）。除了GPU部分，芯片还包含多个CPU核、一个LLC高速缓存以及一个系统代理。这种系统代理包括用于DRAM存储器、显示器和PCIe设备的控制器。Processor Graphics Gen8、CPU、LLC高速缓存和系统代理使用一个环形结构互连，正如Xeon处理器中所示（如图7.16）。

## 19.5何时使用GPU作为协处理器

在本章的最后，我们将从软件设计的角度简要讨论如何确定适合使用GPGPU的程序，同时介绍一些相关的软件工具来辅助这个过程。

  在GPU上（异构计算平台）运行部分代码对程序有什么好处？如本章所说明和讨论的，GPU由数百到数千个处理器核心组成，并且具有SIMD体系结构。因此，具有高度可并行化代码部分的程序（可复制到数千个轻量级线程中以同时处理大型数据集）是加速GPGPU系统上运行时间的最佳候选程序。这里，轻量级线程被定义为一个相对较小、大量可并行的代码片段的实例，该代码片段没有或几乎没有分支。通常，原始串行代码采用大循环迭代或几个嵌入式的循环的形式，代码对迭代之间没有数据依赖的方程（例如，矩阵运算）执行计算。此外，当最初使用类似于基于GNU命令行的gprof或NVIDIA中基于nvprof的可视化剖析器（每种剖析器都针对某种典型的代表性数据而运行）的工具剖析整个程序时，要并行化的部分必须占程序总运行时间相当的百分比。这个要求将最大限度地提高可以获得的加速（Amdahl定律），同时最小化CPU和GPU之间的数据传输时间对总体加速的影响。

  一旦识别出候选的大规模可并行代码段，则需要将其从串行代码转换为并行代码或CUDA内核。如果一个并行编译器能够自动进行这种转换，而不需要用户的输入，并且给出一个接近最优的、正确的解决方案，那么这将节省大量的时间、金钱和努力。不幸的是，这样的工具还不存在。我们还有两个选择：（1）通过复杂规划，使用CUDA、OpenCL或类似语言编程来转换代码；（2）使用编译器指令语言，如OpenACC、hiCUDA或类似语言。虽然使用编译器指令语言在编译器的代码中放置并行化的“提示”可以节省大量的编程时间，但它仍然是一个迭代过程，并且不能保证获得最佳的运行时间。然而，过去几年里，人们对此方法越来越感兴趣，CUDA编译器的新版本支持OpenACC语言。至此，一个精心规划/设计和编码的CUDA程序几乎总是可以收到最佳的运行时间。



## 19. 6关键术语和习题

**关键术语**

|  |  |  |
| --- | --- | --- |
| block 块 | general-purpose computing using a GPU (GPGPU) 通用计算GPU | kernel 内核 |
| cache banking 高速缓存库 | GPU processor core GPU处理器核心 | streaming multiprocessor（SMs） 流式多处理器 |
| central processing unit（CPU）中央处理单元 | graphic processing unit  (GPU) 图像处理单元 | thread 线程 |
| Compute Unified Device Architecture (CUDA) 计算统一设备架构 | grid 网格 | thread block 线程块 |
| CUDA core CUDA核心 |  | warp |
|  |  |  |

习题

19.1 CUDA程序的三个通用部分是什么？

19.2列出CPU和GPU体系结构之间的基本区别。

19.3内核、线程和块之间的区别是什么？

19.4定义高速缓存库。

19.5什么是特殊功能单元？