统一计算设备构架

抽象

在很多电子器件中图形处理器（GPU）是一块不可分割的计算机硬件。它曾经对于用户揭露图型接口并且几乎存在于每个工作站和移动设备。启动于上个时机九十年代中期，3D加速游戏日渐流行（例如地震和虚构系列）为了渲染越来越多的复杂场景导致模数GPU允许的计算能力快速增长。在上个十九年代，制造商扩充GPU的核心功能，高效渲染3D场景，额外支持几何运算的处理，值得注意的是，它开创了NVIDIA Geforce 256[5]及其专用的转换和照明装置。在新的千年里，随着像素语言和Vertex Shader语言的引入，这一过程继续进行，这使得在这一硬线渲染管道之前的操作成为可能。因此，计算机科学家利用GPUs不断增加的计算能力，通过用前面提到的Shader语言来表达它们来实现。这就是所谓的GPUs（GPGPU）通用计算的基础。GPGPU编程的一个主要缺点是缺乏抽象性：处理过的数据必须以纹理编码，并使用Shader语言的指令集进行操作，这极大地限制了复杂程序的设计更通用的算法。在2007年夏天，NVIDIA发布了计算统一设备架构（CUDA）[3]，允许使用C-或基于FORTRAN的复杂应用程序进行方便的编程。结果，一般的算法可以用易于阅读的代码来表达，同时与单线程的CPU实现相比，受益于最多两个数量级的执行时间更快。虽然在其他厂商（例如OpenCL [9]和开放式ACC [17]）的GPUs编程上有统一的方法，但CUDA现在是GPUs上最主要的并行化框架。本章将向您介绍CUDA-C++的基本知识，包括编程模型、底层内存层次结构的有效使用，以及与成千上万线程同步大规模并行任务的机制。

关键词

CUDA, 大规模并行计算，GPU，内核，线程块，设备同步，翘曲，推力。计算机到全球记忆访问，特斯拉，流式多处理器，SIMD，主组件分析。循环展开、合并内存访问、特征值分解、共享内存

GPGPU历史的详尽概述见[13]。  
法律辅助编程，DOI：10.1016/B978-0-12-849890-3.00007-1  
版权所有2018年版权所有

**目录**

**7.1**介绍CUDA(你好world)……………………………………………………226

**7.2**启用cuda的GPU的硬件架构…………………………………………228  
主机与设备之间的互连……………………………………………………228  
视频内存和峰值宽度…………………………………………………………229  
计算资源的组织………………………………………………………………229  
将线程块映射到SMS……………………………………………………230  
将线程映射为翘曲…………………………………………………………233

**7.3**内存访问模式(Egenface)……………………………………………………234  
对普通明星脸的计算…………………………………………………………234  
计算居中的数据矩阵…………………………………………………………241  
协方差矩阵的计算………………………………………………………………244  
特征面的计算……………………………………………………………………252

**7.4**内存层次结构(动态时间调整)………………………………………………256  
介绍………………………………………………………………………………256  
序列dtw的线性记忆算法………………………………………………261  
线性存储器dtw的一个天真CUDA端口………………………………267  
共享记忆中的波前放松……………………………………………………271  
同时调度和银行冲突…………………………………………………………277  
纹理和恒定…………………………………………………………………………278

**7.5**优化准则………………………………………………………………………………281

**7.6**补充练习…………………………………………………………………………282

参考……………………………………………………………………284

7.1**介绍CUDA(你好world)**

在本章中，您将学习如何在CUDA编程模型的帮助下在NVIDIA GPU上编写大量并行程序。由于CUDA基本上是对C和C++的延伸，我们可以很容易地移植出令人尴尬的并行程序，而不需要花费学习一种全新语言的开销。让我们从一个简单的Hello world程序开始，它向命令行打印一个问候语列表7.1中的源代码被分成两部分。它由CPU上执行的一个主要函数和将运行的内核**GPU。**

清单7.1：CUDA Hello世界

让我们看看主要功能。第一个命令cudaset设备通常会选择使用的具有cuda功能的GPU，您的工作站上只会连接一个GPU，因此标识符可以被适当地设置为0 。我们将讨论在本章后面同时使用几个GPU的可能性。第二个调用helo-kineWirl()将调用在GPU上并行运行的四个CUDA线程的一个块。每个线程应该打印一个问候语。目前，我们忽略了内核的实现细节，继续进行cudadeves异步调用。GPU（设备）上的内核和CPU（主机）上的代码是异步执行的。最后的同步迫使主机等待设备上的内核完成计算。这可以防止立即终止全程序。否则，当不使用显式或隐式同步机制时，主机侧主支持将立即返回，而不打印所需的消息。

在hello-内核的签名中，一个明显的新奇之处是返回值空值前面的全局限定符，该限定符指示此方法可从主机调用，但在设备上执行。CUDA编程语言提供了进一步的限定符，可以用来指定执行的范围和刻录行为，最重要的有以下几种：  
.--全局--：可调用的窗体主机，在设备上执行，  
.--主机--：可调用的窗体主机，在主机上执行，  
.--设备--：可调用的窗体设备，在设备上执行。

注意--主机--是隐含地针对每一个主机功能，因此可以添加在CPU执行的每一个传统功能的前面。限定符--主机和设备--可以合并，以迫使编译器为CPU和GPU生成二进制代码。此外，可以使用“非线”和“强制线”限定符来控制内衬。更多的细节可以在CUDA编程指南中找到。

正如前面所描述的，我们在GPU上使用一个CUDA线程块（由四个CUDA线程组成）来调用内核。关于块和线程的详细说明在下一节中讨论。到目前为止，我们只需要知道我们没有生成带有块标识符的块Idx.x=０由块组成Dim.x=4线, 每个都用本地线程标识符枚举Idx.x 范围从0到3（包容的）。结果，第6行中的表达式计算结果为=0\*4+threadIdx.x=threadIdx.x. 或者，当生成每个执行两个线程的两个块或每个执行一个线程的四个块时，我们可以生成相同的索引。正如我们将在第7.4节后面看到的，线程的特定分布会对我们程序的整体性能产生巨大的影响。最后，全局线程标识符通过Printf命令打印。示例代码可以通过调用nvcc hello-world.cu -02 -o hello-world

并像任何其他用主机编译器建立的二进制一样执行，获得以下输出：

hello from thread 0!

hello from thread 1!

hello from thread 2!

hello from thread 3!

注意，优化标志-02只影响代码的主机部分，不影响设备功能的性能。让我们简单地讨论一下终端的输出。您可能想知道为什么这些消息是为了与典型的双线程应用程序形成对比而打印的。这是因为打印语句总是在一个块中以32个连续线程的批量进行序列化，我们将观察到由32个连续线程标识符组成的洗牌批。

7.2**启用cuda的GPU的硬件架构**

顺便提一下，Hello world示例不是很有用，因为打印是序列化的，我们没有数据要处理。但在我们开始计算数字之前，我们必须了解典型的具有cuda功能的GPU的硬件布局，以及CUDA编程模型是如何映射到其各个组件上的。虽然这一节有点理论性，但我们将通过擦除硬件特定的细节来受益于这个知识，这些细节不是以后CUDA编程语言的一部分。用让-雅克·鲁塞安的话来重新措辞：忍耐是苦是好，但它的果实是甜的。

**主机与设备之间的互连**

让我们先从显而易见的具有cuda功能的GPU开始，它通过主板上的专用插槽连接到服务器或工作站。在编写本报告时，大多数显卡都是通过pclev3总线连接的，该总线提供多达16GB/s的带宽（见图7.1）。首先，这听起来相当快，但你很快就会发现这是CUDA应用的主要瓶颈。针对这一点，NVIDIA引入了NVIDIA Link，这是一种提供高达80gb/s峰值带宽的新型总线[6]，允许主机与设备之间或多位设备之间的高效通信与同一计算节点。

主机（RAM）的内存和设备（VRAM）的视频内存是物理分离的，作为一个单元，我们必须独立地在两个平台上分配内存，然后管理它们之间的内存传输。这意味着要在GPU上处理的数据必须明确地从主机转移到设备上。必须将驻留在GPU VRAM中的计算结果复制回主机，以便将其写入磁盘。这意味着不能从设备和副服务器直接访问主机上分配的数据。相应的cudacommands将在第7.3节详细讨论.

注意，Nvidia提供了与CUDA捆绑在一起的强大的库，例如：推力[7]，其特点是可以从主机维护的设备向量。然而，您应该意识到，抽象层可能会模糊代码的次优部分。例如，在主机侧循环中更改推力设备矢量的所有条目会导致对微小内存传输的过度垃圾处理.

**数字7.1**

处理器（CPU和GPU）和主机和设备内存组件的架构概述。内存传输通过pcle v3总线执行，提供高达16gb/s的带宽。从概念上讲，主机和设备可以被视为独立的计算平台，通过互连网络进行通信

对于NVIDIA的统一记忆层，我们也可以做出类似的陈述，尽管不那么苛刻，它把RAM和VRAM的地址空间作为一个整体。因此，我们将对两个内存空间进行传统的区分。这种透明的方法将帮助您确定应用程序的性能瓶颈，而不必猜测统一内存寻址的内部细节。

**视频内存和峰值宽度**

从概念上讲，图形卡可以被视为一个独立的计算平台，拥有自己的计算单元和Memry。与一般主板上附加的内存（在编写DDR4 DRAM的时候）相比，gaphics卡上的视频记忆要快得多。例如，基于Pascal的Titan X具有12GB的GDDR 5X DRAM模块，提供最多480gb/swebadp，相比之下，当前Xeon工件的mempleg带宽小于100 Gb/s。来自专业特斯拉系列如特斯拉P100或特斯拉V100的加速器卡，其特点是更快的hbm2堆叠记忆体，分别拥有高达720gb/s或900gb/s的频宽。

尽管表面上我们可以用几乎一个Tb/s来访问全局内存，但我们仍然需要关注底层内存的层次结构。例如，假设您希望增加存在于VRAM（全局mempry）中的单精度浮点值（FP32）。首先，每个32位值都被加载到寄存器中。其次，寄存器由一个用于单个浮点运算（Flop）的寄存器递增。最后，增量阀被写入全局内存。图7.2可视化描述的模式。这对应于计算机对全球内存访问（CGMA）比率为1flop/8B（读取为4字节，增量后写入为4字节），导致内存宽度为1Tb/s的整体计算性能为125GFlop/s。这仅仅是Tesla P100卡的11TFlop/s FP32峰值性能的1%，因此适当的内存访问模式和快速缓存的有效使用将是我们主要关注的问题之一。

**计算资源的组织**

现代GPU的巨大计算能力可以用处理单元的数量来解释，这些处理单元的数量远远超过几千个核心。相较于多核心CPU有几个以复杂指令集和控制流为特征的几十种单片处理单元，GPU可以被视为指令集有限、控制流受限的巨大的轻量级处理单元阵列。现代GPU的硬件布局是在分层树中组织的。

**数字7.2**

存储在主机上的数组a=(a0,a1,……)的内存传输和控制流程。首先，主机数组a被复制到相同大小的设备数组a。  
第二，线程块的每个经线同时增加数组的32个连续项。第三，驻留在设备数组a中的增量值被复制回主机数组A。

**数字7.3**

在Tesla P40卡片中构建的GP 102 GPU的架构布局，GPU被划分为六个图形处理集群（gpcs），每个集群提供十个流式多处理器（SMS）。60个SMs中的每一个共享相同的L2缓存和全局内存。消费者GFUs，如基于Pascal的GeForce GTX 1080或Titan X，其相似布局的GPCs或SMs数量较少（见表7.1）.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 表7.1高端单GPU视频卡跨代技术规范。双GPU像泰坦Z，特斯拉180，或者Teslam60被省略了，因为它们基本上都是他们单台GPU的堆叠版本。来自kepier和Maxwell一代的Tesla卡配备了较慢的纠错（ECC）RAM。基于Pascal的Tesla P100和基于Volta的Tesla V100提供了明显更快的hbm2堆叠记忆体。注意，核心和VRAM大小的数量随着时间的推移而略有增加。 | | | | | |
| 视频卡 | 一代人 | GPCs | SMs | FP32/GPU | VRAM@宽带 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 表7.2 GPU架构的技术规格横跨四代，每GPC的SMS数量增加，而随时间的推移每SM dcrecase的核心数量增加。因此，一个pascal/volta SM上的核心可以使用比上一代的SMS上更多的寄存器。 | | | | | | |
| 一代人 | | SM/GPC | FP32/SM | FP64/SM | FP64/SM | 注册/SM |

图7.3.GPCs是一个硬件专用的细节，在CUDA编程模型中没有对应关系。然而，第二级的SMS松散地映射到前面提到的CUDA线程块上。一个或多个线程块可以同时在一个SM上执行。具体数字取决于所需和可用硬件的比例。  
请注意，运行时环境没有提供任何关于特定执行顺序的信息，也就是说，我们既不能影响也不能推导块碎片的规划方案。我们程序中在不同块上执行的部分应该是真正独立于数据的。

在CUDA编程模型中，块标识符可以使用1D,2D和3D网格定义。这使得在访问多维域的数据时可以方便地进行索引。  
例如，在处理图像的不同部分时，你可以为每个图指定一个带有坐标的块（sigrickyx.x，sigridkex.y）。网格尺寸可以定义为1D网格的插值器，也可以定义为一般情况下的dim 3。可以通过griDim.x、gridDim.y和gridDim.z在内核中访问这些内容。我们前面的Hello world示例使用了一个包含一个、两个或四个块的1D网格。我们演示了在列表7.2中调用一个由3D块组成的3D网格。

|  |
| --- |
|  |

列出7.2：由三维块组成的三维网格的内核调用。

在我们着手处理一个SM的内部组件之前，我们必须做一个重要的评论：同一个内核的不同CUDA线程块在执行一个内核的过程中不能同步，因为在CUDA版本8之前不存在可以从内核内部调用。  
注意CUDA 9引入了提供设备侧多网格同步调用的合作组的概念。cuda9在编写本书时并没有发布，因此我们坚持传统的方法：必须通过在主机上堆叠几个内核来实现块间的依赖关系，在单个内核调用之间执行一个障碍。  
NVIDIA的这一架构决定有几个方面。首先，低端和高端GPU的硬件布局主要取决于提供的sm数。因此，一个复杂的编程模型应该在任意数量的块上透明地扩展。同步是非常昂贵的，特别是在强制执行全局障碍时，所有的SMS都必须等待最后一个块完成。这一观察结果可以概括为同步并没有扩展，因此，我们不得不重构甚至重新设计过度依赖于全局障碍或变异的代码。

串流多处理器

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**数字7.4**

Pascal生成中使用的流式多处理器（SM）的模式布局。一个SM提供了由32个单精度计算单元（FP32）、16个双精度计算单元（FP64）、8个负载和存储单元（LDST）各组成的两个块，以及8个特殊功能单位（SFU）。这两个块中的每一个可以访问32,768 32位寄存器，其整体大小为256KB，用于存储变量。

**将线程映射为翘曲**

每个SM由多个计算和指令单元组成，如图7.4所示。Pascal一代具有64个单精度（FP32），32个双精度（FP64）计算单元，16个负载和存储单元（LDST），16个特殊函数单元（SFU），每个SM。因此，FP 64的性能不能直接从FP 32单位的数量中推断，例如，梅威尔一代提供了FP 64与FP 32的比例仅为1/32，与Kepler（1/3）、Pascal（1/2）、和Volta（1/2）世代。

直至CUDA 8，一个由32个连续的CUDA线程组成的经线以锁步方式同时在SM上处理，因此所有32个计算单元都必须同时执行与单指令多数据（SIMD）相似的操作。计算单元可以访问非连续存储器或允许控制流分支的马克指令。后者称为分支发散，应避免，因为硬件将分支逐一执行。这个稍微更灵活的互斥模型被NVIDIA创造为单指令多分线程（simt）。严格地说，我们可以说SIMD是simt模型的子集。在这种情况下，一个经线可以被看作是SIMD向量单位，有32个SIMD车道。因此，我们可以很容易地用CUDA编程语言来表达SIMD算法。注意CUDA 9将传统的以wap为中心的编程范式转移到合作组中，这是经纱概念的推广。CUDA 9中的曲线可能不再以锁定步骤执行，这对隐式同步有严重的影响。  
在撰写本报告时（2017年夏季），我们无法预测合作团队对未来代码开发的影响。然而，要意识到这一主要的范式转变，因为它可能是游戏的改变者。

一个Pascal SM的两个寄存器文件可以处理多达65,536 32位元的变量。每个计算单元1024寄存器的数量相当高，对于高效调度轻量级线程至关重要。如果一个曲速耗尽，例如，当等待数据时，调度器可以切换到另一个曲速，而不需要倾倒或加载相应的寄存器。因此，维护的线程数量可能很容易超过可用的计算单元的数量。快速切换（曲速流）可以有效地隐藏计算背后全局内存的延迟。我们无法控制曲柄的执行顺序。不过，可以依靠两个属性高达CUDA版本8。首先，曲速内的所有线程都是同时执行的。其次，所有的翘曲和所有的块都在内核终止后完成了计算。

一个SM的单元可以访问多达64KB的芯片内存，用于线程之间的通信和缓存。最后，一个现代的支持cuda的GPU由1,000个核心组成（见表7.1），可以执行数万个线程。因此，与多核体系结构中粗粒度并行化方案相比，并行化必须在一个细粒度的尺度上组织。除了大规模并行计算方面，为了有效地利用GPU，我们还必须考虑simt计算模型施加的额外限制。

7.3**内存访问模式(Egenface)**

在重新定义了一个典型的具有cuda功能的GPU的硬件架构之后，我们无法开始编写一些更有用的代码。我们的任务是实现主成分分析（PCA）的CUDA，PCA的应用是多方面的，从物理中刚体内在坐标系计算过程中的信号处理中的有损音频/视频压缩，到数据挖掘中潜在变量的确定。处理由202,599个对齐名人脸组成的名人数据集[16]（见图7.5）。特别是，我们得到了大量的量，如平均名人面，中心数据矩阵，对应的协方差矩阵，以及可以用于有损图像压缩的特征向量基。

从编程的角度，您将学习如何在主机和设备之间传输数据，如何使用合并的访问模式正确定位全局内存，以及如何手动缓存共享内存中的冗余数据。您将能够编写基本的CUDA程序，同时处理千兆字节的数据。此外，我们还将讨论什么时候你的程序的CUDA并行化在运行时是有益的，但是在多线程CPU实现可能是更好的选择的情况下也是有益的。

**对普通明星脸的计算**

希瓦数据集的202,599张图像以形状178×218的rgb-值矩阵存储。为了简单起见，我们通过计算红色、绿色和蓝色强度的像素平均值，将三个颜色通道折叠成灰度。此外，在预处理阶段，这些图像的采样率已经降低了大约4倍，产生的图像仍然需要超过1千兆字节的内存，因为（45×55）×大小（浮动）×202,599>1.8 gb.然而，我们讨论的算法和源代码在原始分辨率上也做了非常好的工作（例如在使用带有32GB VRAM的GPU时）。此外，通过独立地处理每个颜色通道中的像素，对图像颜色的扩展是直接的。

**数字7.5**

有八张与名人网数据相似的照片。请注意，由于版权不明确，原始照片没有被描绘出来。在我们的实验中，我们使用了一个变型的数据集，在这个变型的数据集中，人脸特征是对齐的，而RGB通道通过普通平均法折叠成一个单一的灰度通道。

在下面，我们将每个图像解释为一个扁平的向量v(I)=Rn，其中n=45.55=2475像素按行主序索引。简单地说，我们忘记了矩阵的形状，并连续地将每个像素映射到向量的一个槽上。M=202,599向量V(I)存储在数据矩阵D=(V(0),V(1)的行中，……V(M-1)使得dij=vj表示i-th向量(图像)的j-th维度(像素)。然后，我们通过计算所有图像的归一化像素强度值和来确定平均图像U：

得到的平均向量U如图7.6所示。注意，这种一维的更高维度数据的表示推广到了Rn.中由M固定长度向量组成的任何数据集，然而，[参考译文]为了获得有意义的结果，向量V的位置应该是松散关联的。因此，如果不使用面部特征的近似对齐或者使用平移不变的表示，这种方法就不可能在不对齐的数据上运行良好]或图像注册技术[10]。

**数字7.6**

原来分辨率（左面板：178×218像素）中的名人数据集的平均名人面及其下位变体（右面板：45×55像素）。

让我们开始编码。我们包括一些有用的页眉，与本书一起分发（见第7.3列表）。hpc助手。用于测量执行时间（timerstart（标签）和用于报告错误（cuerr）的hpp文件方便宏。错误检查很重要，因为当主机代码继续工作时，CUDA在错误的内存访问或不成功的内存分配后往往会无声地失败。因此，您将观察到我们代码中cuerr宏的频繁使用。  
其中两个头提供了加载二进制文件和以微软位图格式写入图像的功能。我们定义了一个用于平均计算的模板内核，它可以专门针对索引的自定义数据类型（通常为uint32-t oruint 64-t）和浮点数值的表示（通常为浮动或双浮）。设备指针数据和均值用于对数据矩阵和均值向量进行定位.整数的数字条目和数字特征对应于图像的数量（M）和特征/维度的数量（N）。

|  |
| --- |
|  |

清单7.3：平均计算程序的标头

让我们继续执行清单7.4的主要职能。形式行15到19，我们选择CUDA设备，随后定义三个常数，指定图像的数量（imgs）及其对应的形状（行和cols）。接下来，我们在CPU和GPU上为数据矩阵和平均向量分配内存，使用专用命令cudamallocost和cudamalloc，这两个函数都以分配后存储内存位置的指针的地址作为第一个参数。因此，该参数是一个指针的指针，以允许cudamchost和cudamalloc将地址从nullptr更改为相应的值。返回值表示分配是否成功，并将由cuerr宏处理。虽然主机侧内存也可以通过其他命令来分配，例如，通过呼叫malloc或新的，但是当我们明确地将内存保留在特定设备上时，我们是有限的。请注意，在整个Chaper中，我们使用大写变量名来表示devuce-side指针，并为主机使用小写字母。另一个流行的惯例，虽然在本书中没有使用，但在变量名后面附加-H（主机）和后缀，以在视觉上区分地址空间。

|  |
| --- |
|  |

清单7.4：主要功能：内存分配

我们从提供的二进制-i 0开始使用负载二进制()方法将矩阵D从磁盘加载到主机上的数据数组。hpp预报员文件。  
提默斯特宏和提默斯特宏决定执行时间。此时，在处理自己的数据时，将使用行主序寻址手动填充数组。

|  |
| --- |
|  |

清单7.5：主要功能：从磁盘加载数据。

如前所述，处理GPU上的数据必须明确地从主机转移到设备上。复制可以通过调用cudamamby实现，如第7.6行所示。该命令遵循传统memcpy调用的语义，即第一个参数对应于目标sddress，第二个对应于源指针。第三个参数表示传输的字节数，第四个参数指定所涉及的平台。这些相当不方便的常数是用来区分从主机复制到设备到主机的。请注意，通过定义自定义变量或宏，可以大大缩短这些表达式，以减少poilerpiate的冗余类型。例如，hpc助手。hpp头文件包含两行文件夹。

# 定义H2D（计算机设备）

#定义 D2H (cudamemcpydevicetohost)

在第41行中对cudamemset的调用用零覆盖了设备矢量的意思。这是一种安全机制，可以避免我们在编程实践中经常看到的以下错误。此外，假设您在下一个版本中引入了一个错误，它会立即导致内核无声地失败。很有可能（几乎可以保证），结果向量（在我们的例子中是平均值）从设备到主机的随后的cudamempy将把以前运行中的旧数据（正确的）转移到仍然存在于全局内存中。通过单机引导的程序？完成、重置结果和错误检查是强制性的。

在将矩阵D复制到设备后，我们在第46行调用内核。指的是情商。(7.1)，在J(yhe像素指数)上并行是可取的，因为每个N=55.45=2475sum都可以独立地被评估。当使用32个CUDA线程每个线程块时，我们必须调用至少[n/32]多个块。  
如果剩余的n2与零不同，也就是说，N不是块大小的倍数，我们必须生成一个额外的块来处理剩下的几个像素。

在hpc-help中定义了sdiv宏。hpp头文件。在内核终止后，使用cudamemcpy将结果复制回主机（见第52行）。

|  |
| --- |
|  |

清单7.7：主要功能：内存传输和内核调用。

列表7.7中的主要函数的剩余部分是直方向的，平均图像是用位图-I0的哑位图写入磁盘的。此外，分配的内存被明确地释放为主机，并为设备释放为防止内存被明确地释放为主机和设备释放为防止内存泄漏。

|  |
| --- |
|  |

清单7.7：主要功能：写结果和释放记忆。

最后，我们讨论了在清单 7.8中的计算-意义-内核实现。首先，全局头标识符在第76行从介绍部分模拟地计算到Hello world内核。其次，线程标识符的范围在第79行中检查，以避免当数字特性不是块的多派x。每个线程在一个专门的寄存器累积中总结数据矩阵的一列。或者，您可以在第86行中对它进行微调，给编译器一个提示，让它按32的大小分批展开循环。最后,对计算的和进行了归一化,并给出了平均数组.

|  |
| --- |
|  |

清单7.8：CUDA内核实现了平均面计算。

代码可以用以下命令编译：

nvcc -02 -std=c++11 -arch=sm\_61 -o mean\_computation

我们需要设置-STD=C++11标志来激活对C++11标准的支持，因为我们的代码使用的是自动关键字。可配置的计算能力。当在基于Pascal-based的TitanX上执行程序时，我们观察到执行时间的如下输出：

TIMING:13294.3 ms(从磁盘读取数据)

TIMING:170.136 ms(资料H2D)

TIMING:8.82074 ms （计算的意思是内核）

TIMING:0.03174 ms（意思D2H）

TIMING:155.921 ms(将平均影像写入磁碟)

从旋转盘（~150mbs/s）读取1.86GB的图像数据大约需要13秒。通过pcle总线从主机到设备的内存传输大约用170ms（~11gb/s）完成。在CUDA内核的执行过程中，我们只访问数据集的每个像素一次，然后应用一个补充，因此9MS对应于大约208GB/s的有效全局内存带宽和大约52gflop/s的计算性能。将平均影像(~9.5KB)传回主机的时间可以忽略不计。

尽管内核的性能令人印象深刻，但我们仅利用了全球内存480GB/s理论峰值带宽的一半和11tflops/s峰值性能的0.5%。N=2475衍生线程的数量明显小于3584可用核心的数量（见表7.1）。一个更好的方法将会产生数以万计的线程。如果我们在PCIe上包含相对较慢的内存传输或从磁盘读取数据所需的时间，则情况会变得更糟。因此，所描述的算法被认为是独立的应用程序，不太适合GPU，因为52gflopp/s的计算性能很容易被最先进的多核心计算机所击败。该内核可以作为高级算法中的子例程，存储了几个CUDA内核。结论是，当处理的数据太少时，性能是没有意义的。

最后，让我们做一个关于同步的重要声明。你可能已经注意到缺少的cudadevision（）命令在46行中调用内核后同步。下面使用cudamemcpy的内存传输隐式地同步设备，使显式同步成为冗余。然而，异步内存传输和发射是可能的，将在第8.2节讨论。在使用cudamalloc（主机）/cudafree（主机）分配/释放内存时，这种隐式同步行为也会被强制执行，用cudamemset或者一个开关或者一个在l1/共享内存配置之间的开关使用cudadevesetcheconfig.一个完整的列表可以在CUDA编程指南[4]中找到。

**计算居中的数据矩阵**

PCA通常在中心数据上进行，也就是说，我们必须从每个向量中减去平均向量U，中心向量V\_的分量得到如下：

结果，居中的数据矩阵d=V的列和到零。从概念上讲，平行有两个层次。一方面，我们可以将指数J上的平均修正与上一个平行。另一方面，每个图像V也可以独立处理。我们将看到，第一种方法的速度是后者的八倍。这可以用两种方法的内存访问数据来解释。然而，在我们深入讨论细节之前，让我们先编写一些代码，kemel并行地将焦点放在prxel指数上，并在图像指数上序列化循环（见清单7.9）

|  |
| --- |
|  |

**清单7.9：**Cuda内核执行平均校正(像素指数J)

该代码类似于平均计算内核。首先，全局线程标识符在第10行确定。其次，我们检查线程索引的范围，以防止内存访问违规（第12行）。第三，平均向量的j-th分量被写在第14行的寄存器值上。第四，我们从循环中的每个向量减去相应的值。

正交方法为图像生成一个线程，并在像素指标上序列化循环。相应的源代码在列表7.10中提供。在这里，我们再次计算全局线程标识符，然后检查它的范围。最后，循环在像素索引上执行。

|  |
| --- |
|  |

清单7.10：CUDA内核执行平均校正(图像索引I)

相应的内核调用主要不同于现在对应于图像数量的生成块和线程的数量。

当在一个基于帕斯卡的泰坦X上执行两个内核时，我们大约测量弗里斯特内核的60 MS和正交方法的大约500 MS。乍一看，这个观测似乎是楔形的，因为像素的数量是以far表示的。尽管如此，第二个内核由于其次优内存访问模式而不能受益于并行级别的增加。让我们看看正交方法的for-loop本体：

数据项不是连续访问的，尽管变量专长列举了内部循环，因此似乎是地址连续的内存。请记住，Y的32个线程同时执行。结果，我们在循环的每次迭代中都访问住在同一列中的32个条目，这会导致缓存线的过度失效。

**数字7.7**

在一个10×10矩阵x上的聚合和非聚合内存访问模式的例子左面板显示了8个连续线程如何能够同时使用整个缓存线（灰盒）。当访问右面板所示矩阵的列时，每个线程都会发出一条完整线的负载。此外，在操作第一个项时，每个缓存行的其余七个项将立即失效，该项在下一个迭代中强制执行一个新负载。灰色阴影区域的大小与预期执行时间相对应。

当连续线程访问连续内存时，更准确地说，当一系列线程(TK,TK+1……,TK+31)同时读取或写入连续内存位置时(PL,对于K和L的固定值Pl+1……Pl+31），我们称这种模式为“聚合”或“非聚合”（见图7.7）。在命令中将带宽饱和到Globalo内存的时候，加密的访问模式是非常可取的。非合并的读或写的速度和随机访问一样慢，应该不惜一切代价避免。可以应用几种技术来有效地防止随机存取，例如索引的重新排序，输入数据的转换（这是一种练习），或者在更快的内存类型上执行高度不规则的存取模式。后者将在第7.4节中加以说明。最后，让我们就在多维区块中的联合处理问题作一个总结。本地线程标识符susadidx.x比susadidx.y和susadidx.z修改得更快。因此，依赖于susadidx.x的变量（在我们的例子中）应该总是维护索引模式中最小的有效位。

Data[threadId.y\*matrix\_width+theadIdix.x]=some\_value;

一般比它的非聚合对应方要快得多

Data[threadId.x\*matrix\_width+theadIdix.y]=some\_value;

记住，这种微妙的区别可能会使你的算法大约慢一个数量级。在传统的CPU上可以观察到类似的行为，其中行主寻址（对应于合并后的内存访问）的性能相当优于列主索引（见第3节）

**协方差矩阵的计算**

PCA通过其协方差矩阵的对角化穿孔，确定了一个新的内径坐标收缩向量V一个条目C描述了两个像素J和J是如何相互关联的，因为它将对应的强度向量的标量积沿着图像轴重新组合。

在这里，()表示欧几里得空间中的标准标量积。例如，如果J表示左眼的位置，J表示右眼的位置，我们期望C是一个相当高的值。在一般情况下，我们确定在向量空间R.协方差矩阵C中所表示的N特征的所有对对相关关系，协方差矩阵C表现出许多理想的数学性质：

**.** （对称）一个条目C在指数J和J的交换下是不变的，因为两个实数的乘法是交换的。  
**.**（正态）C是一个正规矩阵，因为CT.C=c.c.因此C可以使用特征分解对角化。  
**.**（正谱）EQ中的标量积。(7.5)是正定双线性的，因此C只显示非负特征值。

C的实值特征向量{}是相互正交的，并跨越一个R上的N维向量空间，你可能想知道为什么我们对这个特定的基础感兴趣，因为我们可以坚持到标准的基础或任何其他线性无关向量集。假设我们想要描述的图像来自高维度向量空间R只有一个坐标。数学上，我们对最优基向量B感兴趣，它平均捕获了存储在中心数据矩阵中的大部分信息。让我们用标量积的方法将中心向量V投影到赋范基向量B=B/上。然后我们需要确定提供所有向量VI的最优最小二乘近似值的最小子U：

**数字7.8**

从R2 100点计算出的协方差矩阵的特征值解。点云是从一个多变量正态高斯采样的，该高斯由一个沿x轴旋转45度的因子展开，然后由U=(u0,.U1）因此两个特征向量u 0=-lrb-）描述了旋转矩阵的列，特征值=4和=1等价于旋转的内在坐标系中的标准差。

最后一部分在文献中称为瑞利。我们可以通过简单地选择具有最高特征值的C的遗传向量来最大化这个表达式可对角化矩阵C。B=B。与之相对应的是，我们可以证明，我们的0<k<n向量数据的最优基是由K特征向量的突集给出的，一个广泛的证明可以在[8]中找到。图7.8可视化描述的过程。

我们将计算一个合适的基础，为（有损的）近似的名人面所谓的Eigenes面本征值分解将完成使用coso verdnsgesvd方法从cosolver库。下面，我们将计算出名人脸（有损）近似值的合适基础。所谓的Eigenface本征值分解将使用从与CUDA捆绑在一起的库索弗库中的共解dnsgesvd方法来完成。因此，协方差矩阵计算的实现由我们来完成。我们假设平均调整后的图像存储在中心数据矩阵dij中，并且我们已经为设备上的协方差矩阵C J分配了内存。协方差内核的天真实现如下：

|  |
| --- |
|  |

清单7.11：库达内核执行一个幼稚的协方差计算。

源代码类似于平均计算内核，我们已经将数据矩阵的条目沿图像轴进行了总结。在这种情况下，对像素值的所有N种产品进行了累积：

// feel free to experiment with different block sizes

dim3 blocks(SDIV(rows\*cols,8),SDIV(rows\*cols,8)

dim3 threads(8,8);//64 threads(2 warps)per block

covariance\_kernel<<<blocks,threads>>>

(Data,cov,imgs,rows\*cols); CUERR

内核大约需要36秒来计算M=202,599张图像的全部协方差矩阵，每张图像都由N=2475像素组成。如果我们利用c.的对称性，执行时间可以减半。

|  |
| --- |
|  |

清单7.12：库达内核执行对称协方差矩阵计算。

如果我们移除对全局内存的冗余访问，那么天真和对称内核都可以被进一步加速至少10个因子。假设您想要计算协方差矩阵的第一行。

这个简单的内核从全局内存中为J的每一部分加载了V的所有M项，这就导致了mn的全局内存访问。对称方法将其减少了2倍，但仍然显示了O(mn)的依赖性。如果我们存储M项，这是可以避免的属于独立缓存中的像素零，它的访问速度明显快于全局内存。当忽略访问这类内存的时间时，我们可以在计算单行时将属于像素零（左因数）的所有值的负载减少到O（M）。

从概念上讲，我们的目标是实现只从全局内存中加载整个M×N矩阵D一次，然后执行计算c.结果所需要的nm许多加法，我们可以合理地增加对全球内存访问的计算率。这种方法可以在假定的GPU上实现，它大约有2GB的芯片内存。不幸的是，由于需要大量的晶体管和这种设备的相关成本，这个要求听起来相当乌托邦式。基于Pascal的Titan X为56个SMS中的每个SMS（见表7.1和图7.4）提供64KB的共享内存，因此整个设备的总体大小大约为3.5MB。它通常用于存储冗余数据或执行高度不规则的内存访问模式，在全局内存上执行时效率会很低。请注意，在文献中，由于描述的用法很小，一些作者倾向于将共享内存称为暂存器为组织目的存储。下面，我们将使用共享内存显式缓存数据矩阵的小瓷砖，以大幅减少协方差内核的执行时间。

假设我们要计算所有存在于形状W×W的二次瓷砖中的cjj条目，在M图像上的求和过程中的贡献只依赖于在该瓷砖范围内的像素指数J和j'。因此，我们必须为指数J和j存储长度为M的数据矩阵中的M列。不幸的是，这不适合48ke的共享内存，因为M列=202,599的长度非常大。然而，我们可以利用加法的关联性，然后在W像素的邻域中对W图像进行总结,

其中(I)等于1，如果I<m和0，则等于1。注意，辅助项(I)只有在M不是外和(循环)的每个贡献(迭代)的w的倍数时才需要用到，我们将所有由W映像产生的值存储在在暂存器内存中的两个独立数组中围绕J和J的W像素邻域。在内部和（循环）中，我们现在可以从第一内存中加载J和J的所有指数组合的因子。W)×sdiv(N,M)N×N协方差矩阵C的许多瓷砖将由一个二维网格中生成的单个螺纹块处理。我们可以再次利用C的对称性，只考虑这些在对角线以下不包含指数组合（J、J）的瓷砖。

首先，我们定义了一些辅助变量，方便地对瓷砖进行索引。窗口W对应于模板中的块大小参数。一个线程块将由块大小×块大小的线程组成，这些线程在二维网格中组织。此外，我们立即中止计算第25行对角线以上的图块的线程块。这可以通过只生成显示对角线的块来优化。然而，结果的指数计算是不平凡的，并且将使这个例子变得不合理。

**数字7.9**

从形状M×N=9×6的中心数据矩阵D出发，有效计算形状N×N=6×6的协方差矩阵C的跟踪方案。C和D都细分为形状为W×W=3×3的瓷砖。C瓷砖内的贡献以m/w=3迭代的形式连续地相加，在每次迭代中，瓷砖必须被加载到共享内存中，然后我们才能处理相应瓷砖的矩阵乘积。矩阵产品由W×W在一个线程块中的许多线程执行，这些线程可以同时访问存储在共享内存中的值。我们可以通过位于对角线上方的w瓦的系数来减少对全局内存的访问，但是由于c.的对称性，对角线上的瓦的贡献只需要部分的评估。

|  |
| --- |
|  |

**清单7.13：**CUDA内核的初始部分执行高效的协方差矩阵计算。

其次，我们使用第27和28行中的\_\_\_\_共享限定符来分配共享内存。它的语义类似于多维静态数组的定义。维度必须用恒定整数来指定，所以不能使用内核或其信号中定义的变量。因此我们必须将块大小作为模板参数来传递，这在编译时是已知的。在调用内核时，可以定义共享内存的大小，这将在第7.4节中详细讨论。进一步，我们计算要处理的块的数量，并定义一个寄存器，它将在沿着图像轴的求和过程中积累贡献。

|  |
| --- |
|  |

**清单7.14：**在CUDA内核中分配共享内存，进行高效的协方差矩阵计算。

在我们开始汇总贡献之前，我们必须将相应的部分从居中的数据矩阵D加载到共享内存中。在每个图像块中，我们枚举带有本地线程标识符\_Y=的数据矩阵的行（索引I）。为了确定全局行标识符，必须对本地索引进行偏移块\*chunk\_大小的调整。列索引（J和J）用thid\_x=swadiddx.x表示，由偏移项偏移的bas\_x和bas\_y表示，以选择正确的瓷砖（见第40和41行）。随后，我们检查全局标识符是否在有效范围内（见第44-46行）。辅助变量维尔德-行、维尔德-col-x和维尔德-col-y在以后的步骤中用于掩蔽，以防图像m数或像素N数不是窗口大小W的倍数。

|  |
| --- |
|  |

**清单7.15：**CUDA内核中主循环的启动，进行了全面协方差矩阵计算。

在定义了索引的必要变量之后，我们现在可以将瓷砖从全局内存加载到快速共享内存。这是一个简单的任务，如第53-55行所示，上面提到的辅助变量是用来防止恶意记忆的。超过M和N的范围的条目都是以零填充，这不会影响最终结果。注意，我们在左侧使用本地线程标识符，我们在那里访问共享内存，而在右侧使用全局标识符读取全局内存。但是，我们必须确保块中的所有线程都已通过对第60行中的\_\_sync线程的调用完成加载。在我们的例子中，我们产生了一个二维的形状8×8的块，即，两个翘曲（232个线程）都可能并行执行，因此必须明确同步。这对于防止一个WAP在另一个仍在加载数据时继续执行是至关重要的GPU必须通过终止整个内核或使用基于原子的同步方案来实现（见第8.1节）。

|  |
| --- |
|  |

清单7.6：全局到共享内存副本的CUDA内核执行有效的协方差矩阵计算。

在这一点上，我们有两个瓷砖在共享内存。部分标量积现在使用像素维度上的sinmple循环来计算（见第66行）。同样，我们可以利用对称和对角线以上的线63和75。这只影响协方差矩阵对角线上的瓷砖（见图7.9中的灰色十字）。第二个对第71行中的同步线程的调用强制执行一个块范围的屏障，以防止下一个迭代器的过早执行。最后，将归一化的结果写在对角线以上的项和它们的对称对应项的第76行的协方差Matix上。

|  |
| --- |
|  |

**清单7.17：**主要计算在CUDA内核执行有效的协方差矩阵计算。

描述的内核仅在基于Pascal的泰坦X上的980 MS中计算整个协方差矩阵，而对于专门在全局内存上运行的对称天真内核则是18S。这个令人印象深刻的加速大约18是由共享内存中冗余条目的缓存引起的。理论上，我们可以在从全局内存中读取数据矩阵的W列之后再使用它们。因此，如果我们增加窗口参数，那么SpeedUp应该单调地增长。然而，每个块使用的共享内存的数量也会影响执行时间。第7.4节详细讨论了这个主题。

**协方差矩阵的计算**

在确定了协方差矩阵之后，我们可以计算特征向量的集合和特征值的频谱，这是用库弗库提供的奇异值分解（SvD）的库弗格斯维夫例程完成的。在正定对称矩阵的情况下svd将一般M×N矩阵D分解为三个矩阵的乘积 :

其中U和V都是正交线性映射，即， 奇异值E的矩阵除对角线外处处为零，C=D的特征值分解是容易的。dd可通过SvD获得：

然而，与其计算巨大中心数据矩阵D的代价昂贵的SvD，我们直接将其应用于小得多的协方差矩阵c.忆及C是对称的(C=CT)，因此我们观察到U=V自

注意，这不是一个严格的数学证明，但对编程书来说已经足够了。我们可以进一步利用SvD返回的特征向量是按照它们的特征值的大小排序的，在我们的例子中，这些特征向量与奇异值一致储存在。因此，第一个特征向量捕获了存储在中间的Datamatrix D中的大部分方差。图7.10描述了希瓦数据集的前16个特征向量.

**数字7.10**

形状55×45的前16个特征向量（UO……U15）分别从存储在welliba数据中的中心图像V的协方差矩阵C计算得出。这些图像按它们对应的特征值的大小按降序排序（行主索引）。这些特征向量由于与人的面部有很强的相似性，非常适合用于名人照片的有损压缩。

为了简单起见，我们在SvD中定义的设备函数中隐藏了对cusverdnsgesvd的调用。hpp标题，这是与这本书有关的。

形状N×N的矩阵cov、U和V必须作为设备指针提供。此外，奇异值的对角矩阵表示为长度为N的器件数组，长度为N的N特征向量分别存储在矩阵U的行中。我们把你转移到主机矩阵eigs上，然后将前16名候选人写入磁盘。

|  |
| --- |
|  |

清单7.18：主要功能骨架进行SvD分解。

已经走到了尽头，让我们简单地讨论一下本征面的一个有用的应用。假设我们想用仅仅的K<<n 值来近似一个面V的图像，而不是储存所有的N位元。这可以通过图像V=v的投影由K的特征组成（f 0,……,fk-1）来实现：

**数字7.11**

通过在顶k特征面的基础上展开图像对人脸进行有损压缩，当使用所有的特征向量时，我们可以完美地重建原始图像（右下角）。这些图像的采样分辨率为N=10989=9701像素。用1000个基本向量（左下角）组合起来的图像的压缩比大约为1:10。当只使用几个特征向量时，就可以清楚地观察到明星数据集中的女性偏差，因为平均图像主导了重建。

其中表示欧几里得空间中的标准标量积，原始图像V的重建是直线的-我们简单地在特征向量的基础上展开平均调整后的图像V：

注意，在K=N的情况下，我们可以完美地重构V，因为我们已经将它展开为一个完全的基础N线性无关向量(u0,……,n-1)。对于K，我们得到原始图像的有损逼近（见图7.11）。

7.4**内存层次结构(动态时间调整)**

在上一节中，您已经了解了如何正确利用内存访问模式和共享内存，以显著加快CUDA kernels的性能。即纹理记忆体与常数记忆体，以进一步减少CUDA应用的执行时间。  
在本节中，我们将实现一种时间序列的弹性比较算法，这种时间序列产生于时间序列数据的领域所谓的动态时间扭曲相似度（dtw）。在我们开始编码之前，让我们定义术语时间序列并讨论dtw算法。

**介绍**

本小节主要用于简要介绍时间序列的弹性匹配，并确定一些重要的注释。如果你已经熟悉了主题，可以跳过它。  
定义1（统一时间序列）.设R为实数的进一步T=(t 0,T1,……,TJ,……,TN-1)一个有限序列N实值,均匀间隔,有序时间戳,即,

而不是将时间戳映射到实值上，TJ stj在实域上称为实值和一致的时间序列。

时间邮票的特殊价值往往被忽视。稍微滥用记谱法，N时间戳的映射TJ stj可以通过列举从O到N-1的量值wih指数来改写为纯向量：

这个定义可以自然延伸到更高维度的值SJ RD或非均匀间隔或连续的时间域。然而，为了简单起见，我们把自己限制在这个简单的场景中。