第七章：统一计算设备构架

摘要

A Graphics Processing Unit (GPU) is an integral piece of hardware in many electronic devices. It is used to expose graphical interfaces to the user and can be found in almost every workstation or mobile device. Starting from the mid-nineties of the last century, the increasing popularity of 3D-accelerated games (such as the Quake and Unreal series) caused a rapid growth of computational capabilities of modern GPUs allowing for the rendering of more and more complex scenes.

图形处理器（GPU）作为一个一个整体部件，是很多电子设备中不可或缺的一部分。 在常见的工作站和移动设备中，GPU常被用来作为图像用户接口展现在用户面前。从上个世纪90年代中期开始，随着3d加速游戏的日益流行(如《雷神之锤》和虚幻系列)，带来了现代GPU计算能力的迅速增长，GPU允许渲染越来越复杂的场景。

In the late nineties, manufacturers extended the GPU’s core functionality, the efﬁcient rendering of 3D scenes, to additionally support the processing of geometric operations, notably the pioneering NVIDIA GeForce 256 [5] and its dedicated Transform and Lighting (T&L) unit.

In the late nineties, manufacturers extended the GPU’s core functionality, the efﬁcient rendering of 3D scenes, to additionally support the processing of geometric operations, notably the pioneering NVIDIA GeForce 256 [5] and its dedicated Transform and Lighting (T&L) unit.

This trend continued in the new millennium with the introduction of pixel and vertex shader languages which allowed for the manipulation of the prior to this hard-wired rendering pipeline.

九十年代末，制造商扩展了GPU的核心功能，从3D场景的高效渲染，到几何运算的处理，特别是开创性的NVIDIA GeForce 256 产品[5]，它采用了专门的转换和照明(T&L)单元。

随着Pixel 和 Vertex Shader语言的引入，这种趋势在新千年继续保持，允许在这个硬连线的渲染管道之前对其进行操作。

Consequently, computer scientists used the increasing computational power of GPUs to implement more general algorithms by expressing them in the aforementioned shader languages. This was the birth of the so-called general-purpose computing on GPUs (GPGPU).

因此，计算机科学家借助GPUs不断增加的计算能力，通过前面提到的shader语言完成更多通用算法的实现。这就是所谓的GPU通用计算的诞生

A major drawback of GPGPU programming was its lack of abstraction: the processed data had to be encoded in textures and manipulated with the instruction set of the shader language which substantially limited the design of complex programs.

GPGPU编程的一个主要缺点是缺乏抽象性：待处理的数据必须用纹理来编码并用shader语音提供的指令集来操作。这极大地限制了复杂程序的设计。

In summer 2007, NVIDIA released the Compute Uniﬁed Device Architecture (CUDA) [3], that allows

for the convenient programming of complex applications using a C- or FORTRAN-based language.

2007年夏天，NVIDIA发布了计算统一设备架构（CUDA）[3]，CUDA允许使用基于C或 FORTRAN编程语言完成复杂应用程序的编程。

As a result, general algorithms can be expressed in easily readable code and, at the same time, beneﬁt from the up to two orders-of-magnitude faster execution time compared to single-threaded CPU

implementations

因此，可以用简单可读的代码完成通用算法的开发，此外，与单线程的CPU实现相比，CUDA程序可以获得多达两个数量级执行时间的提升。

Although there exist uniﬁed approaches for the programming of GPUs from other vendors (e.g., OpenCL [9] and OpenACC [17]), CUDA is nowadays the predominant parallelization framework on GPUs. This chapter will teach you the basics of CUDA-C++ covering the programming model, the efﬁcient use of the underlying memory hierarchy, and mechanisms for the synchronization of massively parallel tasks with many thousands of threads.。

虽然也存在其他供应商提供的gpu编程统一的方法（例如OpenCL [9]和OpenACC [17]）），但CUDA是目前gpu编程主要的并行处理框架。

本章将为你介绍CUDA-C++的基本知识，包括编程模型、分层存储层次结构的高效使用，以及成千上万线程大规模并行任务的同步机制。

关键词

CUDA, 大规模并行计算，GPU，内核，线程块，设备同步，Warp，Thrust。计算机到全局存储访问，Tesla，流式多处理器，SIMD，主成分分析。循环展开、合并内存访问、特征值分解、共享内存

GPGPU历史的详尽概述可参见[13]。

**目录**

**7.1** CUDA介绍(Hello World)……………………………………………………226

**7.2**支持CUDA的GPU硬件架构…………………………………………228  
主机与设备之间的互连……………………………………………………228  
显存和峰值宽度…………………………………………………………229  
计算资源的组织………………………………………………………………229  
映射线程块到SMS……………………………………………………230  
映射线程到Warps…………………………………………………………233

**7.3**访存模式(Eigenfaces)……………………………………………………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  
序列化DTW的线性存储算法………………………………………………261  
线性存储器dtw的一个天真CUDA端口………………………………267  
共享内存中的波前放松……………………………………………………271  
并发调度和bank冲突…………………………………………………………277  
纹理和恒定…………………………………………………………………………278

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

**7.6**课外习题…………………………………………………………………………282

参考……………………………………………………………………284

7.1 **CUDA介绍(Hello World)**

In this chapter you will learn how to write massively parallel programs on NVIDIA GPUs with the help of the CUDA programming model. Since CUDA is basically an extension to C and C++, we can easily port embarrassingly parallel programs without the overhead of learning a completely new language. Let us start with a simple Hello World program that prints a greeting message to the command line. The source code in Listing 7.1 is separated into two parts. It consists of a main function that is executed on the CPU and a kernel method hello\_kernel that will be running on the GPU.

在本章中，你将学习到如何借助于CUDA编程模型在NVIDIA GPU上编写大规模并行程序。因为CUDA基本上是C和c++的扩展，因此，我们无需学习一门全新的语言就可以很容易地移植复杂的并行程序。让我们从一个简单的Hello world程序开始，这个程序仅仅是在命令行打印一条问候语。列表7.1中的源代码被分成两部分。一部分是在CPU端执行的main函数,一部分是在GPU端执行的hello\_kernel kernel。

清单7.1：CUDA Hello World

Let us have a look at the main function. The ﬁrst command cudaSetDevice selects the used CUDA-capable GPU. Usually, you will only have one GPU attached to your workstation and thus the identifier can be safely set to 0. We will talk about the possibility to simultaneously utilize several GPUs later in this chapter. The second call hello\_kernel«<1, 4»>() will invoke one block of four CUDA threadsrunning in parallel on the GPU. Each thread is supposed to print a greeting message.

For the moment, we ignore the implementation details of the kernel and proceed with the cudaDeviceSynchronize() call. Kernels on the GPU (device) and code on the CPU (host) are executed asynchronously. The final synchronization forces the host to wait until the kernel on the device has ﬁnished its computation. This prevents an immediate termination of the whole program. Otherwise, the host-sided mainfunction will instantly return without printing the desired message when using no explicit or implicit synchronization mechanism.

让我们来看看它的主要功能。第一个命令udaSetDevice会选择可以使用的支持CUDA功能的GPU，通常一个工作站只会安装一个GPU，这样标识符可以被安全地设置为0。我们将在本章后面讨论同时使用多个GPU的可能性。第二个函数调用hello\_kernel«<1, 4»>() 将会发起一个包含四个CUDA threads的一个Block。每个线程打印一个问候语。目前，我们忽略了内核的实现细节，继续进行cudaDeviceSynchronize()。因为GPU（设备）上的内核和CPU（主机）上的代码是异步执行的。最后的同步语句迫使主机等待直到设备上的内核完成计算，这可以防止整个程序立刻终止。否则，如果不使用显式或隐式同步机制，主机端main函数将立即返回，从而不会成功打印所期望的消息。

An obvious novelty in the signature of hello\_kernel is the \_\_global\_\_ qualiﬁer right in front of the return value void indicating that this method is callable from the host but executed on the device.The CUDA programming language provides further qualiﬁers that can be used to specify the scope of execution and inlining behavior. The most important are the following:

hello\_kernel中最明显的新奇之处是返回值void前面的全局限定符\_\_global\_\_，该限定符指示此函数可从主机调用，但在设备上执行。CUDA编程语言还提供了一些限定符，可以用来明确函数执行的范围和内联行为，最重要的标识符有以下几种：

. \_\_global\_\_：主机端调用，设备上执行，  
. \_\_host\_\_：主机端调用，主机上执行，  
. \_\_device\_\_：设备端调用，设备上执行。

Note that \_\_host\_\_ is implicitly deﬁned for every host function and thus could be added in front of every traditional function which is executed on the CPU. The qualiﬁers \_\_host\_\_and \_\_device\_\_can be combined in order to force the compiler to generate binary code for both the CPU and GPU. Additionally, the \_\_noinline\_\_ and \_\_forceinline\_\_ qualiﬁers can be used to control inlining behavior.More details can be found in the CUDA programming guide [4].

请注意，\_\_host\_\_它是为每个主机函数隐式定义的，因此可以在CPU上执行的每个传统函数前面添加此限定符。限定符\_\_host\_\_与\_\_device\_\_可以合并，强制编译器产生可以在CPU和GPU执行的二进制代码。另外，\_\_noinline\_\_和\_\_forceinline\_\_可以用于函数内联行为的控制。更多细节可以参考CUDA编程指南【4】。

As mentioned before, we invoke the kernel on the GPU using one CUDA thread block consisting of four CUDA threads. Further details on the speciﬁcation of blocks and threads are discussed in the next section. Up to now it is enough to know that we spawn one block with block identiﬁer blockIdx.x=0consisting of blockDim.x=4threads, each of them enumerated with a local thread identiﬁer threadIdx.x ranging from 0 to 3 (inclusive). As a result, the expression in Line 6 evaluates to thid = 0\*4 + threadIdx.x = threadIdx.x. Alternatively, we could have generated the same indices when spawning two blocks each executing two threads, or four blocks each executing one thread. As we will see later in Section 7.4, the particular distribution of threads can have a massive impact on the overall performance of our program. Finally, the global thread identiﬁer thidis printed via the printf command. The example code can be compiled with a call to nvcc hello\_world.cu -O2 -o hello\_world and executed like any other binary built with a host compiler obtaining the following output:

Hello from thread 0!

Hello from thread 1!

Hello from thread 2!

Hello from thread 3!

Note that the optimization ﬂag -O2 only affects the host part of the code and does not inﬂuence the performance of device functions. Let us brieﬂy discuss the output in the end. You might wonder why the messages are printed in order in contrast to typical multithreaded applications. This is because the print statements are always serialized in batches of 32 consecutive threads within a thread block, a so-called warp of threads. If the kernel invocation is modiﬁed to use 512 threads within one block, we will observe shufﬂed batches each consisting of 32 contiguous thread identiﬁers.

2Note that kernels can also be called recursively by the same kernel. However, the concept of Dynamic Parallelism will not bediscussed in this book due to its infrequent occurrence in the literature.

如前所述，我们使用由四个CUDA threads组成的一个CUDA thread Block来调用GPU上的内核。关于blocks 和 threads的更多细节将下一节中讨论。

到目前为止，我们只需要知道我们生成了一个Block，即blockIdx.x=0，该Block包含4个thread即：blockDim.x=4, 它们都使用本地线程标识符threadIdx.x来枚举，范围从0到3。因此，第6行中的表达式计算结果为=0\*4+threadIdx.x=threadIdx.x. 另外，当生成两个block,每个block包含2个thread，或生成4个block,每个block包含1个thread时，我们都可以生成相同的索引。正如我们将在第7.4节看到的，线程的特定分布会对程序的整体性能产生巨大的影响。最后，通过Printf命令打印全局线程标识符thid。示例代码可以通过nvcc hello-world.cu -02 -o hello-world 编译，像其他host编译器构建的二进制一样执行，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host端的代码部分，不影响device 函数的性能。让我们简单地讨论一下终端的输出。可能疑惑，与传统的多线程应用程序相比，为什么打印的消息是顺序的。

这是因为在一个thread Block中，连续的32 个线程为一组来同时执行，也就是所谓的Warp，所以这些打印语句被序列化了。如果内核调用被修改为在一个Block中包含512个thread，我们将会观察到32个连续线程标识符为一组的乱序输出。

7.2**支持cuda的GPU的硬件架构**

无可否认，Hello world示例不是很有用，因为打印是序列化的，而且没有处理数据。但在我们开始计算数字之前，我们必须了解支持CUDA功能的典型GPU的硬件布局，以及CUDA编程模型如何映射到GPU的各个组件。这一节有点理论性，GPU的特定硬件细节不属于CUDA编程语言的一部分，但我们将从这些知识中获益。引用Jean-Jacques Rousseau的话：忍耐是痛苦的，但它的果实是甜蜜的。

**INTERCONNECTION BETWEEN HOST AND DEVICE**

Let us start with the obvious. CUDA-capable GPUs are attached to a server or workstation via dedicatedslots on the mainboard. At the time of writing, the majority of graphics cards are connected via the PCIe v3 bus providing up to 16 GB/s bandwidth (see Fig. 7.1). At ﬁrst sight this sounds reasonably fast but you will soon observe that this is often the main bottleneck of CUDA applications. Addressing this, NVIDIA has introduced NVLink, a novel bus providing up to 80 GB/s peak bandwidth [6], allowing for the efﬁcient communication between the host and the device or between multiple devices attached to the same compute node.

**主机与设备之间的互连**

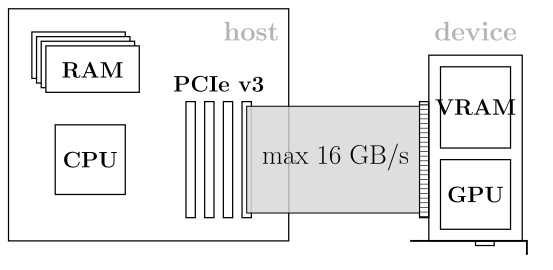
让我们从显而易见的开始。支持CUDA的GPU通过主板上专用的插槽连接到服务器或工作站。在写作本书时，大多数显卡都是通过PCIe v3总线连接，该总线提供高达16GB/s的带宽（见图7.1）。乍一看，这带宽速度相当快，但很快就会发现该带宽常常是CUDA应用程序的主要瓶颈。针对这一点，NVIDIA推出了NVLink，它是一种可以提供高达80GB/s峰值带宽的新型总线[6]，提供host和device之间或同一个计算节点内多个设备之间的有效通信。

The memory of the host (RAM) and the video memory of the device (VRAM) are physically separated. As a result, we have to allocate memory on both platforms independently and afterwards manage memory transfers between them. It means that data which shall be processed on the GPU has to be explicitly transferred from the host to the device. The computed results residing in the VRAM of the GPU have to be copied back to the host in order to write them to disk. This implies that data allocated on the host cannot be directly accessed from the device and vice versa. The corresponding CUDA commands will be discussed in detail in Section 7.3.

主机（RAM）的内存和设备（VRAM）的视频内存是物理分离的。因此，我们必须在两个平台上独立分配内存，然后管理它们之间的内存传输。这意味着要在GPU上处理的数据必须显式地从主机传输到设备上。必须将驻留在GPU VRAM中的计算结果拷贝回主机，以便将其写入磁盘。也就是说， Device端不能直接访问在主机端分配的数据，反之亦然。相关的CUDA 命令细节将在7.3节讨论.

Note that NVIDIA provides powerful libraries bundled with CUDA, e.g. Thrust [7], which features device vectors that can be manipulated from the host. Nevertheless, **you should be aware that these fancy abstraction layers may obscure suboptimal parts of your code**. As an example, altering all entries of a Thrust device vector in a host-sided for-loop results in the excessive spamming of tiny memory transfers. **A similar though less harsh statement** can be made about NVIDIA’s uniﬁed memory layer that treats the address spaces of the RAM and VRAM as one. Consequently, we will pursue the traditional distinction between both memory spaces. This transparent approach will help you to pinpoint performance bottlenecks of your applications without having to guess the internal details of uniﬁed Memory addressing.

请注意，NVIDIA提供了与CUDA绑定的功能强大的类库, 例如：Thrust [7], 它支持可以从Host直接操作的设备端 vectors。尽管如此，我们应该注意到这些抽象层可能会使代码变得模糊。例如，在Host端for-loop中更改Thrust设备vector的所有入口条目会产生微小内存传输造成的过量垃圾信息。NVIDIA将RAM和VRAM的地址空间视为一体的统一内存层也可以造成类似的问题。因此，我们将继续研究这两个存储空间之间的传统区别。这种透明的方法将帮助您确定应用程序的性能瓶颈，同时又不必猜测统一内存寻址的内部细节

****

**图7.1**

Schematic overview of the processors (CPU and GPU) and memory components for host and device. Memory transfers are executed over the PCIe v3 bus providing up to 16 GB/s bandwidth. Conceptually, the host and the device can be treated as independent compute platforms which communicate over an interconnection network.

处理器（CPU和GPU）和主机和设备内存组件的示意图概述。通过PCIe v3总线，提供高达16GB/s带宽的内存传输。从概念上讲，主机和设备可以看作是通过互连网络进行通信的独立计算平台。

**VIDEO MEMORY AND PEAK BANDWIDTH**

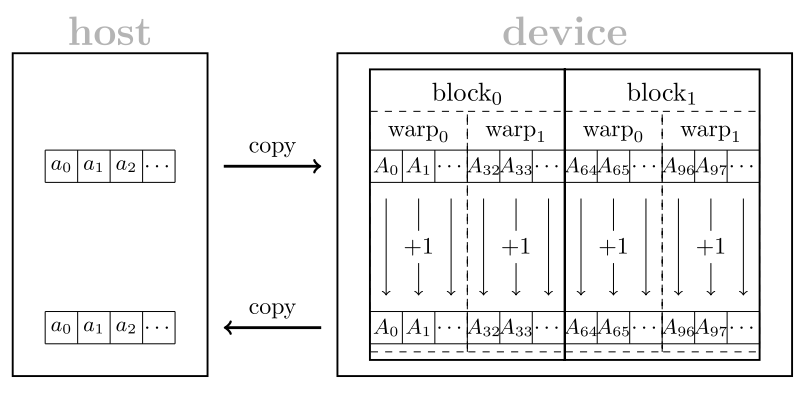
Conceptually, the graphics card can be treated as an independent compute platform with its own compute unit and memory. In contrast to the memory attached to a typical mainboard (at the time of writing, DDR4 DRAM) the video memory of a graphics cards is signiﬁcantly faster. As an example, the Pascal-based Titan X features 12 GB of GDDR5X DRAM modules providing up to 480 GB/s bandwidth compared to the memory bandwidth of current Xeon workstations of less than 100 GB/s. Accelerator cards from the professional Tesla series such as the Tesla P100 or Tesla V100 feature even faster HBM2 stacked memory with up to 720 GB/s or 900 GB/s bandwidth, respectively.

**显存和峰值宽度**

从概念上讲，拥有自己的计算单元和内存的显卡可以被认为一个独立的计算平台。与一般主板上附加的内存（写这本书的时候，DDR4 DRAM）相比，显卡上显存的访问要快很多。例如，基于Pascal的Titan X具有12GB的GDDR 5X DRAM模块，提供高达480 GB/s的带宽，而当前Xeon工作站的内存带宽不足100 GB/s。专业级Tesla系列的加速卡如Tesla P100或Tesla V100，配备更快的HBM2堆叠式存储体，分别可提供高达720GB/s或900GB/s的带宽 。

Despite the fact that we can access the global memory with almost one TB/s, we still have to care about the underlying memory hierarchy. As an example, assume you want to increment single-precision ﬂoating point (FP32) values residing in VRAM (global memory). First, each of the 32-bit values is loaded into a register. Second, the register is incremented by one which accounts for a single ﬂoating point operation (Flop). Finally, the incremented value is written back to global memory. Fig. 7.2 visu- alizes the described scheme. This corresponds to a *compute-to-global-memory-access* (CGMA) ratio of 1 Flop/8 B (four bytes for reading and four bytes for writing after incrementation) resulting in an overall compute performance of 125 GFlop/s for a memory bandwidth of 1 TB/s. This is merely 1% of the 11 TFlop/s FP32 peak performance of a Tesla P100 card. Therefore proper memory access patterns and the efﬁcient use of fast caches will be one of our main concerns.

尽管访问全局内存的带宽可以达到1 TB/s，但我们仍然需要关心底层的内存层次结构。例如，假设要完成驻留在VRAM(全局内存)中的单精度浮点数(FP32)值加1 操作。首先，将每个32位值加载到寄存器中。其次，寄存器作加1操作，即一个浮点运算(FLOP)。最后，递增后的值被写回全局内存。图7.2对运算机制进行了图形化描述。所对应的计算访存比compute-to-global-memory-access为1 Flop/8 B（读取4字节，加1操作后写回4字节）。因此，在内存带宽为1 TB/s的情况下，总的计算性能为125 GFlop/s，这仅仅是Tesla P100加速卡11 TFlop/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。



**图7.2**

Memory transfers and control ﬂow of a kernel that increments the entries of an array *a* = *(a*0*,a*1*,...)* stored on the host. First, the host array *a* is copied to the device array *A* of same size. Second, each warp of a thread block simultaneously increments 32 consecutive entries of the array *A*. Third, the incremented values residing in the device array *A* are copied back to the host array *a*.

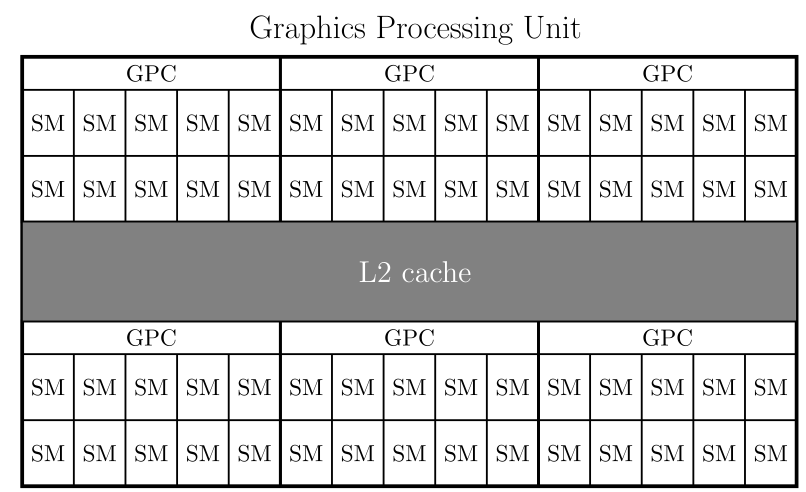
内存传输和kernel控制流，该kernel完成对存储在host上的数组a=(a0,a1,……)的所有元素进行递增操作。首先，将主机数组a复制到相同大小的设备数组a中。其次，线程block中每个Warp同时对数组a中连续的32个元素进行递增操作。第三，将驻留在设备内存数组A中递增后的值复制回主机数组A

ORGANIZATION OF COMPUTATIONAL RESOURCES

The enormous compute capability of modern GPUs can be explained by the sheer number of processing units which usually exceeds a few thousand cores. In contrast to multi-core CPUs **with a few tens monolithic** processing units featuring complex instruction sets and control ﬂows, GPUs can be treated as a huge array of lightweight processing units with limited instruction set and constrained control ﬂow. The hardware layout of a modern GPU is organized in a hierarchical tree.

**计算资源的组织**

现代GPU巨大的计算能力可以用处理单元的绝对数量来解释，这些处理单元通常超过几千个内核。拥有几十个单片处理单元的多核cpu，通常具有复杂的指令集和控制流程，而GPU可以被看作是一个巨大的轻量级处理单元数组，每个处理单元支持的指令集和控制流都非常有限。现代GPU的硬件布局通常采用层次树的形式来组织。



**图7.3**

Schematic layout of the GP102 GPU built in the Tesla P40 card. The GPU is partitioned into six Graphics Processing Clusters (GPCs) each providing ten Streaming Multiprocessors (SMs). Each of the 60 SMs share the same L2 cache and global memory. Consumer GPUs such as the Pascal based GeForce GTX 1080 or Titan X have similar layouts with a smaller number of GPCs or SMs (see Table 7.1).

Tesla P40内置的GP102 GPU示意图。GPU被划分成六个图形处理集群(GPCs)，每个集群提供十个流式多处理器（SMS）。这60个SMs共享相同的L2缓存和全局内存。消费级GPU，如基于Pascal的GeForce GTX 1080或Titan X，具有类似的层次组织但较少的GPCs或SMs（见表7.1）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 7.1 Technical speciﬁcation of high-end single GPU video cards across generations. Dual GPUs like the Titan Z, Tesla K80, or Tesla M60 are omitted since they are basically stacked versions of their single GPU counterparts. Tesla cards from the Kepler and Maxwell generation are equipped with slower error- correcting (ECC) RAM. The Pascal-based Tesla P100 and Volta-based Tesla V100 provide signiﬁcantly faster HBM2 stacked memory. Note that the number of cores and VRAM size are slightly increasing over time.**  表7.1跨代高端单GPU显卡技术规范。像Titan Z、Tesla K80或Tesla M60这样的双GPU被省略了，因为它们基本上是它们的单GPU版本的叠加版本。从开普勒和麦克斯韦的特斯拉显卡配备了较慢的纠错(ECC) RAM。基于 Pascal的Tesla P100和基于 Volta 的Tesla V100，配备更快的HBM2堆叠式内存。核心数量和VRAM大小会随着时间的推移而略有增加。 | | | | | |
| **Video card 显卡** | **Generation 代** | **GPCs** | **SMs** | **FP32/GPU** | **VRAM@Bandwidth** |
| Titan (Black) |  |  |  |  |  |
| Tesla K40 |  |  |  |  |  |
| Titan X |  |  |  |  |  |
| Tesla M40 |  |  |  |  |  |
| GTX 1080 |  |  |  |  |  |
| Titan X |  |  |  |  |  |
| Tesla P40 |  |  |  |  |  |
| Tesla P100 |  |  |  |  |  |
| Tesla V100 |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 7.2 Technical speciﬁcation of GPU architectures across four generations. The number of SMs per GPC increases while the amount of cores per SM decreases over time. Thus a core on a Pascal/Volta SM can use more registers than on SMs from previous generations. Further, note the relatively small FP64/FP32 ratio of the Maxwell architecture.**  表7.2跨越四代的GPU架构技术规范。随着时间的推移，每个GPC的SMs数量在增加，而每个SM的内核数量在减少。因此，与上一代相比，Pascal/Volta SM上的core可以使用更多的寄存器。此外，请注意Maxwell体系结构具有相对较小的FP64/FP32比率。 | | | | | |
| **Generation** | **SM/GPC** | **FP32/SM** | **FP64/SM** | **FP64/FP32** | **Registers/SM** |
| Kepler |  |  |  |  |  |
| Maxwell |  |  |  |  |  |
| Pascal |  |  |  |  |  |
| Volta |  |  |  |  |  |

**Mapping Thread Blocks Onto SMs**

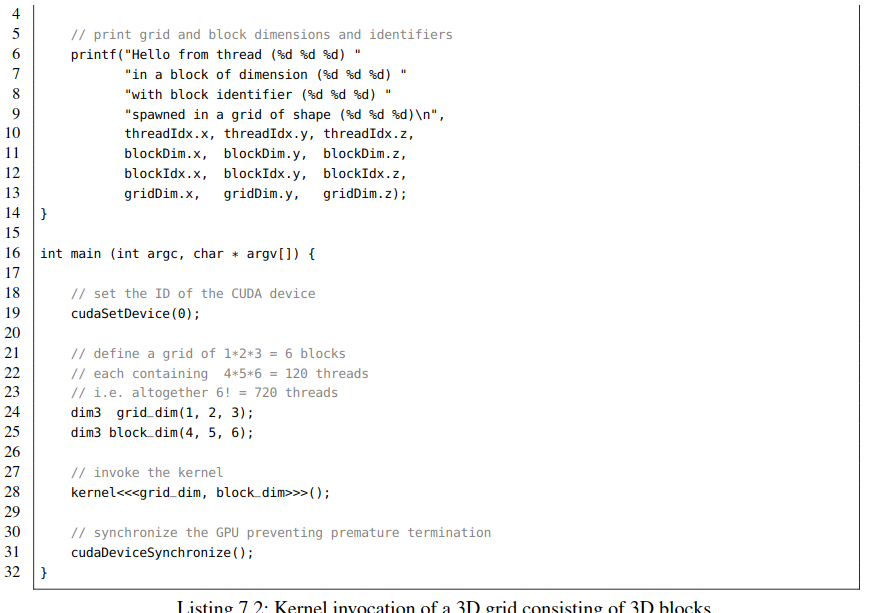
The ﬁrst level consists of a small number (see Table 7.1) of Graphics Processing Clusters (GPCs) each of them containing roughly ten (see Table 7.2) Streaming Multiprocessors (SMs) as shown in Fig. 7.3. The GPCs are a rather hardware-speciﬁc detail that have no correspondence in the CUDA programming model. However, the SMs in the second level loosely map onto the aforementioned CUDA thread blocks. One or more thread blocks can be simultaneously executed on a single SM. The speciﬁc number is dependent on the ratio of required and available hardware resources as shown in Section 7.4. Note that the runtime environment does not provide any information about a speciﬁc order of execution, i.e., we can neither inﬂuence nor deduce the planning scheme of the block scheduler. As a result, the parts of our program that are executed on different blocks should be truly data independent.

线程块到SMs的映射

第一层由少量图形处理集群(GPCs)组成(见表7.1)，每个集群包含大约10个(见表7.2)流式多处理器(SMs)，如图7.3所示。GPCs是一个相当具体的硬件细节，在CUDA编程模型中没有与此对应的关系。但是，第二层中的SMs松散地映射到前面提到的CUDA线程块。一个或多个线程块可以同时在一个SM上执行。如图7.4所示，具体数量取决于所需要的和可用硬件资源的比例。请注意，运行时环境不提供关于特定执行顺序的任何信息，即，我们既不能影响也不能推出block调度的规划方案。因此，在不同块上执行的程序是真正数据独立的。

Within the CUDA programming model the block identiﬁers can be deﬁned using 1D, 2D, and 3D grids. This allows for a convenient indexing when accessing data on multidimensional domains. As an example, when processing distinct parts of an image one could assign a block with coordinates (blockIdx.x, blockIdx.y)to each tile. The grid dimension can be deﬁned as an integer for 1D grids or as a dim3struct for the general case. They can be accessed within a kernel via gridDim.x, gridDim.y, and gridDim.z. Our Hello World example from the previous section utilized a 1D grid with either one, two, or four blocks. We demonstrate the invocation of a 3D grid consisting of 3D blocks in Listing 7.2.

在CUDA编程模型中，可以使用1D、2D和3D网格定义块标识符。这使得在访问多维域上的数据时可以方便地建立索引。例如，在处理图像的不同部分时，对每一个tile，可以分配一个包含坐标(blockIdx.x, blockIdx.y)的Block。网格维度可以定义为一维网格的整数或者通常情况的三维结构。它们可以通过gridDim x,gridDim。y,gridDim.z在内核中访问。上一节中的Hello World示例就使用了一个、两个或四个块的1D网格。Listing 7.2展示了由3D Block组成的3D grid的使用。

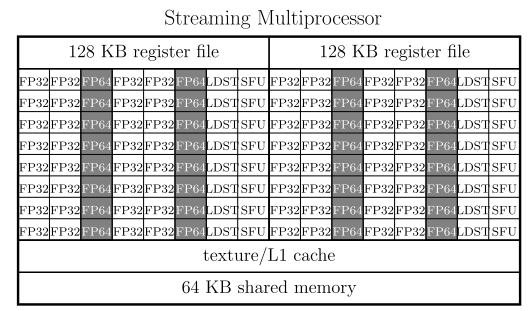


|  |
| --- |
| 1 #include <stdio.h> // printf  2  3 \_\_global\_\_ void kernel() { |

列表7.2：由3D Block组成的3D Grid的内核调用

Before we proceed with the internal components of an SM we have to make an important remark: different CUDA thread blocks of the same kernel cannot be synchronized during the execution of a kernel since there exists no device-wide barrier up to CUDA version 8 which could be called from within a kernel. Note that CUDA 9 introduces the concept of cooperative groups that provide device- sided multi-grid synchronization calls. However, CUDA 9 is not released at the time of writing this book and thus we stick to the traditional approach: inter-block dependencies have to be realized by stacking several kernels on the host enforcing a barrier between individual kernel invocations. This architectural decision made by NVIDIA has several motivations. First, the hardware layouts of low-end and high-end GPUs mainly differ in the number of provided SMs. Thus an elaborate programming model should scale transparently over an arbitrary number of blocks. Second, synchronization is ex- pensive especially when enforcing a global barrier where all SMs have to wait for the last block to ﬁnish. This observation can be summarized in a statement that *synchronization does not scale*.3 As a consequence, we are forced to refactor or even redesign code that relies excessively on global barriers or mutexes.

在我们继续讨论SM的内部组件之前，我们必须注意: 同一内核的不同CUDA线程块不能在内核执行期间同步, 因为在CUDA版本8之前不存在device范围内，且可在内核中调用的barrier。 注意，CUDA 9引入了提供设备端多网格同步调用的协作组的概念。然而，在写这本书的时候CUDA 9还没有发布，所以我们坚持传统的方法: 块间依赖关系必须通过在主机上叠加几个内核来实现，在各个内核调用之间设置一个屏障。NVIDIA设计这种架构主要有几个原因。第一，低端gpu和高端gpu的硬件布局主要区别在于SMs的数量。因此，一个精心设计的编程模型应该透明地扩展任意数量的Block。其次，同步是有代价的，特别是当强制设置一个全局屏障时，所有的SMs都必须等待最后一个块完成。这一观察结果可以总结为一句: *synchronization does not scale*.3 ( 半自动并行化框架OpenACC遵循同样的原则[14]。) 因此，我们被迫重构甚至重新设计过度依赖全局barrier 或 mutex的代码。



**图7.4**

Schematic layout of a Streaming Multiprocessor (SM) used in the Pascal generation. An SM provides two blocks each consisting of 32 single-precision compute units (FP32), 16 double-precision compute units (FP64), 8 load and store units (LDST), and 8 special function units (SFU). Each of the two blocks can access 32,768 32-bit registers resulting in an overall size of 256 KB for the storage of variables. L1 cache and 64 KB shared memory are shared among all units in the SM.

Pascal一代，流多处理器(SM)布局示意图。SM包含两个Block，每个块由32个单精度计算单元(FP32)，16个双精度计算单元(FP64)，8个加载和存储单元(LDST)和8个特殊功能单元组成。这两个块中的每一个都可以访问32,768个32位寄存器，**从而寄存器变量总大小为256 KB，用于存储变量**。在SM中的所有单元之间共享L1缓存和64KB共享内存。

**Mapping Threads Into Warps**

Each SM is composed of multiple compute and instruction units as shown in Fig. 7.4. The Pascal generation features 64 single-precision (FP32), 32 double-precision (FP64) compute units, 16 load and store units (LDST), and 16 special function units (SFU) per SM. The number and ratio of FP32 and FP64 units depend on the GPU generation. Therefore, FP64 performance cannot be directly inferred from the number of FP32 units. As an example, the Maxwell generation provided a FP64 to FP32 ratio of merely 1/32 in contrast to the reasonably higher fractions of the Kepler (1/3), Pascal (1/2), and Volta (1/2) generations.

**线程到Warp 的映射**

如图7.4所示，每个SM由多个计算和指令单元组成。Pascal一代的每个SM具有64个单精度计算单元（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）组成。FP32和FP64单元的数量和比例取决于某一代GPU。因此，FP64的性能不能直接从FP32单元的数量来推断。例如，麦克斯韦一代提供的FP64到FP32单元的比值仅为1/32，而开普勒(1/3)、帕斯卡(1/2)和Volta (1/2)这一比值要高得多。

Up to CUDA 8, a warp consisting of 32 contiguous CUDA threads is processed simultaneously on an SM in lock-step manner. Thus all 32 compute units have to perform the same operation at the same time similar to the Single Instruction Multiple Data (SIMD) paradigm. In contrast to traditional SIMD architectures, the compute units may access non-contiguous memory or mask instructions allowing for the branching of control ﬂow. The latter is called **branch divergence** and should be avoided since the hardware executes the branches one by another. This slightly more ﬂexible computation model was coined by NVIDIA as Single Instruction Multiple Thread (SIMT). Strictly speaking one can state that SIMD is a subset of the SIMT model. In this context, a warp can be considered as SIMD vector unit with 32 SIMD lanes. Thus we can easily express SIMD algorithms in terms of the CUDA programming language. Note that CUDA 9 shifts the traditional paradigm of warp-centered programming to cooperative groups which are a generalization of the warp concept. As a consequence, warps in CUDA 9 might not be executed in lock-step anymore having severe implications for implicit synchronization. At the time of writing (Summer 2017), we cannot predict the impact of cooperative groups on future code development. However, be aware of this major paradigm shift since it might become a game changer.

直至CUDA 8，由32个连续CUDA线程组成的Warp在SM上以锁步方式同步进行处理。因此，所有32个计算单元必须在同一时间执行类似于单指令多数据(SIMD)范式的相同操作。与传统SIMD架构相比，计算单元可以完成非连续访存或掩码指令，用于完成控制流的分支。后者称为分支分化，要避免分支分化，因为硬件一个接一个地执行分支。这个稍微灵活一些的计算模型是由NVIDIA创建的，称为单指令多线程(Single - Instruction multi - Thread, SIMT)。严格地说，SIMD是SIMT模型的子集。在这种情况下，Warp可以被认为是包含32个SIMD 通道的SIMD矢量单位。因此，我们可以很容易地用CUDA程序语言来表达SIMD算法。请注意，CUDA 9将传统的以Warp为中心的编程范式转换为合作组的形式，它是对warp概念的推广的。在撰写本文时(2017年夏季)，我们无法预测合作组对未来代码开发的影响。但是，请注意这一重大范式转换，因为它可能会改变游戏规则

The two register ﬁles of a Pascal SM can store up to 65,536 32-bit variables. This rather high number of 1024 registers per compute unit is crucial for an efﬁcient scheduling of lightweight threads. If a warp runs out of work, e.g, when waiting for data, the scheduler can switch to another warp without dumping or loading the corresponding registers. Consequently, the number of maintained threads may easily exceed the amount of available compute units. Rapid switching (warp streaming) can be used to effectively hide the latency of global memory behind computation. Similarly to the scheduling of blocks, we have no control over the execution order of warps. However, you can rely on two properties up to CUDA version 8. First, all threads within a warp are executed simultaneously. Second, all warps and thus all blocks have ﬁnished their computation after termination of the kernel. Only the latter is true for CUDA 9.

每个Pascal SM的两个寄存器文件最多可以存储65,536个32位变量,对于轻量级线程的高效调度来说，每个计算单元1024个寄存器的数量非常高。如果某个warp不再处于执行状态，比如，在等待数据，调度器可以切换到另一个warp，而无需转储或加载相应的寄存器。因此，维护线程的数量很容易超过可用计算单元的数量。快速切换(warp流)可以有效地隐藏计算之后的全局内存访问延迟。与块的调度类似，我们无法控制Warp的执行顺序。但是，在CUDA 8之前，可以依赖两个属性，第一，所有线程都是同时执行的。其次，在kernel终止之后，所有的warp和于此对应的Block都完成了它们的计算，**只有后者适用于CUDA 9。**

The units of an SM can access up to 64 KB of fast on-chip memory for inter-thread communication and caching. Concluding, a modern CUDA-enabled GPU consists of a few thousand cores (see Table 7.1) which can execute tens of thousands of threads. Hence, parallelism has to be organized on a ﬁne-grained scale compared to coarse-grained parallelization schemes on multi-core architectures. Besides the aspect of massively parallel computation, we have to take care of the additional constraints imposed by the SIMT computation model in order to efﬁciently utilize modern GPUs.

SM的所有计算单元可以访问64 KB的快速片上内存，用于线程间通信和缓存。一个支持CUDA的现代GPU由数千个内核组成（见表7.1）， 可以执行数万个线程，因此，与多核架构上的粗粒度并行化方案相比，并行性必须在细粒度级别上进行组织。除了大规模并行计算因素之外，为了有效地利用现代GPU，我们还必须考虑SIMT计算模型带来的附加约束。

7.3 MEMORY ACCESS PATTERNS (EIGENFACES)

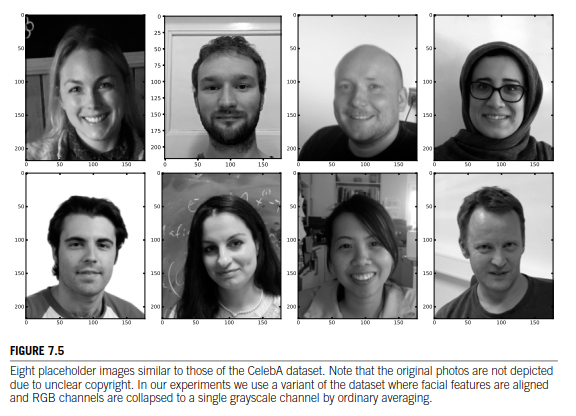
After having revisited the hardware architecture of a typical CUDA-capable GPU, we can now start to write some more useful code. Our task is the CUDA implementation of **Principal Component Analysis (PCA)**, a frequently used dimensional reduction technique in the ﬁeld of machine learning. The applications of PCA are manifold and range from lossy audio/video compression in signal processing over the computation of intrinsic coordinate systems of rigid bodies in physics to the determination of latent variables in data mining. Throughout this section we process the *CelebA* dataset [16] consisting of 202,599 aligned celebrity faces (see Fig. 7.5). In particular, we compute numerous quantities such as the mean celebrity face, the centered data matrix, the corresponding covariance matrix, and an eigenvector basis which can be used for lossy image compression.

From a programming point of view you will learn how to transfer data between host and device, the proper addressing of global memory using coalesced access patterns and the manual caching of redundant data in shared memory. After having ﬁnished this section, you will be able to write basic CUDA programs concurrently processing gigabytes of data. Furthermore, we will discuss when a CUDA parallelization of your program is beneﬁcial in terms of runtime but also cases where a multithreaded CPU implementation might be the better choice.

7.3**内存访问模式(**特征脸**)**

在对支持CUDA功能的典型GPU的硬件架构回顾之后，现在我们可以开始编写一些更有用的代码。我们的任务是用CUDA实现主成分分析（PCA），PCA是一种在机器学习领域经常使用的降维技术。其应用广泛，从信号处理中的有损音频/视频压缩，物理学中刚体固有坐标系的计算，到数据挖掘中潜在变量的确定。在这一节中，我们将处理包含202599张明星脸部图片的 *CelebA*数据集【16】， (见图 7.5)。特别是，通过计算我们得到了许多（**量**），比如名人图片的均值，中心化的Data Matrix，对应的协方差矩阵，和一种可用于有损图像压缩的特征向量基。

从编程的角度，将学习到如何在主机和设备之间传输数据，如何使用合并访存模式完成全局内存寻址，如何手动缓存共享内存数据。在完成这些以后，就能够编写可以并发处理GB级数据的基本CUDA程序了。此外，考虑运行时和多线程CPU实现可能是更好选择的情况下，我们还将讨论什么时候程序的CUDA并行化才是有益的。



**数字7.5**

八个占位符图像与CelebA数据集的图像类似。 请注意，由于版权不明确，原始照片未被描绘。 在我们的实验中，我们使用数据集的变体，其中面部特征被对齐，并且RGB通道通过普通平均折叠成单个灰度通道。

**COMPUTATION OF THE MEAN CELEBRITY FACE**

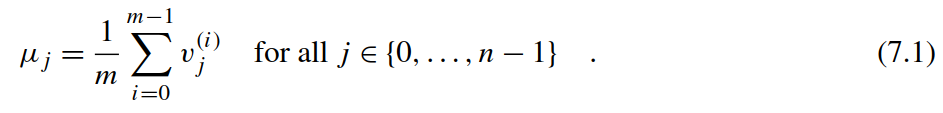
The 202,599 images of the CelebA dataset are stored as RGB-valued matrices of shape 178 × 218. For the sake of simplicity we have collapsed the three color channels to grayscale by computing the pixel-wise mean of the red, green, and blue intensities. Furthermore, the images have been down sampled by a factor of roughly four in a preprocessing phase. The resulting images still need more than one gigabyte of memory since *(*45 × 55*)* × sizeof(float)× 202*,*599 *>* 1*.*8 GB. Nevertheless, the algorithms and source code that we discuss perfectly work on the original resolution too (e.g. when utilizing a GPU with 32 GB of VRAM). Moreover, an extension to color images is straightforward by treating the pixels in each color channel independently.

In the following, we interpret each image as a ﬂattened vector *v(i)* ∈ R*n* where the *n* = 45 · 55 = 2475 pixels are indexed in row-major order. Loosely speaking, we forget the shape of the matrix and consecutively map each pixel onto a slot of the vector. Afterwards, the *m* = 202,599 vectors *v(i)* are stored in the rows of a data matrix *D* = *v(*0*),v(*1*),...,v(m*−1*)* ∈ R*m*×*n* such that *Dij* = *vj(i)* denotes the *j*-th dimension (pixel) of the *i*-th vector (image). We subsequently determine the mean image *μ* by computing the normalized pixel-wise sum of intensity values over all images:

包含202599张图像的CelebA数据集， 存储格式为RGB值矩阵，尺寸为178×218。为了简单起见，通过计算红、绿、蓝通道的像素平均值，我们将RGB三通道分解成灰度图。133 811 86383

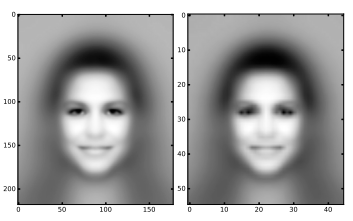
此外，在预处理阶段，图像的下采样率约为4。处理后的结果图像仍然需要超过1G 比特的内存，（ 45 × 55）× sizeof(float)× 202*,*599 *>* 1*.*8 GB。不过，我们讨论的算法和源代码也能很好地处理原始分辨率（比如，使用配有32GB显存的GPU）。此外，直接独立处理每个颜色通道中的像素也是彩色图像的一种扩展。

接下来，我们把每幅图像解释成一个平坦的向量，*v(i)* ∈ R*n* ，*n* = 45 · 55 = 2475，2475个像素以行主序建立索引。大致来说，我们忽略矩阵的形状，并连续地将每个像素映射到向量上。然后，*m* = 202,599 向量，存储在数据矩阵D的行中，*D* = *v(*0*),v(*1*),...,v(m*−1*)* ∈ R*m*×*n，*例如， *Dij* = *vj(i)* 为第i个向量（图像）的第j维（像素）。通过计算所有图像强度值的归一化像素和，最终确定图像均值*μ*



The resulting mean vector *μ* is shown in Fig. 7.6. Note that this one-dimensional representation of higher-dimensional data generalizes to any dataset consisting of *m* ﬁxed length vectors in R*n*. However, the slots of the vectors *v(i)* should be loosely correlated in order to gain meaningful results. Thus you cannot expect this approach to work well on unaligned data without using an approximate alignment of facial features or employing translation-invariant representations. Unaligned data could be tackled with convolutional neural networks [20] or image registration techniques [10].

得到的均值向量*μ*见图7.6. 这种高维数据的一维表示可以推广到由m个固定长度向量组成的任何数据集，然而，为了得到有意义的结果，向量v(i)的槽应该是松散相关的。因此，如果不使用面部特征的近似对齐或使用平移不变表示，就不能期望这种方法能够很好地处理未对齐数据。可以使用卷积神经网络[20]或图像配准技术[10]来解决未对齐的数据。

****

**FIGURE 7.6**

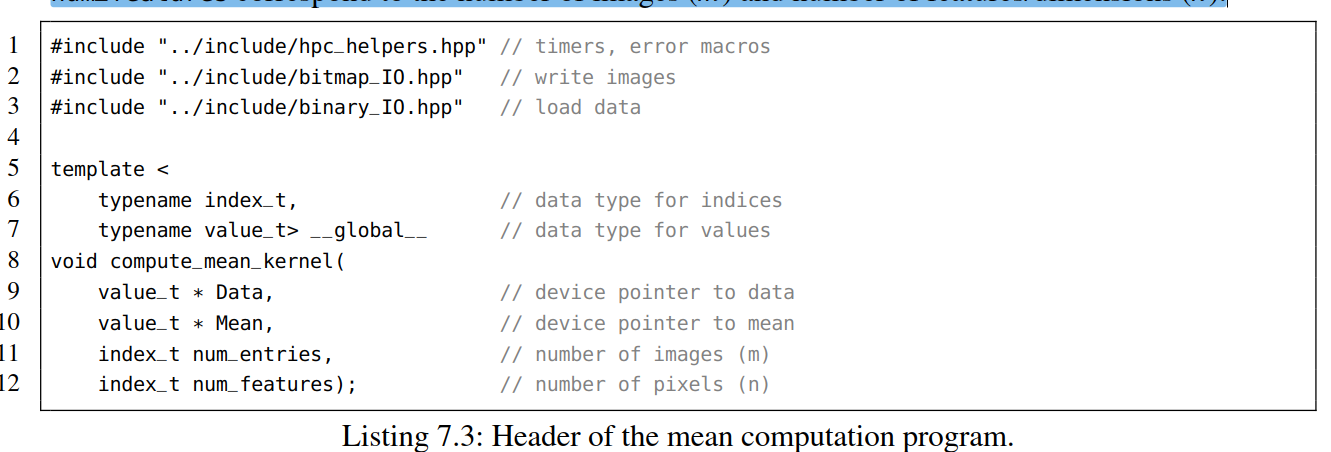
The mean celebrity face of the CelebA dataset in the original resolution (left panel: 178 × 218 pixels) and its downsampled variant (right panel: 45 × 55 pixels).

**数字7.6**

图7.6 CelebA 数据集中明星脸的均值 （左图：178 × 218像素) 和下采样后（右图：45 × 55 像素）

Let us start coding. Initially, we include some useful headers which are distributed with this book (see Listing 7.3). The hpc\_helper.hpp ﬁle contains convenient macros for measuring execution times (TIMERSTART(label) and TIMERSTOP(label)) and for reporting errors (CUERR). Error checking is important since CUDA tends to fail silently after incorrect memory accesses or unsuccessful memory allocations while the host code continues its work. Thus you will observe the frequent use of the CUERR macro in our code. The remaining two headers provide functions for the loading of binary ﬁles and the writing of images in Microsoft bitmap format. Subsequently, we deﬁne a template kernel for the mean computation which can be specialized with custom data types for the indexing (usually uint32\_t or uint64\_t) and representation of ﬂoating point values (usually float or double). The device pointers Data and Mean are used to address the data matrix and mean vector. The integers num\_entries and num\_features correspond to the number of images (*m*) and number of features/dimensions (*n*).

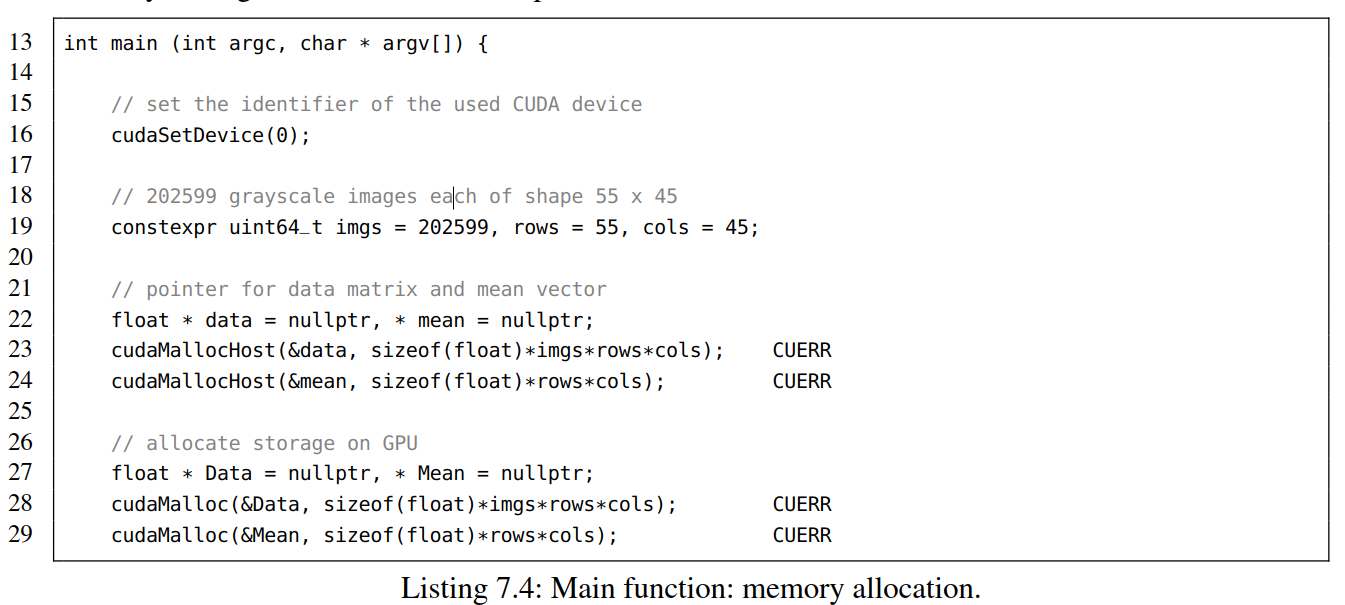
让我们开始编码，我们提供了一些与本书一起分发的有用头文件（见列表7.3），文件hpc\_helper.hpp包含用于方便测量执行时间 (TIMERSTART(label) 和TIMERSTOP(label)) 和报告错误的宏。错误检查很重要，因为在主机代码继续工作时，在不正确的内存访问或内存分配不成功后，CUDA往往会无声地退出执行而主机端代码还会继续执行。因此，将在我们的代码中观察到UERR宏的频繁使用。其余两个头文件分别是提供加载二进制和以微软位图格式写回图像，随后，我们为均值计算定义了一个模板内核，它可以专门用于索引的自定义数据类型(通常是uint32\_t或uint64\_t)和浮点值的表示(通常是float或double)。设备指针Data和Mean分别指向数据矩阵和均值向量。整数num\_entries和num\_features对应于图像的数量（m）和特征/维度的数量（n）。



**列表 7.3: 计算均值头文件.**

Let us proceed with the main function in Listing 7.4. From Lines 15 to 19, we select the CUDA device and subsequently deﬁne three constants specifying the number of images (imgs) and their correspond- ing shape (rows and cols). Next, we allocate the memory for the data matrix and the mean vector on both the CPU and the GPU using the dedicated commands cudaMallocHost and cudaMalloc, respectively. Both functions take as ﬁrst argument the address of a pointer storing the memory location after allocation. Consequently, the argument is a pointer of a pointer in order to allow cudaMallocHost and cudaMalloc to alter the address from nullptr to the corresponding value. The second argument is the number of accessible bytes from that position. The return value denotes whether the allocation was successful and will be handled by the CUERR macro. While host-sided memory could also be allocated with alternative commands, e.g., by a call to malloc or new, we are limited to cudaMalloc when explicitly reserving memory on a speciﬁc device. Note that throughout this chapter we use capitalized variable names for device-sided pointers and lower case letters for the host. Another popular convention, though not used in this book, appends \_h (host) and \_d (device) sufﬁxes to variable names to visually distinguish between address spaces.

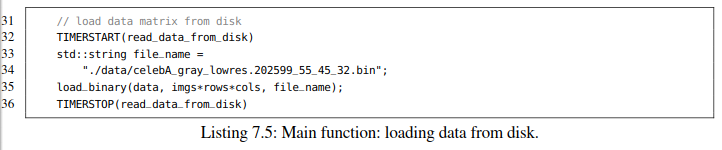
让我们继续讨论列表7.4main函数，从第15行到第19行，我们选择CUDA设备，然后定义三个常量，指定图像的数量（imgs）及其相应的形状(rows and cols).接着，我们使用专用命令cudaMallocHost 和 cudaMalloc 分别为CPU和GPU分配数据矩阵和均值向量的内存。两个函数都将第一个参数作为分配后存储内存位置的指针的地址。因此，参数是指针的指针，以便允许cudaMallocHost和cudaMalloc将地址从nullptr更改为相应的值。第二个参数是从该位置开始要访问的字节数。 返回值表示分配是否成功并将由CUERR宏处理。虽然主机端内存也可以使用替代命令进行分配，例如，通过调用malloc或new 。但当在特定设备上预留内存时，我们仅限于cudaMalloc。请注意，在本章中，我们使用对设备端指针使用大写的变量名 和对主机端指针使用小写的变量。 虽然没有在本书中使用的另一个约定，但它将\_h（主机）和\_d（设备）后缀附加到变量名称，以便在视觉上区分地址空间。



清单7.4：main函数：内存分配

We proceed with the loading of the matrix *D* from disk to the data array on the host using the load\_binary()method from the provided binary\_IO.hppheader ﬁle. The TIMERSTARTand TIMERSTOP macros determine the execution time. At this point you would manually ﬁll the array using row-major order addressing when processing your own data.

我们继续使用binary\_IO.hppheader文件提供的load\_binary（）方法将矩阵D从磁盘加载到主机上的data数组。用TIMERSTART和TIMERSTOP宏来测量执行的时间。 此时，您将在处理自己的数据时使用行主顺序寻址手动填充数组。



清单7.5：主要功能：从磁盘加载数据。

As mentioned before, data that shall be processed on the GPU has to be explicitly transferred from the host to the device. The copying can be achieved with a call to cudaMemcpy as shown in Line 39 of Listing 7.6. The command follows the semantics of the traditional memcpy call, i.e., the ﬁrst argument corresponds to the target address and the second to the source pointer. The third argument denotes the number of transferred bytes and the fourth argument speciﬁes the involved platforms. The rather unhandy constants cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are used to distinguish between copying from host to device and from device to host. Note that these expressions could be signiﬁcantly shortened by deﬁning custom variables or macros in order to reduce redundant typing of boilerplate. As an example, the hpc\_helpers.hppheader ﬁle contains the following two lines:

如前所述，在GPU上处理的数据必须明确地从主机传输到设备。 可以通过调用cudaMemcpy来实现复制，如清单7.6中的第39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地址，第二个参数对应于源指针。 第三个参数表示传输的字节数，第四个参数表示涉及的平台。不可以改变的常量cudaMemcpyHostToDevice 和cudaMemcpyDeviceToHost用于区分从主机到设备的复制以及从设备到主机的复制。 请注意，通过定义自定义变量或宏可以显着缩短这些表达式，以减少样板的冗余类型。 例如，hpc\_helpers.hppheader文件包含以下两行：

#define H2D (cudaMemcpyHostToDevice)

#define D2H (cudaMemcpyDeviceToHost)

The call to cudaMemset in Line 41 overwrites the device vector Meanwith zeros. This is a safety mechanism to avoid the following pitfall that we have frequently observed during programming practicals. Assume you have a working CUDA program but neglect return value checking despite the repetitive warnings of your tutor. Moreover, assume you have introduced a bug in the next version which instantly causes your kernel to fail silently. It is highly probable (almost guaranteed) that the subsequent cudaMemcpy of the result vector (in our case Mean) from the device to the host will transfer the old (correct) data from the previous run still residing in global memory. What could be worse than a defective program that passes unit tests until reboot? Concluding, resetting of results and error checking is mandatory.

第41行中对cudaMemset的调用，用零覆盖设备向量Mean。这是一种安全机制，可以避免我们在编程实践中经常发现的以下陷阱。假设您有一个有效的CUDA程序，尽管您的导师重复警告，但还是忽略了返回值检查。 此外，假设在下一个版本中引入了一个错误，该错误会立即导致内核无声地失败。很可能（几乎可以肯定）从设备到主机的结果向量（在我们的例子中是平均值）的后续cudaMemcpy将传输仍然驻留在全局存储器中的先前运行的旧（正确）数据。 还有什么比有缺陷的程序更糟糕的呢？ 总之，重置结果和错误检查是强制性的。

After having copied the matrix *D* to the device we invoke the kernel in Line 46. Referring to Eq. (7.1), parallelization over *j* (the pixel indices) is advisable since each of the *n* = 55 · 45 = 2475 sums can be evaluated independently.4 When utilizing 32 CUDA threads per thread block we have to invoke at least ⌊*n/*32⌋ many blocks. If the remainder *n*% 32 is different from zero, i.e., *n* is not a multiple of the block size, we have to spawn an additional block to process the few remaining pixels. The described scheme can be summarized in a closed expression for safe integer division:

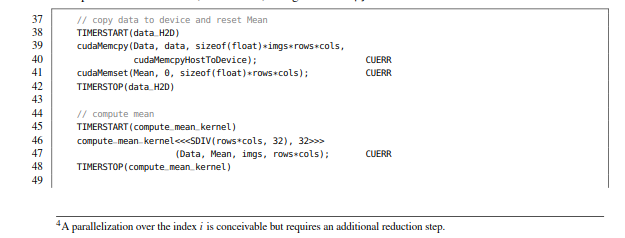


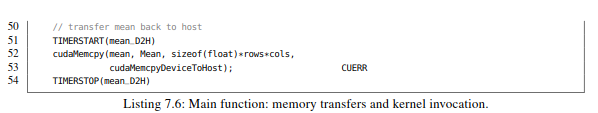
我们在第46行调用内核，将矩阵D复制到设备后。参考公式7.2， 在 j 上并行化（像素的索引）是可取的，因为每个n = 55·45 = 2475求和可以独立计算.4。当每个线程块使用32个CUDA线程时，我们必须调用至少⌊*n/*32⌋多个block。如果余数n％32不为零, 即，n不是Block大小的倍数, 我们必须生成一个额外的块来处理剩下的几个像素。所描述的方案可以用闭合表达式来概括以进行安全整数除法：



The SDIV macro is deﬁned in the hpc\_helper.hppheader ﬁle. After termination of the kernel the result is copied back to the host (see Line 52) using cudaMemcpy.

SDIV宏在hpc\_helper.hppheader文件中定义。 内核终止后，使用cudaMemcpy将结果复制回主机（参见第52行）。



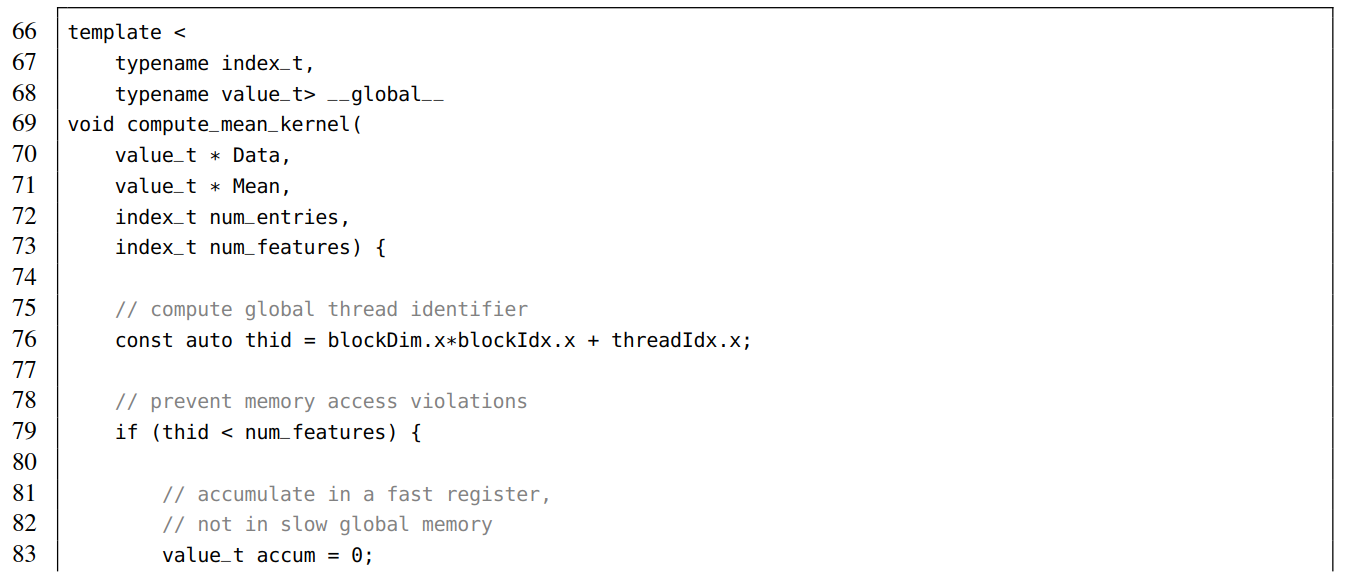


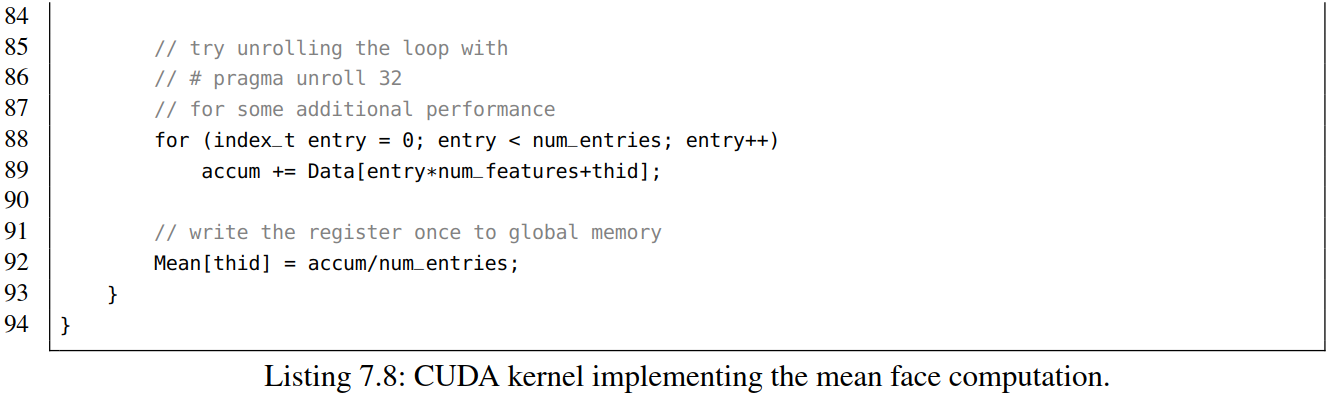
The remaining part of the mainfunction in Listing 7.7 is straightforward. The mean image is written to disk using dump\_bitmapfrom the bitmap\_IO.hppheader ﬁle. Moreover, allocated memory is explicitly freed using cudaFreeHost for the host and cudaFree for the device in order to prevent memory leaks.

清单7.7中的main函数的其余部分很简单。 使用bitmap\_IO.hppheader文件中的dump\_bitmap将均值图像写入磁盘。 此外，使用cudaFreeHost和cudaFree分别为主机和设备显式释放已分配的内存，以防止内存泄漏。

Finally, we discuss the compute\_mean\_kernel implementation in Listing 7.8. First, the global thread identiﬁer is calculated in Line 76 analogically to the Hello World kernel from the introductory section. Second, the range of the thread identiﬁer is checked in Line 79 to avoid potential memory access violations when num\_features is not a multiple of blockDim.x. Third, each thread sums up a column of the Data matrix in a dedicated register accum. Optionally, you can tweak this in Line 86 by giving the compiler a hint to unroll the for-loop in batches of size 32. Finally, we normalize the calculated sum and write it to the Mean array.

最后，我们讨论代码清单7.8中的compute\_mean\_kernel实现。首先，类似于介绍部分中的Hello World内核，76行用来计算全局线程标识符。其次，第79行检查线程标识符的范围，以避免num\_features不是blockDim.x的倍数时发生潜在的内存访问冲突。第三，每个线程在专用寄存器accum中累加数据矩阵的列。或者可以选择在第86行中对此进行调整，方法是给编译器提供一个指导语句，以批量32的大小进行for循环展开。最后，我们将计算的和规范化，并将其写入平均数组。





可以使用以下命令编译代码:

nvcc -O2 -std=c++11 -arch=sm\_61 -o mean \_computing

We need to set the -std=c++11ﬂag which activates support for the C++11 standard since our code uses the auto keyword. Moreover, -arch=sm\_61 speciﬁes the compute capability of the Pascal generation. Please refer to the CUDA programming guide [4] for an extensive overview of conﬁgurable compute capabilities. When executing the program on a Pascal-based Titan X we observe the following output of execution times:

因为我们的代码使用了auto关键字，需要设置-std=c++11标志，它激活了对c++11标准的支持。-arch=sm\_61明确指定Pascal一代的计算能力，有关可配置计算能力的广泛概述，请参阅CUDA编程指南[4]。在基于Pascal的Titan X上执行程序时，我们观察到以下执行时间的输出：

TIMING: 13294.3 ms (read\_data\_from\_disk)

TIMING: 170.136 ms (data\_H2D)

TIMING: 8.82074 ms (compute\_mean\_kernel)

TIMING: 0.03174 ms (mean\_D2H)

TIMING: 155.921 ms (write\_mean\_image\_to\_disk)

It takes roughly 13 seconds to read 1.86 GB of image data from spinning disk (≈ 150 MB/s). The memory transfer from host to device over the PCIe bus is accomplished in roughly 170 ms (≈ 11 GB/s). During the execution of the CUDA kernel we access each pixel of the dataset exactly once and apply one addition. Thus the 9 ms correspond to an effective global memory bandwidth of approximately 208 GB/s and a compute performance of about 52 GFlop/s. The time for transferring the average image (≈ 9.5 KB) back to the host is negligible.

从旋转磁盘(≈ 150 MB/s)读取1.86GB的图像数据大约需要13秒. 通过PCIe总线从主机到设备的内存传输大约在170 ms（≈11 GB/s）内完成。在CUDA内核的执行过程中，访问数据集的每个像素一次，并应用一个加法。大约208 GB/s的全局访存带宽和约52 GFlop/s的计算性能耗时9ms。将均值图像（≈9.5 kb）传回主机的时间可以忽略不计。

Despite the impressive performance of the kernel we utilize merely half of the 480 GB/s theoretical peak bandwidth of the global memory and only 0.5% of the 11 TFlop/s peak performance. Moreover, the amount of *n* = 2475 spawned threads is signiﬁcantly smaller than the number of 3584 available cores (see Table 7.1). A better approach would spawn tens of thousands of threads. The situation gets even worse if we include the reasonably slower memory transfers over PCIe or the time needed for reading the data from disk. As a result, the described algorithm considered as standalone application is not well suited for the GPU since the 52 GFlop/s compute performance can easily be beaten by state-of-the-art multi-core CPUs. Nevertheless, the kernel can be used as subroutine in high-level algorithms which stack several CUDA kernels. Concluding, raw compute performance is meaningless when having too few data to process.

尽管内核的性能令人还挺可观。但我们仅使用了全局访存带宽理论峰值480GB/s的一半，仅使用了峰值性能11 TFlop/s的0.5%。此外，n=2475生成线程的数量明显小于3584可用内核的数量(见表7.1)。更好的方法将产生数万个线程。如果我们包含 PCIe相当慢的内存传输或从磁盘读取数据所需的时间，情况会变得更糟。因此，将所述算法作为独立应用程序不太适合GPU，因为当前最先进的多核CPU很容易击败52 Gflop/s的计算性能。然而，内核可以用作高层算法中的子程序，这些高层算法可以堆叠几个CUDA内核。最后，如果需要处理的数据太少时，原始计算性能也就毫无意义

Finally, let us make an important statement about synchronization. You may have noticed the missing cudaDeviceSynchronize() command after calling the kernel in Line 46. The following memory transfer using cudaMemcpy implicitly synchronizes the device rendering an explicit synchronization redundant. However, asynchronous memory transfers and kernel launches are possible and will be discussed in Section 8.2. This implicit synchronization behavior is also enforced when allocating/freeing memory using cudaMalloc(Host)/cudaFree(Host), setting memory with cudaMemset or a switch bebween the L1/shared memory conﬁgurations using cudaDeviceSetCacheConfig. A complete list can be found in the CUDA programming guide [4].

最后，让我们重点介绍一下同步。在第46行调用内核后，可能已经注意到缺少cudaDeviceSynchronize（）命令。使用cudaMemcpy的内存传输隐式地

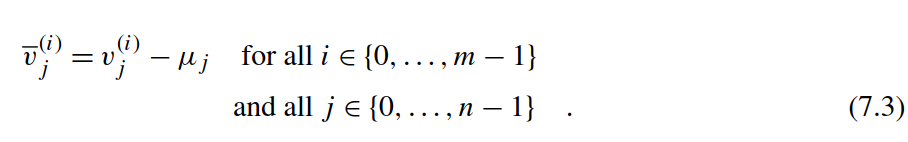
同步设备，从而避免显式同步冗余。异步内存传输和内核启动是可能的，将在8.2节中讨论。当使用cudaMalloc（Host）/ cudaFree（Host）分配/释放内存，使用cudaMemset对内存赋值或使用cudaDeviceSetCacheConfig来设置L1与共享内存配置之切换是，也会强制执行此隐式同步行为。完整列表可以在CUDA编程指南[4]中找到。

**COMPUTING THE CENTERED DATA MATRIX**

计算中心化的Data Matrix

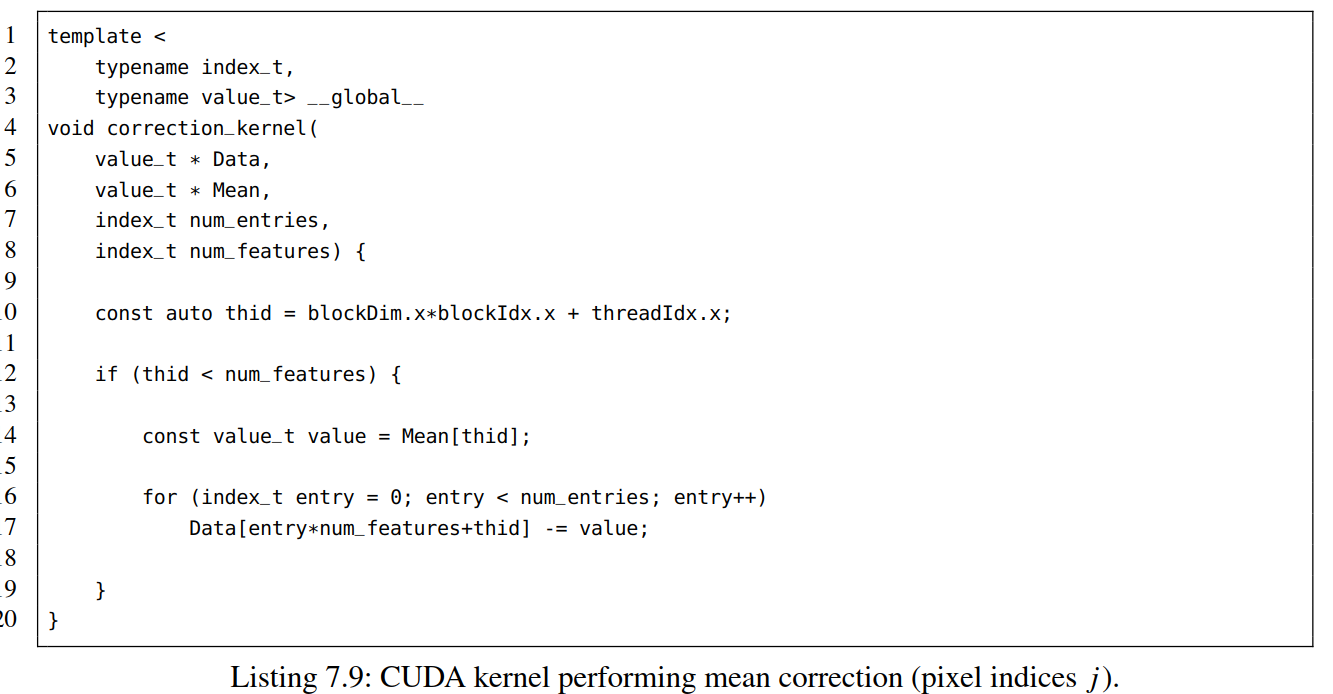
PCA is usually performed on centered data, i.e., we have to subtract the mean vector *μ* from each of the vectors *v(i)*. The components of the centered vectors *v(i)* are obtained as follows:

PCA经常应用在中心化数据，我们必须从*v(i)*中的每个向量减去均值向量μ。中心化向量vectors *v(i)*的分量可以通过一下获得：



As a result, the columns of the centered data matrix *Dij* = *vj(i)* sum up to zero. Conceptually, there are two levels of parallelism. On the one hand, we could parallelize the mean correction over the pixel indices j analogically to the previous subsection. On the other hand, each image *v(i)* can an also be treated independently. We will see that the ﬁrst approach is about eight times faster than the latter. This can be explained by the different memory access patterns of both approaches. However, before we dive into details let us write some code. The ﬁrst kernel parallelizes the centering over the pixel indices and serializes the loop over the image indices (see Listing 7.9).

结果，居中数据矩阵的列总和为零。从概念上讲，有两个级别的并行性。一方面，前一小节类似，我们可以在像素索引 j上 的并行化均值校正。另一方面，每个图像v(i)也可以独立处理。我们发现第一种方法比后者快八倍。这可以通过两种方法的不同访存模式来解释。但是，在我们深入细节之前，让我们编写一些代码。第一个内核将像素索引上的中心并行化，并将循环序列化为图像索引(see Listing 7.9).



清单7.9:执行均值矫正CUDA kernel

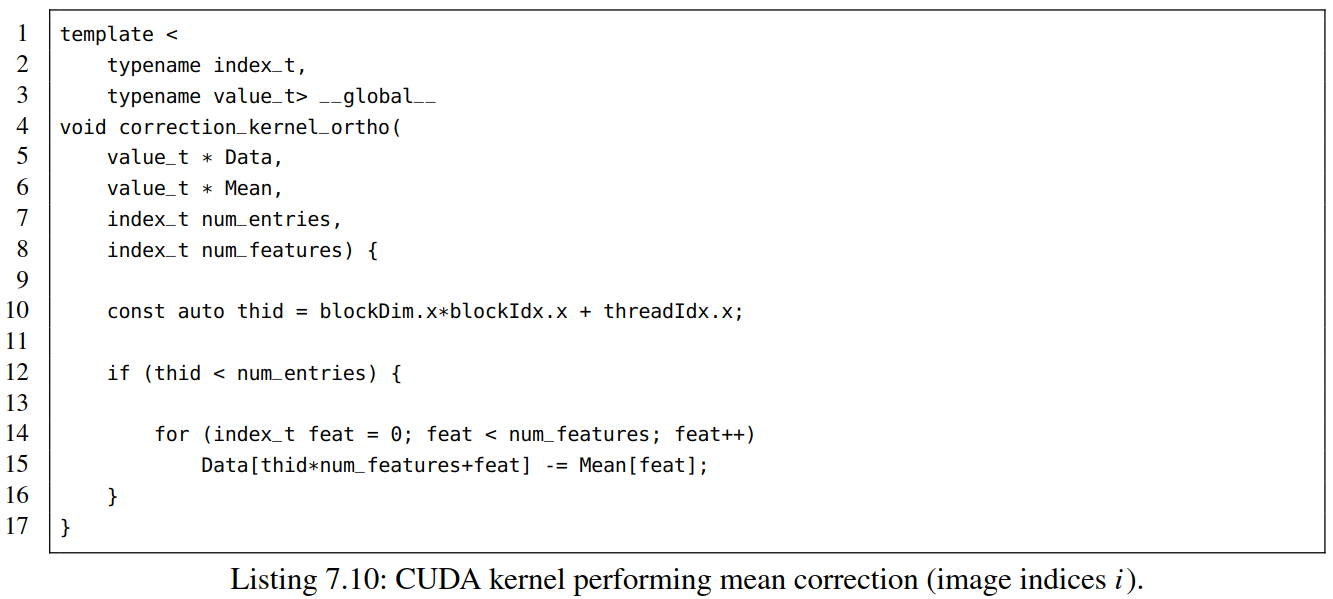
The code is similar to the mean computation kernel. First, the global thread identiﬁer thid is determined in Line 10. Second, we check the range of the thread indices to prevent memory access violations (Line 12). Third, the *j*-th component of the Mean vector is written to the register value in Line 14. Fourth, we subtract the corresponding value from each vector in the loop body. Finally, the kernel is launched with the following call:

代码与平均计算内核类似。首先， 第10行完成全局线程标识符thid计算，第二，我们检查线程索引的范围以防止内存访问冲突（第12行），第三，第14行，Mean向量的第j个分量被写入寄存器value。第四，通过循环体中，每个向量元素减去相应的值。最后，使用以下调用启动内核：

correction\_kernel<<<SDIV(rows\*cols, 32), 32>>> (Data, Mean, imgs, rows\*cols); CUERR

An orthogonal approach spawns one thread for each of the images and serializes the loop over the pixel indices. The corresponding source code is provided in Listing 7.10. Here, we compute again the global thread identiﬁer and subsequently check its range. Finally, the loop is executed over pixel indices.

另外一种通用方法是为每个图像生成一个线程，并串行完成以像素索引的循环。相应的源代码见列表7.10. 在这里，我们再次计算全局线程标识符，然后检查其范围。最后，执行像素索引的循环。



清单7.10：执行均值矫正CUDA kernel（图像索引i）

The corresponding kernel call differs mainly in the number of spawned blocks and threads which now correspond to the number of images.

相应的内核调用主要区别在于根据对应于图像数量所产生的Blocks和线程的数量。

correction\_kernel\_ortho<<<SDIV(imgs, 32), 32>>> (Data, Mean, imgs, rows\*cols); CUERR

When executing both kernels on a Pascal based Titan X we measure roughly 60 ms for the ﬁrst kernel and approximately 500 ms for the orthogonal approach. On ﬁrst sight, this observation seems to be counterintuitive since the number of images exceed the number of pixels by far. Nevertheless, the second kernel cannot beneﬁt from the increased level of parallelism due to its suboptimal memory access pattern. Let us have a look at the for-loop body of the orthogonal approach:

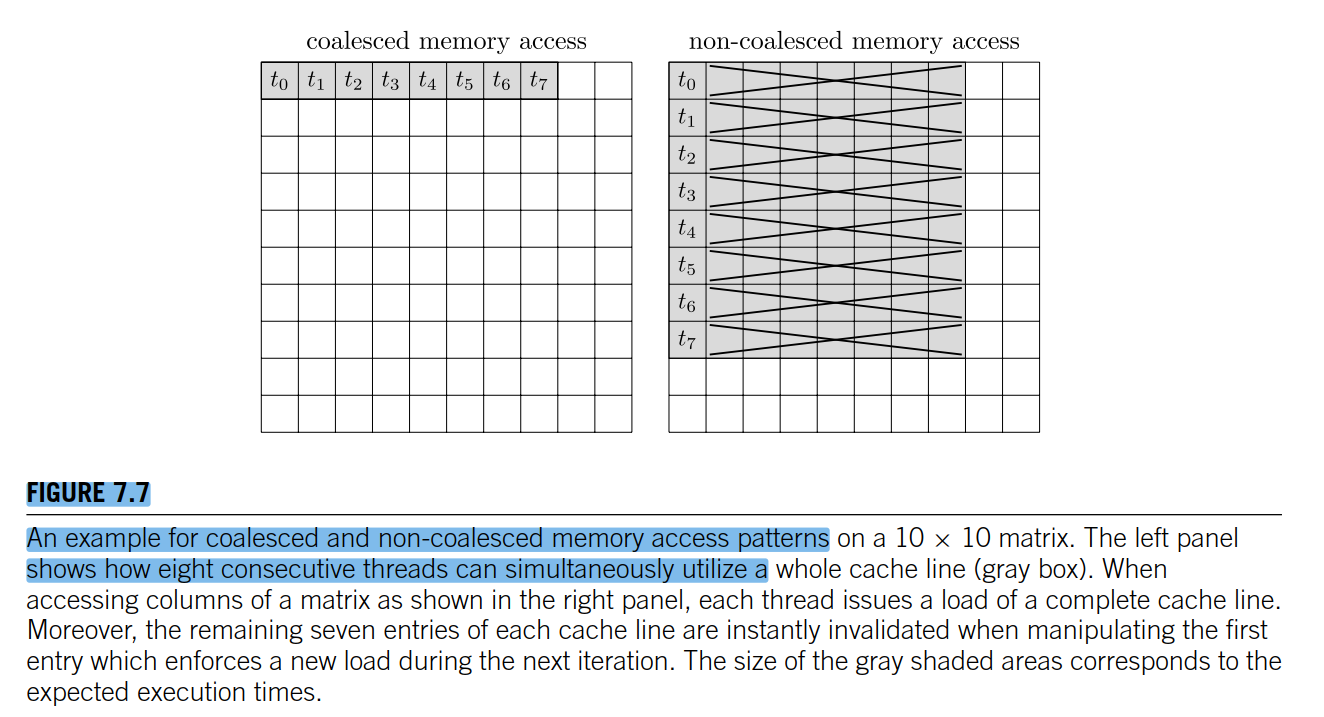
在基于Pascal的Titan X上执行两个内核时，第一个内核执行大约60 ms，正交方法内核大约500 ms。乍一看，结果似乎出人意料，因为图像的数量远远超过了像素的数量。然而，由于并不最优的访存模式，第二个kernel不能从提高的并行度中获益。让我们看一下正交方法的for循环体：

for (index\_t feat = 0; feat < num\_features; feat++)

Data[thid\*num\_features+feat] = some\_value;

The entries of Data are not accessed consecutively although the variable feat enumerates the inner loop and thus seems to address contiguous memory. Recall that 32 threads of a warp are executed simultaneously. Therefore the index thid is altered faster than the variable feat. As a result, we access 32 entries residing in the same column during each iteration of the loop. This causes excessive invalidation of cache lines. By contrast, the access pattern of the ﬁrst kernel, addresses consecutive memory during each iteration.

虽然 变量feat枚举内循环且看起来访存连续，但是 Data访存并不连续。回想一下，32个warp线程同时执行。因此，索引thid的变化速度比变量feat要快。循环体中的每一次迭代就会访问驻留在同一列中的32个条目，这会导致缓存行的过度无效。相比之下，第一个内核的访问模式,每次迭代都是连续访存。



**图7.7**

An example for coalesced and non-coalesced memory access patterns on a 10 × 10 matrix. The left panel shows how eight consecutive threads can simultaneously utilize a whole cache line (gray box). When accessing columns of a matrix as shown in the right panel, each thread issues a load of a complete cache line. Moreover, the remaining seven entries of each cache line are instantly invalidated when manipulating the ﬁrst entry which enforces a new load during the next iteration. The size of the gray shaded areas corresponds to the expected execution times.

10×10矩阵上的合并访存和非合并访存模式示例。左侧面板显示了八个连续线程如何同时使用整个缓存行（灰色框）。如右面板所示，当访问矩阵的列时，每个线程发射一个完整缓存行的负载。此外，当操作在下一次迭代期间强制执行新加载的第一个条目时，每个高速缓存行的剩余七个条目立即无效。灰色阴影区域的大小对应于预期的执行时间。

for (index\_t entry = 0; entry < num\_entries; entry++)

Data[entry\*num\_features+thid] = some\_value;

Let us revisit what we have seen. Whenever consecutive threads access contiguous memory, more precisely when a sequence of threads *(tk,tk*+1*,...,tk*+31*)* simultaneously reads or writes contiguous memory positions *(pl,pl*+1*,...,pl*+31*)* for ﬁxed values of *k* and *l*, we call this pattern **coalesced** and otherwise **non-coalesced** (see Fig. 7.7). Coalesced access patterns are highly advisable in order to saturate the bandwidth to global memory. Non-coalesced reads or writes are often as slow as random accesses and should be avoided at any cost. Several techniques can be applied to effectively prevent random access, e.g. the reordering of indices, transposition of input data (which is an exercise), or the execution of highly irregular access patterns on faster memory types. The latter will be demonstrated in Section 7.4. Finally, let us make a concluding remark related to coalesced addressing in multidi- mensional blocks. Usually, the local thread identiﬁer threadIdx.x is altered faster than threadIdx.y and threadIdx.z. Therefore variables that depend on threadIdx.x (in our case thid) should always manipulate the least signiﬁcant bits of the indexing scheme. As an example, the coalesced scheme

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

is typically signiﬁcantly faster than its non-coalesced counterpart

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

keep in mind that this subtle distinction may slow down your algorithm by roughly one order-of- magnitude. A similar behavior can be observed on traditional CPUs where row-major addressing, which corresponds to coalesced memory access, reasonably outperforms column-major indexing (see Section 3).

让我们重温一下我们所看到的。每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）同时读取或写入固定值k和l的连续内存位置（pl，pl+1，…，pl+31）时，我们称此模式为“合并访存”或“非合并访存” (见 Fig. 7.7)。为了全局存储带宽饱和，建议使用合并的访问模式

非合并读或写通常和随机访问一样慢，应该尽量避免。有几种技术可以有效地防止随机访问，例如，索引的重新排序、输入数据的换位（这是一个练习）或在更快的内存类型上执行高度不规则的访问模式。后者将在第7.4节中演示。最后，让我们做一个关于多维block中合并寻址的总结。通常，本地线程标识符threadidx.x的更改速度比threadidx.y和threadidx.z快。因此，依赖于threadidx.x的变量（在我们的例子中是thid）应该总是操作索引方案中最小的显著位。例如，合并方案

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

通常比非合并方案快很多

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

请记住，这种微妙的区别可能会使您的算法减慢大约一个数量级。 在传统的CPU上可以观察到类似的行为，其中行主序寻址（对应于合并的存储器访问）理所当然地优于列主要索引（参见第3节）。

COMPUTATION OF THE COVARIANCE MATRIX （20190304）

计算协方差矩阵

|  |
| --- |
|  |

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。

第八章： 高级CUDA编程

摘要：

In the recent past, CUDA has become the major framework for the programming of massively parallel accelerators.

在最近的过去，CUDA已成为大规模并行加速器编程的主要框架。

NVIDIA estimates the number of CUDA installations in the year 2016 to exceed one million.

NVIDIA估计2016年CUDA安装数量将超过100万。

Moreover, with the rise of Deep Learning this number is expected to grow at an exponential rate in the foreseeing future.此外，随着深度学习的兴起，在不远的将来，预计这个数字将会以指数速度增长。

Hence, extensive CUDA knowledge is a fundamental pursuit for every programmer in the ﬁeld of High Performance Computing.

因此，深厚的CUDA知识是高性能计算领域中每个程序员的根本追求。

The previous chapter focused on the basic programming model and the memory hierarchy of modern GPUs.

前一章，重点介绍了基本编程模型和现代GPU的内存层次结构。

We have seen that proper memory utilization is key to obtain efﬁcient code.

我们已经发现合理利用内存是获得高效代码的关键。

While our examples from the previous chapter focused on thread-level implementations, we investigate now warp-level parallelization and the efﬁcient use of atomic functions.

上一章的示例我们主要关注在线程级别的实现，现在我们会研究warp级并行和原子函数的有效使用。

Both techniques in combination enable further code optimization

这两种技术组合能够进一步优化代码。

Moreover, we discuss overlapping of communication and computation in single-GPU and multi-GPU scenarios using streams.

此外，我们讨论使用流在单GPU和多GPU场景中的通信和计算的重叠。

We conclude the chapter with a brief discussion of CUDA 9 and its novel features.

我们在本章结束时会简要讨论CUDA 9及其新颖的特性。

Keywords

CUDA, GPU, Warp intrinsics, Atomic operations, Z-normalization, Compare and swap loop, Parallel

preﬁx scan, Multiple GPUs, CUDA streams, Asynchronous memory transfer, Dynamic parallelism,

CUDA-aware MPI

关键词

CUDA, GPU, Warp intrinsics, Atomic operations, Z-normalization, Compare and swap loop, Parallel

preﬁx scan, Multiple GPUs, CUDA streams, Asynchronous memory transfer, Dynamic parallelism,

CUDA-aware MPI

CONTENTS目录

8.1 Warp Intrinsics and Atomic Operations (Parallel Reduction) ------------------- 288

8.1 Warp Intrinsics及原子操作(并行规约) ------------------- 288

Segmented Parallel Reduction 分段并行规约.............................................................................. 288

Global Parallel Reduction 全局并行规约................................................................................... 292

Arbitrary Atomic Operations任意原子操作................................................................................. 294

Outlook 展望观点见解......................................................................................................... 295

8.2 Utilizing Multiple GPUs and Streams (Newton Iteration) ....................................................... 296

8.2 多GPU及流的利用(牛顿迭代法) ....................................................... 296

Newton’s Iteration 牛顿迭代............................................................................................ 296

Harnessing Multiple GPUs 利用多GPU .................................................................................. 299

Interleaving Communication and Computation 通信与 计算交替........................................................... 302

Streamed Computation on Multiple GPUs .多GPU的流式计算............................................................... 305

8.3 Outlook展望.................................................................................................................. 307

Uniﬁed Memory统一内存............................................................................................... 307

Dynamic Parallelism.动态并行......................................................................................... 308

Cooperative Groups Groups间协作 ........................................................................................... 308

Tensor Cores . 张量核.................................................................................................. 309

Distributed Computation on GPU Clusters ................................................................ 309

8.4 Additional Exercises..课外习题................................................................................................ 309

References.参考.................................................................................................................. 312

**8.1 WARP INTRINSICS AND ATOMIC OPERATIONS (PARALLEL REDUCTION)**

8.1 Warp Intrinsics及原子操作(并行规约)

Up to this point, we have used registers to exclusively store state variables such as indices or intermediate values.

到目前为止，我们已经使用寄存器来专门存储状态变量，例如索引或中间值。

This section shows you how to use the enormous amount of registers per SM as data storage.

本节介绍如何将每个SM的大量寄存器用作数据存储。

Historically, registers have been designed as thread-local memory that can exclusively be manipulated in the scope of a single thread.

传统上，寄存器常被设计为线程本地存储器，可以在单个线程的范围内进行操作。

Since 32 threads within a warp are concurrently executed in lock-step manner, one might want to share information between them.

由于warp中的32个线程以锁步方式同时执行，因此人们想在它们之间共享信息。

The traditional approach would employ shared memory for inter-thread communication.

传统方法是将使用共享内存实现进行线程间通信。

Starting from the Kepler generation, CUDA introduced so-called warp intrinsics to achieve the same.

从Kepler开始，CUDA引入了所谓的warp intrinsics来实现这样的目标。

They offer two advantages: ﬁrstly they communicate more efﬁciently and secondly we can save valuable space in shared memory which can be now used to cache other quantities.

它们具有两个优点：首先线程间可以更有效地进行通信; 其次，我们可以在共享内存中节省出宝贵的空间，这些空间可以用来缓存其他数量。

Another important technique allowing for the concurrent access of memory without race-conditions is the use of atomic operations. Throughout the rest of this section we will demonstrate both techniques in detail.

另一个重要技术是使用原子操作，它可以实现在没有竞争条件的情况下并发访问内存。 在本节的其余部分中，我们将详细介绍这两种技术。

SEGMENTED PARALLEL REDUCTION 分段并行规约

In the following, we develop a simple algorithm for parallel reduction based on warp intrinsics.

在下文中，我们基于warp内在函数开发了一种简单的并行规约算法。

Assume you want to process a data matrix  storing m one-dimensional time series of ﬁxed length n.

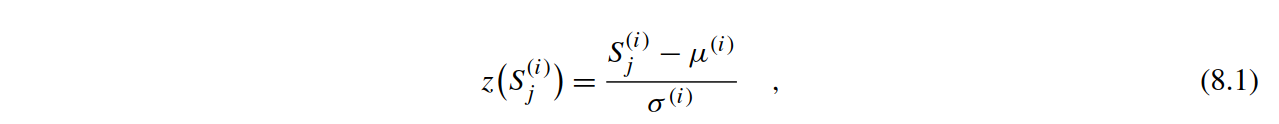
假设您要处理数据矩阵 存储m个一维固定长度为n的时间序列。

The index i enumerates the time series and j denotes the time ticks.

索引i枚举时间序列，j表示时间滴答。

A popular preprocessing technique in the ﬁeld of time series data mining is z-normalization which adjusts the mean and variance of each sequence.

z-归一化是时间序列数据挖掘领域一种流行的预处理技术，z-归一化用来调整每个序列的均值和方差。



where μ(i) and σ(i) are the individual means and standard deviations of each of the m time series.

Note that a similar technique is applied during batch normalization of deep neural networks [12].

Afterwards, each time series has vanishing mean and unit variance. Z-normalization is usually applied

to remove offsets and variability in amplitudes in order to allow robust classiﬁcation in a subsequent

phase. A traditional ﬁne-grained parallelization approach would process one time series per thread

block. The corresponding sums could be evaluated in shared memory using parallel reduction