CUDA Fortran 高效编程实践

科学家和工程师特供

小小河 译

CUDA Fortran for Scientists and Engineers

Best Practices for Efficient CUDA Fortran Programming

Gregory Ruetsch and Massimiliano Fatica

法律条款

1. 对本文档的任何使用都被视为完全理解并接受本文档所列举的所有法律条款。
2. 此处的作者特指本文的译者。本文档的所有权利归作者所有，作者保留所有权利。
3. 未经作者书面同意，禁止商业使用是指在商业活动中或有商业目的活动中使用。商业使用形式包括但不限于存储、出版、复制、传播、展示、引用、编辑。
4. 本文档允许以学术研究、技术交流为目的使用。复制、传播过程中不得对本文档作任何增减编辑，引用时需注明出处。
5. 实施任何侵权形为的法人或自然人都必须向作者支付赔偿金，赔偿金计算方法为：

   赔偿金 = 涉案人次×涉案时长（天）×涉案文档页数×受众人次×100 元人民币，

   涉案人次、涉案时长、涉案文档页数、受众人次小于 1 时，按 1 计算。
6. 对举报侵权行为、提供有价值证据的自然人或法人，作者承诺奖励案件实际赔偿金的 50%.
7. 涉及本文档的法律纠纷由作者所在地法院裁决。
8. 本文档所列举法律条款的最终解释权归作者所有。

译者序

2007 年以来，以 nVidia GPU 为代表的加速器并行计算风起云涌，带有加速器的超级计算机在 TOP500 中的份额逐年增加，支持加速器的主流应用软件也呈爆炸式增长，研究加速器计算的技术人员数以百万计，世界范围内的大学、研究机构竞相开设相关课程。

目前流行的 GPU 通用编程语言是 CUDA C 和 OpenCL. 它们均是 C/C++语言的扩展，因此可以方便地将 C/C++代码移植到 GPU 上。但对于科学与工程计算中的重要编程语言 Fortran，无法直接地改写为 CUDA C 或 OpenCL。很多久经考验的应用程序都是用 Fortran 开发，如果完全改写，工作量巨大，而且有不可估量的稳定性风险。

为使 Fortran 应用能够使用 GPU 加速，The Portland Group 设计了 CUDA Fortran 语言，并在自家的 PGI 编译器中支持。气象、理论物理等领域的应用经过简单的改造，就能够利用 GPU 的强大计算能力。

本书英文版名为《CUDA Fortran for Scientists and Engineers Best Practices for Efficient CUDA Fortran Programming》，由 GPU 厂商 nVidia 公司的两名技术人员撰写，介绍了 2013 年 11 月之前的最新产品与技术。译者本着技术研究、学术交流的目的利用业余时间翻译本文，方便中文读者阅读。

本人水平有限，错误疏漏在所难免，欢迎批评指正。原始英文版请购买正版图书。本文的最新版请到 www.bytes.me/cudafortran 或技术交流 QQ 群中下载。联系译者请发送电子邮件至 hpcfan@foxmail.com.

CUDA Fortran 技术交流 QQ 群：284876008，欢迎加入，共同进步。

2014 年 4 月 22 日

第1章 引述

1.1 GPU 计算简史

并行计算已经以多种形式发展了几十年。在早期阶段，它基本上局限于昂贵大型机器的用户。今天，情况大不相同。几乎所有的消费级别台式机和笔记本电脑都具有多核中央处理器(CPU)。甚至大多数手机和平板电脑都有多个核心。多核 CPU 几乎无处不在的主要原因是 CPU 制造厂商无力推高单核时钟速度以提高性能。因此，大约从 2005 年开始，CPU 设计时不再“纵向”增加时钟速度，转而“横向”发展为多个核心。尽管 CPU 已经有几个至几十个核心，但与图形处理单元(graphics processing unit,GPU)的核心数量比起来，这个规模的并行仍显得苍白无力。例如，英伟达(nVidia)Tesla® K20X 包含 2688 个核心。自 1990 年代中期开始，GPU 就拥有高度并行的架构，这是因为图像处理是一个与生俱来的并行任务。

使用 GPU 作通用计算，经常被称为 GPGPU[1]，起初是一个挑战性工作。人们不得对应用编程接口(API，下称“接口”)进行编程，事实证明接口对能够映射到 GPU 上的算法类型要求非常严格。即便可以映射，它要求的编程技术对计算图形学专业之外的科学家和工程师来说也相当困难，且不直观。正因如此，GPU 在科学和工程计算领域接受缓慢。

2007 年，随着英伟达 CUDA 架构的出现，情况发生了变化。CUDA 架构不但包含英伟达 GPU 硬件，而且包含一个软件编程环境，该环境去除了 GPGPU 的推广障碍。2007 年 CUDA 一经推出，就获得了广泛接受，到 2010 年 11 月，TOP500 榜单的前五台超级计算机有三台都采用 GPU. 在 2012 年 11 月的 TOP 500 榜单中，世界上最快的超级计算机由 GPU 提供动力。CUDA 被快速接受的一个原因是它的编程模型非常简单。CUDA C 是 CUDA 架构的第一个接口，它本质上是在 C 中添加一些扩展，以便将算法的某些部分卸载到 GPU 上运行。它混合使用 CPU 和 GPU，从而 GPU 承担的计算量可以逐渐增加。

在 2009 年下半年，波兰集团(The Portland Group®，PGI®)和英伟达共同开发了 CUDA Fortran 编译器。就像 CUDA 是 C 的扩展一样，CUDA Fortran 本质上是在 Fortran 90 上添加一些扩展，以便用户在计算中扩大 GPU 的效力。有人已经写了许多书、文章和其它类型的文档来帮助开发高效的 CUDA C 应用程序(例如 Sanders and Kandrot, 2011; Kirk and Hwu, 2012; Wilt, 2013)。因为比较新，所以 CUDA Fortran 在代码开发中用得相对较少。编写高效 CUDA C 代码的许多资料都可以轻松平移到 CUDA Frotran 上，因为底层架构是相同的，但仍然需要资料指导如何编写高效 CUDA Fortran 代码。这里有两个原因。第一，尽管 CUDA C 和 CUDA Frotran 类似，但是它们仍然会些不同之处，会影响代码的编写方式。这并不奇怪，因为 CPU 代码用 C 编写，随着项目的增长，Fortran 将承担不同的角色。其次，CUDA C 的一些特性 CUDA Fortran 里没有，例如，某些类型的纹理。相反地，CUDA Fortran 的一些特性 CUDA C 中没有，例如用来标记数据驻留在 GPU 上的变量属性 `device`.

本书的目标读者是希望以并行为工具来完成其它工作的人，而不是为阅读而阅读的人。本书的目标是使读者掌握编写适当优化的 CUDA Fortran 代码所必须的基本技术，充分利用英伟达计算硬件的优势。采用这种方法而不是讲授如何挖出硬件的最后一点性能，原因是假设读者将 CUDA Fortran 作为一个工具，而不是研究的对象。这样的用户通常看重清晰、易维护的代码，编写简单，在多代 CUDA 硬件和 CUDA Fortran 软件上都有不错的性能表现。

但是，付出与性能之间如何权衡？开发者将最终决定投入多少努力来优化代码。在作决定的过程中，

[1] General-Purpose GPU. -- 译者注。

需要知道去除各种瓶颈之后能得到什么样的回报，需要付出什么样的努力。本书的目标之一就是帮助读者培养评估这种投入与回报的直觉。为达到此目标，书中将讨论用 CUDA Fortran 编写科学和工程应用中见算法时遇到的瓶颈。尽可能地展示多样的工作环境，并分析每一种优化措施对性能的影响。

1.2 并行计算

在开始编写 CUDA Fortran 代码之前，就 CUDA 与其它并行编程模型的相通之处说两句。熟悉和理解其它并行编程模型不是阅读本书的前提要求，但对有并行编程经验的读者来说，本节有助将 CUDA 归类。

前面提到过 CUDA 是一个混合计算模型，应用既使用 CPU 又使用 GPU. 这有利于应用开发，因为可以将一个现成的 CPU 代码逐步迁移到 GPU 上。也可以将 CPU 上的计算和 GPU 上的计算相互重叠，因此这也是一种并行方式。

GPU 内部发生的是更大规模的并行。许多线程在并行地执行运行在 GPU 上的子例程。尽管执行相同的代码，但是这些线程通常会操作不同的数据。这样的*数据级并行*是一种细粒度并行，使相邻的线程操作相邻的数据，例如一个数组的元素，效率最高。这种并行模型与消息传递接口（Message Passing Interface），即广为人知的 MPI，有很大不同。MPI 是一个粗粒度并行模型，通常将数据分割为几大部分，每一个 MPI 进程负责执行某部分数据上的所有计算任务。

CUDA 编程模型的一些特点与基于 CPU 的并行模型有很大区别。一个区别就是创建 GPU 线程的开销非常小。在快速创建线程之外，与 CPU 相比，GPU 线程的上下文切换也非常快，线程切换是指线程由活动状态变为空闲状态或者作反向的变化。GPU 上的上下文切换基本上瞬间完成，原因是 GPU 不必像 CPU 那样将线程在活动与空闲间切换时保留现场。线程快速切换的后果就是，超额使用 GPU 核心是有益的，即驻留线程比 GPU 核心多得多，从而可以隐藏内存延时。使 GPU 上驻留线程的数量为该 GPU 核心数多一个数量级并不是不常见。在 CUDA 编程模型中，大致上是程序员编写一份串行代码让 GPU 上的许多核心并行执行。执行该代码的每个线程都有一个识别自身的方法，以便操作不同的数据，但是 CUDA 线程执行的代码与将要编写的串行 CPU 代码非常相似。另一方面，许多颗并行 CPU 的编程模型代码与串行 CPU 代码有巨大差别。在后面遇到的时候，将再次逐一讨论 CUDA 编程模型的诸多方面和体系架构。

1.3 基础概念

本节包含一个简单的 CUDA Fortran 示例代码的演进过程，它用来展示 CUDA Fortran 编程中的基础概念。

开始之前需要先定义几个术语。CUDA Fortran 是一个混合编程模型，这意味着代码块的执行要么在 CPU 上要么在 GPU 上，或者更准确地说是在*主机*或*设备*上。在 CUDA Fortran 编程语境中，术语*主机*是指 CPU 及其内存，术语*设备*是指 GPU 及其内存。进而，术语 *CPU 代码*是指一个仅用到 CPU 的实现。一个在设备上执行但被主机调用的子例程称为一个*内核*。

1.3.1 第一个 CUDA Fortran 程序

作为参照，从一个数组自加的 Fortran 90 代码开始。代码刻意用一个子例程处理自加操作，子例程又放在一个 Fortran 90 模块中。子例程使用传入参数 b 的值对一个数组的每个元素进行循环自加。

```
1  module simpleOps_m
2  contains
3    subroutine increment(a, b)
```

```
 4      implicit none
 5      integer, intent(inout) :: a(:)
 6      integer, intent(in) :: b
 7      integer :: i, n
 8
 9      n = size(a)
10      do i = 1, n
11        a(i) = a(i)+b
12      enddo
13
14    end subroutine increment
15  end module simpleOps_m
16
17
18  program incrementTestCPU
19    use simpleOps_m
20    implicit none
21    integer , parameter :: n = 256
22    integer :: a(n), b
23
24    a = 1
25    b = 3
26    call increment(a, b)
27
28    if (any(a /= 4)) then
29      write(*,*) '**** Program Failed ****'
30    else
31      write(*,*) 'Program Passed'
32    endif
33  end program incrementTestCPU
```

实践中不会以这种方式实现这个操作。使用 Fortran 90 的数组语法可以在主程序中仅用一行就完成相同的操作。然而，为了与 CUDA Fortran 版本作对比，也为了凸显 CPU 代码的串行操作本质，特地使用这种格式。

下面是等价的 CUDA Fortran 代码：

```
 1  module simpleOps_m
 2  contains
 3    attributes(global) subroutine increment(a, b)
 4      implicit none
 5      integer , intent(inout) :: a(:)
 6      integer , value :: b
 7      integer :: i
 8
 9      i = threadIdx%x
10      a(i) = a(i)+b
```

```
11
12    end subroutine increment
13  end module simpleOps_m
14
15
16  program incrementTestGPU
17    use cudafor
18    use simpleOps_m
19    implicit none
20    integer , parameter :: n = 256
21    integer :: a(n), b
22    integer , device :: a_d(n)
23
24    a = 1
25    b = 3
26
27    a_d = a
28    call increment <<<1,n>>>(a_d , b)
29    a = a_d
30
31    if (any(a /= 4)) then
32      write(*,*) '**** Program Failed ****'
33    else
34      write(*,*) 'Program Passed'
35    endif
36  end program incrementTestGPU
```

Fortran 90 代码与 CUDA Fortran 代码之间的第一个区别是 CUDA Fortran 实现的第 3 行子例程前缀 `attributes(global)`。属性 `global` 指明这个代码运行在设备上，但需要从主机上调用（与子例程的所有属性一样，术语 `global` 也描述作用域；从主机和设备均可看到本例程）。

注意到第二个主要区别是，Fortran 90 例子第 10-12 行的 do 循环，被替换为 CUDA Fortran 代码的第 9 行初始化指标 i 的语句和第 10 行的循环体内容。这个区别来自于这两份代码的串行执行与并行执行本质。CPU 代码中，数组“a”元素的自加操作由单个 CPU 线程在 do 循环中完成。在 CUDA Fortran 版本中，许多 GPU 线程同时执行子例程。每一个线程通过在所有设备代码中可用的内置变量 `threadIdx` 来识别自己，并将该变量用作数组下标。注意，连续的线程修改一个数组的相邻元素，这种并行方式称为*细粒度并行*。

CUDA Fortran 代码里的主程序在主机上执行。CUDA Fortran 定义和派生类型包含在模块 `cudafor` 中，该模块用在第 17 行，一起使用的还有第 18 行的 `simpleOps_m` 模块。前面提到过，CUDA Frotran 要和两个相互分离的内存空间打交道，一个在主机上，一个在设备上。两个空间都对主机代码可见，属性 `device` 用来声明变量驻留在设备内存上 -- 例如 CUDA Fortran 代码的第 22 行就用它声明设备变量 `a_d`．变量后缀“`_d`”不强调要求，但这个惯例有助于区分主机变量和设备变量。因为 CUDA Fortran 是强类型，所以可以用赋值语句来完成主机与设备间的数据传递。这种情况出现在第 27 行，数组在主机上初始化之后，数据被传递到动态随机访问内存（DRAM）里的设备内存上。数据一旦被传送到 DRAM 里的设备内存，内核或运行在设备上的子例程就可以启动了，就像第 28 行做的那样。第 28 行上，子例程名字与参数列表之间三尖号内

的一组参数称为执行配置，它决定使用多个 GPU 线程来执行这个内核。稍后将详解执行配置，此刻只要知道执行配置<<<1,n>>>指定 n 个 GPU 线程来执行内核就够了。

尽管 a_d 这样的内核数组参数必须驻留设备内存，但标量参数并非如此，例如第二个内核参数 b 就驻留主机内存。CUDA 运行时将特别留意主机标量参数的传递，它期望这些参数按值传递。Fortran 缺省传递参数地址，但使用变量属性 value 可以实现参数按值传递，就像 CUDA Fortran 代码第 6 行显示的那样。作为与 C 代码互操作的一部分，属性 value 被引入 Fortran 2003.

在 CUDA 这样的混合编程模型中，必须解决的一个问题是主机与设备间的同步。为保证这个程序正确执行，在内核开始执行之前需要知道第 27 行的主机至设备数据传递已经完成，在第 29 行的设备至主机数据传递启动之前需要知道内核已经执行完毕。这些要求确实满足，因为第 27 行和 29 行通过赋值语句的数据传递是阻塞传递或同步传递。这种传递直到前面的 GPU 操作全部完成后才开始，而且数据传递完成之前后续的 GPU 操作不会开始。这些数据传递的阻塞特性有利于隐式同步 CPU 和 GPU.

通过赋值语句的数据传递是阻塞操作或同步操作，内核启动却是非阻塞的或异步的。第 28 行的内核一旦启动，控制权会立即返回主机。然而程序行为仍像期待的那样，因为具有阻塞传递特性，第 29 行的数据传递不会开始。

有一些例程可以实现异步传递，因此，只要提供一个主机与设备的同步方法，设备上的计算与主机设备间的通信就可以重叠，详细讨论在 3.1.3 节。

1.3.2 扩展到大型数组

前面的例子中，执行配置<<<1,n>>>中的参数 n 有上限，因此数组尺寸必须要小。这个上限依赖于所使用的具体 CUDA 设备。对基于 Kepler™和 Fermi™的产品，例如 Tesla K20 和 C2050 卡，上制 n=1024，上一代卡的上限是 n=512. 这些上限参见附录 A。 适应大数组的方法是修改第一个执行配置参数，因为本质上执行代码的 GPU 线程数量由这两个执行配置参数的乘积来指定。那么，为什么要这样做？为什么 GPU 线程要以这种方式分组？编程模型中的线程分组方式是在模仿 GPU 上硬件处理单元的分组方式。

GPU 上的基本计算单元是一个线程处理器，也可以简称为核心(core)。本质上，一个线程处理或核心是一个浮点运算单元。线程处理器被组合为多处理器(multiprocessor)，它包含数量有限的资源供驻留线程使用，这些资源就是寄存器(register)和共享内存(shared memory)。图 1.1 显示了这个概念，图中一个 CUDA 设备包含一块带四个多处理器的 GPU，每个多处理器包含 32 个线程处理器。

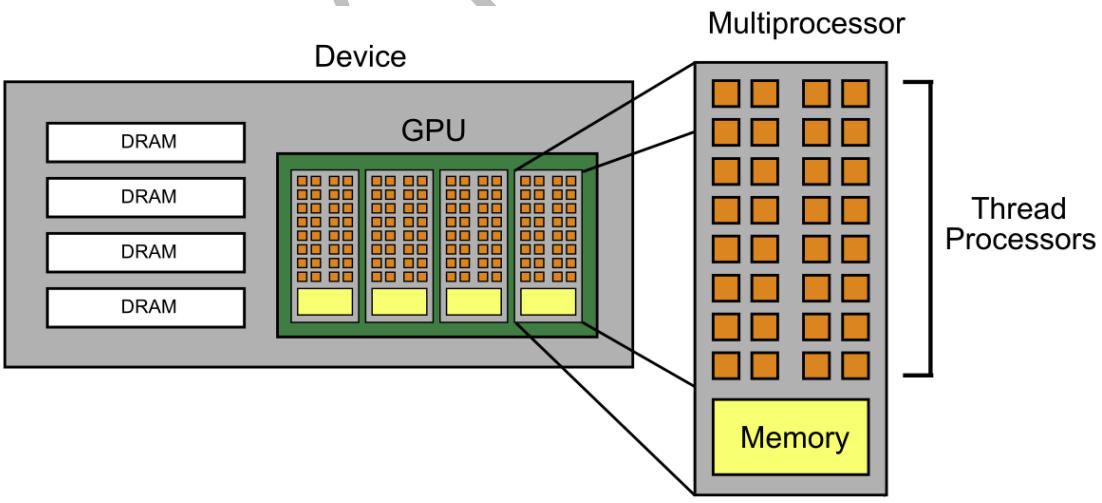


图 1.1

GPU 内计算单元的层级，线程处理器组成多处理。

编程模型中用来模拟多处理器的是线程块(thread block)。线程块是指派组一个多处理上的一组线程，一旦指派就不再迁移。多个线程块可以驻留在同一个多处理器上，但可以同时驻留在一个多处理器上的线程块数量受单个多处理器上的可用资源和每个线程块需求的资源限制。

回到示例代码，内核被调用时，它启动一个线程网格(grid)。启动的线程块数量由执行配置的第一个参数指定，每个线程块内的线程数量由第二个参数指定。因此，第一个 CUDA Fortran 程序启动的网格包含一个线程块，这个线程块包含 256 个线程。可以启动多个线程块来适应大型数组，代码如下：

```
module simpleOps_m
  contains
  attributes(global) subroutine increment(a, b)
    implicit none
    integer , intent(inout) :: a(:)
    integer , value :: b
    integer :: i, n

    i = blockDim%x*(blockIdx%x-1) + threadIdx%x
    n = size(a)
    if (i <= n) a(i) = a(i)+b

    end subroutine increment
end module simpleOps_m


program incrementTest
  use cudafor
  use simpleOps_m
  implicit none
  integer , parameter :: n = 1024*1024
  integer , allocatable :: a(:)
  integer , device , allocatable :: a_d(:)
  integer :: b, tPB = 256

  allocate(a(n), a_d(n))
  a = 1
  b = 3

  a_d = a
  call increment <<<ceiling(real(n)/tPB),tPB >>>(a_d , b)
  a = a_d

  if (any(a /= 4)) then
    write(*,*) '**** Program Failed ****'
  else
```

```
37       write(*,*) 'Program Passed'
38     endif
39     deallocate(a, a_d)
40   end program incrementTest
```

主机代码中将主机数组和设备都声明为可分配。使用大数组时不是必须这样，这样做仅为表明设备数组也可以像主机数组一样分配和撤销。事实上，像本例第 26 行和第 39 行那样，主机数组和设备数组都可以用在同一个 `allocate()`和 `deallocate()`语句中。

除了使用可分配数组，本程序还包含了对 1.31.节中 CUDA Fortran 代码的另外一些修改。主机代码中，第 24 行定义的参数 `tPB` 代表每块的线程数量。当以多个线程块的方式启动一个内核时，本次启动的所有线程块必须具有相同的尺寸，这个尺寸由第二个执行配置参数指定。在本例中，当数组元素数量不能被每块线程数量整除时，需要确保启动足够多的线程来处理数组的每一个元素，还要同样确保不能越界访问数组。第 31 行的函数 `ceiling` 用来确定处理所数组元素需要的线程块数量。在设备代码中，第 10 行的 Fortran 90 固有函数 `size()` 用来确定数组里的元素数量，这个数量被用到第 11 行的 `if` 条件中以确保内核读写不会超越数组末尾。

除了检查越界内存访问，设备代码与 1.3.1 节中单块示例的区别还有第 9 行的计算数组指标 `i`. 预定义变量 `threadIdx` 是一个线程在其线程块内的指标。当使用多个线程块时，如这里的情形，需要用这个值来计算对上一个线程块的线程数量的偏移量，以获得访问数组元素需要的特定整数。这个偏移量由预定义变量 `blockDim` 和 `blockIdx` 决定，它们分别是一个块内的线程数量和本块在网格内的指标。图 1.2 展示了设备代码中利用预定义变量计算全局数组指标的方法。

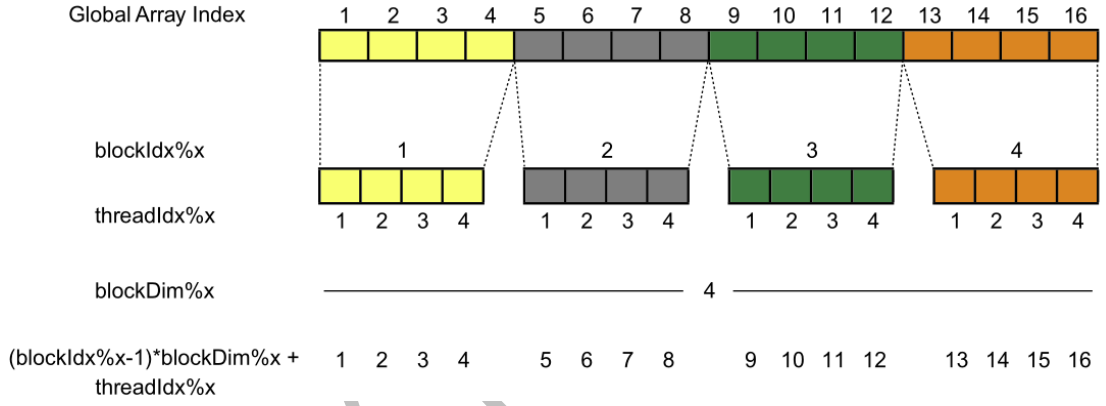


图 1.2

用预定义变量 blockDim、blockIdx 和 threadIdx 计算全局数组指标。简单起见，这里使用四个线程块，每个线程块有四个线程。实际的 CUDA Fortran 代码，线程块会有多得多的线程。

在叙述过的两个 CUDA Fortran 示例中，内核代码访问预定义变量的 `x` 域，就像有人可能期待的那样，这些数据类型可以适应多维数组，这就是接下来探讨的内容。

1.3.3 多维数组

很容易就能将前面示例扩展到多维数组上。代码中预定义变量属于派生类型 `dim3`，该类型包含 `x` 域、`y` 域和 `z` 域，因此扩展工作挺简单。就主机代码来说，目前为止执行配置参数中，每个网格内的块数和每块内的线程数均指定为整数，但这些参数也可以是 `dim3` 类型。利用 `dim3` 类型的其它域，多维版代码就变成：

```
1  module simpleOps_m
2  contains
3    attributes(global) subroutine increment(a, b)
4    implicit none
5      integer :: a(:,:)
6      integer , value :: b
7      integer :: i, j, n(2)
8
9      i = (blockIdx%x-1)* blockDim%x + threadIdx%x
10     j = (blockIdx%y-1)* blockDim%y + threadIdx%y
11     n(1) = size(a,1)
12     n(2) = size(a,2)
13     if (i<=n(1) .and. j<=n(2)) a(i,j) = a(i,j) + b
14   end subroutine increment
15 end module simpleOps_m
16
17
18
19 program incrementTest
20   use cudafor
21   use simpleOps_m
22   implicit none
23   integer , parameter :: nx=1024, ny=512
24   integer :: a(nx,ny), b
25   integer , device :: a_d(nx,ny)
26   type(dim3) :: grid , tBlock
27
28   a = 1
29   b = 3
30
31   tBlock = dim3(32,8,1)
32   grid = dim3(ceiling(real(nx)/tBlock%x), &
33               ceiling(real(ny)/tBlock%y), 1)
34   a_d = a
35   call increment <<<grid ,tBlock >>>(a_d , b)
36   a = a_d
37
38   if (any(a /= 4)) then
39     write(*,*) '**** Program Failed ****'
40   else
41     write(*,*) 'Program Passed'
42   endif
43 end program incrementTest
```

对这个二维示例，在声明参数 nx 和 ny 以及主机数组和设备数组之后，第 26 行又声明了用在执行配

置里的两个 dim3 类型变量。第 31 行设定 dim3 类型的三个分量，这些分量指定每块的线程数量；本例中每个线程块有 32 × 8 个线程。接下来的两行中，函数 ceiling 用来决定访问数组所有元素时在 x 维度和 y 维度上需要线程块数量。在第 35 行，以这些变量做为执行配置参数启动内核。在内核代码里，形参 a 被声明为二维数组，变量 n 被声明为两个元素的数组，在第 11 和 12 行，n 被设定为保存 a 的每个维度的尺寸。与第 9 行对 i 的赋值方法类似，第 10 行对另一个指标 j 赋值，在 a(i,j) 自加之前检查 i 和 j 以保证界内访问。

1.4 查清 CUDA 硬件的特性和限制

有许多不同 CUDA 设备可用，它们覆盖多代体系架构和不同的产品线(GeForce®、Quadro® 和 Tesla)。前面已经讨论过每块里面的线程数量上限，基于 Kepler 和 Fermi 的硬件是 1024，基于更早架构的硬件是 512，还有其它许多特性和限制会因设备而异。本节论及设备管理接口，接口包含的例程可探测一个特定系统上 CUDA 卡的数量和类型以及这些卡具有的特性和限制。

进入设备管理接口话题之前，应该先简要阐述一下计算能力(compute capability)的概念。CUDA 设备的计算能力标识体系架构，并以主要.次要的形式给出。计算能力的主要部分反映体系架构属于哪一代，次要部分是本代内的修订版本。第一批 CUDA 卡的计算能力是 1.0. Fermi 一代卡的计算能力是 2.x，Kepler 一代卡的计算能力是 3.x. 若干 CUDA 特性与计算能力相关联；例如，双精度仅在具有 1.3 及更高计算能力的卡上可用。其它的特性与计算能力无关，但也可以利用设备管理接口查明。

设备管理接口的例程用来获取系统上可用的多种卡的信息，也用来从多块可用的卡中选择某一块卡。该接口利用派生类型 cudaDeviceProp 查询单块卡的特性，接下来的程序展示了查询方法：

```
1  program deviceQuery
2    use cudafor
3    implicit none
4
5    type (cudaDeviceProp) :: prop
6    integer :: nDevices=0, i, ierr
7
8    ! Number of CUDA -capable devices
9
10   ierr = cudaGetDeviceCount(nDevices)
11
12   if (nDevices == 0) then
13     write(*,"(/,'No CUDA devices found ',/)")
14     stop
15   else if (nDevices == 1) then
16     write(*,"(/,'One CUDA device found ',/)")
17   else
18     write(*,"(/,i0,' CUDA devices found ',/)") nDevices
19   end if
20
21   ! Loop over devices
22
```

```
23    do i = 0, nDevices -1
24
25      write(*,"('Device Number: ',i0)") i
26
27      ierr = cudaGetDeviceProperties(prop , i)
28
29      ! General device info
30
31      write(*,"(' Device Name: ',a)") trim(prop%name)
32      write(*,"(' Compute Capability: ',i0,'.',i0)") &
33           prop%major , prop%minor
34      write(*,"(' Number of Multiprocessors: ',i0)") &
35           prop%multiProcessorCount
36      write(*,"(' Max Threads per Multiprocessor: ',i0)") &
37           prop%maxThreadsPerMultiprocessor
38      write(*,"(' Global Memory (GB): ',f9.3,/)") &
39           prop%totalGlobalMem /1024.0**3
40
41      ! Execution Configuration
42
43      write(*,"(' Execution Configuration Limits ')")
44      write(*,"(' Max Grid Dims: ',2(i0,' x '),i0)") &
45           prop%maxGridSize
46      write(*,"(' Max Block Dims: ',2(i0,' x '),i0)") &
47           prop%maxThreadsDim
48      write(*,"(' Max Threads per Block: ',i0 ,/)") &
49           prop%maxThreadsPerBlock
50
51    enddo
52
53  end program deviceQuery
```

这份代码在第 10 行从例程 `cudaGetDeviceCount()`中探明系统上附属 CUDA 设备数量，然后对每一个设备循环，用例程 `cudaGetDeviceProperties()`取回设备属性。这份代码仅列出类型 `cudaDeviceProp` 的一小部分可用成员。派生类型 `cudaDeviceProp` 成员的完整列表由 *CUDA Toolkit Reference Manual* 提供，也可到英伟达官网在线查看。

此处列出这份代码运行在具有不同计算能力的多种 Tesla 设备上的输出结果。第一个 Tesla 设备是 2007 年发布的 Tesla C870，它的计算能力是 1.0. 在带有此设备的系统上获得下列结果：

```
One CUDA device found

Device Number: 0
  Device Name: Tesla C870
  Compute Capability: 1.0
  Number of Multiprocessors: 16
```

```
  Max Threads per Multiprocessor: 768
  Global Memory (GB): 1.500

  Execution Configuration Limits
    Max Grid Dims: 65535 x 65535 x 1
    Max Block Dims: 512 x 512 x 64
    Max Threads per Block: 512
```

注意这里的计数是从零开始而不是从一开始。Max Threads per Multiprocessor 后面的数字是指多处理器上可以并发驻留线程的数量。Global Memory 表示本行后面的输出内容是设备 DRAM 上的可用内存容量。Execution Configuration Limits 的前两行表示执行配置的前两个参数在每一个维度的上限：内核启动时的线程块配置和线程块数量、一个线程块内部的线程配置和线程数量。注意，对此计算能力，网格必须是一个线程块的二维配置，而线程块则可以是线程的三维摆放，各维度均可达到指定的上限。对本设备，执行配置里指定的线程块的三个分量的乘积必须小于等于 Max Threads per Block，即 512.

下一个更高计算能力的 Tesla 产品是 Tesla C1060:

```
One CUDA device found

Device Number: 0
  Device Name: Tesla C1060
  Compute Capability: 1.3
  Number of Multiprocessors: 30
  Max Threads per Multiprocessor: 1024
  Global Memory (GB): 4.000

  Execution Configuration Limits
    Max Grid Dims: 65535 x 65535 x 1
    Max Block Dims: 512 x 512 x 64
    Max Threads per Block: 512
```

除了比 C870 拥用更多的多处理器之外，C1060 的每个多处理器上的线程数量上限更高。但两个设备间最重要的区别恐怕就是 C1060 为第一款有能力实施双精度算术运算的 Tesla 设备。执行配置上限与 C870 相同。

Tesla C2050 是费米(Fermi)一代设备的一个代表：

```
One CUDA device found
  Device Number: 0
  Device Name: Tesla C2050
  Compute Capability: 2.0
  Number of Multiprocessors: 14
  Max Threads per Multiprocessor: 1536
  Global Memory (GB): 2.624

  Execution Configuration Limits
    Max Grid Dims: 65535 x 65535 x 65535
    Max Block Dims: 1024 x 1024 x 64
    Max Threads per Block: 1024
```

与 C1060 具有 30 个多处理器不同，C2050 只有 14 个，但费米多处理器设计得比上一代多处理强大得多。相对于上一代设备，执行配置上限也有本质性改变。线程块最大尺寸由 512 升至 1024，在一个网格内三维摆放线程块也成为可能，三维问题的分解极为便利。

下一代卡是开普列(Kepler)一代设备，具有 3.x 的计算能力，例如 Tesla K20：

```
One CUDA device found
  Device Number: 0
  Device Name: Tesla K20
  Compute Capability: 3.5
  Number of Multiprocessors: 13
  Max Threads per Multiprocessor: 2048
  Global Memory (GB): 4.687

  Execution Configuration Limits
    Max Grid Dims: 2147483647 x 65535 x 65535
    Max Block Dims: 1024 x 1024 x 64
    Max Threads per Block: 1024
```

对开普列，每个多处理器上的线程数量又增加了，能用作网格第一个参数的线程块的数量上限也增加了。上限增长是因为需要仅用一维线程块和一维网格来启动带有巨多线程的内核 。在计算能力小于 3.0 的设备上，能以这种方式启动的最大线程数量为 64×1024^2. 将线程与数据元素 1-1 映射，它对应一个 256 MB 的单精度数组。可以采用二维网格或采用每个线程处理多个数组元素的方法来规避这个限制，但在 3.0 或更高计算能力的设备不必再用些变通方法。本书后面将会讲解开普列架构引入的许多其它特性。属于开普列一代的另外一个 Tesla 设备是 Tesla K10. 在带有单个 Tesla K10 的系统上获得下列输出：

```
2 CUDA devices found

Device Number: 0
  Device Name: Tesla K10.G1.8GB
  Compute Capability: 3.0
  Number of Multiprocessors: 8
  Max Threads per Multiprocessor: 2048
  Global Memory (GB): 4.000

  Execution Configuration Limits
    Max Grid Dims: 2147483647 x 65535 x 65535
    Max Block Dims: 1024 x 1024 x 64
    Max Threads per Block: 1024

Device Number: 1
  Device Name: Tesla K10.G1.8GB
  Compute Capability: 3.0
  Number of Multiprocessors: 8
  Max Threads per Multiprocessor: 2048
  Global Memory (GB): 4.000

  Execution Configuration Limits
    Max Grid Dims: 2147483647 x 65535 x 65535
    Max Block Dims: 1024 x 1024 x 64
    Max Threads per Block: 1024
```

每块 Tesla K10 包含两颗 GPU，每颗 GPU 拥用 4 GB DRAM 内存。从 CUDA Fortran 程序员的角度看，一个系统带有一块 K10 与带有两块单 GPU 设备没有区别。第四章会讲述如何这样的多 GPU 系统上编程。

表 1.1 总结了这个 deviceQuery 代码得到的一些数据[2]。表中的所有数据都能够从派生类型 `cudaDeviceProp` 的成员中获得，但能够同时驻留在一个多处理器上的最大线程块数量除外。

表 1.1 多种 Tesla 设备的特征参数

	Tesla C870	Tesla C1060	Tesla C2050	Tesla K10	Tesla K20

[2]在这些设备和其它 Tesla 设备上的更多信息在附录 A 列出。

计算能力	1.0	1.3	2.0	3.0	3.5
多处理器数量	16	30	14	2×8	13
每个多处理器上的最大线程数量	786	1024	1536	2048	2048
每个多处理器上的最大线程块数	8	8	8	16	16
每个线程块内的最大线程数	512	512	1024	1024	1024
全局内存(GB)	1.5	4	3*	2×4*	5*
*打开 ECC 会减少可用的全局内存					

从这些设备上的多处理器数量与每个多处理器上的最大线程数量的乘积，可以看出所有设备上的并发线程数都可以达到几万。

正如表中标注的那样，计算能力 2.0 或更高的 Tesla 设备具用内存校正代码(error-correctingcode， ECC)特性。ECC 可以打开或关闭，如果打开，可用的全局内存会比表中列出的数字少一点。这种情况下，减少的数值会被报告出来 就像在前面 Tesla C2050 和 Tesla K20 的输出中看到的那样。可以从 `cudaDeviceProp` 类型的 `ECCEnabled` 域里查看 ECC 是处于打开状态还是关闭状态。

尽管表 1.1 中的数据是从特定的 Tesla 设备采集，但其中许多数据可以应用到具有相同计算能力的设备上。表 1.1 中的数据只有全局内存量和设备上的多处理器数量会在相同计算能力的设备间发生变化。

通过调整多处理器个数 ,可以使用同一个多处理器架构来制造出许多不同的设备。带有一块 GeForce GT 650 M 的笔记本电脑拥有两个计算能力为 3.0 的多处理器，作为对比，Tesla K10 的两颗 GPU 内各有八个相同的多处理器。不考虑处理能力，前节的代码不做任何修改就能在这些设备上运行。这是编程模型里将线程分组为线程块的好处之一。只要有可用空间，调度器就将线程块分配到多处理器上。线程块是相互独立的，因此，它们的执行顺序不会影响执行结果。编程模型中线程块的独立性允许后台调度，从而程序员只需关注一个线程块内的编程。

不管设备上有多少个多处理器，一个内核可以启动的线程块数量都是非常巨大的。甚至能在带有一块 GeForce GT 650 M 的笔记本电脑上得到：

```
One CUDA device found

Device Number: 0
  Device Name: GeForce GT 650M
  Compute Capability: 3.0
  Number of Multiprocessors: 2
  Max Threads per Multiprocessor: 2048
  Global Memory (GB): 0.500

  Execution Configuration Limits
    Max Grid Dims: 2147483647 x 65535 x 65535
    Max Block Dims: 1024 x 1024 x 64
    Max Threads per Block: 1024
```

在这个笔记本电脑 GPU 上启动的内核，使用一维线程块和一维网格，线程数可达 2147483647 × 1024! 再一次，线程块的独立性允许调度器将线程分配给有可用空间的多处理器，调度工作不需要程序员的干预。

在花时间编写 deviceQuery 代码的完整版之前，请注意，包含在 PGI 编译器内的工具 `pgaccelinfo` 也提供这样的信息。在一个带有单块 Tesla K20 的系统上，`pgaccelinfo` 的输出如下：

```
CUDA Driver Version:           5000
NVRM version: NVIDIA UNIX x86_64 Kernel Module 304.52
Sun Sep 23 20:28:04 PDT 2012

CUDA Device Number:            0
Device Name:                   Tesla K20
Device Revision Number:        3.5
```

```
Global Memory Size:             5032706048
Number of Multiprocessors:      13
Number of SP Cores:             2496
Number of DP Cores:             832
Concurrent Copy and Execution:  Yes
Total Constant Memory:          65536
Total Shared Memory per Block:  49152
Registers per Block:            65536
Warp Size:                      32
Maximum Threads per Block:      1024
Maximum Block Dimensions:       1024, 1024, 64
Maximum Grid Dimensions:        2147483647 x 65535 x 65535
Maximum Memory Pitch:           2147483647B
Texture Alignment:              512B
Clock Rate:                     705 MHz
Execution Timeout:              No
Integrated Device:              No
Can Map Host Memory:            Yes
Compute Mode:                   default
Concurrent Kernels:             Yes
ECC Enabled:                    Yes
Memory Clock Rate:              2600 MHz
Memory Bus Width:               320 bits
L2 Cache Size:                  1310720 bytes
Max Threads Per SMP:            2048
Async Engines:                  2
Unified Addressing:             Yes
Initialization time:            44466 microseconds
Current free memory:            4951891968
Upload time (4MB):              1715 microseconds ( 962 ms pinned)
Download time:                  3094 microseconds ( 877 ms pinned)
Upload bandwidth:               2445 MB/sec (4359 MB/sec pinned)
Download bandwidth:             1355 MB/sec (4782 MB/sec pinned)
PGI Compiler Option:            -ta=nvidia ,cc35
```

输出里最后一行的 `PGI Compiler Option` 提到了 PGI Accelerator 对接 CUDA 使用的选项。1.6 节将探究 CUDA Fortran 的编译器选项。

1.4.1 单精度和双精度

多处理器里的线程处理器有能力实施单精度浮点算术运算，而双精度浮点算术运算由多处理器内包含的独立双精度核心来承担。表 1.2 总结了每个多处理器和每个设备包含的单精度和双精度核心数量。

表 1.2 各种 Tesla 设备上的单精度和双精度资源

计算能力	1.0	1.3	2.0	3.0	3.5
代表设备	**Tesla C870**	**Tesla C1060**	**Tesla C2050**	**Tesla K10**	**Tesla K20**
多处理器数量	16	30	14	2×8	13
每个多处理器上的单精度核心	8	8	32	192	192
单精度核心总数	128	240	448	2×1536	2496
每个多处理器上的双精度核心	–	1	16*	6	64
单精度核心总数	–	30	224*	2×64	832
每个多处理器上的最大线程数	768	1024	1536	2048	2048

*GeForce GPU 的双精度单元少一些。

已经提到过，具有计算能力 1.3 的设备，例如 Tesla C1060，首次支持双精度。总之，每一代卡里的单精度和双精度资源都会显著增加。Tesla K10 的双精度能力是个例外。Tesla K10 的核心时钟比 K20 高得多，

设计得单精度性能优秀。对双精度性能，Tesal K20 是合适的开普列设备。

在表 1.2 的最后一行添加每个多处理器上的最大线程数，意为显示，一个多处理器上的驻留线程数量可以远远超过计算资源的 10 倍以上。这是刻意设计的。GPU 线程间的上下文切换是如此的高效，全局内存延时是如此的巨大，因此希望使用过量的线程来隐藏巨大的全局内存延时。

1.4.1.1 适应变量的精度

常常需要先在一个较小规模的问题上使用单精度变量开发代码，然后在更大规模的问题上使用双精度开发代码。Fortran 90 的 kind 类型可以相当容易地在单精度和双精度间切换。仅需用备选类型定义一个模块：

```
module precision_m
  integer , parameter :: singlePrecision = kind(0.0)
  integer , parameter :: doublePrecision = kind(0.0d0)

  ! Comment out one of the lines below
  integer , parameter :: fp_kind = singlePrecision
  !integer , parameter :: fp_kind = doublePrecision
end module precision_m
```

声明单精度浮点变量时使用这个模块和参数 `fp_kind`：

```
use precision_m
real(fp_kind), device :: a_d(n)
```

这允许程序员通过简单地改精度模块里的定义来在两种精度间切换。（可能不得不编写一些通用接口，以适应 NVIDIA CUDA® Fast Fourier Transform, 或 CUFFT 等库、例程的调用。）

在单精度和双精度间切换的另一个可选方法不需要修改源代码，利用预处理功能，精度模块可以改写为：

```
module precision_m
  integer , parameter :: singlePrecision = kind(0.0)
  integer , parameter :: doublePrecision = kind(0.0d0)
#ifdef DOUBLE
  integer , parameter :: fp_kind = doublePrecision
#else
  integer , parameter :: fp_kind = singlePrecision
#endif
end module precision_m
```

用编译器选项`-Mpreprocess -DDOUBLE` 编译来精度模块，如果使用`.CUF` 文件扩展名，用编译器选`-DDOUBLE` ，就可以编译为双精度。

本书将广泛使用这个精度模块，理由有好几个。第一，使读者无论有什么卡都能使用这些示例代码。其次，方便在多种代码上评估两种精度的性能特征。最后，它是代码重用方面的良好实践。

这个技术可以扩展到编写混合精度代码。例如，在一个反应流模拟代码中，希望对不同精度的流量变量和化学物质实验。为达到这个目标，可以在下面的代码里这样声明变量：

```
real(flow_kind), device :: u(nx,ny,nz), v(nx,ny,nz), w(nx,ny,nz)
real(chemistry_kind), device :: q(nx,ny,nz,nspecies)
```

在模块 `precision_m` 中，`flow_kind` 和 `chemistry_kind` 被声明为要么是单精度要么是双精度。

这种编程风格里，还可以用一个指定的 kind 来定义单精度常量 — 例如：

```
real(fp_kind), parameter :: factorOfTwo = 2.0_fp_kind
```

1.5 错误处理

设备查询示例中的主机 CUDA 函数和所有主机 CUDA 接口函数的返回值都可以用来检测它们执行过程中发生的错误。为演示这样的错误处理方式，1.4 节例子 deviceQuery 第 10 行上 cudaGetDeviceCount() 的成功执行可作如下检测：

```
ierr = cudaGetDeviceCount(nDevices)
if (ierr/= cudaSuccess) write(*,*) cudaGetErrorString(ierr)
```

变量 cudaSuccess 在本代码使用的模块 cudafor 中定义。如果有错误，函数 cudaGetErrorString() 会来返回一个描述错误的特征字符串，而不仅仅列出错误的数字代码。这个代码可能发生错误的一个情形是代码运行在不带 CUDA 设备的机器上。没有可以在其上运行的设备运时，命令无法执行，从而不会修改 nDevices 的值，最后返回一个错误。正因如此，第 6 行声明变量 nDevices 时将它初始化为 0.

内核的错误处理稍微复杂一点，因为内核是子例程，没有返回值，还因为内核相对于主机异步地执行。为帮助检测内核及其它异步操作的执行错误，CUDA 运行时维护着一个错误变量，每次发生错误时就会重写这个变量。函数 cudaPeekAtLastError() 返回这个变量的值，函数 cudaGetLastError() 返回这个变量的值并将它重置为 cudaSuccess. 内核执行的错误检测可以使用下面的方法实现：

```
call increment <<<1,n>>>(a_d , b)
ierrSync = cudaGetLastError()
ierrAsync = cudaDeviceSynchronize()
if (ierrSync/= cudaSuccess) &
  write (*,*) 'Sync kernel error:', cudaGetErrorString(ierrSync)
if (ierrAsync/= cudaSuccess) &
  write(*,*) 'Async kernel error:', cudaGetErrorString(ierrAsync)
```

这个代码同时检测同步和异步错误。非法执行配置参数，例如每个线程块里放太多的线程，将会反映到 cudaGetLastError() 的返回值 ierrSync 里。控制权返回主机后发生在设备上的异步错误，就需要诸如 cudaDeviceSynchronize() 的同步机制来阻止主机线程继续执行，直到前面提交到设备上的诸如内核启动的命令全部执行完毕。也可以像下面这样修改最后一行代码来检测异步错误并重置运行时维护的变量：

```
call increment <<<1,n>>>(a_d , b)
ierrSync = cudaGetLastError()
ierrAsync = cudaDeviceSynchronize()
if (ierrSync/= cudaSuccess) &
  write(*,*) 'Sync kernel error:', cudaGetErrorString(ierrSync)
if (ierrAsync/= cudaSuccess) write (*,*) 'Async kernel error:', &
  cudaGetErrorString(cudaGetLastError ())
```

1.6 编译 CUDA Fortran 代码

CUDA Fortran 代码用 PGI Fortran 编译器编译。带有.cuf 或.CUF 扩展名的文件会自动支持 CUDA Fortran，编译带有其它扩展名的文件时，用编译器选项-Mcuda 打开对 CUDA Fortran 的支持。此外，因为使用的是标准 PGI 编译器，CPU 代码中使用的诸如 OpenMP 和 SSE 向量化等全部特性都可以用到主机代码里。编译 CUDA Fortran 代码可以简单到仅提交一个命令：

```
pgf90 increment.cuf
```

后台进行了多步处理。设备源码被编译成一种称为并行线程执行(Parallel Threade Xecution, PTX)的中间表示码。这种向前兼容 PTX 表示码将来可以为不同计算能力的设备编译成可执行代码。主机代码由主机编译器编译。

利用编译器选项-Mcuda=ptxinfo 可以查看编译目标是哪种计算能力。用这个选项编译自加例子产生如下输出：

```
% pgf90 -Mcuda=ptxinfo increment.cuf
ptxas info    : Compiling entry function 'increment' for 'sm_10'
ptxas info    : Used 4 registers , 24+16 bytes smem
ptxas info    : Compiling entry function 'increment' for 'sm_20'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 6 registers , 56 bytes cmem[0]
ptxas info    : Compiling entry function 'increment' for 'sm_30'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 8 registers , 344 bytes cmem[0]
```

用 -Mcuda=ptxinfo 的编译输出包含许多关于从 PTX 编译成二进制码的有用信息，例如内核使用的寄存器数量和不同类型内存的数量，但此刻仅关注目标的计算能力。输出显示产生的二进制码针对三种计算能力：1.0、2.0 和 3.0(这里标记为 sm_10、sm_20、和 sm_30). 回顾前文，计算能力的第一个数字指设备架构的代次，第二个数字指本代内的修订版本。对任何修订版本大于等于编译目标的同一代设备，二进制代码都是兼容的。就这样，这个应用可以运行在计算能力为 1.X、2.X 和 3.X 的 CUDA 设备上。在运行的时候，主机代码将选择加载、执行最合适的代码。

如果改变自加代码，数组是双精度浮点型而不是整型，得到：

```
% pgf90 -Mcuda=ptxinfo incDP.cuf
ptxas info    : Compiling entry function 'increment' for 'sm_13'
ptxas info    : Used 5 registers , 24+16 bytes smem
ptxas info    : Compiling entry function 'increment' for 'sm_20'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 8 registers , 56 bytes cmem[0]
ptxas info    : Compiling entry function 'increment' for 'sm_30'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 8 registers , 344 bytes cmem[0]
```

CUDA Fortran 编译器追踪程序里特定计算能力的所有特性，例如双精度算术运算，并针对每一代中的最低版本设备产生代码。因为双精度是由计算能力 1.3 的设备首次支持，所以编译器针对计算能力 1.3、2.0 和 3.0 产生代码，得到的应用将能运行在支持双精度的任何设备上。

除了包含多种计算能力的二进制代码，可执行文件还包含 PTX 代码。因为新的计算能力包含新的特性，不同版本的 PTX 对应不同的计算能力。包含在可执行文件内的 PTX 版本对应最高的目标计算能力，例如在本示例中对应计算能力 3.0. 这个内嵌的 PTX 代码可以被及时编译为计算能力大于等于 PTX 版本的二进制代码。因此，尽管设备二进制码兼容本代内更新修订版设备，但 PTX 可以为更新代次的设备产生代码(本代内更新修订版的设备也可以)。本例中，当计算能力 4.0 及更高的设备出现后，可执行文件可以在它们上面正确地运行 ,因为设备代码将由内嵌的 PTX 产生。尽管可以过环境变强制编译 PTX(参见 B.1.3 节)，但在默认情况下，只要二进制码可用，应用总是使用兼容的二进制码，而不是及时编译 PTX.

这里介绍的产生设备二进制码的默认机制保证 CUDA Fortran 应用对所有合适设备的兼容性，但偶尔

也需要针对特定的计算能力。但最后产生的臃肿二进制文件的尺寸可能是一个麻烦。尽管计算能力 3.0 的二进制码可以运行在计算能力 3.5 的设备上，但它可能达到为计算能力 3.5 创建的代码那样的性能。用编译器选项-Mcuda=ccXY 可以指定计算能力 X.Y. 例如这样编译代码：

```
% pgf90 -Mcuda=cc20 ,ptxinfo increment.cuf
ptxas info    : Compiling entry function 'increment' for 'sm_20'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 6 registers , 56 bytes cmem[0]
```

得到的可执行文件可以用二进制码运行在任何计算能力为 2.X 的设备上，归功于及时编译 PTX 码，它也可以正确地运行在带有计算能力 3.X 设备的机器上。

也可以利用代次名字来指定架构。例如用-Mcuda=fermi 编译等价于 -Mcuda=cc20.

在生成 PTX 信息和指定具体设备架构之外，编译器选项-Mcuda 还有许多其它参数。用 pgf90 -Mcuda -help 可以生成这些参数的清单。该命令的输出包含：

```
emu              Enable emulation mode
tesla            Compile for Tesla architecture
cc1x             Compile for compute capability 1.x
fermi            Compile for Fermi architecture
cc2x             Compile for compute capability 2.x
kepler           Compile for Kepler architecture
cc3x             Compile for compute capability 3.x
cuda4.0          Use CUDA 4.0 Toolkit compatibility
cuda4.1          Use CUDA 4.1 Toolkit compatibility
cuda4.2          Use CUDA 4.2 Toolkit compatibility
cuda5.0          Use CUDA 5.0 Toolkit compatibility
fastmath         Use fast math library
[no]flushz       Enable flush -to-zero mode on the GPU
keepgpu          Keep kernel source files
keepbin          Keep CUDA binary files
keepptx          Keep PTX portable assembly files
maxregcount:<n>  Set maximum number of registers to use on the GPU
nofma            Don't generate fused mul -add instructions
ptxinfo          Print informational messages from PTXAS
[no]rdc          Generate relocatable device code
```

在指定计算能力之外，还可以利用选项-Mcuda=emu 编译 CUDA Fortran 代码使之能够运行在主机 CPU 上。这允许程序员在一个不带 CUDA 设备的系统上开发 CUDA Fortran 代码，还能在内核代码里使用主机调试器。然而，以仿真模式的执行有很大不同，典型地，一次只执行一个线程块，因此仿真中也许不能暴露数据竞争。

CUDA Fortran 连同几个版本的 CUDA 工具包库一起发布。从 pgf90 -Mcuda -help 的输出中可以查看可用的 CUDA 库。通常，默认版本是第二新版本--这个例子中是 CUDA 4.2 工具箱库。

CUDA 有一组快速但精度较低的内置单精度函数，诸如 sin()和 cos()，可以用-Mcuda=fastmath 打开。选项-Mcuda=maxregcount:N 能将每个线程使用的寄存器数量限定为 N 以下。选项 keepgpu、keepbin、keepptx 会分别将内核源码、CUDA 二进制码和 PTX 保存到本地目录的文件中。

尽管不是 CUDA 专用，有用的其它编译器选项是-v 和-V. 用选项-v 编译将提供编译、连接步骤的详细输出。选项-V 可以用来核实 PGI 编译器的版本，或者配上合适的参数用来从机器上安装的众多版本中选择某个编译器版本，举例来说，-V12.10 用来选择 12.10 版本的 PGI 编译器。

1.6.1 分离编译

CUDA Frotran 总是允许主机代码启动定义在多个模块中的内核，无论是这些模块在同一个文件中还是分散在不同的文件中。主机代码需要简单地 `use` 包含被启动内核的每一个模块。

类似地，模块间共享设备数据也相对简单，且在任何计算能力的 GPU 上都可以使用。例如，文件 `b.cuf` 包含一个简短模块 `b_m`，该模块包含设备数据 `b_d`：

```
1  module b_m
2    integer , device :: b_d
3  end module b_m
```

文件 `a.cuf` 包含模块 `a_m`，该模块里的一个内核使用(以 Fortran 90 的方式)模块 `b_m`：

```
1  module a_m
2    integer , device :: a_d
3  contains
4    attributes(global) subroutine aPlusB()
5      use b_m
6      implicit none
7      a_d = a_d + b_d
8    end subroutine aPlusB
9  end module a_m
```

这个模块反过来被主机代码 `aPlusB.cuf` 使用：

```
 1  program twoPlusThree
 2    use a_m
 3    use b_m
 4    implicit none
 5    integer :: a
 6
 7    a_d = 2
 8    b_d = 3
 9    call aPlusB <<<1,1>>>()
10    a = a_d
11    write(*,"('2+3=',i0)") a
12  end program twoPlusThree
```

接着，整个应用可以用下面一系列命令来编译和运行：

```
% pgf90 -c b.cuf
% pgf90 -c a.cuf
% pgf90 aPlusB.cuf a.o b.o
aPlusB.cuf:
% ./a.out
2+3=5
```

跨模块共享设备数据简单明了，但自从编译器 13.3 才能跨模块使用设备例程。分离编译仅在计算能力 2.0 或更高的设备上可用，且要求 5.0 或更高版本的 CUDA 工具包。用下面的例子演示跨模块使用设备代码。文件 `d.cuf` 中定义模块 `d_m`，该模块包含设备数据 `d_d` 以及例程 `negateD()`：

```
1  module d_m
2    integer , device :: d_d
```

```
3  contains
4    attributes(device) subroutine negateD()
5      d_d = -d_d
6    end subroutine negateD
7  end module d_m
```

用 attributes(device) 声明的例程此前没有见过。这样的例程像内核一样在设备上执行，但从设备代码(内核和其它 attributes(device)代码)中调用而不是从主机代码调用，例如文件 c.cuf 中内核 cMinusD()的第 7 行上：

```
1  module c_m
2    integer , device :: c_d
3  contains
4    attributes(global) subroutine cMinusD()
5      use d_m
6      implicit none
7      call negateD()
8      c_d = c_d + d_d
9    end subroutine cMinusD
10  end module c_m
```

注意，调用例程 negateD()时没有像启动一个内核时那样提供执行配置。它的调用方式与 Fortran 90 子程序或函数的调用方式一样。调用一个 attributes(device) 函数时并没有启动一个内核，因为遭遇该调用时函数由已经存在的设备线程来执行。必须指出，内核中可用的预定义变量(threadIdx、blockIdx、blockDim 和 gridDim) 在用 attributes(device)声明的代码里也可用，这个仅由一个设备线程执行的简单例子里面没有用到这些变量。这个例子的主机代码是：

```
1  program twoMinusThree
2    use c_m
3    use d_m
4    implicit none
5    integer :: c
6
7    c_d = 2
8    d_d = 3
9    call cMinusD <<<1,1>>>()
10   c = c_d
11   write(*,"('2-3=',i0)") c
12 end program twoMinusThree
```

如果用前面编译 b.cuf 和 a.cuf 的方式来编译文件 d.cuf 和 c.cuf，将得到下面的错误：

```
% pgf90 -c d.cuf
% pgf90 -c c.cuf
PGF90 -S-0155-Illegal call of a device routine from another module
- negated (c.cuf: 7)
  0 inform , 0 warnings , 1 severes , 0 fatal for cminusd
```

为了使设备例程能够跨模块访问，编译、连接两个步骤都需要使用-Mcuda=rdc 选项，该选项也称为可重

分配设备代码 (relocatable device code)：

```
% pgf90 -Mcuda=rdc -c d.cuf
% pgf90 -Mcuda=rdc -c c.cuf
% pgf90 -Mcuda=rdc cMinusD.cuf c.o d.o
cMinusD.cuf:
% ./a.out
2-3=-1
```

使用选项-Mcuda=rdc 时必须显式指定一个大于等于 2.0 的计算能力或 CUDA 5 工具箱，编译器清楚这些选项和隐式包含的必要选项等特性要求哪种架构和哪个工具包版本。用选项-Mcuda=ptxinfo 可以查看到，当用-Mcuda=rdc 编译时，默认以计算能力 2.0 和 3.0 为目标：

```
$ pgf90 -Mcuda=rdc ,ptxinfo -c c.cuf
ptxas info : 16 bytes gmem , 8 bytes cmem[14]
ptxas info : Compiling entry function 'c_m_cminusd_' for 'sm_20'
ptxas info : Function properties for c_m_cminusd_
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info : Used 8 registers , 32 bytes cmem[0]
ptxas info : 16 bytes gmem
ptxas info : Compiling entry function 'c_m_cminusd_' for 'sm_30'
ptxas info : Function properties for c_m_cminusd_
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info : Used 8 registers , 320 bytes cmem[0]
```

第2章 性能测量和评价指标

性能优化的一个先决条件是要有一个对代码区段精确计时的方法，然后描述如何利用这些计时信息评估代码的性能。本章首先论述如何使用 CPU 计时器、CUDA 事件、命令行测绘器和 nvprof 测绘工具对内核执行进行计时。接下来论述如何利用计时信息找出内核执行的限制因素。最后论述如何计算性能指标，特别是与带宽相关的指标，以及该如何解读这些指标。

2.1 测量内核执行时间

测量内核执行时间有好几种方法。这里使用传统的 CPU 计时器，但为保证测量精确，操作中必须确保主机与设备间的正确同步。从主机代码调用的 CUDA 事件接口例程可以使用设备时钟为内核执行计时。最后论述如何使用命令行测绘器器和 nvprof 测绘工具给出这些计时信息。

2.1.1 主机与设备同步和 CPU 计时器

用传统 CPU 计时器时必须小心。从主机的角度看，内核执行和许多 CUDA Fortran 接口例程一样是非阻塞的或者称为异步的：在完成 GPU 上的工作之前它们就已将控制权返回至主调 CPU 线程。例如，考虑如下代码段：

```
1  a_d = a
2  call increment <<<1,n>>>(a_d , b)
3  a = a_d
```

第二行的内核 increment 一旦启动，控制权立刻返还给 CPU. 与此相反，启动内核前后的数据传输却是同步的，或者称为阻塞的。在前面递交的所有 CUDA 调用都已经完成之前，这些数据传输不会开始；在所有的数据传输都已经完成[1]之前，后续的 CUDA 调用也不会开始。由于内核执行相对于主机线程是异步的，在调用语句前后使用 CPU 计时器只能简单地记录内核启动的时间。为了用主机代码计时器对内核执行精确地计时，需在利用 cudaDeviceSynchronize() 显式地同步 CPU 线程：

```
1  a_d = a
2  t1 = myCPUTimer()
3  call increment <<<1,n>>>(a_d , b)
4  istat = cudaDeviceSynchronize()
5  t2 = myCPUTimer()
6  a = a_d
```

函数 cudaDeviceSynchronize() 阻止调用主机线程，直至前面由主机线程递交的所 CUDA 调用均已完成，这个效果恰是正确测量 increment 所必需的。最佳方法是在任何计时调用之前都调用一下 cudaDeviceSynchronize(). 例如，既使不是必须要求，也最好在第 2 行之前插入一个 cudaDeviceSynchronize() ,因为程序员可能会将第 1 行的传输改为异步传输却忘记添加这个同步调用了。

函数 cudaDeviceSynchronize() 的一个替代方法是将环境变量 CUDA_LAUNCH_BLOCKING 设定为

[1] 注意，利用例程 cudaMemcpy*Async() 可以实现异步数据传输，详述在 3.13 节。

1, 它将内核调用变成同步函数调用。然而，这将应用到程序中所有的内核启动，从而任何带有内核执行的CPU 代码都将被串行化。

2.1.2 通过 CUDA 事件计时

用函数 cudaDeviceSynchronize()和环境变量 CUDA_LAUNCH_BLOCKING 产生主机与设备间的同步点有一个问题，它们会暂停 GPU 的处理管线。可惜，这些同步点是使用 CPU 计时器所必需的。所幸，CUDA 提供一个相对轻量级的 CUDA 事件接口来替代 CPU 计时器。CUDA 事件接口提供的调用可以创建和销毁事件、记录事件（通过一个 GPU 时间戳）、将时间戳的差值转换为一个以毫秒为单位的浮点数值。

CUDA 事件利用了 CUDA 流的概念，编写 CUDA 事件代码之前需要就 CUDA 流说几句。一个 CUDA 流仅仅是一系列在设备上顺序执行的操作。不同流中的操作可以交错执行，甚至某些情形下还可以重叠执行—这个性质能用来隐藏主机与设备间的数据传输，后有详述。到目前为止，GPU 上的所有操作全部发生在默认流即零号流之中。

这里秀出事件接口的典型用法：

```
type(cudaEvent) :: startEvent , stopEvent
real :: time
integer :: istat

istat = cudaEventCreate(startEvent)
istat = cudaEventCreate(stopEvent)

a_d = a
istat = cudaEventRecord(startEvent , 0)
call increment <<<1,n>>>(a_d , b)
istat = cudaEventRecord(stopEvent , 0)
istat = cudaEventSynchronize(stopEvent)
istat = cudaEventElapsedTime(time , startEvent , stopEvent)
a = a_d

if (any (a/= 4)) then
  write(*,*) '**** Program Failed ****'
else
  write(*,*) ' Time for kernel execution (ms): ', time
endif

istat = cudaEventDestroy(startEvent)
istat = cudaEventDestroy(stopEvent)
```

CUDA 事件属于 cudaEvent 类型，分别用 cudaEventCreate()和 cudaEventDestroy()来创建和销毁。这个代码里，cudaEventRecord()用来将开始事件和结束事件放入默认流，即零号流。每当到达流中的一个事件时，设备就会为该事件记下一个时间戳。函数 cudaEventElapsedTime()返回 GPU 上开始事件与结束事件之间流逝的时间。这个数值以微秒为单位表示，精确度接近半微秒。因为是 cudaEventRecord()非阻塞的，所以要求在调用 cudaEventElapsedTime()之前有一个同步，以确保

stopEvent 事件能够被记录下来，这就是第 12 行调用 cudaEventSynchronize() 的原因。cudaEventSynchronize()阻塞 CPU 执行，直到指定的事件被记录在 GPU 上。

对非常简单的内核（例如例子 increment），CPU 端的抖动会导致用 CUDA 事件计时会有稍许不精确。这种情况下，简单地在第一个 CUDA 事件调用的紧邻上方添加一个无操作内核(初始化相关设备—译者注)，从而 cudaEventRecord()和后内核调用都将在 GPU 上排队，这样就能得到更加精确的结果。

2.1.3 命令行测绘器(profiler)

计时信息也可以从命令行测绘器获得。这种方法不必像 CUDA 事件那样注入代码。它甚至不需要用特殊的选项重编译源代码。就像分析 CUDA C 代码时做的那样，将环境变量 COMPUTE_PROFILE 设定为 1 能开打测绘。其它几个环境变量控制着测绘什么以及结果输出到哪里。这些环境变量的论述包含在 B.1.2 节中，但现在仅讨论 COMPUTE_PROFILE 设定为 1 时这个简单例子的输出。命令行分析器的输出默认发送到文件 cuda_profile_0.log;输出包含一些基本信息，诸如方法名字、GPU 和 CPU 的执行时长、内核执行的占用率。例如，这里就是 1.3.3 节中多维数组自加代码的测绘器输出：

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff693dc2e2f28
method ,gputime ,cputime ,occupancy
method=[memcpyHtoD] gputime =[382.304] cputime =[712.000]
method=[memcpyHtoD] gputime =[1.632] cputime =[8.000]
method=[increment] gputime =[153.472] cputime =[24.000]
   occupancy =[1.000]
method=[memcpyDtoH] gputime =[433.504] cputime =[1787.000]
```

前四行是头信息，包括执行代码的设备号和设备名。第五行列出域，下面每个被执行的方法都需要显示。默认情况下，这些域是正在执行的方法的名字、GPU 报告的微秒时间、CPU 报告的微秒时间、占用率，占用率仅为内核执行报告。每个多处理器上的实际并发线程数与最大可能线程数的比例就是*占用率*。3.5.1 节将详述占用率。后面各行显示了每个方法的测绘结果。主机到设备的数据传输有两种：第一种是数组的传输，第二种是内核执行参数和数据参数的传输，参数传输由 CUDA 运行时隐式完成。接下来内核 increment 启动，再后面就是结果数组的设备到主机传输。

测绘器输出的 gputime 域的解释一目了然—GPU 报告的微秒时间—但解释 cputime 时要留意几分。对诸如内核这样的非阻塞方法，cputime 报告的值仅仅是启动该方法的 CPU 开销，这种情况下墙钟时间是 cputime+gputime. 对诸如数据传输这样的阻塞方法，cputime 包括 gputime 和 CPU 开销，因此等价于墙钟时间。除了启动开销，第一个被调用方法的计时还包括相关设备的初始化开销。

注意，数据传输时间比内核执行时间大得多。部分原因是这里采用一个非常简单的内核，而 PCIe 总线上的数据传输常常是一个性能瓶颈。后续的优化章节将论述如何最小化和隐藏此类传输。

就像前面提到的，除 COMPUTE_PROFILE 之外还有几个环境变量，它们决定测绘什么以及输出结果如何排版。它们的详细论述见 B.1.2 节。

2.1.4 测绘工具 nvprof

命令行测绘器的一个替代工具是包含在 CUDA 5 工具包中的 nvprof 应用。命令行测绘器和 nvprof 是互斥的，因此使用 nvprof 时必须将 COMPUTE_PROFILE 设定为 0. 除了这个注意事项外，nvprof 的使用方法简单到只需将 CUDA 应用程序命令作为 nvprof 的一个参数来运行。再次使用多维自加代码，运行 nvprof ./a.out 时得到下列输出：

```
======== NVPROF is profiling a.out...
======== Command: a.out
 Program Passed
======== Profiling result:
Time(%) Time Calls Avg Min Max Name
44.56 385.19us 2 192.59us 1.31us 383.88us [CUDA memcpy HtoD]
37.93 327.84us 1 327.84us 327.84us 327.84us [CUDA memcpy DtoH]
17.51 151.36us 1 151.36us 151.36us 151.36us increment
```

这个输出中，每个方法的所有调用被总结为一行，例如两个主机到设备的数据复制只占一行。利用选项 --print-gpu-trace 可以将所有调用分别输出。

结束论述命令行测绘器和 nvprof 之前，需要注意这些工具均是“跟踪”执行，即收集时间线数据。也可以利用这些工具“测绘”执行，即收集硬件计数器。执行 nvprof --query-events 能得到可用来监视的一个硬件计数器列表。收集硬件计数器比收集时间线数据更加深入，这也导致某并发操作可能被串行化。

nvprof 和命令行测绘器的更多信息，参见 CUDA 5 工具包提供的 CUDA 测绘器用户指南(*CUDA Profiler Users Guide*).

2.2 指令约束、带宽约束和延时约束的内核

现在已经有能力对内核执行计时，可以说说如何探测一个内核执行的限制因素了。有好几种方法可以达到这个目标。一个可选方案是利用测绘器的硬件计数器，但这个分析方法使用的计数器好像每一代硬件都会发生改变。作为替代方案，本节描述的方法更加通用，无论是哪代硬件，同一个程序都能正常工作。实际上，这种方法不但能用到 GPU 上而且能够用到 CPU 平台上。这种方法会为内核创建多个版本，它们暴露出全内核的内存密集方面和数学密集方面。对每一个内核计时，对比这些时间就能暴露出内核执行的限制因素。通读一个例子才能真正理解这个过程。接下来的代码包含三个内核：

- 基础(base)内核，执行期望的全部操作
- 内存(memory)内核，具有与基础内核相同的设备内存访问模式，但没有数学操作
- 数学(math)内核，执行基础内核的数学操作，但不访问全局内存

```
1  module kernel_m
2  contains
3    attributes(global) subroutine base(a, b)
4      real :: a(*), b(*)
5      integer :: i
6      i = (blockIdx%x-1)* blockDim%x + threadIdx%x
7      a(i) = sin(b(i))
8    end subroutine base
```

```
 9
10    attributes(global) subroutine memory(a, b)
11      real :: a(*), b(*)
12      integer :: i
13      i = (blockIdx%x-1)* blockDim%x + threadIdx%x
14      a(i) = b(i)
15    end subroutine memory
16
17    attributes(global) subroutine math(a, b, flag)
18      real :: a(*)
19      real , value :: b
20      integer , value :: flag
21      real :: v
22      integer :: i
23      i = (blockIdx%x-1)* blockDim%x + threadIdx%x
24      v = sin(b)
25      if (v*flag == 1) a(i) = v
26    end subroutine math
27  end module kernel_m
28
29  program limitingFactor
30    use cudafor
31    use kernel_m
32    implicit none
33    integer , parameter :: n=8*1024*1024 , blockSize = 256
34    real :: a(n)
35    real , device :: a_d(n), b_d(n)
36
37    b_d = 1.0
38    call base <<<n/blockSize ,blockSize >>>(a_d , b_d)
39    call memory <<<n/blockSize ,blockSize >>>(a_d , b_d)
40    call math <<<n/blockSize ,blockSize >>>(a_d , 1.0, 0)
41    a = a_d
42    write(*,*) a(1)
43  end program limitingFactor
```

对数学内核，必须注意骗过编译器的技巧，因为编译器能够探测并去除对设备内存无写入的操作。因此，需要将写入操作放在永远为假的条件语句里，前面代码的第 25 行就是这样做的。这个条件应该不仅依赖于传入子例程的一个标签，还要依赖于一个中间结果；否则，编译器会将整个操作移入条件中。

如果在一块 Tesla C2050 上运行该代码，使用命令行分析器，得到三个内核的如下输出：

```
method=[base] gputime =[850.912] cputime =[5.000] occupancy =[1.000]
method=[memory] gputime =[625.920] cputime =[6.000] occupancy =[1.000]
```

```
method=[math] gputime =[784.384] cputime =[5.000] occupancy =[1.000]
```

比较多个内核的 gputime，观察到数学操作和内存操作有相当数量的重叠，因为数学[2]内核与内存内核的 gputime 之和比基础内核的 gputime 大。因为数学内核的时间是基础内核的 92%，而内存内核的时间是基础内核的 73%, 所以这个例子的性能限制因素就是数学操作。如果不要求完全精度，可以利用快速的内置数学函数加速数学内核，简单地用选项-Mcuda=fastmath 重新编译，就能用硬件来计算函数 sin()．结果为：

```
method=[base] gputime =[635.424] cputime =[7.000] occupancy =[1.000]
method=[memory] gputime =[626.336] cputime =[7.000] occupancy =[1.000]
method=[math] gputime =[261.280] cputime =[7.000] occupancy =[1.000
```

正如所料，数学内核的时间显著下降，基础内核时间也随之下降。现在，基础内核受内存约束，因为内存内核和基础内核的运行时间几乎相同 :数学操作几乎完全被内存操作隐藏。此时此刻 ,如果可以优化的话，进一步改善只能靠优化设备内存访问。

在一块 K20 上运行这个代码，并行监测其执行，观察到另外一番景象：

```
method=[base] gputime =[529.568] cputime =[7.000] occupancy =[1.000]
method=[memory] gputime =[473.792] cputime =[7.000] occupancy =[1.000]
method=[math] gputime =[273.344] cputime =[8.000] occupancy =[1.000]
```

比对 Tesla K20 和 C2050 的测绘器输出，除了内核在 Tesla K20 上跑得更快之外，基础内核在 K20 上受到的内存约束比 C2050 上更加严重。可以预料，用选项-Mcuda=fastmath 重新编译在 K20 上提高性能的百分比将没有 C2050 上那么多，这个现象也能从测绘器的输出中观察到：

```
method=[base] gputime =[481.632] cputime =[7.000] occupancy =[1.000]
method=[memory] gputime =[474.816] cputime =[6.000] occupancy =[1.000]
method=[math] gputime =[210.624] cputime =[8.000] occupancy =[1.000]
```

再一次，用选项-Mcuda=fastmath 时，基础内核受到内存约束，进一步的改善只能靠优化设备内存访问。是否可以提高内存访问促使下一节对内存带宽的研究。但在进入带宽评价指标之前，需要对这种修改源码以探测内核限制因素的作法做一些总结。当数学操作和内存操作重叠非常小的时候，内核很可能受到延时的约束。这种情况常常发生在占用率较低的时侯；某一时刻设备上没有足够的线程来重叠任何操作。这种情形的应对办法常常是修改执行配置。

这个分析中使用测绘器来测量时间有两个原因。第一个就是它不要求注入主机代码（已经额外写了两个内核，因此测绘器受欢迎）。第二个是希望确保所有内核都有相同的占用率。从内核中移除数学操作时，很可能减少了寄存器使用量（可以用选项-Mcuda=ptxinfo 检验）。如果寄存器用量变化足够大，那么占用率，或者称为驻留在一个多处理器上的实际线程数量与最大数量的比例，就会发生变化，进而影响运行时长。在本例中，占用率处处都是 1.0，如要不是这样，可以用执行配置的第三个参数来在内核中分配动态共享内存，降低占用率。第三个参数是可选项，它是动态分配的共享内存的字节数，分配的共享内存供每一个线程块使用。3.3.3 节会进一步讲述共享内存，但目前仅需知道，简单地提供每个线程块需要的字节数就能为一个线程块预留一块共享内存，这里的字节数就是执行配置的第三个参数。

2.3 内存带宽

回到 2.2 节的示例代码，利用内置快速数学函数减少花费在计算 sin()上的时间之后，留下了一个内

[2] 原文为“基础内核”，有误，已更正。--译者注。

存约束内核。这个阶段要探究内存系统的使用情况以及是否有提升空间。为回答这个问题，必须计算内存带宽。

带宽—传速数据的速度—是性能的最关键因素之一。几乎对代码的所有修改都要从它们如何影响带宽方面来考虑。存储哪些数据、数据如何布局、数据的访问顺序和其它因素都会剧烈地影响带宽。

评定内存效率时，要用到理论最大内存带宽和实际内存带宽。当一个代码受内存约束且有效内存带宽远低于最大带宽时，优化努力就应该集中于提高有效带宽。

2.3.1 理论最大带宽

理论最大内存带宽可以从内存时钟频率和内存总线宽度计算出来。这两个量值可通过设备管理接口查询得到，下面的代码计算所有附属设备的理论最大带宽，同时演示了查询方法。

```
 1  program peakBandwidth
 2    use cudafor
 3    implicit none
 4
 5    integer :: i, istat , nDevices=0
 6    type (cudaDeviceProp) :: prop
 7
 8    istat = cudaGetDeviceCount(nDevices)
 9    do i = 0, nDevices -1
10      istat = cudaGetDeviceProperties(prop , i)
11      write(*,"(' Device Number: ',i0)") i
12      write(*,"(' Device name: ',a)") trim(prop%name)
13      write(*,"(' Memory Clock Rate (KHz): ', i0)") &
14           prop%memoryClockRate
15      write(*,"(' Memory Bus Width (bits): ', i0)") &
16           prop%memoryBusWidth
17      write(*,"(' Peak Memory Bandwidth (GB/s): ', f6.2)") &
18           2.0 * prop%memoryClockRate * &
19           (prop%memoryBusWidth / 8) * 1.e-6
20      write(*,*)
21    enddo
22  end program peakBandwidth
```

在最大内存带宽的计算过程中，因子 2.0 对应于每个时钟周期内随机访问内存的双倍数据速率，除以 8 是将总线宽度由位转换为字节，因子 `1.e-6` 处理千赫兹到赫兹和字节到吉字节的转换[3]。在多种 Tesla 硬件上运行此代码，得到：

```
Device Number: 0
  Device name: Tesla C870
  Memory Clock Rate (KHz): 800000
  Memory Bus Width (bits): 384
```

[3] 注意，某些计算中用 $1{,}024^3$ 而不是用 10^9 来将字节换算成吉字节。无论采用哪个因子，重要的是用同一个因子来计算理论带宽和有效带宽，这样的比较才合理。

```
  Peak Memory Bandwidth (GB/s): 76.80
```

```
Device Number: 0
  Device name: Tesla C1060
  Memory Clock Rate (KHz): 800000
  Memory Bus Width (bits): 512
  Peak Memory Bandwidth (GB/s): 102.40
```

```
Device Number: 0
  Device name: Tesla C2050
  Memory Clock Rate (KHz): 1500000
  Memory Bus Width (bits): 384
  Peak Memory Bandwidth (GB/s): 144.00
```

```
Device Number: 0
  Device name: Tesla K10.G1.8GB
  Memory Clock Rate (KHz): 2500000
  Memory Bus Width (bits): 256
  Peak Memory Bandwidth (GB/s): 160.00
Device Number: 1
  Device name: Tesla K10.G1.8GB
  Memory Clock Rate (KHz): 2500000
  Memory Bus Width (bits): 256
  Peak Memory Bandwidth (GB/s): 160.00
```

```
Device Number: 0
  Device name: Tesla K20
  Memory Clock Rate (KHz): 2600000
  Memory Bus Width (bits): 320
  Peak Memory Bandwidth (GB/s): 208.00
```

对带有错误校正代码(error-correcting code, ECC)的设备，例如 Tesla C2050、K10、和 K20，需要考虑到，当 ECC 打开时，最大带宽将会减小。

2.3.2 有效带宽(Effective Bandwidth)

对指定的程序活动计时并且知道该程序如何访问数据，就可以计算有效带宽了。用这个公式计算：

$$BW_{\text{Effective}} = \frac{(R_B + W_B)/10^9}{t}$$

这里的$BW_{\text{Effective}}$是以 GB/s 为单位的有效带宽，R_B是每个内核读取的字节数，W_B是每个内核写入的字节数，t是以秒为单位给出的流逝时间。

在多种设备上运行一个简单的复制数据内核，例如 2.2 节限制因素代码里的 memory()内核，用来获取

有效带宽是很有用的。表 2.1 列出了一个简单复制数据内核多次运行中获得的最好有效带宽，内核运行时使用不同的数组尺寸、不同的执行配置，在支持 ECC 的设备上测试 ECC 打开和关闭时的情况[4]。

表 2.1 一个简单复制数据内核的实际带宽。修改了块尺寸和数组长度以获得每种情况下的最佳结果。.

	复制数据的实际带宽(GB/s)			
	Tesla C870	**Tesla C1060**	**Tesla C2050**	**Tesla K20**
ECC 关	65	78	119	164
ECC 开	–	–	107	145

返回到 2.2 节的例子，那里对 8×1024^2 个元素中的每个都执行一次读和写，接下来的公式用来计算使用 `-Mcuda=fastmath` 选项时的基础方法在 C2050（ECC 打开）上的有效带宽：

$$BW_{\text{Effective}} = \frac{(8 \times 1024^2 \times 4 \times 2)/10^9}{635 \times 10^{-6}} = 106\ GB/s$$

元素数量乘以每个元素的尺寸(一个 float 4 个字节)，再乘以 2(因为有读和写)，再除以 10^9 得到内存传输的 GB 数量。基础内核的测绘器结果给出 GPU 时间是 635 μs，由此得到有效带宽大约是 106 GB/s. 这个结果可以与 C2050 的 144 GB/s 理论最大带宽相比较，但这没有考虑 ECC 的影响。替代地，可以用 表 2.1 中的合适数字 107 GB/s 比较。从而，在这上设备上不指望本代码能获得更多的实质性加速。

为获得这个内核在 Tesla K20 上的有效带宽，再次打开 ECC，简单地在前面的公式里替换上基础内核 481 μs 的测绘器时间，得到一个数值 139 GB/s. 与表 2.1 中的数值 145 GB/s 对比，再一次，不指望这个代码能在这个设备获得更多的实质性加速。

2.3.3 实际数据吞吐率对阵有效带宽

使用测绘器的计数器可以估计数据吞吐率。这样计算出来的吞吐率与 2.3.2 节中计算出来的有效带宽对比时，一定要小心谨慎。一个区别是，测绘器使用一部分 GPU 多处理器测量传输，然后将这个数字外推至整个 GPU，因此它报告的是一个估算的数据吞吐率。

需要注意的另一个区别是，所用计数器表示的是实际数据的吞吐率量，还是被请求数据的吞吐量。这个区别很重要，因为最小传递尺寸大于大多数字长，所以实际数据的吞吐量将大于等于被请求数据的吞吐量。有效带宽是根据与算法相关的数据计算出来的，因此对应于被请求数据吞吐率。实际数据吞吐率和被请求数据吞吐率都很有用。实际数据吞吐率显示出代码接近硬件极限的程度，对比有效带宽和实际吞吐率能显示有多少带宽被不理想的内存访问模式所浪费。

实际数据吞吐率和有效带宽的差异还不是目前示例代码的议题，因为所有数据访问都使用连续数据。但以跳跃方式访问内存时，第 3 章会深入探讨，实际数据吞吐率和有效带宽的数值就会有显著差别。

[4] 附录 B 中 `nvidia-smi` 那节论述如何打开和关闭 ECC.

第3章 优化

之前章节论述如何利用计时信息来确定内核执行的限制因素。事实表明，许多科学和工程代码都受带宽约束，这也是为什么这个相对较长的一章将大部分篇幅都放在内存优化上。CUDA 设备有多种不同的内存类型，为了高效地编程，需要有效利用这些内存类型。

数据传输可分为两个主要类别：主机内存和设备内存之间的数据传输；设备上不同内存之间的数据传输。先讲化主机与设备间传输的优化，然后讲述设备上的不同类型内存以及如何有效地使用它们。为演示这些内存的多种优化技术，本章将完整讲解一个优化矩阵转置内存的例子。

除了内存优化，本章还讲述如何选取执行配置参数的决定因素，从而能够有效地利用硬件。最后讲述指令优化。

3.1 主机与设备间的传输

设备内存和 GPU 之间的顶峰带宽（例如，在 Tesla K20 上为 208GB/s）要远高于主机内存和设备内存之间的顶峰带宽(在 PCIe x16 Gen3 上为 16GB/s, 在 PCIe x16 Gen2 上为 8GB/s)。因此，为达到程序的最佳整体性能，尽量最小化主机与设备间的数据传输就格外重要，必须传输数据时，要确保这些传输已经优化过。

当刚开始编写或移植一个应用到 CUDA Fortran 上时，通常先将几段关键代码转换为 CUDA Fortran 内核。如果这些代码段是相互孤立的，那么它们将需要到达主机或源自主机的数据传输，整体性能也可能受这些数据传输的限制。在这个阶段，对带与不带这些传输的代码分别评定性能是有帮助的。包括数据传输在内的总时间是当前代码性能的一个精确评定，不带这些传输的时间表示的是，当更多的代码改写为在设备上运行时能达到的性能。在这个阶段不应该在主机与设备间传输的优化上花费时间，因为随着更多的主机代码被转换为内核，许多这样的中间数据传输将会消失。当然，总会需要一些主机与设备间的传输，程序员需要保证它们尽可能高效地执行，但是不值得花时间去优化那些最终将被移除的数据传输。

从执行时间上看，可能会有些操作在设备上运行时没有显示出任何加速。如果在主机上执行，这些操作将要求额外的主机与设备间传输，因此在设备上执行这些操作可能对整体性能有利。

在有些情形中，可以避免主机与设备间的数据传传输。可以在设备内存中创建中间数据结构，并在设备上操作和销毁，永远不需要被主机端映射也不需要复制到主机内存中。

至此，已经讲述了如何尽可能避免主机与设备间的传输。本节后续部分将讲述如何高效地执行必要的主机与设备间传输。它包括使用钉固内存、一起执行批量小传输和异步数据传输。

3.1.1 钉固内存(Pinned Memory)

为驻留在主机上的变量分配内存时，默认使用可分页内存。可分页内存可以交换到硬盘上，从而允许程序使用比主机系统 RAM 上可用容量更多的内存。当数据在主机与设备间传输时，GPU 上的直接内存访问(direct memory access, DMA)引擎必须瞄准页锁定(page-locked)或钉固(*pinned*)主机内存。钉固内存不能被交换出去，因此对这样的传输来说它总是可用的。为容许可分页内存到 GPU 的数据传输，主机操作系统先分配一个临时钉固主机缓冲区，将数据复制到这个钉固缓冲区，然后将数据传输到设备，如图 3.1 所示。钉固内存缓冲区可能比保存主机数据的可分页内存小，这种情况下要分多阶段传输。类似地，设备到主机的传输中也使用钉固内存缓冲区。如果将主机数组声明为使用钉固内存，那么就可以避免可分页内存与钉固

主机缓冲区间的传输开销。

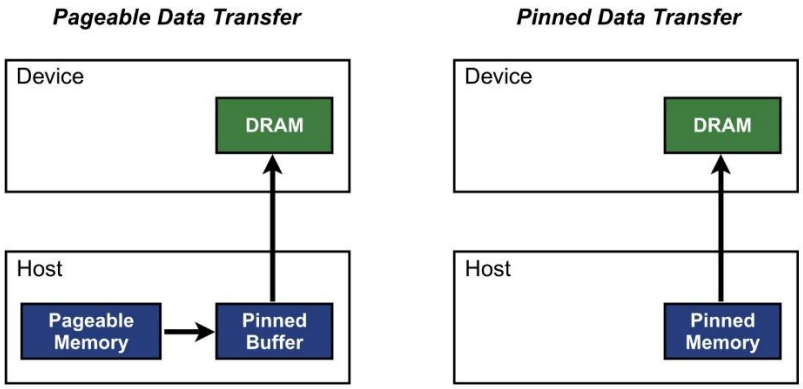


图 3.1

从可分页主机内存（左）和钉固主机内存（右）出发的主机到设备传输。对可分页主机内存，数据被传到设备上之前，先被传送到主机上的一个临时钉固内存缓冲区。右边从一开始就使用钉固内存，去掉了外的主机数据复制。

CUDA Fortran 中，变量修饰词 `pinned` 表示使用钉固内存，并且这种类型的内存还必须使用变量修饰词 `allocatable` 声明为可分配的。`allocate` 语句可能会分配钉固内存失败，这种情况下将尝试分配可分页内存。接下来的代码将展示带有错误检查的钉固内存分配，还将展示使用钉固内存时可以期待的加速：

```
 1  program BandwidthTest
 2
 3    use cudafor
 4    implicit none
 5
 6    integer , parameter :: nElements = 4*1024*1024
 7
 8    ! host arrays
 9    real(4) :: a_pageable(nElements), b_pageable(nElements)
10    real(4), allocatable , pinned :: a_pinned(:), b_pinned(:)
11
12    ! device arrays
13    real(4), device :: a_d(nElements)
14
15    ! events for timing
16    type (cudaEvent) :: startEvent , stopEvent
17
18    ! misc
19    type (cudaDeviceProp) :: prop
20    real(4) :: time
21    integer :: istat , i
22    logical :: pinnedFlag
23
24    ! allocate and initialize
25    do i = 1, nElements
```

```
26        a_pageable(i) = i
27     end do
28     b_pageable = 0.0
29
30     allocate(a_pinned(nElements), b_pinned(nElements), &
31     STAT=istat , PINNED=pinnedFlag)
32     if (istat /= 0) then
33        write(*,*) 'Allocation of a_pinned/b_pinned failed'
34        pinnedFlag = .false.
35     else
36        if (.not. pinnedFlag) write(*,*) 'Pinned allocation failed'
37     end if
38
39     if (pinnedFlag) then
40        a_pinned = a_pageable
41        b_pinned = 0.0
42     endif
43
44     istat = cudaEventCreate(startEvent)
45     istat = cudaEventCreate(stopEvent)
46
47     ! output device info and transfer size
48     istat = cudaGetDeviceProperties(prop , 0)
49
50     write(*,*)
51     write(*,*) 'Device: ', trim(prop%name)
52     write(*,*) 'Transfer size (MB): ', 4*nElements /1024./1024.
53
54     ! pageable data transfers
55     write(*,*)
56     write(*,*) 'Pageable transfers'
57
58     istat = cudaEventRecord(startEvent , 0)
59     a_d = a_pageable
60     istat = cudaEventRecord(stopEvent , 0)
61     istat = cudaEventSynchronize(stopEvent)
62
63     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
64     write(*,*) ' Host to Device bandwidth (GB/s): ', &
65          nElements*4/time/1.e+6
66
67     istat = cudaEventRecord(startEvent , 0)
68     b_pageable = a_d
69     istat = cudaEventRecord(stopEvent , 0)
```

```
70    istat = cudaEventSynchronize(stopEvent)
71
72    istat = cudaEventElapsedTime(time , startEvent , stopEvent)
73    write(*,*) ' Device to Host bandwidth (GB/s): ', &
74         nElements*4/time/1.e+6
75
76    if (any(a_pageable /= b_pageable)) &
77         write(*,*) '*** Pageable transfers failed ***'
78
79    ! pinned data transfers
80    if (pinnedFlag) then
81       write(*,*)
82       write(*,*) 'Pinned transfers'
83
84       istat = cudaEventRecord(startEvent , 0)
85       a_d = a_pinned
86       istat = cudaEventRecord(stopEvent , 0)
87       istat = cudaEventSynchronize(stopEvent)
88
89       istat = cudaEventElapsedTime(time , startEvent , stopEvent)
90       write(*,*) ' Host to Device bandwidth (GB/s): ', &
91            nElements*4/time/1.e+6
92
93       istat = cudaEventRecord(startEvent , 0)
94       b_pinned = a_d
95       istat = cudaEventRecord(stopEvent , 0)
96       istat = cudaEventSynchronize(stopEvent)
97
98       istat = cudaEventElapsedTime(time , startEvent , stopEvent)
99       write(*,*) ' Device to Host bandwidth (GB/s): ', &
100           nElements*4/time/1.e+6
101
102      if (any(a_pinned /= b_pinned)) &
103           write(*,*) '*** Pinned transfers failed ***'
104   end if
105
106   write(*,*)
107
108   ! cleanup
109   if (allocated(a_pinned)) deallocate(a_pinned)
110   if (allocated(b_pinned)) deallocate(b_pinned)
111   istat = cudaEventDestroy(startEvent)
112   istat = cudaEventDestroy(stopEvent)
113
```

```
114   end program BandwidthTest
```

钉固内存的分配在第 30 行上执行，它带有可选关键字参数 STAT 和 PINNED，这些参数可以用来检查分配是否完成，如果已经完成，检查分配结果是否在钉固内存中，就像 32-37 行做的那样。

数据传输速率依赖于主机系统类型和 GPU 类型。例如，在带有一块 Tesla K20 的 Intel Xeon E5540 系统上，本代码得到：

```
  Device: Tesla K20
  Transfer size (MB):     16.00000

  Pageable transfers
    Host to Device bandwidth (GB/s):     1.659565
    Device to Host bandwidth (GB/s):     1.593377

  Pinned transfers
    Host to Device bandwidth (GB/s):     5.745055
    Device to Host bandwidth (GB/s):     6.566322
```

然而，在 Intel Xeon E5-2667 系统上，同样带有一块 Tesla K20，得到：

```
  Device: Tesla K20m
  Transfer size (MB):     16.00000

  Pageable transfers
    Host to Device bandwidth (GB/s):     3.251782
    Device to Host bandwidth (GB/s):     3.301395

  Pinned transfers
    Host to Device bandwidth (GB/s):     6.213710
    Device to Host bandwidth (GB/s):     6.608200
```

这两个系统之间的钉固数据传输速率相似。然后，主机与设备间的可分页数据传输速率受主机系统影响很大，这归咎于主机端从可分页内存到钉固缓冲区的隐式复制。

在测绘器配置文件中指定选项 memtransferhostmemtype，就可以从命令行测绘器中验证主机与设备之间的一次数据传输中是否使用了钉固主机内存。例如，测绘 BandwidthTest 代码，得到：

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff69b0066e8b8
method, gputime, cputime, occupancy, memtransferhostmemtype
method=[ memcpyHtoD ] gputime=[ 9018.912 ] cputime=[ 9937.000 ]
       memtransferhostmemtype=[ 0 ]
method=[ memcpyDtoH ] gputime=[ 9216.160 ] cputime=[ 10160.000 ]
       memtransferhostmemtype=[ 0 ]
method=[ memcpyHtoD ] gputime=[ 2786.464 ] cputime=[ 3127.991 ]
       memtransferhostmemtype=[ 1 ]
method=[ memcpyDtoH ] gputime=[ 2501.312 ] cputime=[ 2555.000 ]
       memtransferhostmemtype=[ 1 ]
```

这里的 memtransferhostmemtype 值为 0 表示可分页内存，值为 1 表示钉固内存。

不应过度使用钉固内存，过量使用会降低整个系统的性能。很难事前说出多少是过量，因此，需要测试应用和它们运行时所用的系统，找出性能最优参数。

3.1.2 批量传输小数据

无论使用可分页内存还是钉固内存，主机与设备间的每一次数据传输都带有一个额外开销。对于小规模数据传输，这个开销对传输速率的影响可能会很大，因此，将批量小传输一起处理能提高效率。

对多种尺寸的数组运行 3.1.1 节中的代码，就能看明白如何一起批量处理多个数据传输。图 3.2 和图 3.3 显示的是在 3.1.1 节中的那两个系统上，可分页数据传输和钉固数据传输的传输速率，传输尺寸从几 KB 变化到接近 1GB. 如果正在实施中的多次传输的尺寸恰好位于这些曲线的陡峭部分，那么一起批量处理这些单个传输可能会显著减少传输的总体时间。

3.1.2.1 使用 cudaMemcpy()显式传输

CUDA Fortran 有可能将赋值语句里的隐式数据传输分裂为若干次传输。近期的编译器版本已经大大降低了这种情况发生的几率，但仍然有可能发生。(可以用命令行测绘器探明来自单个赋值语的传输次数。)为避免这种情况，可以用函数 `cudaMemcpy()` 显式地指定连续数据的单个传输。例如，用下面的语句替换代码中第 59 行上的隐式数据传输：

```
istat = cudaMemcpy(a_d , a_pageable , nElements)
```

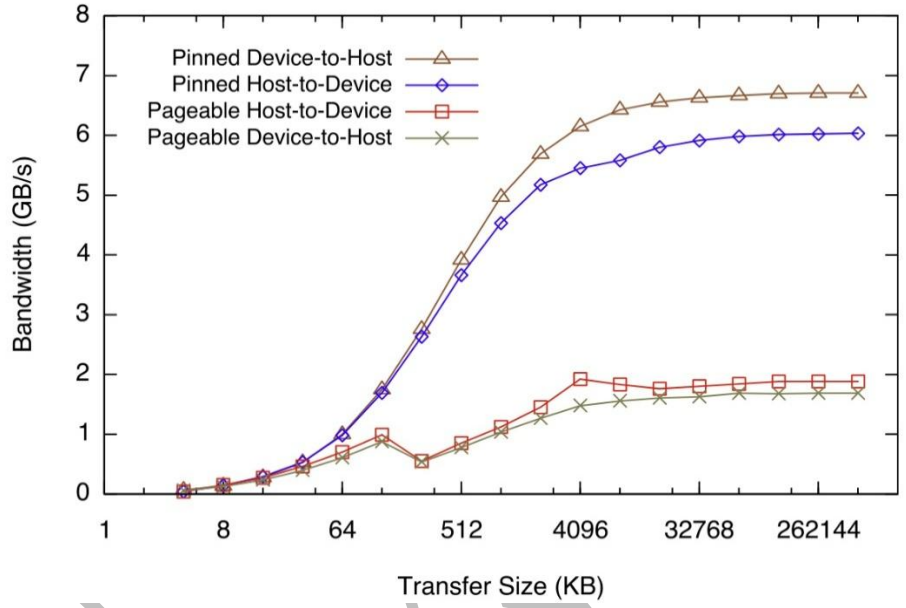


图 3.2

在带有一块 Tesla K20 的 Intel Xeon E5440 系统上，相对于传输尺寸，可分页内存和钉固内存的主机到设备带宽与设备到主机带宽。

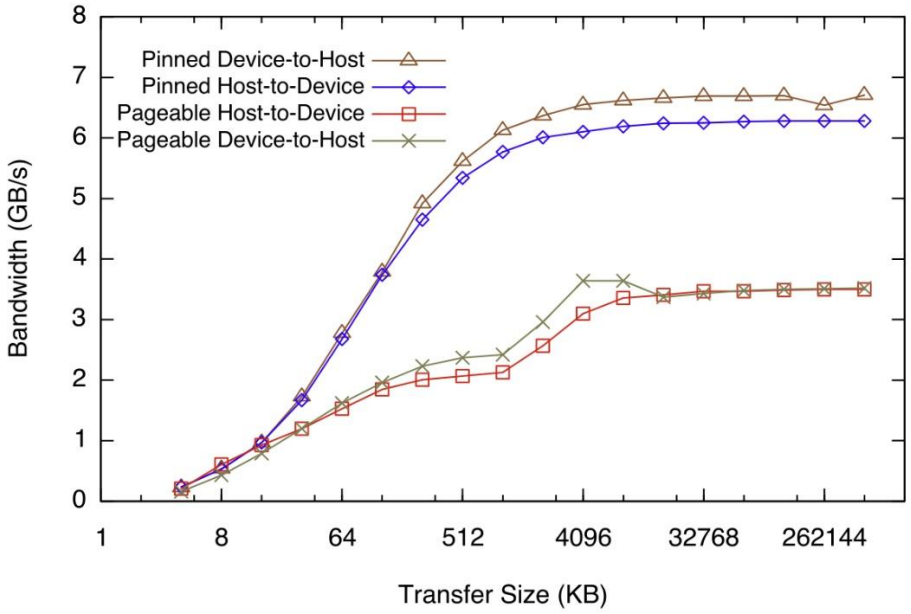


图 3.3

在带有一块 Tesla K20 的 Intel Xeon E5-2667 系统上，相对于传输尺寸，可分页内存和钉固内存的主机到设备带宽与设备到主机带宽。

cudaMemcpy()的参数是终点数组、源头数组和待传输元素[1]的个数。因为 CUDA Fortran 是强类型语言，所以没有必要指定传输方向。依据声明中是否使用修饰符 device，编译器有能力探测前两个参数中的数据驻留在哪里，并执行恰当的数据传输。然而，如果程序员要求，可以用可选的第四个参数指定传输方向，该参数可取的值为 cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost、cudaMemcpyDeviceToDevice. 使用可选的第四个参数时，编译器更加宽容，它会忽略前两个参数的变量类型。这种情形下，元素个数是指源头数组的元素个数。

CUDA Fortran 中，可以用赋值语句在设备与主机间传输数组区块，像这里：

```
a_d(n1_l:n1_u , n2_l:n2_u) = a(n1_l:n1_u , n2_l:n2_u)
```

这样的操作一般会分裂为多个单独的传输。实施此类传输的一个更有效方法是使用例程 cudaMemcpy2D(). 下面的代码段演示如何使用 cudaMemcpy2D()来实施前面赋值语句里的那个数组区块传输：

```
istat = cudaMemcpy2D(a_d(n1_l , n2_l), n, &
                     a(n1_l , n2_l), n, &
                     n1_u -n1_l+1, n2_u -n2_l+1)
```

第一个和第三个参数分别是终点数组和源头数组的第一个元素。第二个和第四个参数是这些参数的领头维度，这里假定为 n；最后两个参数是子数组的尺寸，即每个维度里的元素个数。还有一个类似的例程 cudaMemcpy3D()用来传输三维数组区块。

3.1.3 异步数据传输（高级话题）

使用赋值语句或函数 cudaMemcpy()实施的主机与设备间数据传输，无论是哪个方向，都是阻塞传输；即，只有在传输完成之后控制权才返回主机线程。函数 cudaMemcpyAsync()是一个非阻塞变体，它立即将控制权返回主机线程。与赋值语句及 cudaMemcpy()不同，异步传输版本要求钉固主机内存，它还包含一个附加参数：一个流标识。一个流就是一个简单的操作序列，这些操作在设备上顺序执行。不同流中的操作可能会相互交错，某此情况下也可以重叠 – 这个特性可以用来隐藏主机与设备间的数据传输。

异步数据传输使得能够用计算来重叠数据传输，有两种不同的方法。在所有的 CUDA 设备上都可以使用异步数据传输和设备计算来重叠主机计算。例如，下面的代码段展示了在向设备传输数据和内核执行期间，例程 cpuRoutine()里的主机计算是如何实施的。

```
istat = cudaMemcpyAsync(a_d , a_h , nElements , 0)
call kernel <<<gridSize ,blockSize >>>(a_d)
call cpuRoutine(b)
```

cudaMemcpyAsync 的前三个参数与 cudaMemcpy 的前三个参数相同。最后一个参数是流标识，这里用的是默认流，流 0. 这个内核也使用默认流。因为内核与异步数据传输都在同一个流中，所以内核不会开始执行，直至内存复制完毕；因此不需要显式同步。因为内存复制与内核都立即将控制权返回主机，所以主机例程 cpuRoutine()可以重叠它们的执行。

在前面的例子中，内存复制和内核执行顺序发生。在有能力“并发复制和执行”的设备上，有可能重叠设备上的内核执行和主机与设备间的数据传输。一个设备是否具备这种能力，可以从一个 cudaDeviceProp

[1]与 CUDA C 调用 cudaMemcpy()的第三个参数指定需要传输的字节数量，而这里指定元素数量的方法与之不同。

变量的 deviceOverlap 域查看，也可以从 pgaccelinfo 的输出中查看。在一个具备此能力的设备上，重叠再一次要求钉固内存，还要求数据传输和内核必须使用不同的、非默认的流(具有非 0 标识的流). 这种重叠要求非默认流是因为，只有设备上的所有前序调用都完成以后，内存复制、内存设置函数和使用默认流的内核调用才能开始，并且在他们完成之前，设备上的操作(任何流中)不会开始。在下面的代码中：

```
istat = cudaStreamCreate(stream1)
istat = cudaStreamCreate(stream2)
istat = cudaMemcpyAsync(a_d , a, n, stream1)
call kernel <<<gridSize ,blockSize ,0,stream2 >>>(b_d)
```

创建两个流，并用在数据传输和内核执行中，两个流被指定在 cudaMemcpyAsync()的最后一个参数和内核的执行配置[2]中。

如果内核中数据上的操作是逐点进行的，即操作独立于其它数据，那么这样的情形适合以流水线的形式实施数据传输和内核执行：数据可以分割成多块并分多阶段传输，启动多个内核来操作每一块到达的数据，当相关内核完成时再将每块的结果传回主机。下面列出的完整代码展示了这种技术，分裂数据传输和内核以便隐藏传输时间：

```
  1  ! This code demonstrates strategies hiding data transfers via
  2  ! asynchronous data copies in multiple streams
  3
  4  module kernels_m
  5  contains
  6    attributes(global) subroutine kernel(a, offset)
  7      implicit none
  8      real :: a(*)
  9      integer , value :: offset
 10      integer :: i
 11      real :: c, s, x
 12
 13      i = offset + threadIdx%x + (blockIdx%x-1)* blockDim%x
 14      x = i; s = sin(x); c = cos(x)
 15      a(i) = a(i) + sqrt(s**2+c**2)
 16    end subroutine kernel
 17  end module kernels_m
 18
 19  program testAsync
 20    use cudafor
 21    use kernels_m
 22    implicit none
 23    integer , parameter :: blockSize = 256, nStreams = 4
 24    integer , parameter :: n = 4*1024* blockSize*nStreams
 25    real , pinned , allocatable :: a(:)
 26    real , device :: a_d(n)
 27    integer(kind=cuda_stream_kind) :: stream(nStreams)
```

[2] 执行配置中的的最后两个参数是可选项。执行配置中的第三个参数与内核中使用的共享内存相关，本章后面会讲述。

```
28    type (cudaEvent) :: startEvent , stopEvent , dummyEvent
29    real :: time
30    integer :: i, istat , offset , streamSize = n/nStreams
31    logical :: pinnedFlag
32    type (cudaDeviceProp) :: prop
33
34    istat = cudaGetDeviceProperties(prop , 0)
35    write(*,"(' Device: ', a,/)") trim(prop%name)
36
37    ! allocate pinned host memory
38    allocate(a(n), STAT=istat , PINNED=pinnedFlag)
39    if (istat /= 0) then
40       write(*,*) 'Allocation of a failed'
41       stop
42    else
43       if (.not. pinnedFlag) &
44            write(*,*) 'Pinned allocation failed'
45    end if
46
47    ! create events and streams
48    istat = cudaEventCreate(startEvent)
49    istat = cudaEventCreate(stopEvent)
50    istat = cudaEventCreate(dummyEvent)
51    do i = 1, nStreams
52       istat = cudaStreamCreate(stream(i))
53    enddo
54
55    ! baseline case - sequential transfer and execute
56    a = 0
57    istat = cudaEventRecord(startEvent ,0)
58    a_d = a
59    call kernel <<<n/blockSize , blockSize >>>(a_d , 0)
60    a = a_d
61    istat = cudaEventRecord(stopEvent , 0)
62    istat = cudaEventSynchronize(stopEvent)
63    istat = cudaEventElapsedTime(time , startEvent , stopEvent)
64    write(*,*) 'Time for sequential ', &
65         'transfer and execute (ms): ', time
66    write(*,*) ' max error: ', maxval(abs(a-1.0))
67
68    ! asynchronous version 1: loop over {copy , kernel , copy}
69    a = 0
70    istat = cudaEventRecord(startEvent ,0)
71    do i = 1, nStreams
```

```
72        offset = (i-1)* streamSize
73        istat = cudaMemcpyAsync( &
74             a_d(offset+1),a(offset+1),streamSize ,stream(i))
75        call kernel <<<streamSize/blockSize , blockSize , &
76                       0, stream(i)>>>(a_d ,offset)
77        istat = cudaMemcpyAsync( &
78             a(offset+1),a_d(offset+1),streamSize ,stream(i))
79     enddo
80     istat = cudaEventRecord(stopEvent , 0)
81     istat = cudaEventSynchronize(stopEvent)
82     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
83     write(*,*) 'Time for asynchronous V1 ', &
84          'transfer and execute (ms): ', time
85     write(*,*) ' max error: ', maxval(abs(a-1.0))
86
87     ! asynchronous version 2:
88     ! loop over copy , loop over kernel , loop over copy
89     a = 0
90     istat = cudaEventRecord(startEvent ,0)
91     do i = 1, nStreams
92        offset = (i-1)* streamSize
93        istat = cudaMemcpyAsync( &
94             a_d(offset+1),a(offset+1),streamSize ,stream(i))
95     enddo
96     do i = 1, nStreams
97        offset = (i-1)* streamSize
98        call kernel <<<streamSize/blockSize , blockSize , &
99                       0, stream(i)>>>(a_d ,offset)
100    enddo
101    do i = 1, nStreams
102       offset = (i-1)* streamSize
103       istat = cudaMemcpyAsync(&
104            a(offset+1),a_d(offset+1),streamSize ,stream(i))
105    enddo
106    istat = cudaEventRecord(stopEvent , 0)
107    istat = cudaEventSynchronize(stopEvent)
108    istat = cudaEventElapsedTime(time , startEvent , stopEvent)
109    write(*,*) 'Time for asynchronous V2 ', &
110         'transfer and execute (ms): ', time
111    write(*,*) ' max error: ', maxval(abs(a-1.0))
112
113    ! asynchronous version 3:
114    ! loop over copy , loop over {kernel , event},
115    ! loop over copy
```

```
116     a = 0
117     istat = cudaEventRecord(startEvent ,0)
118     do i = 1, nStreams
119        offset = (i-1)* streamSize
120        istat = cudaMemcpyAsync( &
121        a_d(offset+1),a(offset+1),streamSize ,stream(i))
122     enddo
123     do i = 1, nStreams
124        offset = (i-1)* streamSize
125        call kernel <<<streamSize/blockSize , blockSize , &
126                       0, stream(i)>>>(a_d ,offset)
127        istat = cudaEventRecord(dummyEvent , stream(i))
128     enddo
129     do i = 1, nStreams
130        offset = (i-1)* streamSize
131        istat = cudaMemcpyAsync( &
132             a(offset+1),a_d(offset+1),streamSize ,stream(i))
133     enddo
134     istat = cudaEventRecord(stopEvent , 0)
135     istat = cudaEventSynchronize(stopEvent)
136     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
137     write(*,*) 'Time for asynchronous V3 ', &
138          'transfer and execute (ms): ', time
139     write(*,*) ' max error: ', maxval(abs(a-1.0))
140
141     ! cleanup
142     istat = cudaEventDestroy(startEvent)
143     istat = cudaEventDestroy(stopEvent)
144     istat = cudaEventDestroy(dummyEvent)
145     do i = 1, nStreams
146        istat = cudaStreamDestroy(stream(i))
147     enddo
148     deallocate(a)
149
150  end program testAsync
```

这份代码用四种方法处理数组数据。第一种方法是串行处理，将所有数据传输到设备(第 58 行)，然启动一个内核，内核带有足够多的线程来处理数组中的每个元素(第 59 行)，接着是一个设备到主机的数据传输(第 60 行)。别外三种方法牵扯到重叠异步内存复制与内核执行的不同策略。

异步情形与串行情形相似，只是有多个数据传输和内核启动，这些传输、内核由不同的流和数组偏移量来区分。尽管没理由不让大型数组使用更多的流，但为叙述方便，这里将流的数量限制为 4。注意，代码里的串行情形和异步情形使用同一个内核，因为传送给内核的偏移量可以使之适应不同流中的数据。前两个异步版本间的区别在复制和内核执行的顺序上。第一个版本(从第 68 行开始)对每个流循环，每个流递交一个主机至设备复制、一个内核、一个设备至主机复制。第二个版本(从第 87 行开始)先递交所有的主机至

设备复制，然后递交所有的内核启动，接后递交所有的设备至主机复制。第三个异步版本(从 113 行开始)在递交每个内核之后记录一个虚假事件，该事件与紧临上方的内核处于同一个流，除此之外第三个版本与第二个版本相同。

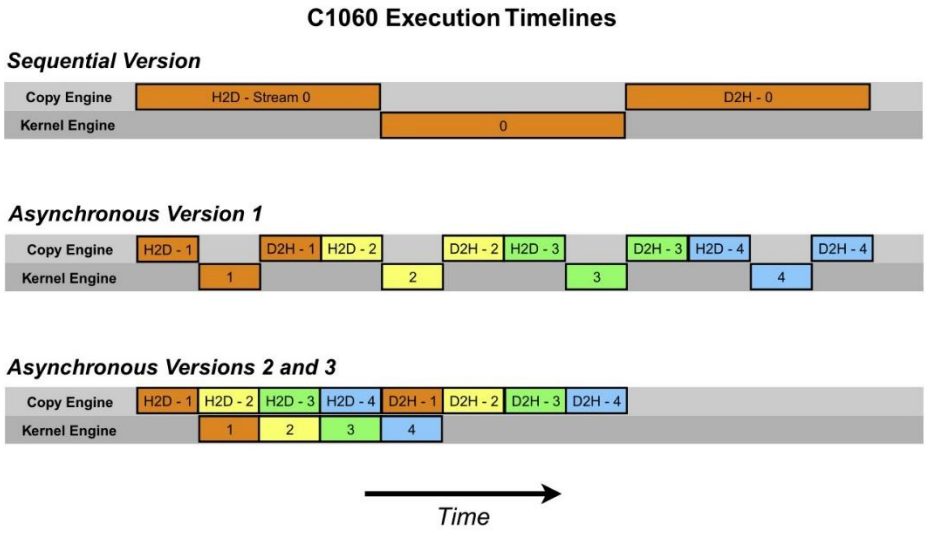
此处你可能会问异步情形为什么有三个版本。原因就是这些变体在不同代次的硬件上有会有不同的表现。在 NVIDIA Tesla C1060 上运行这个代码产生：

```
Device: Tesla C1060

Time for sequential transfer and execute (ms):    12.92381
  max error:    2.3841858E-07
Time for asynchronous V1 transfer and execute (ms):    13.63690
  max error:    2.3841858E-07
Time for asynchronous V2 transfer and execute (ms):    8.845888
  max error:    2.3841858E-07
Time for asynchronous V3 transfer and execute (ms):    8.998560
  max error:    2.3841858E-07
```

而在 NVIDIA Tesla C2050 得到：

```
Device: Tesla C2050

Time for sequential transfer and execute (ms):    9.984512
  max error:    1.1920929E-07
Time for asynchronous V1 transfer and execute (ms):    5.735584
  max error:    1.1920929E-07
Time for asynchronous V2 transfer and execute (ms):    7.597984
  max error:    1.1920929E-07
Time for asynchronous V3 transfer and execute (ms):    5.735424
  max error:    1.1920929E-07
```

为解读这些结果，需要对设备如何调度和执行各样任务多一点理解。CUDA 设备包含多样任务的对应引擎，操作被递交以后就在这些引擎里排队。系统维护着不同引擎中任务的依赖关系，但是在任何一个引擎的内部，所有的依赖关系都丢弃了，因为一个引擎队列里的任务将按照主机线程递交它们的顺序执行。例如 ,C1060 只有一个复制引擎和一个内核引擎。对前面的代码 ,设备上的执行时间线显示在图 3.4 的顶部。这个示意图中假定主机至设备传输、内核执行和设备至主机传输的时间近似相等(特意选择内核代码以使在 Tesla C1060 和 C2050



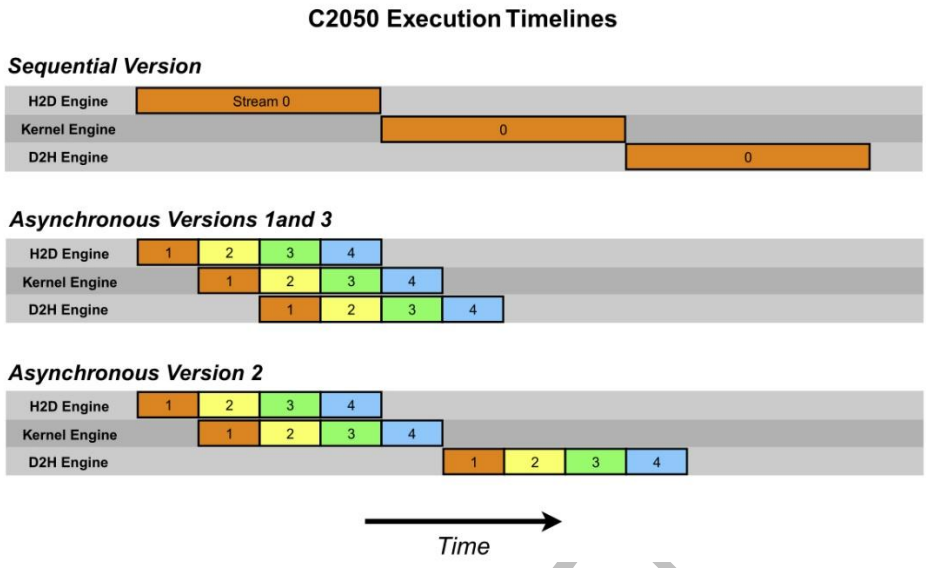


图 3.4

在 Tesla C1060 和 C2050 上，串行和三种异步策略下数据传输和内核执行的时间线。C1060 仅有一个复制引擎，然后 C2050 有两个单独的设备至主机和主机至设备复制引擎。在每个引擎内，数据传输按照它们从主机递交过来的顺序执行。从而，不同的策略会在这些不同的架构上实现重叠。

上时这些时间相当)。对串行内核，正如期待的那样，任何操作之间都没有重叠。对代码里的第一个异步版本，复制引擎的执行顺序是 H2D stream(1), D2H stream(1), H2D stream (2), D2H stream(2), 如此等等。 这就是为什么在 C1060 上运行第一个异步版本时没有看到任何加速：任务按照一定顺序递交给复制引擎，这个顺序会阻止内核与数据传输的任何重叠。然而对于版本二和版本三，所有主机至设备传输的顺序递交排在任何设备至主机传输的前面，因此重叠是有可能的，执行时间减少说明发生了重叠。从示意图来看，可以期待版本二和版本三的执行时间是串行版本的 8/12，或者是 8.7 ms，这正是在前面的计时结果里观察的现象。

在 C2050 上，两个特性相互影响带来与 C1060 上不一样的行为。除了拥有一个内核引擎，C2050 还拥有两个复制引擎，一个负责主机至设备传输，另一个负责设备至主机传输。拥有两个复制引擎解释了为什么第一个异步版本能在 C2050 上获得良好加速：stream(i)中数据的设备至主机传输不会阻塞 stream(i+1)中数据的主机至设备传输，就像 C1060 上做的那样，这是因为这两个操作在 C2050 上处于不同的流中，图 3.4 底部的示意图显示了这个原理。从示意图中看，可以期待将执行时间砍为串行版本的一半，从前面列出来的计时数据中观察到大致如此。但这无法解释在第二个异步方式中观察到的性能下降，它与 C2050 支持并发运行多个内核有关。当紧挨着递交多个内核时，调度器尝试开启这些内核的并发执行，从而延迟了一个信号，这个信号正常情况下发生在每个内核完成之后(该信号负责剔除设备至主机传输)直到所有内核都完成。所以，尽管异步代码的第二个版本中有主机至设备传与内核执行的重叠，却没有内核执行与设备至主机传输之间的重叠。从图 3.4 知，可以期待第二个异步版本的时间为串行版本时间的 9/12，或者 7.5 ms，从上面计时数据中确实观察到的了这个现象。这种情况可以通过在每个内核之间记录一个虚假 CUDA 事件来改正，这样将禁止并发执行内核，但开启了数据传输和内核执行的重叠，第三处异步版本就是这么做的。

3.1.3.1 Hyper-Q

计算能力 3.5 的设备(本书撰写时的最高计算能力)，例如 Tesla K20, 包含一个称为 Hyper-Q 的特性。先前的 CUDA 架构拥有单个的工作队列，上面观察到的复制引擎和内核执行串行化现象中其实已经介绍过工作队列。Hyper-Q 引入 32 个独立的工作队列。在异步代码例子中，有 Hyper-Q 时，每个流都由它自己的

硬件工作队列管理。从而，某个流里的操作将不会阻塞其它流里的操作。在 NVIDIA Tesla K20 上运行这个代码，得到：

```
Device: Tesla K20

Time for sequential transfer and execute (ms):    7.963808
  max error:    1.1920929E-07
Time for asynchronous V1 transfer and execute (ms):    5.608096
  max error:    1.1920929E-07
Time for asynchronous V2 transfer and execute (ms):    5.646880
  max error:    1.1920929E-07
Time for asynchronous V3 transfer and execute (ms):    5.506816
  max error:    1.1920929E-07
```

这里观察到每个异步方法都获得了大致相同的性能。你可能已经注意到，K20 上同步和异步版本间的相对加速不像 C2050 上的最优异步版本的相对加速那么大。这归根于在 K20 上内核执行时间明显少于数据传输时间，如同测绘器输出所示：

```
method=[ memcpyHtoDasync ] gputime=[ 712.608 ] cputime=[ 19.000 ]
method=[ kernel ] gputime=[ 442.816 ] cputime=[ 29.000 ]
       occupancy=[ 1.000 ]
method=[ memcpyDtoHasync ] gputime=[ 1295.520 ] cputime=[ 9.000 ]
```

在 C050 上，数据传输和内核执行时间大致相同，从而产生一个相对较大的加速。可以调整内核使它在 K20 上达到跟 C2050 上一样的加速，但这里的重点是获得最好加速的努力。Hyper-Q 消除了程序员在 K20 上对多个流实施最优调度工作的必要性，然而调整异步复制和内核的提交顺序仍然是在 C1060 和 C2050 上获得最好结果的必要手段。

3.1.3.2 测绘异步事件

监测异步性能的一个好方法是使用测绘器，使用包含下列内容的配置文件：

```
conckerneltrace
timestamp
gpustarttimestamp
gpuendtimestamp
streamid
```

不像硬件计数器，这些项目不会将设备上的执行串行化，不会阻碍测量行为。应该注意，在前面的代码里开启测绘将高效地实现内核调用之间插入一个 `cudaEventRecord()` 所实现的功能，因此这种情况下，测量行为不会改变测量对象。

在用异步数据传输重叠内核计算这个话题结束之前，应该注意，这个例子中内核选用的计算数值 1.0 的方法让人非常迷惑。这样选取就是为让主机与设备间传输时间能够和内核执行时间相当，至少对 C1060 和 C2050 来说时间相当。如果使用更简单的内核，诸如此前讲述过的一些内核，那么就会因为内核执行时间比数据传输时间小太多而导致很难检测到重叠。

3.2 设备内存

本章至此都聚焦在将数据放入与取出设备 DRAM 的有效手段。更准确地说，这些数据都存储在全局内存(global memory)中，而全局内存驻留在 DRAM 中。全局内存可以被设备和主机访问，可以在应用的生存期内一直存在。除了全局内存，还有其它类型的数据存储在 DRAM 中，它们具有不同的作用域、生存期和缓冲行为。还有几种位于芯片上面的内存类型。本节讲述这些不同类型的内存，以及如何最好地使用它们。

图 3.5 给出了 CUDA 中的多种内存类型。在设备 DRAM 中有全局内存、本地内存、常量内存和纹理内存。芯片上有寄存器、共享内存、和多种缓存(L1 缓存、常量缓存、纹理缓存)。稍后本章将逐一深入讲解这些内存并提供示例，但此刻先给出他们的简短概括。

全局内存(global memory)是设备内存，在主机代码里用 `device` 属性声明。它可以被从主机端和设备端读写。它对设备上启动的所有线程可用，并持续到应用的整个生存期(或者直至被撤销，如果声明为 allocatable)。

假如有足够的寄存器可用，设备代码里定义的本地变量将存储在片上寄存器中。如果没有足够的寄存器，数据将被存在芯片之外的本地内存(local memory)中。(本地内存中的形容词本地是指作用域，而不是物理位置。) 每个线程都可以访问寄存器内存和本地内存。

共享内存(shared memory)是一个线程块内所有线程都能访问的一种内存。在设备代码里用变量修饰符 `shared` 来声明它。它能用来共享数据加载和数据存储，还能用来避免全局内存的低效访问模式。

常量内存(constant memory)可以从主机代码读写，但对设备里的线程是只读的。它在一个 Fortran 模块里用修饰符 `constant` 声明，并可以在包含该模块的任何代码中使用，也可以在使用该模块的任何代码里使用。常量数据在芯片上缓冲，当同一时刻执行的线程都访问同一数值时，它最高效。

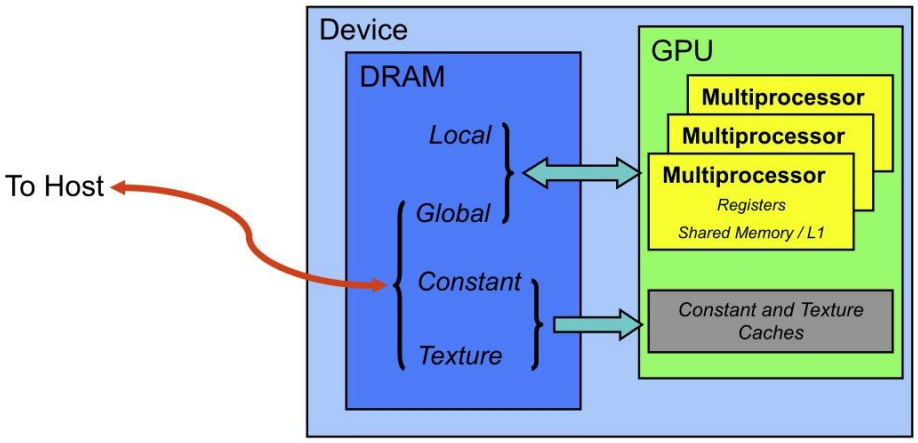


图 3.5

DRAM 中和芯片上设备内存类型示意图。

表 3.1 设备内存特征

内存	位置	被缓存	设备访问	作用域	生存期
寄存器	芯片上	N/A	读/写	一个线程	线程
本地	DRAM	Fermi, Kepler	读/写	一个线程	线程
共享	芯片上	N/A	读/写	块中所有线程	线程块
全局	DRAM	Fermi, Kepler*	读/写	所有线程和主机	应用
常量	DRAM	是	读	所有线程和主机	应用
纹理	DRAM	是	读	所有线程和主机	应用
*Tesla K10, K20 和 K20X 仅在 L2 缓存里缓存全局内存					

纹理内存(texture memory)与常量内存相似，对设备代码也是只读的，且在 GPU 上缓冲。它仅仅是访问全局内存的另一条通路，有时对避免设备代码的糟糕内存访问模式有帮助。

表 13.1 总结了所有设备内存类型的特征。

3.2.1 在设备代码中声明数据

在讲述如何高效利用不同类型的内存之前，应该说一下关于设备代码中如何声明数据的几个要点。绝大多数情况下，设备代码里的数据声明与主机代码里的声明或常规 Fortran 90 的声明一样。然而，有几个例外值得注意。

在设备代码里声明数据时必须明白，因为主机和设备都具有单独的内存空间，而且 Fortran 默认按引用传递参数，所以内核参数要么必须位于设备上，要么参数是主机标量，此时参数必须具有 `value` 属性[3]。

第二个例外就是 CUDA Fortran 在设备代码里不支持 `save` 属性，无论是显式地还是隐式地。因为在声明时初始化的变量会隐式地获得 `save` 属性，所以 CUDA Fortran 设备代码里不允许变量在声明处初始化。当然了，带有 `parameter` 属性的变量是允许的，而且必须在声明时赋值，这是因为编译器要把这些值转换为代码里的字面常量。下面的代码片断展示了这些要点：

```
attributes(global) subroutine increment(array , incVal)
  integer :: array(*)
  integer , value :: incVal
  integer :: otherVal=1 ! illegal
  integer , parameter :: anotherVal = 2 ! OK
```

3.2.2 合并访问全局内存

在对 CUDA 架构的编程中，最重要一个性能要素也许就是合并全局内存访问。介绍如何访问全局内存之前，需要完善一下编程模型。已经讲过如何将线程分组为线程块，线程块被指派给设备上的多处理器。线程被近一步分组为幅(warp)，或包含 32 个线程的组，幅是以单指令多线程方式进行计算的线程实际分组。设备上的每一条指令都被提交给一幅线程，幅内的每一个线程步调一致地执行该指令。一个线程块内的不同的幅可能正在执行着设备代码的不同指令，所有这些活动皆由每个多处理器上的调度器在幕后协调。为了当前的目标，程序员仅需知道指令会被一个叫做幅的线程组同时执行。将进程分组为幅不仅与计算有关，还与全局内存访问有关。当满足特定访问要求时，半幅线程(对计算能力 1.x 的设备)或一幅线程(对计算能力 2.0 及更高设备)对全局内存的加载与存储会被设备尽可能少地合并为一次作业。为理解这些访问要求，以及它们在不同的 Tesla 架构上是如何演变的，这里在代表不同计算能力的 Tesla 卡上做几个简单实验。实验采用单精度和双精度(可能的时候)。

这里运行两个实验代码，它们是引述里用到的自增内核的变体 – 一个带有数据偏移或称为数组的未对齐访问，另一个以相似的方式实施跨跃访问。执行这些访问的代码是：

```
module kernels_m
  use precision_m
contains
  attributes(global) subroutine offset(a, s)
    real(fp_kind) :: a(*)
```

[3] value 属性要求按值传递参数—译者注。

```
    integer , value :: s
    integer :: i
    i = blockDim%x*(blockIdx%x-1)+ threadIdx%x + s
    a(i) = a(i)+1
  end subroutine offset

  attributes(global) subroutine stride(a, s)
    real(fp_kind) :: a(*)
    integer , value :: s
    integer :: i
    i = (blockDim%x*(blockIdx%x-1)+ threadIdx%x) * s
    a(i) = a(i)+1
  end subroutine stride
end module kernels_m

program offsetNStride
  use cudafor
  use kernels_m

  implicit none

  integer , parameter :: nMB = 4 ! transfer size in MB
  integer , parameter :: n = nMB*1024*1024/ fp_kind
  integer , parameter :: blockSize = 256
  ! array dimensions are 33*n for stride cases
  real(fp_kind), device :: a_d(33*n), b_d(33*n)
  type(cudaEvent) :: startEvent , stopEvent
  type(cudaDeviceProp) :: prop
  integer :: i, istat
  real(4) :: time


  istat = cudaGetDeviceProperties(prop , 0)
  write(*,'(/,"Device: ",a)') trim(prop%name)
  write(*,'("Transfer size (MB): ",i0)') nMB

  if (kind(a_d) == singlePrecision) then
     write(*,'(a,/)') 'Single Precision'
  else
     write(*,'(a,/)') 'Double Precision'
  endif

  istat = cudaEventCreate(startEvent)
  istat = cudaEventCreate(stopEvent)
```

```
  write(*,*) 'Offset , Bandwidth (GB/s):'
  call offset <<<n/blockSize ,blockSize >>>(b_d , 0)
  do i = 0, 32
     a_d = 0.0
     istat = cudaEventRecord(startEvent ,0)
     call offset <<<n/blockSize ,blockSize >>>(a_d , i)
     istat = cudaEventRecord(stopEvent ,0)
     istat = cudaEventSynchronize(stopEvent)

     istat = cudaEventElapsedTime(time , startEvent , &
          stopEvent)
     write(*,*) i, 2*n*fp_kind/time*1.e-6
  enddo

  write(*,*)
  write(*,*) 'Stride , Bandwidth (GB/s):'
  call stride <<<n/blockSize ,blockSize >>>(b_d , 1)
  do i = 1, 32
     a_d = 0.0
     istat = cudaEventRecord(startEvent ,0)
     call stride <<<n/blockSize ,blockSize >>>(a_d , i)
     istat = cudaEventRecord(stopEvent ,0)
     istat = cudaEventSynchronize(stopEvent)
     istat = cudaEventElapsedTime(time , startEvent , &
          stopEvent)
     write(*,*) i, 2*n*fp_kind/time*1.e-6
  enddo

  istat = cudaEventDestroy(startEvent)
  istat = cudaEventDestroy(stopEvent)

end program offsetNStride
```

3.2.2.1 未对齐访问

现在开始查看单精度数据的未对齐访问结果，如图 3.6 所示。当一个数组在设备内存中分配后，不管是显式分配还是隐式分配，这个数组都被对齐到 256 字节的内存段。全局内存可以通过 32、64、或 128 字节的作业来访问，这些作业都对齐到它们的尺寸。当一幅(或半幅)内的线程在尽可能少的内存作业里访问数据时能获得最佳性能，正如图 3.6 中偏移为零的情形。这个情形下，一幅(或半幅)线程请求的数据被合并成一个 128 字节(或 64 字节)的作业，该笔作业中所有的字都是被请求的数据。对计算能力 1.0 的 C870 和此计算能力的其它卡，这个性能还要求半幅内的连续线程必须访问 64 字节内存段的连续字。

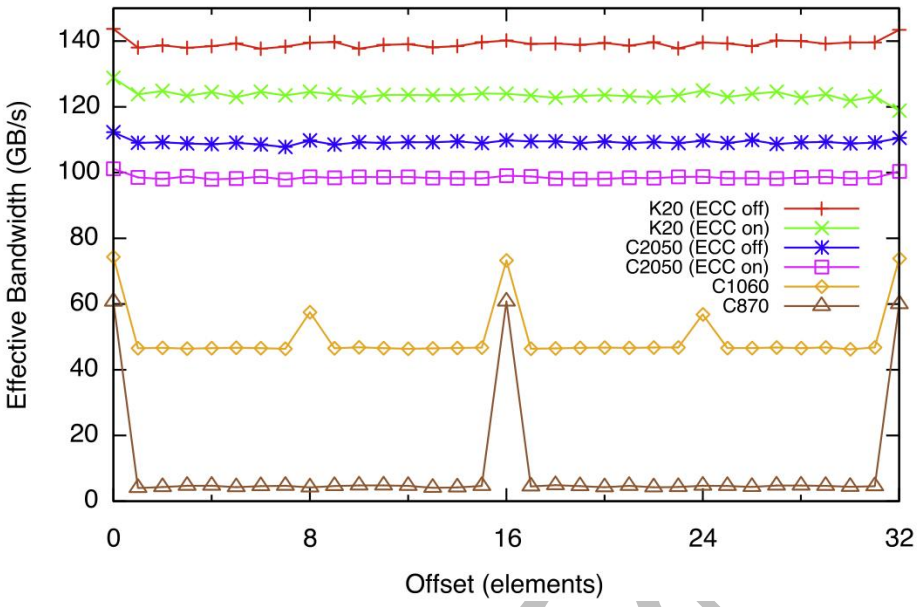


图 3.6

对单精度数据，数组自增内核在各种偏移量下的实际带宽。

对不同的计算能力，非对齐访问的性能有很大变化。对计算能力 1.0 的 C870，半幅线程的未对齐访问(或者半幅线程不按顺序访问对齐的内存)导致 16 笔单独的 32 字节作业。因为每笔 32 字节作业仅有 4 个字节是被请求数据，所以可以预测有效带宽会减小至八分之一。这与图 3.6 中偏移量不是 16 个元素倍数时的性能大致吻合，16 个元素对应半幅线程。

对计算能力为 1.3 的 C1060，未对齐访问的问题少点。基本上，对连续数据的未对齐访问，用几笔作业就能覆盖被请求数据。相对于对齐的情形，仍然有性能损失，这归咎于传输了未被请求的数据以及不同的半幅所请求的数据相互重叠。稍后会分析 C1060 性能的三个级别，但这里先给出决定将要发生的传输类型的算法。在 C1060 上，用来决半幅线程带来的作业类型和作业数量所用的准确算法是：

- 找出编号最低的活动线程所请求数据所在的内存片段。片段尺寸对 8 位数据来说是 32 字节，对 16 位数据来说是 64 字节，对 32 位、64 位和 128 数据来说是 128 字节。
- 找出所请求的地址都落在同一个片段内的所有其它活动线程，并尽量减少作业尺寸：
  - ✧ 如果作业尺寸是 128 字节且只有上半段或下半段被用到，那么作业尺寸被缩减为 64 字节。
  - ✧ 如果作业尺寸是 64 字节且只有上半段都下半段有用，那么作业尺寸缩小为 32 字节。
- 完成这笔作业并将服务过的线程示记为不活动。
- 反复上述步骤，直到半幅内的所有线程都被服务过。

现在将这个算法应用到偏移例子中，瞧瞧偏移为 0、1 和 8 时发生了什么。

从 0 偏移对应的最优情形开始。前两个半幅的数据访问模式如图 3.7 所示。图中，两行方块代表同一个 256 字节的内存片段，顶部显示的是各种尺寸作业的对齐位置。对每半幅线程，请求的数据带来一笔 64 字节的作业。尽管仅显示了两个半幅，但是所有半幅上发生的事情都是一样的。没有传输未被请求的数据，没有数据被传输两次，因此这是最优的情形，就像图 3.6 中反映的那样。注意，任何为 16 个元素倍数的偏移都将获得同样的性能，因为它恰好将这个示意图平移一个 64 字节的片段。

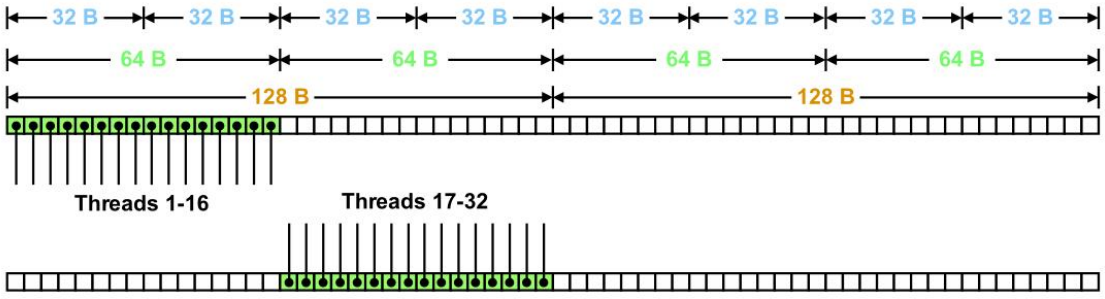


图 3.7

对齐访问或零偏移访问单精度数据时，C1060 上两个半幅的作业示意图。32 字节、64 字节和 128 字节的片段显示在顶部，代表相同内存的两行方块显示在下方。第一行用来刻画第一个半幅线程的访问，第二行用来刻画第二个半幅线程的访问。这是最优的情形，每个半幅的请求都带来单笔 64 字节业务，两个半幅总共传输了 128 字节，没有传输未被请求的数据，也没重复传输数据。

对 C1060 而言，访问模式平移一格就会进入最糟糕的情形。前两个半幅的访问模式及产生的作业如图 3.8 所示。对第一个半幅来说，即使只请求了 64 字节，但是整个 128 字节片段都要传输。这是因为第一个半幅所请求的数据同时落在了 128 字节片段的下半段和上半段；作业不能缩减。第二半幅线程访问的数据跨越两个 128 字节片段，每个片段中的业务都能缩减。注意，对这两个半幅，传输了一些未请求数据，还有一些数据传输了两次。而且，这种模式对后面成对的半幅反复使用，所以第二个半幅的 32 字节作业将会重叠第三个半幅的 128 字节作业。为这两个半幅需要传输 224 字节，与之对比，对齐或零偏移的情形只需传输 128 字节。由基于此，可以期期它的有效带宽会比 0 偏移情形的一半高一些。这能从图 3.6 中看到。对偏移 2–7、9–15、17–23、和 25–31，发生相同数量的作业，也有相同的有效带宽。

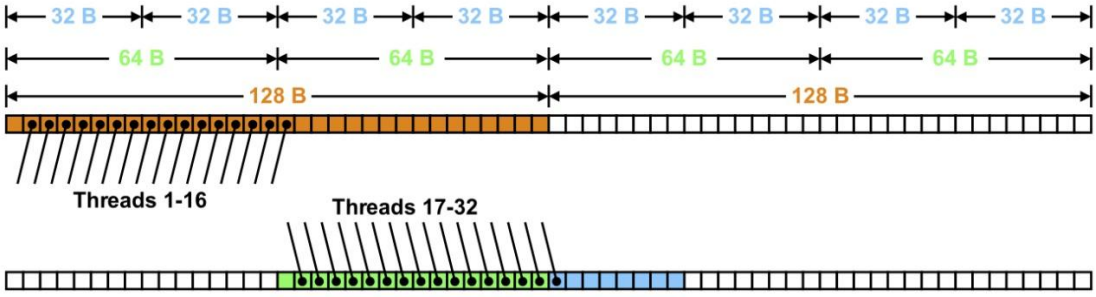


图 3.8

对偏移一个元素的未对齐单精度数据，C1060 上两个半幅的作业示意图。两行方块代表相同的内存。第一行用来描绘前半幅线程的访问，第二行用来描绘后半幅线程的访问。这两个半幅的请求由三次作业来服务，总共 224 字节。

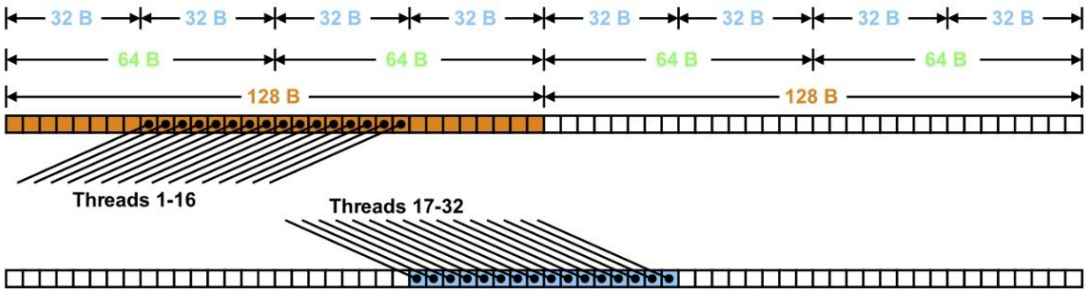


图 3.9

对偏移 8 个元素的未对齐单精度数据，C1060 上两个半幅的作业示意图。两行方块代表相同的内存。第一行用来描绘前半幅线程的访问，第二行用来描给后半幅线程的访问。这两个半幅的请求由三次作业来满足，总共 192 字节。

为 C1060 考虑的最后一种未对齐访问情形是偏移 8 或 24 个元素，如图 3.9 所示。这与偏移一个元素的情况相似，只是来自第二个半幅的请求由两个 32 字节的作业来服务，而不是由一个 64 字节作业和一个 32 字节作业来服务。这样以来两个半幅带来 192 个元素的传输，并且有效带宽约为对齐有效带的 2/3，从图 3.6 可以看出这一点。

C2050 的情况与前面的情形大不相同，这是因为费米(Fermi)架构引入了全局内存缓冲。还有，内存作业是由每幅线程递交而不再是每半幅。在费米架构上，每个多处理器拥有 64KB 内存，这些内存在共享内存和 L1 级缓存间分割，要么 16KB 共享内存和 48KB L1 级缓存，要么反过来。这个 L1 缓存使用 128 字节的缓存线(cache line)。 当一条缓存线被取到多处理器上的 L1 缓存时，驻留在该多处理器上的任意线程幅都可以使用这个缓存线。因此，一些未被请求的数据也可能被取到 L1 缓存，但应该只有很少的重复数据被取到芯片上。从图 3.6 中的结果可以看出，任意偏移都有一小点性能损失 – 未对齐访问导致的性能损失是如此之小，以致于它实际上比 ECC 导致的性能损失还小。

在 K20 上，未对齐访问导致的有效带宽变动小于 C2050 上的变动，当全局内存访问在未对齐时能看到轻微的性能损失。在 K20 上，L1 缓存仅用于本地内存，全局内存在 L2 上缓冲，L2 是一个所有多处理器共享的片上缓存。再一次，未对齐访问损失很小 – 比 ECC 带来的损失要小得多。

前面对单精度数据的讲述也适用于双精度数据，就像图 3.10 中看到的那样，只是有个例外，C870 不支持双精度，因此图中没有显示。在 NVIDIA Tesla C2050 和 K20 上，再一次看到，未对齐访问只有轻微的性能下降。在 NVIDIA Tesla C1060 上，因为半幅线程请求的双精度数据跨越 128 个字节，相对于单精度数据请求，双精度会有更多的片段组合形式。图 3.11 展示了带有 0 至 16 偏移的半幅线程请求对应的作业。

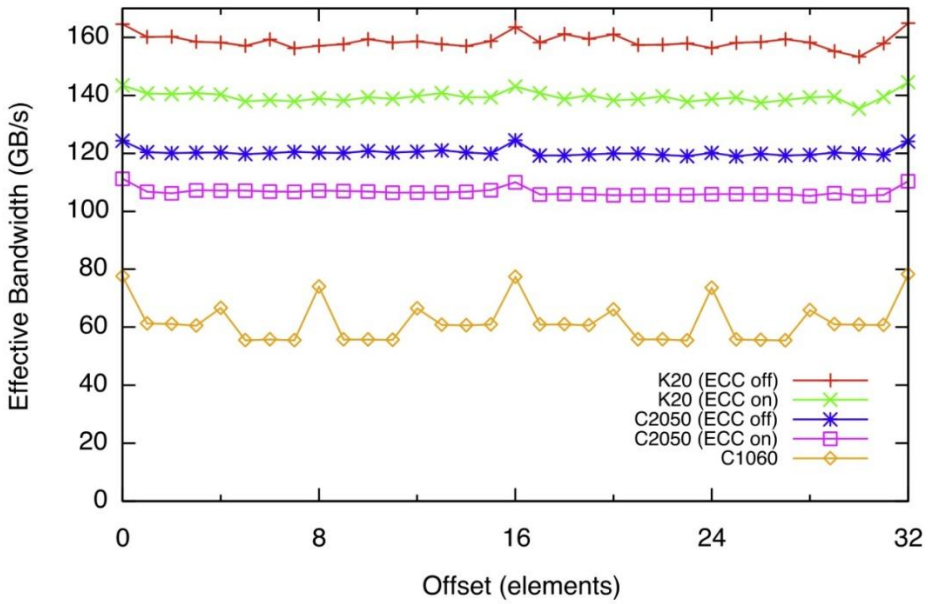


图 3.10

使用数组自增内核时，相对于偏移量的双精度有效带宽。

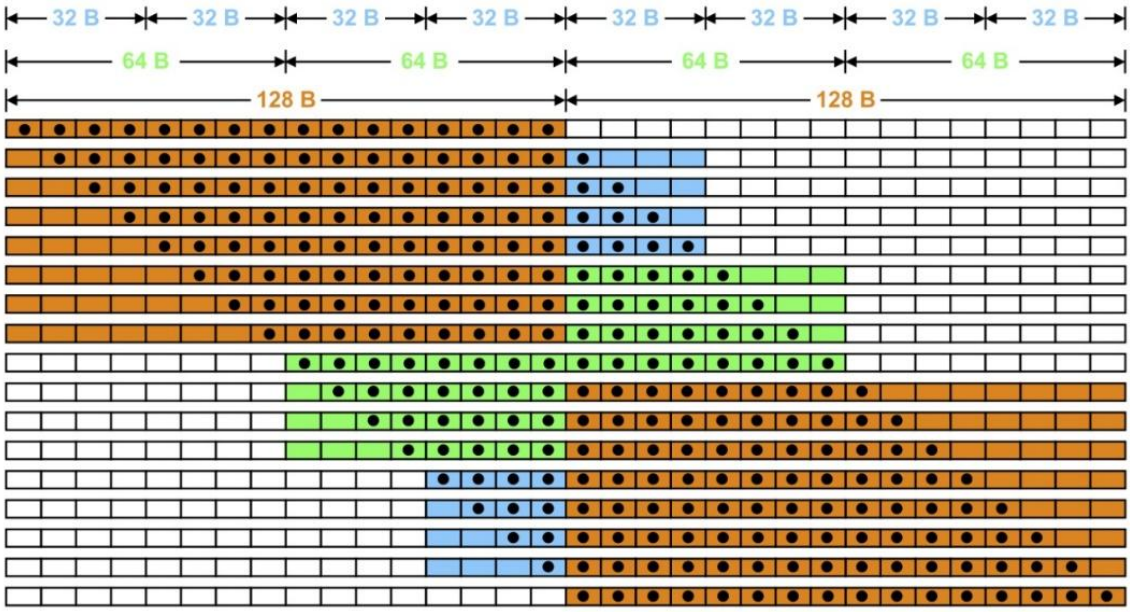


图 3.11

在 C1060 上，半幅线程访问连续双精度数据带来的作业，各行代表的偏移量为从 0 到 16. 对双精度数据，奇数半幅和偶数半幅的访问模式相同，这一点与单精度数据情形不一样。

在转移到讲述全局内存跳跃访问之前，这里应该提一下，计算能力 2.0 的设备上，更复杂内核的访问未对齐时，开启 ECC 将带来更大的损失。例如，在内核中使用异位自增操作 `b(i)=a(i)+1`，而不是原来的 `a(i)=a(i)+1`，那么在开启 ECC 的 C2050 上会观察到大幅的性能下降，正如图 3.12 中单精度情形表现的那样。作为一个通用准则，只要可能就将访问对齐，这对代码总是最好的，特别适用于开启 ECC 的 Tesla C2050. 对天然偏移的访问，例如有限差分操作中发生的访问，片上共享内存可以用来促成对齐访问，本章稍后会有讲述。在 Tesla K20 上，当 ECC 开启时看不到异位内核带来的性能下降，如图 3.13 所示，这是因为开普列 GPU 上的 ECC 实现方法已经改进。

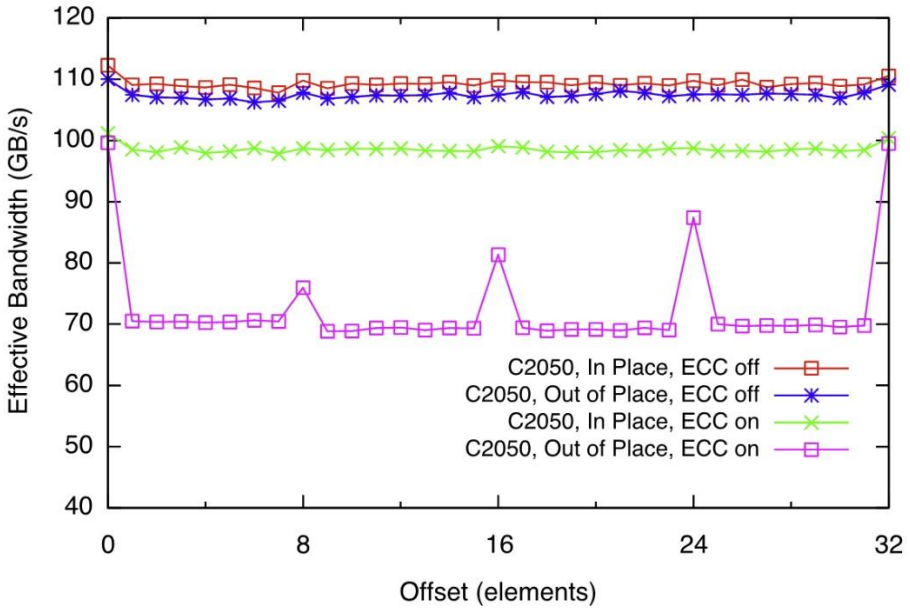


图 3.12

Tesla C2050 上，ECC 开启和关闭时，单精度数据上同位自增操作和异位自增操作的有效带宽。ECC 关闭状

态下，同位和异位具有相似的性能。然而，ECC 开启状态下，偏移访问就会给异位操作带来性能损失。实践经验表明，只要 ECC 开启，在 C2050 上就要尽可能确保访问对齐。

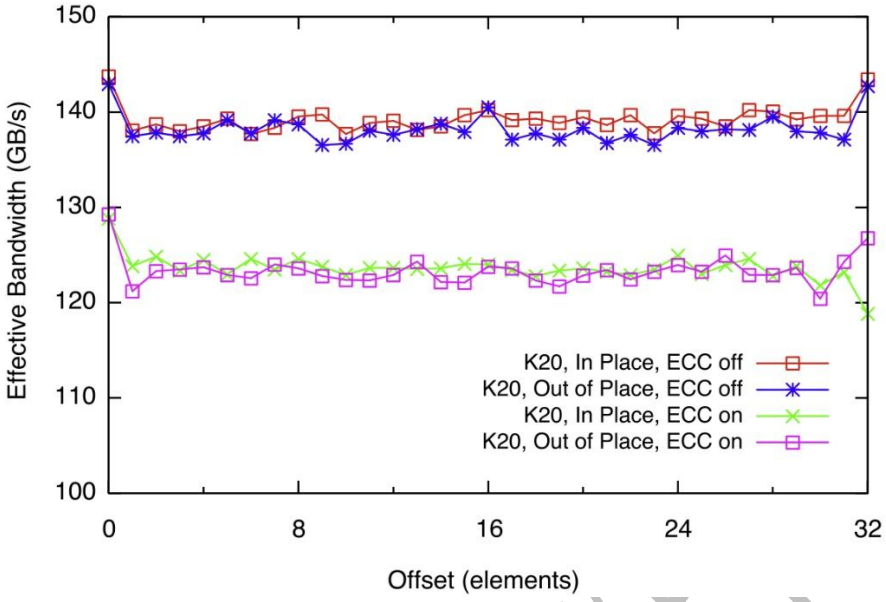


图 3.13

Tesla K20 上，ECC 开启和关闭时，单精度数据上同位自增操作和异位自增操作的有效带宽。与图 3.12 中 C2050 的结果不一样，无论 ECC 是处于开启状态还是关闭状态，同位和异位带宽都大致相等。

3.2.2.2 跳跃访问

未对齐数据访问内核中讲述的合并规则同样适用于跳跃访问内核。区别是半幅或一幅线程请求访问的数据不再连续，而且可能会跨越许多内存片段。图 3.14 展示了跨越多达 32 个元素的结果，使用单精度数据，包括设备开启 ECC 和关闭 ECC 两种情况。

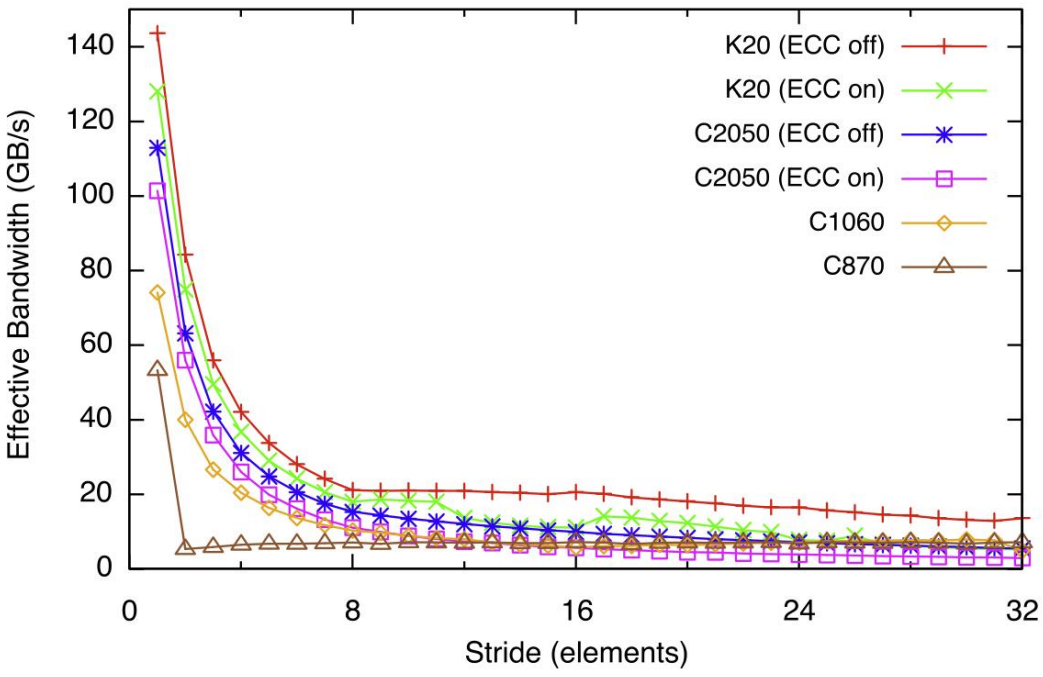


图 3.14

不同跨度下的有效带宽，数组自增内核中使用单精度数据。

与未对齐数据访问性能一样，C870 具有最严格的数据合并条件，半幅线程请求的跨度不是 1 的任何跳跃访问都将由 16 笔单独的 32 字节作业来服务。对应地，观察单位跨度的有效带宽接近 53 GB/s，而其它任何偏移的有效带宽都在 10 GB/s 以下。

对 C1060，更大跨度的有效带宽减少幅度更加平缓，因为随着跨度增大，有更多的片段需要传输。对大跨度，在 C1060 上，半幅线程由 16 个独立的 32 字节作业来服务，与 C870 上一样。

对 C2050，尽管在单位跨度上有较大的有效带宽，但在大跨度上的性能比 C1060 还低，这归咎于 C2050 上传输的是 128 字节的 L1 缓存线，而 C1060 上传输的是 32 字节的片段。通过编译器选项`-Mcuda=noL1`可以在 L1 缓存里关闭缓冲全局加载。单精度数据的结果显示在图 3.15 中，双精度数据的结果显示在图 3.16 中。对单精度和双精度，跨度分别到达 8 和 4 时，在 L1 缓存关闭的情况下，传输小于 128 节的片段能得到更高的有效带宽。

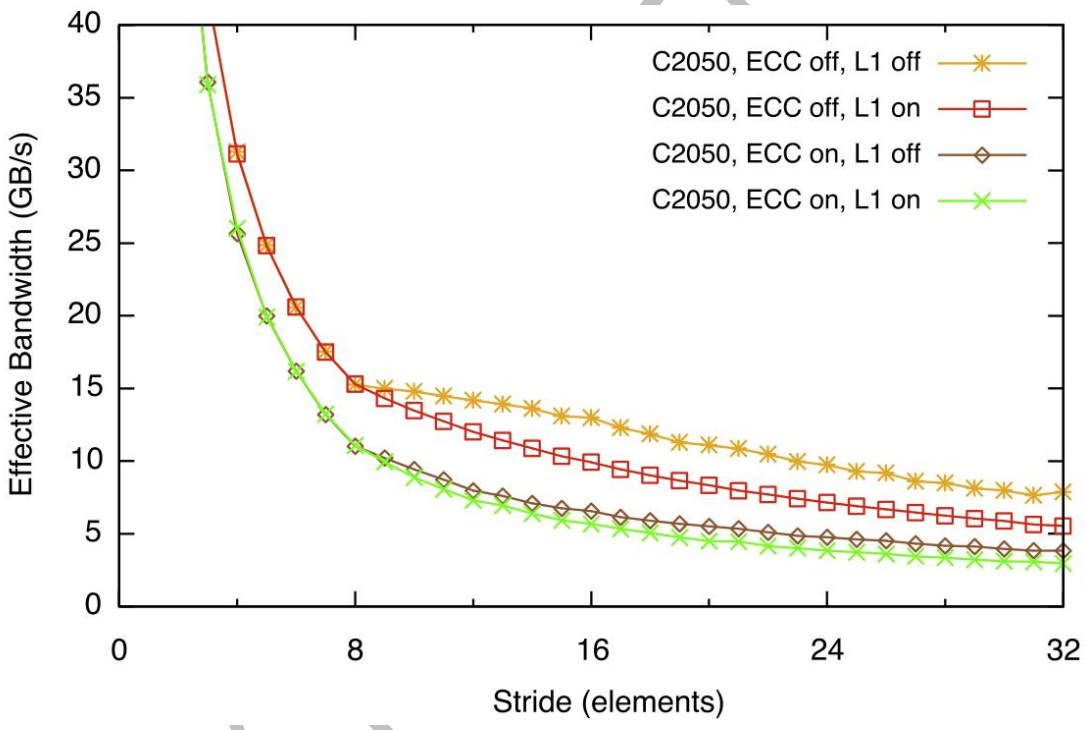


图 3.15

ECC 和 L1 缓存开启或关闭情形下，C2050 上单精度数据在不同跨度下的有效带宽。坐标范围已调整过，以便显示曲线尾部的差异。一旦到达跨度 8，关闭 L1 缓存将带来更高的有效带宽。

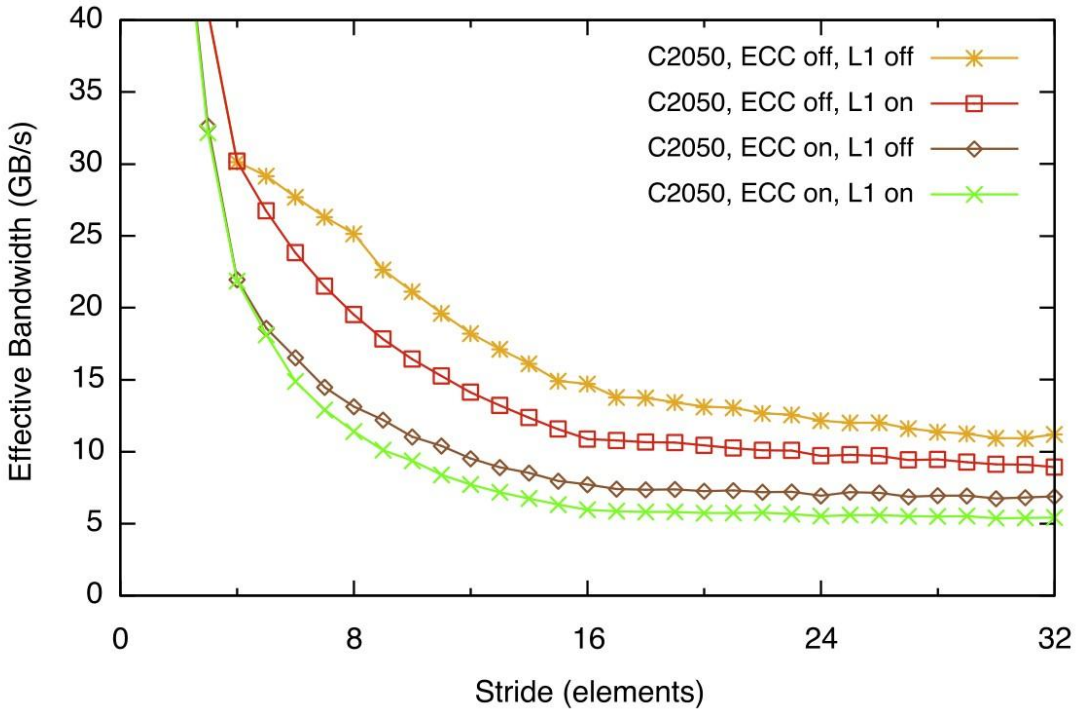


图 3.16

ECC 和 L1 缓存开启或关闭情形下，C2050 上双精度数据在不同跨度下的有效带宽。一旦到达跨度 4，关闭 L1 缓存将带来更高的有效带宽。

在 K20 上，L1 里只缓冲本地变量，因此，实际上选项`-Mcuda=noL1` 是隐式打开的，如图 3.17 所示。与 C2050 对全局加载禁用 L1 缓存相似，当跨度达到 8 时，传输更小的数据片段，并观察到有效带宽减小得更慢。

合并全局加载的讲述较长，牵扯面广。之所以如此详细，是因为数据合并是 CUDA Fortran 中获得良好性能最重要手段之一。

回顾前面的讲述，应该牢记的有几个要点。首先，更新的 GPU 架构不仅仅提高了原生性能(即最高带宽)，而且去除了获得良好性的限制和障碍。在计算能力 1.0 的设备(即 C870)中，对齐数据访问是获取良好性能的关键。在计算能力 3.5 的设备(即 K20)中，错对齐访问[4]只会带来微不足道的性能损失。

尽管数据访问的对齐在近期的 CUDA 架构上不是一个大问题，但是大跨度访问数据在所有的设备上都将导致较低的有效带宽。在高性能计算领域这不是一个新课题；数据局部性一直是应用性能调节中的一个重要课题。处理内存中跳跃的最好办法是尽量避免它。然而，有些情形下它是无法避免的，例如沿着第一维之外的某个维度访问多维数据。这些情形下有几个选项可以实施，以便获得良好性能。如果跳跃访问发生在只读数据上，可以使用纹理。另一个选项是使用片上共享内存，它被一个线程块内的所有线程共享。可以以合并的方式将数据带入共享内存，然后以跳跃方式访问它，没有任何性能损失。本章后面将讲述共享内存，但接下来先瞧一下纹理内存。

[4] 对应英文术语为 misaligned accesses －译者注。

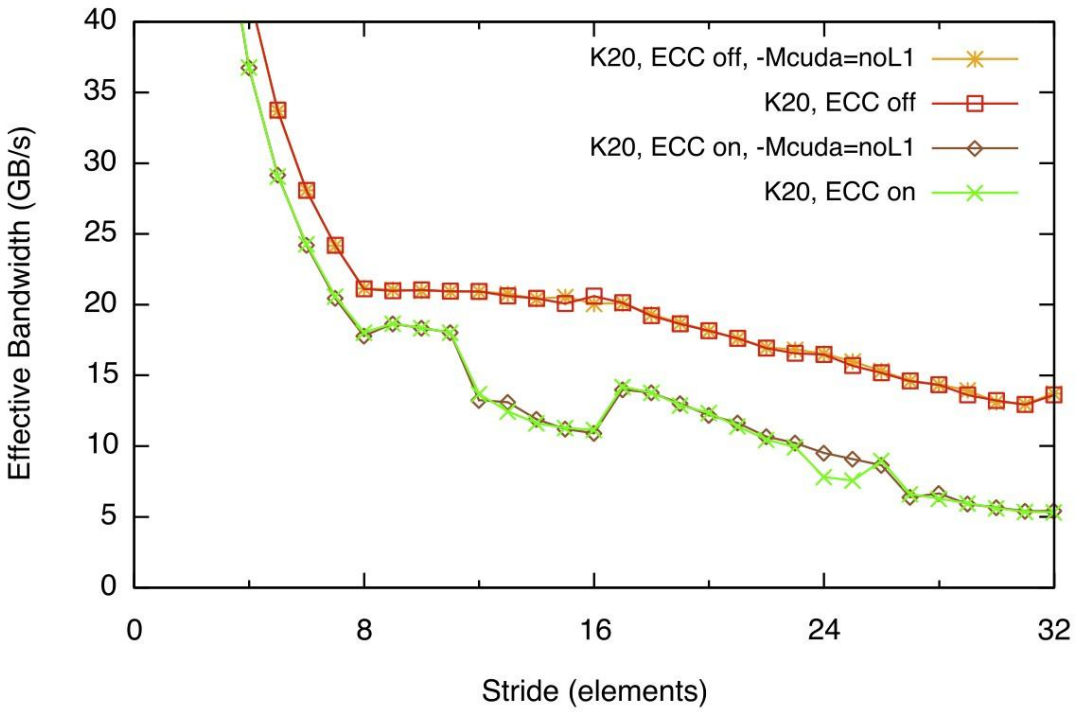


图 3.17

ECC 和 L1 缓存开启或关闭，以及用和不用编译器选项-Mcuda=noL1 的情形下，K20 上双精度数据在不同跨度下的有效带宽。因为 K20 不在 L1 中缓冲全局变量，所以使用编译选项-Mcuda=noL1 对性能无影响。

3.2.3 纹理内存(Texture Memory)

TextureswereaddedtoCUDAFortraninversion12.8ofthecompilers.Forthosefamiliarwithtexturesin CUDAC,thisimplementationisasubsetofthetexturefeaturesofferedinCUDAC,essentiallycovering the functionality offered by `tex1Dfetch()`. The filtering and wrapping/clamping capabilities of textures are not currently available in CUDA Fortran. In addition, only single precision is currently supported in CUDA Fortran textures.

Textures in CUDA Fortran allow us to access global memory in a read-only fashion through the texture cache. In addition to utilizing additional on-chip cache, textures may be advantageous in cases where access by sequential threads is to noncontiguous data, such as the strided data access pattern previously discussed. Such data access through textures may be advantageous because the minimum transaction size for textures is 32 bytes, as opposed to, say, the 128-byte cache line of the L1 cache on devices of compute capability 2.x. Although we can disable the caching of globals in L1 to obtain 32-byte transactions, doing so prohibits accessing other variables in global memory from using the L1 cache. In addition, textures can have more load requests in flight compared to global memory. We can see the benefit of textures from a modified version of the strided memory access kernel:

```
 1  module kernels_m
 2    real , texture , pointer :: aTex(:)
 3  contains
 4    attributes(global) subroutine stride(b, a, s)
 5      real :: b(*), a(*)
 6      integer , value :: s
 7      integer :: i, is
 8      i = blockDim%x*(blockIdx%x-1)+ threadIdx%x
 9      is = (blockDim%x*(blockIdx%x-1)+ threadIdx%x) * s
10      b(i) = a(is)+1
11    end subroutine stride
```

```
12
13    attributes(global) subroutine strideTex(b, s)
14      real :: b(*)
15      integer , value :: s
16      integer :: i, is
17      i = blockDim%x*(blockIdx%x-1)+ threadIdx%x
18      is = (blockDim%x*(blockIdx%x-1)+ threadIdx%x) * s
19      b(i) = aTex(is)+1
20    end subroutine strideTex
21  end module kernels_m
22
23  program strideTexture
24    use cudafor
25    use kernels_m
26
27    implicit none
28
29    integer , parameter :: nMB = 4 ! transfer size in MB
30    integer , parameter :: n = nMB*1024*1024/4
31    integer , parameter :: blockSize = 256
32    real , device , allocatable , target :: a_d(:), b_d(:)
33    type(cudaEvent) :: startEvent , stopEvent
34    type(cudaDeviceProp) :: prop
35    integer :: i, istat , ib
36    real :: time
37
38    istat = cudaGetDeviceProperties(prop , 0)
39    write(*,'(/,"Device: ",a)') trim(prop%name)
40    write(*,'("Transfer size (MB): ",i0 ,/)') nMB
41
42    allocate(a_d(n*33), b_d(n))
43
44    istat = cudaEventCreate(startEvent)
45    istat = cudaEventCreate(stopEvent)
46
47    write(*,*) 'Global version'
48    write(*,*) 'Stride , Bandwidth (GB/s)'
49    call stride <<<n/blockSize ,blockSize >>>(b_d , a_d , 1)
50    do i = 1, 32
51       a_d = 0.0
52       istat = cudaEventRecord(startEvent ,0)
53       call stride <<<n/blockSize ,blockSize >>>(b_d , a_d , i)
54       istat = cudaEventRecord(stopEvent ,0)
55       istat = cudaEventSynchronize(stopEvent)
```

```
56       istat = cudaEventElapsedTime(time , startEvent , &
57            stopEvent)
58       write(*,*) i, 2*n*4/time*1.e-6
59    enddo
60
61    ! bind the texture
62    aTex => a_d
63
64    write(*,*) 'Texture version'
65    write(*,*) 'Stride , Bandwidth (GB/s)'
66    call strideTex <<<n/blockSize ,blockSize >>>(b_d , 1)
67    do i = 1, 32
68       a_d = 0.0
69       istat = cudaEventRecord(startEvent ,0)
70       call strideTex <<<n/blockSize ,blockSize >>>(b_d , i)
71       istat = cudaEventRecord(stopEvent ,0)
72       istat = cudaEventSynchronize(stopEvent)
73       istat = cudaEventElapsedTime(time , startEvent , &
74            stopEvent)
75       write(*,*) i, 2*n*4/time*1.e-6
76    enddo
77
78    ! unbind the texture
79    nullify(aTex)
80
81    istat = cudaEventDestroy(startEvent)
82    istat = cudaEventDestroy(stopEvent)
83    deallocate(a_d , b_d)
84
85 end program strideTexture
```

Textures in CUDA Fortran make use of the Fortran 90 pointer notation to “bind” a texture to a region of global memory. The texture pointer is declared on line 2 in the preceding code using both the `texture` and the `pointer` variable attributes. The kernel that uses this texture pointer is listed on lines 13–20, and the nontexture version is listed on lines 4–11. Note that the texture pointer, `aTex`, is not passed in as an argument to the kernel, and it must be declared at module scope. If a texture pointer is passed as an argument to a kernel, even if declared in the kernel with the `texture` attribute, the

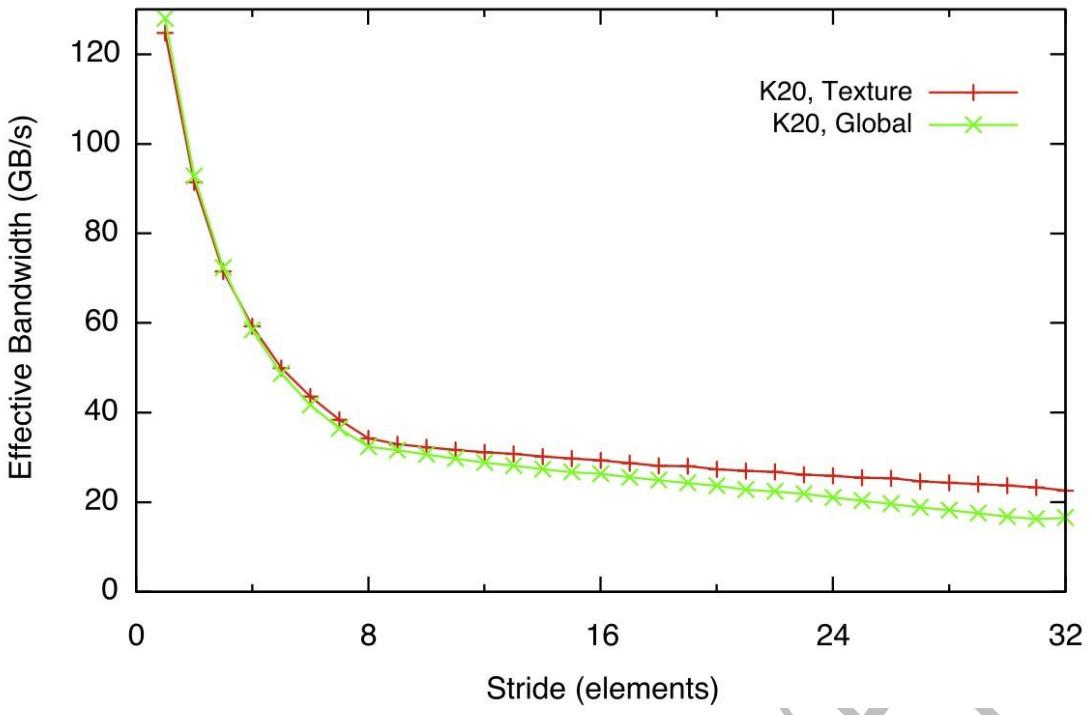


图 3.18

Effective bandwidth versus stride for single-precision data on the K20 using textures and global memory.

data will be accessed through the global memory path, not the texture path.[5] Aside from that scoping aspect, use of CUDA Fortran textures in device code is analogous to use of a global memory array, unlike CUDA C, which uses a `tex1Dfetch()` command to access the data.

Note that these kernels differ from the simple increment kernels used previously. First, since textures areread-only,thesekernelsmustusedifferentvariablesontheright-andleft-handsidesoftheassignment statement. Also, different indices are used to access these two arrays. We write the results in a coalesced fashion in order to highlight the effect of the strided reads.

In host code, the device data to which a texture is bound must be declared with the `target` attribute, as is done on line 32, which is standard practice with Fortran pointers. On line 62, the texture binding occurs using the pointer notation. And on line 79, the texture is unbound using the Fortran 90 `nullify()` command.

Running this code on the K20 we see slightly improved performance at large strides with the texture version, as shown in Figure 3.18. The improved performance at large strides is due to the ability of textures to have more load requests in flight than global memory.

On the K10, K20, and K20X, where the L1 cache is used only for caching local data, the texture cache is especially attractive for read-only data that is reused in the kernel. For example, the following code kernels calculate at each interior point in a 2D mesh the average of the nearest four and eight points using both global and texture memory:

```
1  module kernels_m
2    real , texture , pointer :: aTex(:,:)
3    integer , parameter :: n = 2048
4    integer , parameter :: nTile = 32
5  contains
6    attributes(global) subroutine average4(b, a)
7      implicit none
```

[5] To verify use of the texture path, compile the code with `-Mcuda=keepgpu`, which dumps the generated CUDA C code.

The texture fetch will be denoted in this code by `__pgi_texfetchf()`.

```
 8      real :: b(n,n), a(0:n+1,0:n+1)
 9      integer :: i, j
10      i = blockDim%x*(blockIdx%x-1)+ threadIdx%x
11      j = blockDim%y*(blockIdx%y-1)+ threadIdx%y
12      b(i,j) = 0.25*( &
13                    a(i-1,j)+ &
14           a(i,j-1)+           a(i,j+1)+&
15                    a(i+1,j))
16    end subroutine average4
17
18    attributes(global) subroutine average8(b, a)
19      implicit none
20      real :: b(n,n), a(0:n+1,0:n+1)
21      integer :: i, j
22      i = blockDim%x*(blockIdx%x-1)+ threadIdx%x
23      j = blockDim%y*(blockIdx%y-1)+ threadIdx%y
24      b(i,j) = 0.125*( &
25           a(i-1,j-1)+a(i-1,j)+a(i-1,j+1)+ &
26           a(i,j-1)+           a(i,j+1)+&
27           a(i+1,j-1)+a(i+1,j)+a(i+1,j+1))
28    end subroutine average8
29
30    attributes(global) subroutine average4Tex(b)
31      implicit none
32      real :: b(n,n)
33      integer :: i, j
34      i = blockDim%x*(blockIdx%x-1)+ threadIdx%x
35      j = blockDim%y*(blockIdx%y-1)+ threadIdx%y
36      b(i,j) = 0.25*( &
37                        aTex(i-1,j)+ &
38           aTex(i,j-1)+             aTex(i,j+1)+ &
39                        aTex(i+1,j))
40    end subroutine average4Tex
41
42    attributes(global) subroutine average8Tex(b)
43      implicit none
44      real :: b(n,n)
45      integer :: i, j
46      i = blockDim%x*(blockIdx%x-1)+ threadIdx%x
47      j = blockDim%y*(blockIdx%y-1)+ threadIdx%y
48      b(i,j) = 0.125*( &
49           aTex(i-1,j-1)+aTex(i-1,j)+aTex(i-1,j+1)+ &
50           aTex(i,j-1)+             aTex(i,j+1)+ &
51           aTex(i+1,j-1)+aTex(i+1,j)+aTex(i+1,j+1))
```

```
52    end subroutine average8Tex
53  end module kernels_m
```

Thecomplete code iscontained inAppendix D.1. Thisaccess patternisvery common infinitedifference codes, and in Chapter 6 we will show an example of its use in solving the Laplace equation. Running this code on a Tesla K20, we obtain:

```
Device: Tesla K20

  4-point versions
    Global Bandwidth (GB/s):      90.71741
      Max Error:     0.000000
    Texture Bandwidth (GB/s):     94.64387
      Max Error:     0.000000
  8-point versions
    Global Bandwidth (GB/s):      58.48986
      Max Error:     0.000000
    Texture Bandwidth (GB/s):     82.60018
      Max Error:     0.000000
```

where we see a substantial improvement in bandwidth for textures in the eight-point stencil case, where data reuse is large.

3.2.4 本地内存

本地内存(Local memory)是存储在设备 DRAM 中的线程私有内存。这里的名字本地是指一个变量的作用域(意为线程私有)而不是代表它的物理位置，它物理位置上是位于芯片外部的设备 DRAM 中。依赖于本地内存的使用量和是否缓冲本地内存，本地内存可能会成为一个性能瓶颈。

3.2.4.1 探测本地内存使用情况（高级话题）

通过编译下列内核集来检测在什么条件下对数组使用本地内存：

```
 1  module localmem
 2    implicit none
 3  contains
 4    attributes(global) subroutine k1(a)
 5      real :: a(*), b(2)
 6      integer :: i
 7      i = blockDim%x*(blockIdx%x-1) + threadIdx%x
 8      b(1) = 1; b(2) = 2
 9      a(i) = b(2)
10    end subroutine k1
11
12    attributes(global) subroutine k2(a,j,k)
13      real :: a(*), b(2)
14      integer :: i,j,k
```

```
15      b(j) = 1.0
16      i = blockDim%x*(blockIdx%x-1) + threadIdx%x
17      a(i) = b(k)
18    end subroutine k2
19
20    attributes(global) subroutine k3(a)
21      real :: a(*), b(256)
22      integer :: i
23      b = 1.0
24      i = blockDim%x*(blockIdx%x-1) + threadIdx%x
25      a(i) = b(2)
26    end subroutine k3
27  end module localmem
```

三个内核都声明了变量 b ,它是线程私有数据 :对这个数组 ,执行内核的每一个线程都拥有它自己的版本。在第一个内核中 ,b 仅包含两个元素且使用静态下标访问。第二个内核中 ,b 也是一个有两个元素的数据，但通过变量或动态下标访问。第三个内核中，b 被声明成拥有 256 个元素，由于数据初始化语句 b=1.0，它被以动态形式访问。

在编译过程中使用编译器选项-Mcuda=ptxinfo 可以获取本地内存使用量的反馈信息。如果为计算能力 1.x 的设备编译前面的代码，那么得到如下输出：

```
% pgf90 -c -Mcuda=ptxinfo ,cc10 local.cuf
ptxas info    : Compiling entry function 'k1' for 'sm_10'
ptxas info    : Used 2 registers , 8+16 bytes smem

ptxas info    : Compiling entry function 'k2' for 'sm_10'
ptxas info    : Used 3 registers , 8+0 bytes lmem ,
                24+16 bytes smem

ptxas info    : Compiling entry function 'k3' for 'sm_10'
ptxas info    : Used 3 registers , 1024+0 bytes lmem ,
                8+16 bytes smem , 4 bytes cmem[1]
```

在第一个内核中，编译器反馈中没有提到本地内存；因此数据放在寄存器内存中。这个是理解想情形。因为寄存器内存不能用下标，所以第二个内核里的动态下标迫使在本地内存中分配数组，正如 8+0 bytes lmem 所示 ,这里的记号 8+0 指示不同的编译阶段。第三个内核里的数组赋值相当于动态下标 ,导致 1024+0 bytes lmem，因此这个数组也驻留本地内存中。

为计算能力 2.0 编译过程中，得到：

```
% pgf90 -c -Mcuda=ptxinfo ,cc20 local.cuf
ptxas info    : Compiling entry function 'k1' for 'sm_20'
ptxas info    : Function properties for k1
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 6 registers , 40 bytes cmem[0]
```

```
ptxas info    : Compiling entry function 'k2' for 'sm_20'
ptxas info    : Function properties for k2
    8 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 10 registers , 56 bytes cmem[0]
ptxas info    : Compiling entry function 'k3' for 'sm_20'
ptxas info    : Function properties for k3
    1024 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 19 registers , 40 bytes cmem[0]
```

对计算能力 2.0 及更高，本地内存使用情况由参数 `stack frame` 报告，这里观察到第二个和第三个内核的本地内存使用情况类似，报告为 `8 bytes stack frame` 和 `1024 bytes stack frame`.

确定使用多少本地内存及使用多频繁的另一种方法是检查生成的 PTX 代码。可以使用编译器选项 `-Mcuda=keepptx` 将 PTX 中间代码保存到本地目录中一个带有 `.ptx` 扩展名的文件里。本地内存将使用助记符 `.local` 声明 – 例如：

```
            .local .align 8 .b8 __local_depot2 [1024];
```

且用 `ld.local` 或 `st.local` 来访问：

```
            st.local.u32 [%rl6+-4], %r10;
```

到目前为止已经讲完设备代码中声明的线程私有数组对本地内存的使用情况，但是当源代码超过寄存器上限时，本地内存也被用来保存设备代码中声明的标量变量。寄存器溢出加载和溢出存储都会随着 stack frame 一同报告 – 例如：

```
ptxas info    : Compiling entry function 'jacobian_v1'
                for 'sm_20'
ptxas info    : Function properties for jacobian_v1
     160 bytes stack frame , 164 bytes spill stores ,
                176 bytes spill loads
ptxas info    : Used 63 registers , 4200+0 bytes smem ,
                100 bytes cmem[0], 176 bytes cmem[2],
                40 bytes cmem[16]
```

它显示有 164 次溢出存储和 176 次溢出加载。应该注意，溢出加载和溢出存储实行静态计数，并反映在生成代码中加载指令和存储指令的数量上(用每个加载/存储的尺寸来加权)。这些指令执行的频率没有计算在内。无论寄存器溢出加载和溢出存储是否发生一个循环之内都不会反映在这些数字上。

为建立本地内存的使用频率，无论是来自由于尺寸、动态下标、寄存器溢出而放入本地内存的数组，都应该对测绘器重新排序。请注意，尽管在计算能力 1.x 上要尽最大努力避免使用本地内存，但是在计算能力 2.x 及更高的设备上，本地内存可能不会拉低性能，这是因为 L1 缓存缓冲了本地内存。如果 L1 级存内存里没有资源争抢，那么本地内存可能会包含在 L1 中。通过使用 `l1_local_load_hit` 及其关联计数器，测绘器可以协助评估。可以利用函数 `cudaFuncSetCacheConfig()` 和 `cudaDeviceSetCacheConfig()` 来增强 L1 资源，L1 缓存一节将有详述。此外，还可以借助 `-Mcuda=noL1` 来禁止全局变量使用 L1 缓存，从而为本地变量留出更多资源。

3.2.5 常量内存

所有的 CUDA 设备都拥有 64 KB 常量内存。常量内存对内核来说是只读的，但对主机来说既可读又可写。常量内存在芯片上缓冲，对没有 L1 缓存的设备，以及因使用编译器选项-Mcuda=noL1 而没有设定全局缓冲的设备，这是一个巨大优势。

半幅(计算能力 1.x)或整幅内(计算能力 2.0 及更高)的线程对常量缓存中不同地址的访问会被串行化的，因为只有一个读取端口。从而，当半幅或整幅内的所有线程都访问同一个地址时常量缓存效率最高。它的一个良好应用场景是物理常量。

CUDA Fortran 中，常量数据必须在模块的声明区内声明，即 contains 之前，且能够在本模块内的任何代码中使用，也能在包含该模块的任何主机代码中使用。

用常量内存改写自增例子：

```
module simpleOps_m
  integer , constant :: b
contains
  attributes(global) subroutine increment(a)
    implicit none
    integer , intent(inout) :: a(:)
    integer :: i

    i = threadIdx%x
    a(i) = a(i)+b

  end subroutine increment
end module simpleOps_m


program incrementTest
  use cudafor
  use simpleOps_m
  implicit none
  integer , parameter :: n = 256
  integer :: a(n)
  integer , device :: a_d(n)

  a = 1
  b = 3

  a_d = a
  call increment <<<1,n>>>(a_d)
  a = a_d

  if (any(a /= 4)) then
     write(*,*) '**** Program Failed ****'
```

```
33     else
34        write(*,*) 'Program Passed'
35     endif
36  end program incrementTest
```

第 2 行上用属性 constant 将参数 b 声明常量变量。内核不再将 b 用作一个参数，它也不必在主机代码里声明。除了这些改变(简化)之外，这个代码与引述中用到的代码保持相同。

对模块中声明的变量，用常量内存试验非常简单。将变量属性在 constant 和 device 之间简单切换就能分别将变量放置在常量内存和全局内存。

3.2.5.1 探测常量内存使用情况（高级话题）

像本地内存一样，通过选项-Mcuda=ptxinfo 可以在编译时看到常量内存的使用情况，这里常量内存的使用情况由多个 cmem[]值标识。应该牢记，编译器广泛使用常量内存。编译器对常量内存的使用量依赖于所针对的计算能力。如果针对计算能力 1.0 编译此代码，并且用变量属性 device 来声明 b，将得到：

```
ptxas info    : Compiling entry function 'increment' for 'sm_10'
ptxas info    : Used 4 registers , 16+16 bytes smem , 4 bytes cmem[14]
```

如用变量属性 constant 来声明变量 b，将得到：

```
ptxas info    : Compiling entry function 'increment' for 'sm_10'
ptxas info    : Used 4 registers , 16+16 bytes smem , 16 bytes cmem[0]
```

当针对计算能力 2.0 并且用变量属性 device 声明变量 b，会得到：

```
ptxas info    : Compiling entry function 'increment' for 'sm_20'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 8 registers , 48 bytes cmem[0], 8 bytes cmem[14]
```

用变量属性 constant 声明变量 b，会得到：

```
ptxas info    : Compiling entry function 'increment' for 'sm_20'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 6 registers , 48 bytes cmem[0], 16 bytes cmem[2]
```

在计算能力 2.0 及更高的设备上，内核参数存储在常量内存里，用 cmem[0]标出，这是常量内存用量比计算能力 1.x(这些参数放在共享内存中)要高的原因之一。

计算能力 2.0 及更高的设备支持一致加载(LoaD Uniform, LDU)指令，如果在内核中的某个变量是只读的，数组的话，下标不依赖于变量 threadIdx，那么该指令经过常量缓存来加载这个全局内存中的变量。最后一个必备条件保证一幅内的每个线程都在访问同一个值，最优利用常量缓存。从而，对自增内核来说，无论是用变量属性 constant 将 b 声明在常量内存中还是用变量属性 device 将 b 声明在全局内存中，都会使用常量缓存来缓冲 b. 可以使用选项-Mcuda=keepptx 而后查看 PTX 代码来验证这一点。b 的声明中使用变量属性 device 时，语句

```
            ldu.global.u32 %r8, [_simpleops_m_16];
```

表明经过常量缓存对全局内存的一个 32 位字进行一致加载。对内核参数还有其它的一致加载发生，但是参数_simpleops_m_16 表明加载的是用户定义的模块数据，其中 simpleops_m 是模块名字。与此相

对，当在声明 b 时使用变量属性 constant，PTX 代码包含：

```
ld.const.u32 %r10 , [_simpleops_m_17];
```

表示一个从常量内存出发的加载。两种情形都能从常量缓存中获益。因为有了一致加载指令，在性能调节中，通过变量标识符 constant 显式地使用常量缓存不再那么重要。事实上，DRAM 中预留 64KB 的常量内存空间，对数据量超过 64KB 的一些场景，依靠一致加载指令更有利。但要注意，当 b 声明为 constant 变量时，寄存器用量较少。在一些寄存器压力较大的情形，将一些只读变量声明在常量内存中效果可能比较好。

3.3 芯片上的内存

本节讲述芯片上的多种内存。本节用大部分篇幅讲述共享内存及其用法，把它留到最后。讲述共享内存之前，简要点评一下寄存器的用法和计算能力 2.0 及以上卡片的 L1 缓存。

3.3.1 L1 缓存

在计算能力 2.x 和 3.x 上，每个多处理器都拥有 64 KB 的片上内存，这些内存可以在 L1 缓存和共享内存之间配置。在计算能力 2.x 的设备上有两种设置，48 KB 共享内存/16 KB L1 缓存和 16 KB 共享内存/48 KB L1 缓存。在计算能力 3.x 的设备上有三种设置，刚刚提到的两种和 32 KB 共享内存/32 KB L1 缓存。默认设置是 48 KB 共享内存。

在程序运行期间，可以从主机端配置共享内存/L1 缓存，使用运行时函数 cudaDeviceSetCacheConfig()可对设备上的所有内核生效，使用 cudaFuncSetCacheConfig()对每个内核生效。前一个例程接受一个参数，从下列偏好中选一个：cudaFuncCachePreferNone、cudaFuncCachePreferShared 均对应 48 KB 共享内存和 16 KB L1 缓存，cudaFuncCachePreferL1 对应 16 KB 共享内存和 48 KB L1 缓存，在计算能力 3.x 的设备上，cudaFuncCachePreferEqual 对应 32 KB 共享内存和 32 KB L1 缓存。配置例程 cudaFuncSetCacheConfig 接受函数名字作为第一参数，一个偏好选项做为第二个参数。驱动程序会尽可能遵守偏好设定，当单个线程请求的共享内存超过缓存配置的设定时，驱动程序就不再遵守偏好设定，这也是为什么默认设置倾向于分配较大的共享内存。

内核执行期间请求和使用的缓存配置可以利用测绘器选项 cacheconfigrequested 和 cacheconfigexecuted 来验证。例如，在测绘配置文件里指定这些选项，然后运行自增代码，得到：

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff69c52860010
method, gputime, cputime, occupancy, cacheconfigexecuted,
        cacheconfigrequested
method=[ memcpyHtoD ] gputime=[ 1.440 ] cputime=[ 9.000 ]
method=[ memcpyHtoD ] gputime=[ 0.928 ] cputime=[ 8.000 ]
method=[ increment ] gputime=[ 5.472 ] cputime=[ 521.000 ]
       occupancy =[0.125 ]
       cacheconfigrequested=[ 0 ]cacheconfigexecuted=[ 0 ]
```

```
method=[ memcpyDtoH ] gputime=[ 2.656 ] cputime=[ 60.000 ]
```

这里的值 0 代表 cudaFuncCachePreferNone. 值 1、2 和 3 分别对应 cudaFuncCachePreferShared、cudaFuncCachePreferL1 和 cudaFuncCachePreferEqual.

作为启动内核前的一个主机代码预处理步骤，实践经验是尽可能改变缓存配置，这是因为改动配置可以使用内核执行空转。

对 Tesla K10、 K20 和 K20X，L1 缓存仅被用于本地内存，这是因为驻留全局内存的变量仅在 L2 缓存里缓冲。对计算能力 2.0 的设备，L1 缓存默认用于全局内存里和本地内存里的变量。正如从对全局内存跳跃访问合并的讲述中看到的那样，关闭全局加载的 L1 缓冲来避免 128 字节缓存线加载，可能效果更好。每个编译单元中使用选项-Mcuda=noL1 能够关闭缓冲。

3.3.2 寄存器

寄存器内存是线程私有内存，它在一个多处理器上驻留的所有线程间分割。设备例程本地声明的所有变量，如果不带变量属性 shared，这些变量要么放在寄存器内存，要么放在本地内存。如果有足够的空间，线程私有的标量变量会放到寄存器中；线程私有的数组可能放也可能不放在寄存器中，放与不放取决于数组的尺寸和数组如何编址。至于什么会被放到本地内存中，详情查看 3.2.4 节。因为寄存器位于芯片内部，而本地内存在设备 DRAM 中(尽管它能在芯片上缓冲)，所以线程私有变量倾向于驻留在寄存器中。

每个多处理器拥有的 32 位寄存器的数量，每一代设备都会稳定增长，从计算能力 1.0 的 8K 寄存器到计算能力 3.x 设备的 64K 寄存器。各种设备的寄存器特性描述请参看附录 A. 借助派生类型 cudaDeviceProp 的域 regsPerBlock，可以在运行时查询每个多处理器上的寄存器数量。

内核里每个线程使用的寄存器数量由编译器控制。然而，程序员可以在编译阶段使用编译器选项 -Mcuda=maxregcount:N 来限制每个内核使用的寄存器数量。限制每个线程的寄存器可以提高并发驻留在一个多处理器上的线程块数量 ,从而带来更好的延时隐藏。然而 ,限制寄存器数量会增大寄存器压力。

对一个给定的任务，当没有足够的可用寄存器时就会发生寄存器承压。后果就是寄存器溢出到本地内存。更高占用率和寄存器溢出这两要素相互对立，因此常常需要实验几次才能获得最优配置。使用编译器选项-Mcuda=ptxinfo 可以抓取每个内核的寄存器溢出和本地内存的加载溢出与存储溢出信息。例如，编译自增内核的常量内存版，得到：

```
ptxas info    : Compiling entry function 'increment' for 'sm_20'
ptxas info    : Function properties for increment
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 8 registers , 48 bytes cmem[0], 8 bytes cmem[14]
```

它表明 ,在该设备上 ,这个内核的每个线程使用 8 个寄存器。对一个计算能力 2.0 的设备 ,在满占用率下，每个多处理器最多有 1536 个线程，这意味着每个多处理器上驻留 1536 个线程，每个多处理器上总共使用 12288 个寄存器，远小于可用的 32K 个寄存器。从而可以期待该内核能以满占用率运行。注意，如果寄存器溢出到芯片上的 L1 缓存中且没有被迫溢出到设备内存，那么寄存器溢出不会必然带来性能问题。该问题的详述参见 3.2.4 节。不仅一个多处理器上可用寄存器有上限，每个线程使用的寄存器数量也有上限：对计算能力 1.x 每个线程 127 个寄存器，对计算能力 2.x 和 3.0，每个线程 63 个寄存器；对计算能力 3.5，每个线程 255 个寄存器。

用-Mcuda=ptxinfo 编译可以得到编译时的寄存器使用情况，此外，在配置文件里指定选项

regperthread，命令行测绘器也能提供寄存器的使用情况。

编译器和硬件线程调度器将尽最大努力最优地调度指令，以避免寄存器条带冲突。当每块的线程数量是 64 的整数倍时，它们将获得最好结果。除了遵守这个准则之外，应用程序无法直接控制这些条带冲突。

3.3.3 共享内存

这里讲述的最后一个片上内存是共享内存。不像寄存器内存和片上缓存，程序员能够完全控制共享内存，决定使用多少共享内存，哪些变量使用它，及如何访问它。共享内存按每个线程块分配 ，这是因为块内的所有线程都访问相同的共享内存。因为一个线程从全局内存加载的内享内存能被同一个线程块内的另一个线程访问，所以共享内存用来促成全局内存的合并访问，万一其它办法做不到的时候。

设备代码中用变量修饰符 shared 声明共享内存。根据编译时或运行时内存容量是否已知，内核内部可用多种方法声明共享内存。接下的代码演示了共享内存的多种使用方法：

```
! This code shows how dynamically and statically allocated
! shared memory are used to reverse a small array

module reverse_m
  implicit none
  integer , device :: n_d
contains
  attributes(global) subroutine staticReverse(d)
    real :: d(:)
    integer :: t, tr
    real , shared :: s(64)

    t = threadIdx%x
    tr = size(d)-t+1

    s(t) = d(t)
    call syncthreads()
    d(t) = s(tr)
  end subroutine staticReverse

  attributes(global) subroutine dynamicReverse1(d)
    real :: d(:)
    integer :: t, tr
    real , shared :: s(*)

    t = threadIdx%x
    tr = size(d)-t+1

    s(t) = d(t)
    call syncthreads()
    d(t) = s(tr)
  end subroutine dynamicReverse1
```

```
33
34    attributes(global) subroutine dynamicReverse2(d, nSize)
35      real :: d(nSize)
36      integer , value :: nSize
37      integer :: t, tr
38      real , shared :: s(nSize)
39
40      t = threadIdx%x
41      tr = nSize -t+1
42
43      s(t) = d(t)
44      call syncthreads()
45      d(t) = s(tr)
46    end subroutine dynamicReverse2
47
48    attributes(global) subroutine dynamicReverse3(d)
49      real :: d(n_d)
50      real , shared :: s(n_d)
51      integer :: t, tr
52
53      t = threadIdx%x
54      tr = n_d -t+1
55
56      s(t) = d(t)
57      call syncthreads()
58      d(t) = s(tr)
59    end subroutine dynamicReverse3
60  end module reverse_m
61
62
63  program sharedExample
64    use cudafor
65    use reverse_m
66
67    implicit none
68
69    integer , parameter :: n = 64
70    real :: a(n), r(n), d(n)
71    real , device :: d_d(n)
72    type(dim3) :: grid , tBlock
73    integer :: i, sizeInBytes
74
75    tBlock = dim3(n,1,1)
76    grid = dim3(1,1,1)
```

```
 77
 78    do i = 1, n
 79       a(i) = i
 80       r(i) = n-i+1
 81    enddo
 82
 83    sizeInBytes = sizeof(a(1))* tBlock%x
 84
 85    ! run version with static shared memory
 86    d_d = a
 87    call staticReverse <<<grid ,tBlock >>>(d_d)
 88    d = d_d
 89    write(*,*) 'Static case max error:', maxval(abs(r-d))
 90
 91    ! run dynamic shared memory version 1
 92    d_d = a
 93    call dynamicReverse1 <<<grid ,tBlock ,sizeInBytes >>>(d_d)
 94    d = d_d
 95    write(*,*) 'Dynamic case 1 max error:', maxval(abs(r-d))
 96
 97    ! run dynamic shared memory version 2
 98    d_d = a
 99    call dynamicReverse2 <<<grid ,tBlock ,sizeInBytes >>>(d_d ,n)
100    d = d_d
101    write(*,*) 'Dynamic case 2 max error:', maxval(abs(r-d))
102
103    ! run dynamic shared memory version 3
104    n_d = n ! n_d declared in reverse_m
105    d_d = a
106    call dynamicReverse3 <<<grid ,tBlock ,sizeInBytes >>>(d_d)
107    d = d_d
108    write(*,*) 'Dynamic case 3 max error:', maxval(abs(r-d))
109
110 end program sharedExample
```

这个代码用共享内存来翻转一个 64 元素数组中的数据。所有内核的代码都非常相似；主要区别在于如何声明共享内存数组和如何调用内核。如果共享内存数组的尺寸在编译时已知，就像内核 `staticReverse` 中的那样 ,那么就用那个尺寸值来声明数组 ,无论尺寸值是一个整型参数还是一个字面值 ,就像第 11 行 `s(64)` 做的那样。该内核中，代表原始顺序和翻转后顺序的两个下标分别在第 13 和 14 行上计算。第 16 行上，数据被从全局内存复制到共享内存。反转在第 18 行完成，下标 `t` 和 `tr` 用来将数据从共享内存复制到全局内存。第 18 行上，每个线程访问的数据都是另一线程写入的，执行此行之前，必须确保第 16 行上所有线程对共享内存的加载都已经完成。这个要求由第 17 行上的障碍同步 `syncthreads()` 实现。这个障碍同步发生在一个线程块内的所有线程之间，它意味着在同一个线程块内的所有线程都到达此行之前，没有线程能够

跨过此行。这个例子中使用共享内存是为了促成全局内存合并。因为全局内存总是通过下标 t 来访问，所以读和写均已达到最优的全局内存合并。反转下标 tr 只用来访问共享内存，为了最优性能，共享内存没有全局内存那些访问限制。共享内存唯一的性能问题是条带冲突，下节会有详述。

本例中的另外三个内核都使用动态共享内存，在编译阶段共享内存的数量未知，当调用内核时必须在执行配置的第三个可选参数里指定共享内存的数量(按字节)，正如第 93、99、106 行做的那样。第一个动态共享内存内核，dynamicReverse1，在第 24 行上用假定尺寸数组语法来声明共享内存数组。内核启动时，从第三个执行配置参数隐式指定尺寸。这个内核的其余代码与 staticReverse 内核完全相同。

如 dynamicReverse2 和 dynamicReverse3 所示，可以通过自动数组使用共享内存。这些示例中，动态共享内存的维度由作用域内的一个整数指定。在 dynamicReverse2 中，第 38 行上用子例程参数 nSize 声明共享内存数组的尺寸，在 dynamicReverse3 中，第 50 行上用开头模块中声明的设备变量 n_d 来声明共享内存的尺寸。注意，这个两个示例内核被调用时必须在执行配置的第三个参数中指定动态内存的尺寸。

声明动态共享内存有这些备选方法，应该选用哪个？如果打算使用多个动态共享内存，特别是它们还属于不同的类型，那么需要使用 dynamicReverse2 和 dynamicReverse3 中那样的自动数组。如果想用第 24 行那样的假定尺寸记号来指定多个动态共享内存数组，那么编译器如何知道怎么在这些数组之间分配动态共享内存总量？不但是这个因素，选择也要由程序员做出；这些声明方法的性能没有差别。

3.3.3.1 探测共享内存使用情况（高级话题）

当使用编译器选项-Mcuda=ptxinfo 时，编译过程中会报告每一个内核中每一个线程块的静态共享内存使用情况。例如，针对计算能力 3.0 编译数组反转代码，得到：

```
ptxas info    : Compiling entry function 'staticreverse' for 'sm_30'
ptxas info    : Function properties for staticreverse
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 10 registers , 256+0 bytes smem , 336 bytes cmem[0]

ptxas info    : Compiling entry function 'dynamicreverse1' for 'sm_30'
ptxas info    : Function properties for dynamicreverse1
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 10 registers , 336 bytes cmem[0]

ptxas info    : Compiling entry function 'dynamicreverse2' for 'sm_30'
ptxas info    : Function properties for dynamicreverse2
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 6 registers , 332 bytes cmem[0]

ptxas info    : Compiling entry function 'dynamicreverse3' for 'sm_30'
ptxas info    : Function properties for dynamicreverse3
    0 bytes stack frame , 0 bytes spill stores , 0 bytes spill loads
ptxas info    : Used 10 registers , 328 bytes cmem[0]
```

这里只有内核 staticReverse 指明预留了 256 字节的共享内存。注意，在计算能力 1.x 设备上，系统也会使用静态共享内存。因此，针对计算能力 1.0 设备，得到：

```
ptxas info    : Compiling entry function 'staticreverse' for 'sm_10'
ptxas info    : Used 4 registers , 272+16 bytes smem , 4 bytes cmem[14]

ptxas info    : Compiling entry function 'dynamicreverse1' for 'sm_10'
ptxas info    : Used 4 registers , 16+16 bytes smem , 4 bytes cmem[14]

ptxas info    : Compiling entry function 'dynamicreverse2' for 'sm_10'
ptxas info    : Used 3 registers , 16+16 bytes smem , 4 bytes cmem[14]

ptxas info    : Compiling entry function 'dynamicreverse3' for 'sm_10'
ptxas info    : Used 4 registers , 16+16 bytes smem , 4 bytes cmem[14]
```

这里观察到每个内核都使用了共享内存。

当将选项 stasmemperblock 和 dynsmemperblock 放入配置文件时，命令行测绘器能够报告静态共享内存和动态共享内存的用量。以这种方法测绘数组反转代码，得到：

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff69de87d2020
method ,gputime ,cputime ,dynsmemperblock ,stasmemperblock ,occupancy
method=[ memcpyHtoD ] gputime=[ 1.344 ] cputime=[ 9.000 ]
method=[ memcpyHtoD ] gputime=[ 0.928 ] cputime=[ 8.000 ]
method=[ staticreverse ] gputime=[ 5.568 ] cputime=[ 9.000 ]
       dynsmemperblock=[ 0 ] stasmemperblock=[ 256 ]
       occupancy=[ 0.031 ]
method=[ memcpyDtoH ] gputime=[ 2.560 ] cputime=[ 57.000 ]
method=[ memcpyHtoD ] gputime=[ 0.928 ] cputime=[ 8.000 ]
method=[ memcpyHtoD ] gputime=[ 0.896 ] cputime=[ 7.000 ]
method=[ dynamicreverse1 ] gputime=[ 5.088 ] cputime=[ 9.000 ]
       dynsmemperblock=[ 256 ] stasmemperblock=[ 0 ]
       occupancy=[ 0.031 ]
method=[ memcpyDtoH ] gputime=[ 2.592 ] cputime=[ 57.000 ]
method=[ memcpyHtoD ] gputime=[ 0.928 ] cputime=[ 8.000 ]
method=[ dynamicreverse2 ] gputime=[ 3.904 ] cputime=[ 10.000 ]
       dynsmemperblock=[ 256 ] stasmemperblock=[ 0 ]
       occupancy=[ 0.031 ]
method=[ memcpyDtoH ] gputime=[ 2.144 ] cputime=[ 59.000 ]
method=[ memcpyHtoD ] gputime=[ 0.896 ] cputime=[ 8.000 ]
method=[ memcpyHtoD ] gputime=[ 0.928 ] cputime=[ 7.000 ]
method=[ dynamicreverse3 ] gputime=[ 4.544 ] cputime=[ 8.000 ]
       dynsmemperblock=[ 256 ] stasmemperblock=[ 0 ]
       occupancy=[ 0.031 ]
method=[ memcpyDtoH ] gputime=[ 2.112 ] cputime=[ 55.000 ]
```

3.3.3.2 共享内存条带冲突

为了达到并发访问的最高内存带宽，共享内存被切分成尺寸相等的内存模块(条带)，它们可以被同时访问。因此，跨越 *n* 个不同内存条带的 *n* 个地址的任意内存加载或存储都能被同时服务，产生的有效带宽高达单条带带宽的 *n* 倍。

然而，如果一个内存请求的多个地址映射到同一个内存条带，那么这些访问会被串行化。硬件将具有条带冲突的内存请求按需分裂成多个单独的无冲突请求，有效带宽的减少因子等于单独内存请求的数量。这里有一个例外，半幅或整幅内的所有线程访问同一个共享内存位置将导致一个广播。计算能力 2.0 及更高的设备具有多次广播共享内存访问的额外功能，这意味着同一幅内任意数量的线程多次访问同一个位置将被同时服务。

为最小化条带冲突，理解内存地址是如何映射到内存条带的以及如何最优在调度内存请求就变得格外重要。共享内存这样组织，连续的 32 位字指派给连续条带，每个条带在每个时钟周期都有一个 32 位的带宽。共享内存带宽是每个时钟周期每个条带 32 位。

对计算能力 1.x 的设备，幅尺寸是 32 线程，条带数量是 16. 整幅的一个共享内存请求被拆分成一个服务前半幅的请求和一个服务后半幅的请求。注意，如果半幅线程对每个条带仅访问一个内存位置，那么没有条带冲突。

对计算能力 2.x 的设备，幅尺寸是 32 线程，条带数量也是 32. 整幅的一个共享内存请求不会像计算能力 1.x 设备上那样分裂，这意味着同幅内的前半幅线程和后半幅线程之间可能会发生条带冲突。

在计算能力 3.x 设备上，程序员有能力控制共享内存条带的尺寸。共享内存默认条带尺寸是 32，但是使用带有参数 `cudaSharedMemBankSizeEightByte` 的函数 `cudaDeviceSetSharedMemConfig()` 就可以设定为 64. 当处理双精度数据时，这样做可以帮助避免共享内存条带冲突。函数 `cudaDeviceGetSharedMemConfig(config)` 在 `config` 中返回共享内存条带的当前尺寸。

3.4 内存优化例子：矩阵转置

本节带来一个例子，用它演示本章讲述的各种内存优化技术和前一章讲述的性能测量。这里将要优化的代码是一个单精度值矩阵的转置，转置是异位操作，即输入矩阵和输出矩阵指向相互分离的内存位置。为简化呈现，仅考虑一侧维度为 32 整数倍的方阵。

适用于所有转置情形的主机代码在附录 D.2 给出。主机代码执行典型任务：空间分配和主机与设备间的数据传输，若干内核的启动与计时，以及内核结果的验证，主机内存和设备内存的撤销。

除了实施若干不同的矩阵转置之外，运行一个内核来实施矩阵复制。矩阵复制的性能用作矩阵转置能达到的性能标杆。对矩阵复制和矩阵转置，相对性能的评价标准是以 GB/s 为单位的有效带宽，用矩阵尺寸(单位 GB)的两倍来计算出，一次读矩阵和一次存矩阵，再除以执行时间(单位为秒)。将每个例程调用 `NUM_REP` 次，然后将对应的有效带宽求平均。

这次研究的所有内核均启动若干维度为 32 × 8 的线程块，每个线程块转置(或复制)一个的 32 ×32 基片。就这样，参数 `TILE_DIM` 和 `BLOCK_ROWS` 分别设定为 32 和 8. 线程块里的线程数量小于基片里的元素数量对矩阵转置有利，每个线程将转置若干个矩阵元素，本例中是 4 个，计算下标的大部分开销会被这些元素分摊。

应该提及的最后一个预备事项是线程下标如何映射到数组元素上。说明数组元素的时候使用(*x*, *y*)坐标系，以数组左上角为原点。这个坐标系无缝地映射到预定义变量 `threadIdx`、`blockIdx` 和 `blockDim` 的 `x` 和 `y` 分量。Fortran 中，多维变量的第一维下标变化最快，就像预定义变量的 `x` 分量，在这个解释中连续元

素沿 *x* 方向分布。另一种选择需要将预定义变量的 x 分量和 y 分量解释为矩阵的行和列，这样可以高效地转置矩阵。两种解释没有优劣之分；它们会出现相同的性能瓶颈，但会从读全局数据转换到写全局数据。牢记上面的这些约定，看一下第一个内核，矩阵复制：

```
29    attributes(global) subroutine copySharedMem(odata , idata)
30
31      real , intent(out) :: odata(nx,ny)
32      real , intent(in) :: idata(nx,ny)
33
34      real , shared :: tile(TILE_DIM , TILE_DIM)
35      integer :: x, y, j
36
37      x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
38      y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
39
40      do j = 0, TILE_DIM -1, BLOCK_ROWS
41         tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
42      end do
43
44      call syncthreads()
45
46      do j = 0, TILE_DIM -1, BLOCK_ROWS
47         odata(x,y+j) = tile(threadIdx%x, threadIdx%y+j)
48      end do
49    end subroutine copySharedMem
```

这个复制内核用到了共享内存。在复制中使用共享内存没有必要，用在这里是因为它能模拟最优转置内核里用到的数据访问模式。归咎于复制中使用的共享内存，性能会损失一点，如果有损失的话。

在第 41 行上，数据从全局内存复制到共享内存基片，接着在第 47 行上，数据从共享内存基片复制回全局内存。这两个语句必须放在循环内，这是因为一块中的线程数量少于一个基片中的元素数量，比例因子为 TILE_DIM/BLOCK_ROWS. 每个线程负责复制 4 个矩阵元素。还要注意，第 38 行计算矩阵下标 y 时需要用到 TILE_DIM 而不是 BLOCK_ROWS 或 blockIdx%y，然而在第 37 行计算 x 时，TILE_DIM 就可以替换为 blockDim%x. 对第二维循环而不对第一维循环，是因为每幅线程在第 41 行上从全局内存加载 idata 的连续元素，并在第 47 行上将 odata 的连续元素存到全局内存。因此，从 idata 的读取和对 odata 的写入都是合并的。

注意，第 44 行对 synchthreads() 的调用在技术上是不必要的，这是因为共享内存基片中每个元素的读和写都由同一个线程操作。但是这里包含对 synchthreads() 的调用是为了模拟转置内核使用它时的形为。这里列出共享内存复制内核在不同设备上的性能：

	有效带宽(GB/s)				
例程	Tesla C870	Tesla C1060	Tesla C2050	Tesla K10	Tesla K20
copySharedMem	61.2	71.3	101.4	118.8	149.6

必须指出，本节用到的设备都支持开启了 ECC.

用一个非常简单的内核开始转置的讲述：

```
56    attributes(global) &
57         subroutine transposeNaive(odata , idata)
58
59      real , intent(out) :: odata(ny,nx)
60      real , intent(in) :: idata(nx,ny)
61
62      integer :: x, y, j
63
64      x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
65      y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
66
67      do j = 0, TILE_DIM -1, BLOCK_ROWS
68         odata(y+j,x) = idata(x,y+j)
69      end do
70    end subroutine transposeNaive
```

在 transposeNaive 中，从 idata 中的读取是合并的，但是连续线程对 odata 的写入有一个 1024 元素或 4096 字节的跨度。这恰好进入图 3.14 中的渐进线，从而可以预计这个内核的性能将遭受相应的损失。观察到的 transposeNaive 性能证实了这一点：

	有效带宽(GB/s)				
例程	Tesla C870	Tesla C1060	Tesla C2050	Tesla K10	Tesla K20
copySharedMem	61.2	71.3	101.4	118.8	149.6
transposeNaive	3.9	3.2	18.5	6.5	54.6

依赖于不同的架构，内核 transposeNaive 表现得比内核 copySharedMem 糟糕约 3 至 20 倍。

这个不佳性能补救措施是借助共享内存来避免巨大的跨度。图 3.19 是转置中如何使用共享内存的示意图。一个共享内存基片的使用方法与优化 CPU 代码中用到的缓存阻碍技巧相似(参看 Garg and Sharapov, 2002 或 Dowd and Severance, 1998)。图 3.19 对应的代码是：

```
79    attributes(global) &
80         subroutine transposeCoalesced(odata , idata)
81
82      real , intent(out) :: odata(ny,nx)
83      real , intent(in) :: idata(nx,ny)
84      real , shared :: tile(TILE_DIM , TILE_DIM)
85      integer :: x, y, j
86
87      x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
88      y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
89
90      do j = 0, TILE_DIM -1, BLOCK_ROWS
91         tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
92      end do
93
```

```
 94      call syncthreads()
 95
 96      x = (blockIdx%y-1) * TILE_DIM + threadIdx%x
 97      y = (blockIdx%x-1) * TILE_DIM + threadIdx%y
 98
 99      do j = 0, TILE_DIM -1, BLOCK_ROWS
100         odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
101      end do
102    end subroutine transposeCoalesced
```

第 91 行上，一幅线程从 idata 中读取连续数据并按行放入共享内存 tile 中。在第 96、97 行上重新计算数组下标后，将共享内存 tile 上的一列写入 odata 的一段连续地址。因为一个线程将写入的 odata 数据与业已从中 idata 读取数据的位置不同，所以必须有第 94 行上的按块障碍同步 syncthreads().将有效带宽添加到表中：

	有效带宽(GB/s)				
例程	Tesla C870	Tesla C1060	Tesla C2050	Tesla K10	Tesla K20
copySharedMem	61.2	71.3	101.4	118.8	149.6
transposeNaive	3.9	3.2	18.5	6.5	54.6
transposeCoalesced	36.6	23.6	51.6	65.8	90.4

transposeCoalesced 的结果比 transposeNaive 有很大提升，但它的性能仍然与内核 copySharedMem 相差很远。

尽管共享内存已经提高了转置性能，但是内核 transposeCoalesced 中的共享内存使用方式仍然不是最优的。对一个具有 32×32 个元素的共享内存基片，一列数据中的所有元素都来自于同一个共享内存条带，引发共享内存条带冲突的最坏情况：读取一列(C2050、K20)或半列(C870、C1060)数据将分别导致 32 路或 16 路条带冲突。所幸解决这个问题只需简单地填充共享内存数组的第一个下标，做法如内核 transposeNoBankConflict 的第 114 行所示：

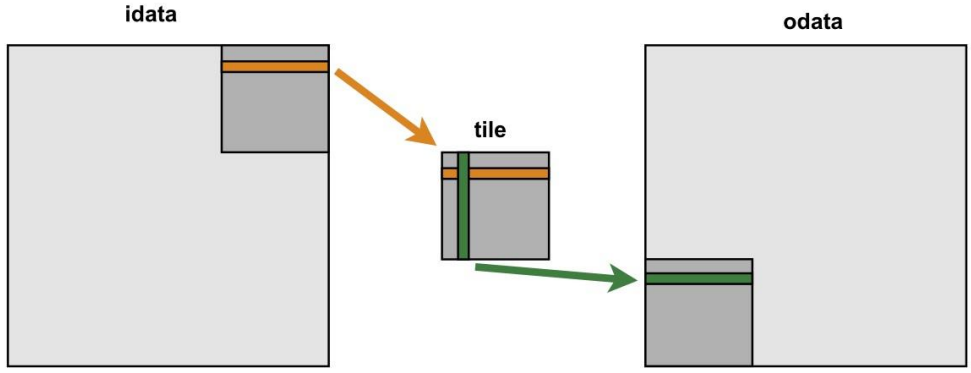


图 3.19

如何使用共享内存基片实现全局内存读、写完全合并的示意图。一幅线程从 idata 行中读取一个片段，将它写入共享内存基片的一行。同一幅线程读取共享内存基片的一列，将它写入 odata 行的一个片段。

```
109    attributes(global) &
110         subroutine transposeNoBankConflicts(odata , idata)
```

```
111
112       real , intent(out) :: odata(ny,nx)
113       real , intent(in) :: idata(nx,ny)
114       real , shared :: tile(TILE_DIM+1, TILE_DIM)
115       integer :: x, y, j
116
117       x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
118       y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
119
120       do j = 0, TILE_DIM -1, BLOCK_ROWS
121          tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
122       end do
123
124       call syncthreads()
125
126       x = (blockIdx%y-1) * TILE_DIM + threadIdx%x
127       y = (blockIdx%x-1) * TILE_DIM + threadIdx%y
128
129       do j = 0, TILE_DIM -1, BLOCK_ROWS
130          odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
131       end do
132    end subroutine transposeNoBankConflicts
```

去除条带冲突就解决了大部分性能问题：

	有效带宽(GB/s)				
例程	Tesla C870	Tesla C1060	Tesla C2050	Tesla K10	Tesla K20
copySharedMem	61.2	71.3	101.4	118.8	149.6
transposeNaive	3.9	3.2	18.5	6.5	54.6
transposeCoalesced	36.6	23.6	51.6	65.8	90.4
transposeNoBankConflict	45.6	23.6	96.1	94.0	137.6

有个例外，在 Tesla C1060 上，转置内核的表现仍然显著低于复制内核。归能差距归咎于分块处理，并与矩阵尺寸有关。对不同尺寸的矩阵，在 C870 上可能会出现类似的性能下降。

3.4.1 区块扎堆 (高级话题)

接下来讲述的区块扎堆(partition camping)通常用于计算能力低于 2.0 的设备，即 C870 和 C1060. 计算能力 2.0 或更高的设备中也会发生区块扎堆，但没那么常见并且也后果也没那么严重。

恰如共享内存被成割为 16 个 32 位宽度的条带那样，全局内存也被分割成 6 个(Tesla C870)或 8 个(Tesla C1060)区块，每个区块宽度为 256 字节。为在这些架构上高效地使用共享内存，半幅之内的线程应当访问不同的条带以使这些访问能够同时进行。如果半幅内的线程仅通过少数几个条带访问共享内存将会发生条带冲突。为有效利用全局内存，所有的活动幅对全局内存的访问应当在区块之间平均分割。当全局内存访问指向一个区块子集时，某些区块处的访问请求必须排队等待，而其它区块却未被使用，使用术语区块扎堆来描述这种景象，它与共享内存条带冲突类似。

合并关注半幅内的全局内存访问，而区块扎堆关注活动半幅间的全局内存访问。因为区块扎堆关注活动线程块在多处理器间的分配行为，线程块在多处理器间如何调度就成为一个重要问题。在计算能力 1.x 设备上启动一个内核时，按线程块在变量 blockIdx 里出现的列优先顺序将线程块指派给多处理器。刚开始以轮询方式指派。一旦达到最大占用率，额外的块只有在需要的时候才指派给多处理器；无法确定内核多快能够完成，也无法确定内核完成的顺序。

如果回到矩阵转置，看一眼那个 1024×1024 矩阵里的块是如何映射到 Tesla C1060 各区块上的，如图 3.20 所示，立即发现区块扎堆是个问题。在 Tesla C1060 上，区块宽度是 256 字节，一个 2048 字节跨度内的所有数据(或 512 个单精度元素)被映射到同一个区块。任意一个列数为 512 整数倍的单精度矩阵，例如前例中的矩阵，它整列的元素都会映射到同一个区块。使用拥有 32×32 个元素(或 128×128 字节)的基片，基片前两列里的所有数据都映射到同一个区块，基片里的其它列对也一样(假设该矩阵已经对齐到区块片段)。

并发块将按行访问 idata 里的基片，这些访问将大致均等地分布在区块上。然而，这些块将按列访问 odata 中的基片，这通常通过仅一两个区块来访问全局内存。

为避免区块扎堆，可像共享内存基片里做的那样填充矩阵。但是用足够多的列来填充以消除区块扎堆的方法可能非常耗内存。一个更高效的选择是从根本上重新解释 blockIdx 分量如何与矩阵关联。

3.4.1.1 对角线重排序

尽管程序员不直接控制块的调度顺序(它由自动内核变量 blockIdx 决定)，但是程序员拥有如何解释 blockIdx 分量的灵活性。给定 blockIdx 分量的命名方式，即 x 和 y，通常假设这些分量关联一个笛卡尔坐标系。然而，不是必须这样，还有其它选择。这样实质上意味着在软件里重新调度各块，这是追求的目标：如何重新调度各块从而使输入矩阵和输出矩阵的操作在区块之间都能平均分布。

为避免从 idata 里读和往 odata 里写的区块扎堆，一个方法是利用对角线来解释 blockIdx 的分量：y 分量代表整个矩阵里的不同对角线上的基片，x 分量表示沿着每条对角线的距离。这样就得一种块的映射关系，如图 3.21 所示。实施这个转换的是：

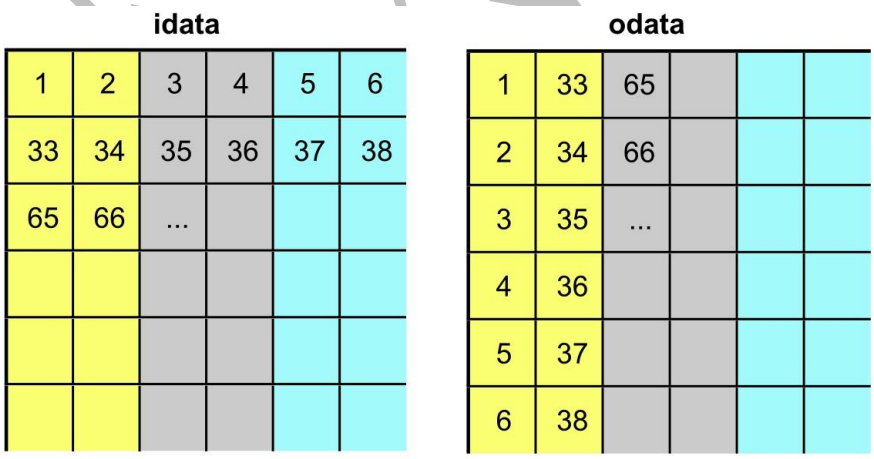


图 3.20

对 idata 和 odata 的左上角，如何将线程块(数字)指派给区块(颜色)的示意图。在 C1060 上，对一个包含 1024×1024 个单精度元素的矩阵，一列中的所有元素都属于同一个区块。从中 idata 读取数值的操作在活动线程块间均匀分布，但是一组 32 个线程块将通过同一个区块写入 odata.

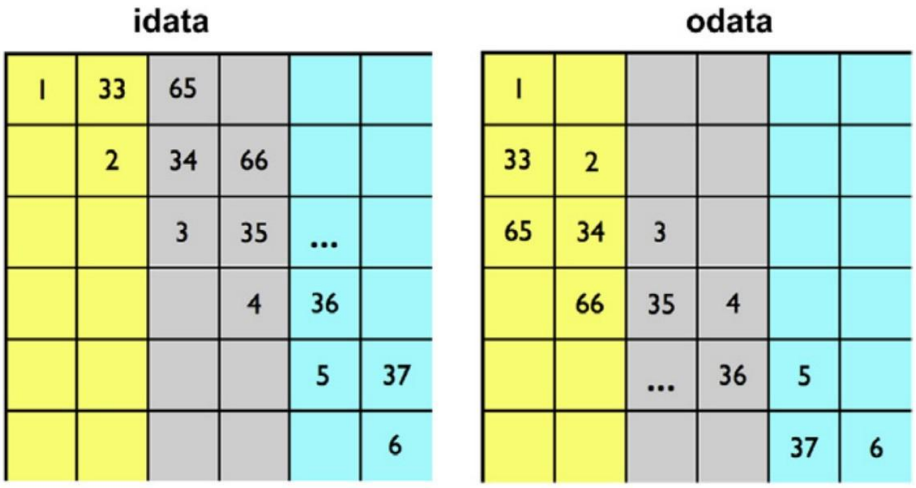


图 3.21

利用 blockIdx 分量的对角线解释，对 idata 和 odata 的左上角，如何将线程块(数字)指派给区块(颜色)的示意图。这里的读和写都均匀分布在各个区块上。

```
142    attributes(global) &
143         subroutine transposeDiagonal(odata , idata)
144
145      real , intent(out) :: odata(ny,nx)
146      real , intent(in) :: idata(nx,ny)
147      real , shared :: tile(TILE_DIM+1, TILE_DIM)
148      integer :: x, y, j
149      integer :: blockIdx_x , blockIdx_y
150
151      if (nx==ny) then
152         blockIdx_y = blockIdx%x
153         blockIdx_x = &
154              mod(blockIdx%x+blockIdx%y-2,gridDim%x)+1
155      else
156         x = blockIdx%x + gridDim%x*(blockIdx%y-1)
157         blockIdx_y = mod(x-1,gridDim%y)+1
158         blockIdx_x = &
159              mod((x-1)/gridDim%y+blockIdx_y -1,gridDim%x)+1
160      endif
161
162      x = (blockIdx_x -1) * TILE_DIM + threadIdx%x
163      y = (blockIdx_y -1) * TILE_DIM + threadIdx%y
164
165      do j = 0, TILE_DIM -1, BLOCK_ROWS
166         tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
167      end do
168
169      call syncthreads()
170
```

```
171       x = (blockIdx_y -1) * TILE_DIM + threadIdx%x
172       y = (blockIdx_x -1) * TILE_DIM + threadIdx%y
173
174       do j = 0, TILE_DIM -1, BLOCK_ROWS
175          odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
176       end do
177    end subroutine transposeDiagonal
```

对本例中的方阵，第 152 和 153 行指定从笛卡尔坐标到对角线坐标的映射关系。这个映射完成后，代码就与 transposeNoBankConflicts 相同 ,除了一个例外 :所有的 blockIdx.x 都替换成了 blockIdx_x，y 分量的情况类似。现在向带宽表格里添加最后一行：

	有效带宽(GB/s)				
例程	Tesla C870	Tesla C1060	Tesla C2050	Tesla K10	Tesla K20
copySharedMem	61.2	71.3	101.4	118.8	149.6
transposeNaive	3.9	3.2	18.5	6.5	54.6
transposeCoalesced	36.6	23.6	51.6	65.8	90.4
transposeNoBankConflict	45.6	23.6	96.1	94.0	137.6
transposeDiagonal	44.2	64.3	90.3	110.4	128.7

内核 transposeDiagonal 将 Tesla C1060 上的转置性能提高到与内核 copySharedMem 性能接近的水平。注意，在其它大多数设备上，重排序对性能无帮助。大多数情况下，下标带来的新增计算量实际上会损害性能。利用对角线重排序，Tesla K10 确实显示有稍微提升，将性能从 copySharedMem 的 80%提高到 93%. 对角线重排序对 Tesla C1060 来说却重要得多，将性能从 copySharedMem 的 33%提高到 90%.

关于区块扎堆有几点需要牢记。在计算能力低于 2.0 的卡上，区块扎堆与问题尺寸相关。如果矩阵的每个边都是 386 个 32 位元素的整数倍，那么将在 C870 上看到区块扎堆，但 C1060 上看不到。在计算能力 2.0 及更高卡上，区块扎堆很不常见，通常也不严重，这是因为块已经以散列方式指派到多处理器上。

3.5 执行配置

既使一个内核已经优化过，所有的全局内存访问均已完美合并，仍然有一个问题需要处理，即这些内存访问具有几百个周期的延时。为得到良好的整体性能，必须保证一个多处理器上有足够的并行度，从而能尽可能多地隐藏内存访问停顿。有两种方法可以获得这种并行效果：调整一个多处理器上的并行线程数量和调整每个线程提交的独立操作数量。第一种方法称为*线程级并行*，第二种称为*指令级并行*。

3.5.1 线程级并行

借助主机用来启动内核的执行配置，可以在一定程度上控制线程级并行。执行配置中指定内核启动时每块中的线程数量和块的数量。对一个给定的内核，可以驻留在一个多处理器上的线程块数量就成为一个重要关注点，它受多种因素制约，图 3.22 给出了不同代次 Tesla 卡上的一些制约因素。这些特性的更详细表格参见附录 A. 不管线程块尺寸或者资源的使用情况怎么样，每个多处理器上的线程块数量都有一个上限。计算能力 1.x 和 2.x 设备上的上限是 8 个线程块，3.x 设备上是 16 个线程块。每块内的线程数量、每个多处理器上的线程数量、寄存器尺寸、可用共享内存都有上限，这些因素都会限制并行线程的数量。

	Tesla C870	Tesla C1060	Tesla C2050	Tesla K20
计算能力	1.0	1.3	2.0	3.5
最大线程/线程块	512	512	2014	1024
最大线程块/多处理器	8	8	8	16
最大幅/多处理器	24	32	48	64
线程/幅	32	32	32	32
最大线程/多处理器	768	2014	1536	2048
32 位寄存器/多处理器	8K	16K	32K	64K

图 3.22

多种 CUDA 架构下的线程块和多处理器上限。

度量标准占用率用来帮助评估内核在一个多处理器上的线程级并行。占用率是每个多处理器上活动幅的数量与活动幅的最大可能数量的比例。用幅来定义是因为幅是同时执行的线程单元，但也可以用线程概念来思考这个度量标准。高占用率不会必然带来高性能，因为可以在内核代码里放置大量的指令级并行。但如果依靠线程级并行来隐藏延时，那么占用率不应该非常小。利用命令行测绘器可以探明所有内核的占用率，占用率是命令行测绘器的一个默认选项。

为演示选取的各种运行配置会如何影响性能，可以利用附录 D.3 中列出的简单复制代码。这个代码里的内核相对简单，例如这探讨的第一个内核是：

```
12    attributes(global) subroutine copy(odata , idata)
13      use precision_m
14      implicit none
15      real(fp_kind) :: odata(*), idata(*), tmp
16      integer :: i
17
18      i = (blockIdx%x-1)* blockDim%x + threadIdx%x
19      tmp = idata(i)
20      odata(i) = tmp
21    end subroutine copy
```

利用双精度数据瞄准 Tesla K20 使用编译器选项-Mcuda=cc35，观察到下列结果：

线程块	占用率	有效带宽(GB/s)
32	0.25	96
64	0.5	125
128	1.0	136
256	1.0	137
512	1.0	137
1024	1.0	133

这个表中，线程块尺寸和有效带宽都从代码的输出获得，占用率从命令行测绘器产生的文件里获得。按照一贯做法，线程块尺寸是一幅线程的整数倍。如果用每块 33 个线程来启动内核，每块将处理两个完整

的幅，每个线程都计算出结果，但第二幅中只有一个线程的结果被显示出来。

因为 Tesla K20 的每个多处理器最多有 2048 个线程和 16 个线程块，所以有尺寸为 32 或 64 的线程块启动内核无法达到完全占用率。从而每块 32 个线程的有效带宽蒙受损失，但是每个线程块 64 线程时，即使占用率只有一半，内核执行的带宽仍然接近观测到的最大值；获得良好性能不必是满占用率。

通常，每块中更多线程不意味着更高占用率。如果看一眼 C2050 上的结果，还是用双精度数据，就能得到：

线程块	占用率	有效带宽 (GB/s)
32	0.167	55
64	0.333	82
128	0.667	103
256	1.0	102
512	1.0	103
1024	0.667	98

Tesla C2050 的每个多处理器能容纳最多 1536 个线程和 8 个线程块，因此，采用一个包含 1024 线程的线程块，一次只能允许一个线程块驻留在一个多处理器上，带来三分之二的占用率。再一次，更高的占用率不意味着更好的性能 ,因为 128 个线程的线程块带来三分之二的占用率却能达到所有运行里最高的带宽。

3.5.1.1 共享内存

在几个情形中共享内存会有帮助，例如帮助合并全局内存访问和消除全局内存的冗余访问。然而，它也可能是占用率的一个约束因素。示例代码内核中没有使用共享内存；然而，通过执行配置的第三个参数改变动态分配共享内存的数量 ,就能探测性能对占用率的敏感程度。简单地增加这个参数的值(不改动内核)，有有可能有效地减小内核占用率并测量它对性能的影响。例如，如果用如下语句启动内核：

```
122       call copy <<<grid , threadBlock , 0.9*smBytes >>>(b_d , a_d)
```

这里的 `smBytes` 是每个多处理器上共享内存的字节尺寸，接着迫使每个多处理器上只有一个并发线程块。在 Tesla K20 上，这个做法产生如下结果，添加到前面的表中：

	无共享内存		共享内存	
线程块	占用率	带宽(GB/s)	占用率	带宽(GB/s)
32	0.25	96	0.016	8
64	0.5	125	0.031	15
128	1.0	136	0.063	29
256	1.0	137	0.125	53
512	1.0	137	0.25	91
1024	1.0	133	0.5	123

*无共享内存*下面各列的结果来自前表中 K20 的结果。*共享内存*下面各列的占用率意味着，当使用共享内存时，任一时刻只有一个线程块驻留在一个多处理器上，带宽数字表明性能正如预计的那样下降。这个练习引发几个问题：在更复杂的内核里，寄存器或共享内存限制了占用率，此时该做什么？必须在这种情况下提升低下的性能吗？答案是不，如果使用指令级并行的话。

3.5.2 指令级并行

在本书中已经看过到一个指令级并行的例子。3.4 节的矩阵转置例子中，大数多内核都用到了一个 32×32 的共享内存基片。但是因为在特定设备上，每块的最大线程数量是 512，不可能用每块 32×32 个线程来启动内核。做为替代方案，不得不采用较少线程的线程块，且让每个线程处理多个元素。在那个转置例子中，启动包含 32×8 个线程的线程块，每个线程处理 4 个元素。

对本节的例子，为利用指令级并行的优势，修改内核 copy 如下：

```
27    attributes(global) subroutine copy_ILP(odata , idata)
28      use precision_m
29      implicit none
30      real(fp_kind) :: odata(*), idata(*), tmp(ILP)
31      integer :: i,j
32
33      i = (blockIdx%x-1)* blockDim%x*ILP + threadIdx%x
34
35      do j = 1, ILP
36         tmp(j) = idata(i+(j-1)* blockDim%x)
37      enddo
38
39      do j = 1, ILP
40         odata(i+(j-1)* blockDim%x) = tmp(j)
41      enddo
42    end subroutine copy_ILP
```

这里的 ILP 设定为 4. 这个内核里，每个线程复制 ILP 个数组元素，因此一个拥用 blockDim%x 个线程的线程块将复制 ILP*blockDim%x 个元素。除了让每个线程复制多个元素，第 35-37 行的循环利用线程私有数组 tmp(ILP)来批量处理所有的加载，这里的 tmp(ILP)驻留寄存器内存。此处使用批量加载的原因是，在 CUDA 中，一个加载命令不会阻塞后来的独立执行，但第一次使用加载命令所请求的数据将会阻塞，直至加载完成。一个加载的递交时刻与使用所请求数据的时刻之间的时间长度或者指令条数用术语取用分离度用来描述。取用分离度越大，加载延时隐藏得越好。通过第 35-37 行循环那样的批量加载，可以让每个线程的 ILP 次加载请求先跑着。利用循环中递交的其它 ILP-1 次加载来提高第一次加载的取用分离度。

如果再一次利用动态分配的共享内存来将占用率限制为每个多处理器一个块，那么可以在表中追加 ILP=4 时的结果：

	无共享内存		共享内存		
线程块	占用率	带宽(GB/s)	占用率	无 ILP 带宽(GB/s)	ILP=4 带宽(GB/s)
32	0.25	96	0.016	8	26
64	0.5	125	0.031	15	50
128	1.0	136	0.063	29	90
256	1.0	137	0.125	53	125
512	1.0	137	0.25	91	140

1024	1.0	133	0.5	123	139

这里看到低占率情形的性能有巨大提升，比没有采用指令级并行的内核提高近三倍。指令级并行的使用从根本上提高了实际占用率，倍数因子等于 ILP. 例如，未采用指令级并行时，一个拥有 512 线程的块(占用率 1/4)获得 91 GB/s 的带宽，未用共享内存时一个拥有 32 线程的块(占用率也是 1/4)获得 96 GB/s 的带宽，而一个具有 128 线程的块(占用率 1/16)在 ILP=4 时就获得 90 GB/s 的带宽，与前面两个接近。

即使占用率不是一个问题，用单个线程处理共享内存数组的多个元素的做法也是有益的。这是因为每个元素都需要做的一些公共操作可以由线程只执行一次，被处理的多个共享内存元素分摊该线程的开销。可以观察到，在四分之一占用率和半占用率上，ILP=4 的结果超越了满占用率的结果，满占用率时未用共享内存去约束每个多处理器上的线程块数量。

指令级并行的一个缺点是像 tmp(ILP)这样的线程私有数组会消耗寄存器，可能进一步增加寄存器压力。从而，采用多少指令级并行需要平衡协调，需要做一些实验来得到最优结果。

3.6 指令优化

至此，已经从数据移动方面讲解了优化手段，包括主机与设备之间的移动、设备内部的移动。前面还讲了通过线程级并行(执行配置和占用率)或指令级并行来确保能暴露出足够的并行度从而使用设备保持忙碌。当一份代码不受内存约束且在设备上有足够的并行度时，为了提高性能就需要关注内核的指令吞吐量。

表 3.2 每个多处理器每时钟期的原生算术操作吞吐量。对一幅 32 个线程，一个指令对应 32 个操作，因此指令吞吐量是操作吞吐量的 1/32. “多指令”条目意为这个操作会被转换成多条指令。

	计算能力				
操作	**1.0**	**1.3**	**2.0**	**3.0**	**3.5**
32 位 iand()、ieor()、ior()	8	8	32	160	160
32 位 ishft()	8	8	16	32	64
32 位整数加、比较	10	10	32	160	160
32 位整数乘、乘加	多指令	多指令	16	32	32
32 位浮点加、乘、乘加	8	8	32	192	192
32 位浮点倒数、平方根倒数	2	2	4	32	32
64 位浮点加、乘、乘加	--	1	16*	8	64
*吞吐量低于 Geforce GPU.					

表 3.2 列出了多种原生指令在不同计算能力设备上的算术吞吐量。(该表的更完整版本可以在 *CUDA C Programming Guide* 中找到。) 不仅是类型转换，其它指令也会映射成多条原生指令，但特定设备内置例程例外。

3.6.1 设备内置例程

通过在设备代码中使用模块 cudadevice ,CUDA Fortran 允许访问许多内置的设备函数。CUDA Fortran 可用的内置函数的完整列表包含在 *CUDA Fortran Programming and Reference Guide* 中。这里简要讲述这些函数中的几类。

3.6.1.1 有向舍入

不必设定舍入模式，通过附加指令就能在 CUDA 使用有向舍入。后缀_ru、_rd、_rn 和_rz 意为向上舍入、向下舍入、向最近偶数舍入和向零舍入。例如，采用__fadd_[rn,rz,ru,rd]和__dadd_[rn,rz,ru,rd]，32 位和 64 位浮点加函数就有多种舍入模式可用。

3.6.1.2 C 内置例程

通过模块 cudadevice 可以使用一些 Fortran 中没有的 C 内置例程。特别地，sincos(x, s, c)同时计算参数 x 的正弦和余弦。相对于分别调用正弦和余弦，这个函数的将吞吐率几乎翻倍，且没有任何精度损失。

3.6.1.3 快速数学内置例程

对 32 位浮点数据，CUDA 拥用一批快速但低精度的内置例程，启用方式为每个编译单元都加上编译器选项-Mcuda=fastmath，或有选择地使用模块 cudadevice 且显式地调用__fdividef(x,y)、__sinf(x)、__cosf(x)、__tanf(x)、__sincosf(x,s,c)、__logf(x)、__log2f(x)、__log10f(x)、__expf(x)、__exp10f(x)和__powf(x,y).

3.6.2 编译器选项

前面已经提过编译器选项-Mcuda=fastmath ,它用来为 32 位浮点数调用更快但精度稍低的内置例程。还有一些其它的编译器选项会影响指令吞吐率。

选项-Mcuda=nofma 触发采用融合乘加指令。如果编译这个简单的代码：

```
1  module mfa_m
2  contains
3    attributes(global) subroutine k(a, b, c)
4      implicit none
5      real :: a, b, c
6      c = a*b+c
7    end subroutine k
8  end module mfa_m
```

并用选项-Mcuda=keepptx 保存产生的 PTX 代码，为计算能力 1.x 产生的 PTX 代码包含内置指令：

```
        mad.f32 %f4, %f2, %f3, %f1;
```

然而为计算能力 2.0 及更高产生的 PTX 代码包含的指令却是：

```
        fma.rn.f32 %f4, %f2, %f3, %f1;
```

MAD 和 FMA 都将乘法操作和加法操作组合为一条指令，但它们的做法却大不相同。在用到加法中之前，MAD 指令会截断乘积的尾数，而 FMA 指令是一条遵循 IEEE-754(2008)标准的融合乘加指令，加法使用是全宽度乘积，接着是一个简单的舍入步骤。与此相反的情形是指定-Mcuda=nofma，对所有的目标计算能力，PTX 代码都包含两条指令：

```
        mul.rn.f32 %f4, %f2, %f3;
        add.f32 %f5, %f1, %f4;
```

MAD 或 FMA 指令将会比分离的 MUL 和 ADD 指令执行得更快，因为这些操作有专用硬件。MUL 舍入到最近的偶数，而 MAD 截断中间结果，因此在计算能力 1.x 设备上，分离的乘法和加法通常比 MAD 更精确。计算能力 2.0 或更高设备上，在 FMA 中，乘积在加法之前没有任何截断或舍入，这意味着 FMA 产出的结果通常将比分离的 MUL 和 ADD 指令的结果更精确。

选项-Mcuda=[no]flushz 控制着对单精度非规范数的支持。用-Mcuda=flushz 编译的代码将非规范数冲刷为零，且比用-Mcuda=noflushz 支持非规范数的代码执行得更快。编译器选项-Mcuda=fastmath 暗含-Mcuda=flushz.计算能力 1.x 设备不支持非规范数，且暗含 Mcuda=flushz.计算能力 2.0 及更高设备支持非规范数，并默认使用。

3.6.3 发散幅

另一个指令优化手段是最小化发散幅的数量。考虑下列设备代码片段：

```
1       i = blockDim%x*(blockIdx%x-1) + threadIdx%x
2       if (mod(i,2) == 0) then
3         x(i) = 0
4       else
5         x(i) = 1
6       endif
```

如果下标 i 是偶数或奇数，代码将 x(i)设定为 0 或 1. 一幅线程并排执行，如果幅中有任何线程满足条件语句的一个分支，那么幅中所有线程都必须执行这个分支。多个执行路径会被串行化，每个线程的指令计数也相应增长。不满足分支条件的线程的结果被高效遮挡。但对性能的影响是，只要是幅中有一个线程满足的每个分支，幅中所有线程都要执行。在本例中，每个分支幅中都有一半线程满足，因此所有的线程都要执行这两个分支。这个简单例子的性能损失不算严重，但是如果一个 if 或 case 构件有很多分支，或者嵌套多层这样流程控制语句，那么幅发散就可能成为一个大问题。另一方面，如果一个条件对一幅线程计算出来都是一样的，那么最多只有一个分支会被执行，下面的例子就是这样：

```
1       i = blockDim%x*(blockIdx%x-1) + threadIdx%x
2       if (mod((i-1)/warpsize ,2) == 0) then
3         x(i) = 0
4       else
5         x(i) = 1
6       endif
```

如果 x(i)属于偶数或奇数幅，代码将它分为设定为 0 或 1. 因此每个线程只执行一个分支。

3.7 内核循环导语(directive)

尽管不是一个严格意义上的性能优化技术，但是内核循环导语或 CUF 内核可以用来简化设备上特定操作的编程。这些导语指导编译器从包含紧密嵌套循环的主机代码区域产生内核。本质上，内核循环导语允许程序员在主机代码里内联内核。

本书中广泛使用过一个数据自增例子。自增例子的 CUF 内核版本是：

```
1  program incrementTest
 2    implicit none
 3    integer , parameter :: n = 1024*1024
 4    integer :: a(n), i, b
 5    integer , device :: a_d(n)
 6    integer , parameter :: tPB = 256
 7
 8    a = 1
```

```
 9    b = 3
10
11    a_d = a
12
13    !$cuf kernel do <<<*,tPB >>>
14    do i = 1, n
15       a_d(i) = a_d(i) + b
16    enddo
17
18    a = a_d
19
20    if (any(a /= 4)) then
21       write(*,*) '**** Program Failed ****'
22    else
23       write(*,*) 'Program Passed'
24    endif
25  end program incrementTest
```

这个代码里没有包含内核。取代显式设备例程的是第 13 行上的导语，它指导编译器从第 14-17 行上的 do 循环里自动生成一个内核。第 13 行提供了一个执行配置 指明启动这个内核时使用包含 tPB 个线程的线程块。第一个执行配置参数指定为*号，让编译器自主计算将要启动的线程块数量，以便执行循环中的操作。执行配置也可以指定为<<<*,*>>>，此时编译器会选择将要启动的线程块尺寸和线程块数量。当在配置文件里指定 gridsize 和 threadblocksize 时，可以从命令行测绘器的输出中查看一个 CUF 内核启动的执行配置参数：

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff68da82e00f0
method ,gputime ,cputime ,gridsizeX ,gridsizeY ,threadblocksizeX ,
  threadblocksizeY ,threadblocksizeZ ,occupancy
method=[ memcpyHtoD ] gputime=[ 1250.176 ] cputime=[ 1593.000 ]
method=[ incrementtest_14_gpu ]
  gputime=[ 67.264 ] cputime=[ 26.000 ]
  gridsize=[ 4096, 1 ] threadblocksize=[ 256, 1, 1 ]
  occupancy=[ 1.000 ]
method=[ memcpyDtoH ] gputime=[ 2111.168 ] cputime=[ 3198.000 ]
```

这里使用执行配置参数<<<4096,256>>>用来启动自动生成的内核 incrementtest_14_gpu.

采用 CUF 内核的代码里实行显式管理数据。设备上的数组用 device 属性声明，像第 5 行那样，主机到设备和设备到主机的数据传输分别在第 11 和 18 行显式实施。标量变量 b 是一个主机变量，它被按值传给生成的内核。

利用 CUF 内核，自增例子的二维版本是：

```
 1  program incrementTest
 2    implicit none
```

```
 3    integer , parameter :: n = 4*1024
 4    integer :: a(n,n), i, j, b
 5    integer , device :: a_d(n,n)
 6
 7    a = 1
 8    b = 3
 9
10    a_d = a
11
12    !$cuf kernel do (2) <<< (*,*), (32,8) >>>
13    do j = 1, n
14       do i = 1, n
15          a_d(i,j) = a_d(i,j) + b
16       enddo
17    enddo
18
19    a = a_d
20
21    if (any(a /= 4)) then
22       write(*,*) '**** Program Failed ****'
23    else
24       write(*,*) 'Program Passed'
25    endif
26  end program incrementTest
```

这种情况下，导语上指定的 do (2)意味着生成的内核将映射到接下来的两个循环。导语中指定的多维线程块和多维网格以从最内层到最外层的方式映射到嵌套循环。例如，对 32×8 的线程块，预定义的内核变量 threadIdx%x 将从 1 跑到 32 并映射到下标 i，threadIdx%y 将从 1 跑到 8 并映射到下标 j. 除了指定线程块尺寸，还能使用<<<(*,*),(*,*)>>>甚至 <<<*,*>>>并让编译器自已选取线程块尺寸和网络尺寸。归功于具体要求 do (2)，为执行配置使用单个星号仍然会将两层循环都映射到内核。如果没有在 do 后面指定(2)，那么只有外层循环会被映射到 threadIdx%x，生成的内核也将只包含对 i 的循环。

3.7.1 CUF 内核中的归约

CUF 内核非常有用的一个领域就是实施归约。CUDA 中的高效归约写起来不简单，就像在蒙特卡洛案例学习中看到的那样。线程块内部和跨越线程块的数据都需要归约。CUF 内核可以自动完成这项工作，如下列代码所示：

```
1  program reduce
 2    implicit none
 3    integer , parameter :: n = 1024*1024
 4    integer :: i, aSum = 0
 5    integer , device :: a_d(n)
 6    integer , parameter :: tPB = 256
 7
 8    a_d = 1
```

```
 9
10    !$cuf kernel do <<<*,tPB >>>
11    do i = 1, n
12       aSum = aSum + a_d(i)
13    enddo
14
15    if (aSum /= n) then
16       write(*,*) '**** Program Failed ****'
17    else
18       write(*,*) 'Program Passed'
19    endif
20  end program reduce
```

这个代码里变量 aSum 是一个声明在主机上的标量变量。像这样，编译器懂得在设备上实施一个归约并将结果入主机变量。这个特殊例子实施一个求和归约，但也能够实施其它类型的归约。

3.7.2 CUF 内核中的流

目前为止，导语只为 CUF 内核使用了前两执行参数。通过指定一个做为可选参数的流标识，程序员就能指定 CUF 内核启动时所在的流。这个目标有两个实现方法。第一种是第四个执行配置参数，例如：

```
!$cuf kernel do <<< *,*,0,streamID >>>
```

这里的 0 是第三个参数。利用关键字 stream 也可以将流编号指定为第三个参数。

```
!$cuf kernel do <<< *,*,stream=streamID >>>
```

3.7.3 CUF 内核中的指令级并行

在 3.5.2 节中已经看到如何利用指令级并行隐藏延时。本质上是让每个线程处理数组的多个元素。通过显式指定不足以覆盖数组元素的线程块参数和网络参数，CUF 内核也能达到同样的效果。编译器将产生一个每个线程都处理多个元素的内核。例如，如果返回第一个 CUF 内核代码，指定块尺寸之外，在导语上额外显式指定网络尺寸，得到：

```
 1  program ilp
 2    implicit none
 3    integer , parameter :: n = 1024*1024
 4    integer :: a(n), i, b
 5    integer , device :: a_d(n)
 6    integer , parameter :: tPB = 256
 7
 8    a = 1
 9    b = 3
10
11    a_d = a
12
13    !$cuf kernel do <<<1024,tPB >>>
14    do i = 1, n
15       a_d(i) = a_d(i) + b
16    enddo
```

```
17
18    a = a_d
19
20    if (any(a /= 4)) then
21       write(*,*) '**** Program Failed ****'
22    else
23       write(*,*) 'Program Passed'
24    endif
25  end program ilp
```

如果每个线程处理单个元素，那么拥有 256 个线程的 1024 个线程块无法处理 1024^2 个元素，因此编译器生成一个循环，导致每个线程处理 4 个元素。

第4章 多 GPU 编程

根据启动的主机线程数量、资源是否分配到集群的多个节点上，一个应用使用多块 GPU 有许多配置方案。CUDA 兼容任何主机线程模型，诸如 OpenMP 和 MPI，每个主机线程可以访问一个或多个 GPU. 本章考察两个常见场景 :单个线程使用多块 GPU 和每个 MPI 进程各用一块 GPU。接下来各节论述这两个多 GPU 用法。

4.1 CUDA 的多 GPU 特性

CUDA 4.0 工具包引入一个高度简化的多 GPU 编程模型。在这版本发布之前，从单个主机线程管理多块 GPU 必须使用驱动程序接口的压入和弹出上下文函数。从 CUDA 4.0 起 ,程序员不必显式地处理上下文，因为可以用 `cudaSetDevice()` 方便地切换到另一个设备。所有的 CUDA 调用都被提交给当前 GPU，`cudaSetDevice()` 用来设定当前 GPU. 下面的代码是它的用法的一个简单示例 ,在不同的设备上为数组赋值。

```
1  module kernel
2  contains
3    attributes(global) subroutine assign(a, v)
4      implicit none
5      real :: a(*)
6      real , value :: v
7      a(threadIdx%x) = v
8    end subroutine assign
9  end module kernel
10
11 program minimal
12   use cudafor
13   use kernel
14   implicit none
15   integer , parameter :: n=32
16   real :: a(n)
17   real , device , allocatable :: a0_d(:), a1_d(:)
18   integer :: nDevices , istat
19
20   istat = cudaGetDeviceCount(nDevices)
21   if (nDevices < 2) then
22      write(*,*) 'This program requires at least two GPUs'
23      stop
24   end if
25
26   istat = cudaSetDevice(0)
27   allocate(a0_d(n))
28   call assign <<<1,n>>>(a0_d , 3.0)
```

```
29     a = a0_d
30     deallocate(a0_d)
31     write(*,*) 'Device 0: ', a(1)
32
33     istat = cudaSetDevice(1)
34     allocate(a1_d(n))
35     call assign <<<1,n>>>(a1_d , 4.0)
36     a = a1_d
37     deallocate(a1_d)
38     write(*,*) 'Device 1: ', a(1)
39  end program minimal
```

第 3-8 行用来赋值的内核代码与为单 GPU 赋值的内核代码没有差别；单 GPU 和多 GPU 代码的所有差别都存在于主机代码。第 17 行的设备数组声明采用了 `allocatable` 变量属性。没有用 `allocatable` 属性声明的设备数组将隐式地分配在默认设备上（设备 0）。为声明打算驻留到其它设备上的数组，必须在将合适的设备设定为当前设备之后才能分配空间；因此必须用变量属性 `allocatable`. 第 20-22 行保证系统上至少有两个 CUDA 设备，如果没有，就终止程序。

第 26 行将当前设备设定为设备 0. 这个操作不必要，因为默认设备就是 0，写上这一句是为清楚明了。这份代码中，空间分配、内核启动、设备至主机传输、第 27-30 的撤销设备数据都要求将当前设备设定为数组 `a0_d` 所驻留的那个设备。第 34-37 行执行类似的操作，只是在设备 1 上分配设备数组。

编译这份代码必须使用 CUDA 4.0 或更新的库，近期的编译器版本都满足这个要求，因此编译和执行这份代码就如此简单：

```
% pgf90 minimal.cuf -o minimal
% ./minimal
Device 0: 3.000000
Device 1: 4.000000
```

4.1.1 端到端通信

到目前为止论述的多 GPU 编程，各 GPU 独立操作本地数据。如果一块 GPU 用到另一块 GPU 上的数据，程序员只能借助主机的两次传输，一次从数据驻留的 GPU 为起点的设备至主机传输，接着一个以 GPU 为终点的主机至设备传输。

在特点条件下，CUDA 允许端到端访问，这样的传输不用借助 CPU 转运。用开启端到端访问的两个设备，在 GPU 间传输数据就像主机与设备间传输数据一样简单：

```
a1_d = a0_d
```

这样做不仅仅使编码更容易，更有重要的性能收益，因为这样跨越 PCIe 总线的*直接传输*没有来自主机的干扰（传输初始化之外），图 4.1 的左边描绘了这个过程。除*直接传输*之外，一个 GPU 上运行的内核能够访问另一个 GPU 上的数据，这个特性称为*直接访问*。CUDA 4.0 引入的一个称为*统一虚拟地址*的特性使这一切成为可能，*统一虚拟地址*简称为*统一虚址*。统一虚址中，主机内存和所有 GPU 的内存组合成单个虚拟地址空间，每个设备的内存占据这个虚拟空间的一段连续地址。对一个给定的变量，运行时能根据它的虚拟地址确定数据驻留在哪里。

4.1.1.1 端到端通信的必要条件

使用端到端特性必须满足几个前提条件。除了要使用 CUDA 工具包 4.0 或更高版本，生成代码的目标还必须是计算能力 2.0 或更高。还有 ,必须是 64 位操作系统 ,执行端到端传输的 GPU 对必须是同一代产品，且位于同一个 IO 集线器芯片组上。最后一条不像其它条件那么容易验证，但可以用端到端编程接口来探测哪些 GPU 能够相互到端访问，具体做法如下：

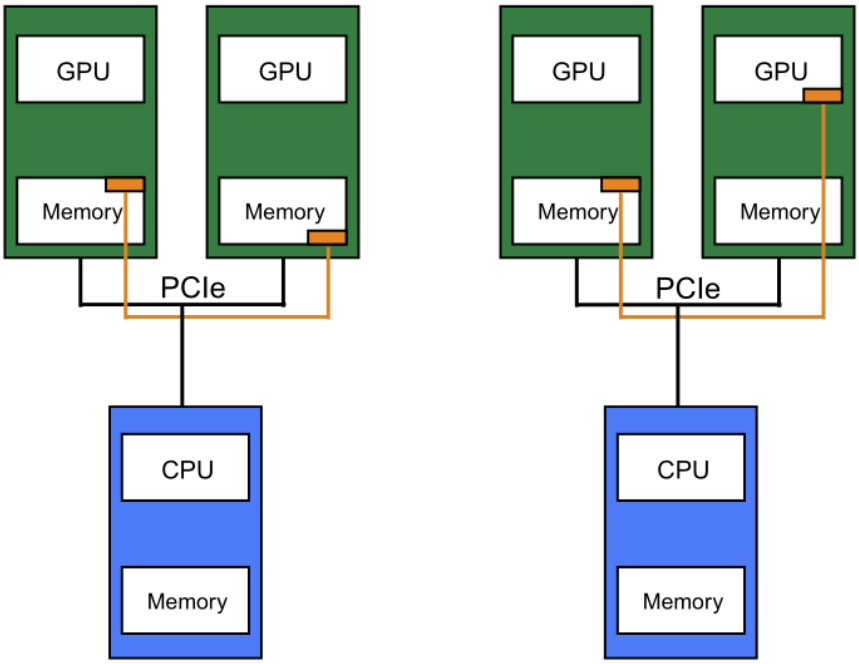


图 4.1

借助端到端通信进行的直接传输(左)和直接访问(右)

```
1  program checkP2pAccess
2    use cudafor
3    implicit none
4    integer , allocatable :: p2pOK(:,:)
5    integer :: nDevices , i, j, istat
6    type (cudaDeviceProp) :: prop
7
8    istat = cudaGetDeviceCount(nDevices)
9    write(*,"('Number of CUDA -capable devices: ', i0 ,/)") &
10        nDevices
11
12   do i = 0, nDevices -1
13      istat = cudaGetDeviceProperties(prop , i)
14      write(*,"('Device ', i0, ': ', a)") i, trim(prop%name)
15   enddo
16   write(*,*)
17
18   allocate(p2pOK(0:nDevices -1, 0:nDevices -1))
```

```
19    p2pOK = 0
20
21    do j = 0, nDevices -1
22       do i = j+1, nDevices -1
23          istat = cudaDeviceCanAccessPeer(p2pOK(i,j), i, j)
24          p2pOK(j,i) = p2pOK(i,j)
25       end do
26    end do
27
28    do i = 0, nDevices -1
29       write(*,"(3x,i3)", advance='no') i
30    enddo
31    write(*,*)
32
33    do j = 0, nDevices -1
34       write(*,"(i3)", advance='no') j
35       do i = 0, nDevices -1
36          if (i == j) then
37             write(*,"(2x,'-',3x)", advance='no')
38          else if (p2pOK(i,j) == 1) then
39             write(*,"(2x, 'Y',3x)",advance='no')
40          else
41             write(*,"(6x)",advance='no')
42          end if
43       end do
44       write(*,*)
45    end do
46 end program checkP2pAccess
```

这份代码中，第 12-15 行的循环列出所有的 CUDA 设备之后，代码执行第 21-26 行的二重循环，评定所有的 GPU 是否能够访问其它 GPU 的内存。

```
21    do j = 0, nDevices -1
22       do i = j+1, nDevices -1
23          istat = cudaDeviceCanAccessPeer(p2pOK(i,j), i, j)
24          p2pOK(j,i) = p2pOK(i,j)
25       end do
26    end do
```

第 23 行的函数 `cudaDeviceCanAccessPeer()`判定设备 `i` 是否能够访问设备 `j` 的内存并根据能或不能将 `p2pOK(i,j)`设定为 1 或 0. 虽然这个函数隐含着传输的方向性，但是任何妨碍到端访问的限制因素都与传输方向无关。本质上， `cudaDeviceCanAccessPeer()`调用可以解释为大致判定用最后两个参数指定的设备之间是否可以到端访问。正因如此，第 22 行的循环判定 `i>j` 时的可访问性，第 24 行将得到的结果应用到 `j>i` 的情况。

剩余的代码打印出一个反映端到端可访问性的矩阵。在带有两个 Tesla S2050 系统的节点上(每个 Tesla S2050 包含四块 GPU)，两个 S2050 连接到同一个 IOH 芯片组，得到：

```
%  pgf90 p2pAccess.cuf -o p2pAccess
%   ./p2pAccess
Number of CUDA -capable devices: 8
Device 0: Tesla S2050
Device 1: Tesla S2050
Device 2: Tesla S2050
Device 3: Tesla S2050
Device 4: Tesla S2050
Device 5: Tesla S2050
Device 6: Tesla S2050
Device 7: Tesla S2050

    0     1     2     3     4     5     6     7
 0  -     Y     Y     Y     Y     Y     Y     Y
 1  Y     -     Y     Y     Y     Y     Y     Y
 2  Y     Y     -     Y     Y     Y     Y     Y
 3  Y     Y     Y     -     Y     Y     Y     Y
 4  Y     Y     Y     Y     -     Y     Y     Y
 5  Y     Y     Y     Y     Y     -     Y     Y
 6  Y     Y     Y     Y     Y     Y     -     Y
 7  Y     Y     Y     Y     Y     Y     Y     -
```

这个结果显示，每个 GPU 都能访问其它 GPU 的内存。要记住一件重要的事情，设备从零开始编号，以便兼容底层的 CUDA C 运行时。

如果一个代码要求在所有 GPU 间到端访问，那么可以利用环境变量 `CUDA_VISIBLE_DEVICES` 枚举哪些设备对 CUDA 程序可用，且能指定设备顺序。例如，继续前面的 shell:

```
% export CUDA_VISIBLE_DEVICES=2,4,1,3
% ./p2pAccess
Number of CUDA -capable devices: 4

Device 0: Tesla S2050
Device 1: Tesla S2050
Device 2: Tesla S2050
Device 3: Tesla S2050

    0     1     2     3
 0  -     Y     Y     Y
 1  Y     -     Y     Y
 2  Y     Y     -     Y
 3  Y     Y     Y     -
```

回想一下，Tesla K10 本质上是单块板子上的两个设备。如果在带有两块 Tesla K10 的系统上运行这个代码，将得到：

```
% pgf90 p2pAccess.cuf -o p2pAccess
% ./p2pAccess
```

```
Number of CUDA -capable devices: 4

Device 0: Tesla K10.G1.8GB
Device 1: Tesla K10.G1.8GB
Device 2: Tesla K10.G2.8GB
Device 3: Tesla K10.G2.8GB

    0     1     2     3
 0  -     Y
 1  Y     -
 2              -     Y
 3              Y     -
```

设备 0 和 1 属于一个 K10，设备 2 和 3 属于另一个。尽管一个 K10 内的两个设备相互可以端到端访问，但是在这个特殊的系统上，两个 K10 附属于不同的 IOH 芯片组从而不能相互到端访问。

除了用 CUDA 编程接口判定哪些卡能够端到端通信外，Linux 命令`/sbin/lspci -tv` 也能用来打印 PCIe 树

4.1.2 端到端直接传输

用下列代码开始对端到端直接传输的论述，代码利用三种方法将一个数组从一个设备复制到另一个设备上：通过赋值传输，它隐式利用了 `cudaMemcpy()`；通过打开到端访问的 `cudaMemcpyPeer()`传输；通过关闭到端访问的 `cudaMemcpyPeer()`传输。在每个设备上，代码都使用事件对传输计时两次。这里先列出代码，稍后详述：

```
 1  program directTransfer
 2    use cudafor
 3    implicit none
 4    integer , parameter :: N = 4*1024*1024
 5    real , pinned , allocatable :: a(:), b(:)
 6    real , device , allocatable :: a_d(:), b_d(:)
 7
 8    ! these hold free and total memory before and after
 9    ! allocation , used to verify allocation is happening
10    ! on proper devices
11    integer(int_ptr_kind()), allocatable :: &
12         freeBefore(:), totalBefore(:), &
13         freeAfter(:), totalAfter(:)
14
15    integer :: istat , nDevices , i, accessPeer , timingDev
16    type (cudaDeviceProp) :: prop
17    type (cudaEvent) :: startEvent , stopEvent
18    real :: time
19
20    ! allocate host arrays
21    allocate(a(N), b(N))
```

```
22    allocate(freeBefore(0:nDevices -1), &
23         totalBefore(0:nDevices -1))
24    allocate(freeAfter(0:nDevices -1), &
25         totalAfter(0:nDevices -1))
26
27    ! get device info (including total and free memory)
28    ! before allocating a_d and b_d on devices 0 and 1
29    istat = cudaGetDeviceCount(nDevices)
30    if (nDevices < 2) then
31       write(*,*) 'Need at least two CUDA capable devices'
32       stop
33    endif
34    write(*,"('Number of CUDA -capable devices: ', i0 ,/)") &
35         nDevices
36    do i = 0, nDevices -1
37       istat = cudaGetDeviceProperties(prop , i)
38       istat = cudaSetDevice(i)
39       istat = cudaMemGetInfo(freeBefore(i), totalBefore(i))
40    enddo
41    istat = cudaSetDevice(0)
42    allocate(a_d(N))
43    istat = cudaSetDevice(1)
44    allocate(b_d(N))
45
46    ! print out free memory before and after allocation
47    write(*,"('Allocation summary ')")
48    do i = 0, nDevices -1
49       istat = cudaGetDeviceProperties(prop , i)
50       write(*,"(' Device ', i0, ': ', a)") &
51            i, trim(prop%name)
52       istat = cudaSetDevice(i)
53       istat = cudaMemGetInfo(freeAfter(i), totalAfter(i))
54       write(*,"(' Free memory before: ', i0, &
55            ', after: ', i0, ', difference: ',i0 ,/)") &
56            freeBefore(i), freeAfter(i), &
57            freeBefore(i)-freeAfter(i)
58    enddo
59
60    ! check whether devices 0 and 1 can use P2P
61    if (nDevices > 1) then
62       istat = cudaDeviceCanAccessPeer(accessPeer , 0, 1)
63      if (accessPeer == 1) then
64         write(*,*) 'Peer access available between 0 and 1'
65      else
```

```
66          write(*,*) 'Peer access not available between 0 and 1'
67        endif
68      endif
69
70      ! initialize
71      a = 1.0
72      istat = cudaSetDevice(0)
73      a_d = a
74
75      ! perform test twice , timing on both sending GPU
76      ! and receiving GPU
77      do timingDev = 0, 1
78         write(*,"(/,'Timing on device ', i0, /)") timingDev
79
80         ! create events on the timing device
81         istat = cudaSetDevice(timingDev)
82         istat = cudaEventCreate(startEvent)
83         istat = cudaEventCreate(stopEvent)
84
85         if (accessPeer == 1) then
86            ! enable P2P communication
87            istat = cudaSetDevice(0)
88            istat = cudaDeviceEnablePeerAccess(1, 0)
89            istat = cudaSetDevice(1)
90            istat = cudaDeviceEnablePeerAccess(0, 0)
91
92            ! transfer (implicitly) across devices
93            b_d = -1.0
94            istat = cudaSetDevice(timingDev)
95            istat = cudaEventRecord(startEvent ,0)
96            b_d = a_d
97            istat = cudaEventRecord(stopEvent ,0)
98            istat = cudaEventSynchronize(stopEvent)
99            istat = cudaEventElapsedTime(time , &
100                startEvent , stopEvent)
101           b = b_d
102           if (any(b /= a)) then
103              write(*,"('Transfer failed ')")
104           else
105              write(*,"('b_d=a_d transfer (GB/s): ', f)") &
106                   N*4/time/1.0E+6
107           endif
108        end if
109
```

```
110       ! transfer via cudaMemcpyPeer()
111       if (accessPeer == 0) istat = cudaSetDevice(1)
112       b_d = -1.0
113
114       istat = cudaSetDevice(timingDev)
115       istat = cudaEventRecord(startEvent ,0)
116       istat = cudaMemcpyPeer(b_d , 1, a_d , 0, N)
117       istat = cudaEventRecord(stopEvent ,0)
118       istat = cudaEventSynchronize(stopEvent)
119       istat = cudaEventElapsedTime(time , startEvent , &
120            stopEvent)
121       if (accessPeer == 0) istat = cudaSetDevice(1)
122       b = b_d
123       if (any(b /= a)) then
124          write(*,"('Transfer failed ')")
125       else
126          write(*,"('cudaMemcpyPeer transfer (GB/s): ', f)") &
127               N*4/time/1.0E+6
128       endif
129
130       ! cudaMemcpyPeer with P2P disabled
131       if (accessPeer == 1) then
132         istat = cudaSetDevice(0)
133         istat = cudaDeviceDisablePeerAccess (1)
134         istat = cudaSetDevice(1)
135         istat = cudaDeviceDisablePeerAccess (0)
136         b_d = -1.0
137
138         istat = cudaSetDevice(timingDev)
139         istat = cudaEventRecord(startEvent ,0)
140         istat = cudaMemcpyPeer(b_d , 1, a_d , 0, N)
141         istat = cudaEventRecord(stopEvent ,0)
142         istat = cudaEventSynchronize(stopEvent)
143         istat = cudaEventElapsedTime(time , startEvent , &
144              stopEvent)
145
146         istat = cudaSetDevice(1)
147         b = b_d
148         if (any(b /= a)) then
149            write(*,"('Transfer failed ')")
150         else
151            write(*,"('cudaMemcpyPeer transfer w/ P2P ', &
152                 ' disabled (GB/s): ', f)") N*4/time/1.0E+6
153         endif
```

```
154       end if
155
156       ! destroy events associated with timingDev
157       istat = cudaEventDestroy(startEvent)
158       istat = cudaEventDestroy(stopEvent)
159     end do
160
161     ! clean up
162     deallocate(freeBefore , totalBefore , freeAfter , totalAfter)
163     deallocate(a, b, a_d , b_d)
164   end program directTransfer
```

声明主机数据之后，22-28 和 36-40 行利用设备管理接口判定系统上的 GPU 类型和数量。这里要特别留意：

```
39        istat = cudaMemGetInfo(freeBefore(i), totalBefore(i))
```

它用来探测分配数组之前每个设备上的可用内存容量。设备数据分配在第 42 行和第 44 行。分配设备空间之后，第 53 行再次利用 `cudaMemGetInfo()` 探测所有设备上分配空间后的可用内存，分配前后可用内存的差值被打印出来。这样做是为验证数组确实被分配在期望的设备上。

61-68 行探明设备 0 和 1 之前是否可以到端访问，接着初始化主机数据，一个循环执行设备间的数据传输并计时。为打开两个设备间的双向到端访问，必须调用 `cudaDeviceEnablePeerAccess()` 两次，但是判定两个设备间是否可以到端访问仅需调用 `cudaDeviceCanAccessPeer()` 一次。

因为设备 0 和 1 都用来对执行计时，所以开始于第 77 行的主循环遍历计时设备 `timingDev`。在每一个设备上计时并不是期待不同的答案；相反，这样做为揭示在多 GPU 代码中用事件计时的一些特性。CUDA 事件使用 GPU 时钟，因此与创建事件时的当前设备相关联。正是因为这个原因，在第 81 行将计算设备设定为当前设备之后，计算设备循环的 82-83 行才创建事件。设定、创建完成之后，如果设备 0 和 1 之间可以到端访问，87-90 行就打开到端访问并在第 96 行用赋值语句完成数据的直接传输。调用任何 CUDA 事件接口之前，当前设备必须设定为 `timingDev`.

注意，在第 101 行将 `b_d` 从设备 1 传输到主机之前，必须将当前设备 `timingDevice` 设定为设备 1. 当前设备不必处于数据传输的发送端或接收端；只需要本次传输涉及的设备可以到端访问。正因如此，第 88 和 90 行才在设备 0 和 1 之间打开了双向访问：以适应当前既不发送也不接收数据时的设备到主机传输。两个设备间的数据传输也适用相同的逻辑。只要当前设备能到端访问数据传输涉及的两个设备的内存，那么这个传输就是一个合法操作。

无论到端访问是否已经打开，显式调用 `cudaMemcpyPeer()` 的数据传输都能够完成。如果到端访问已打开，那么该传输不通过 CPU 中转就能完成，并能获得与上面赋值隐式传近似的传输速度。如果到端访问没有打开，那么 `cudaMemcpyPeer()` 就递交一个从源数给驻留设备开始的设备到主机传输，再递交一个以终点数组驻留设备为目标的主机到设备传输。此外，到端访问没有打开时必须留意当前设备已经恰当设置，如在第 111 行所示的初始化设备数据时，以及第 112 行所示的取回结果时：

```
111       if (accessPeer == 0) istat = cudaSetDevice(1)
112       b_d = -1.0
```

当到端访问已经打开时，只要当前设备能够访问传输涉及的设备，就不必再设定当前设备。

最后，在第 133 行和第 135 行显式地关闭端到端通信之后对传输计时。再一次，在第 140 行使用 cudaMemcpyPeer().在一个带有两块具有端到端能力卡的系统上运行这个程序，得到如下结果：

```
Number of CUDA -capable devices: 2

Allocation summary
  Device 0: Tesla M2050
    Free memory before: 2748571648 , after: 2731794432 , difference: 16777216

  Device 1: Tesla M2050
    Free memory before: 2748571648 , after: 2731794432 , difference: 16777216

 Peer access available between 0 and 1

Timing on device 0

b_d=a_d transfer (GB/s): 5.0965576
cudaMemcpyPeer transfer (GB/s): 5.3010325
cudaMemcpyPeer transfer w/ P2P disabled (GB/s): 3.4764111

Timing on device 1

b_d=a_d transfer (GB/s): 5.2460275
cudaMemcpyPeer transfer (GB/s): 5.2518082
cudaMemcpyPeer transfer w/ P2P disabled (GB/s): 3.5856843
```

不出所料，关闭端到端的传输速率明显低于打开时的传输速率。

在一个带有两块 Tesla K10 GPU 的系统上，得到如下结果：

```
Number of CUDA -capable devices: 4

Allocation summary
  Device 0: Tesla K10.G1.8GB
    Free memory before: 4240695296 , after: 4223918080 , difference: 16777216

  Device 1: Tesla K10.G1.8GB
    Free memory before: 4240695296 , after: 4223918080 , difference: 16777216

  Device 2: Tesla K10.G2.8GB
    Free memory before: 4240695296 , after: 4240695296 , difference: 0

  Device 3: Tesla K10.G2.8GB
    Free memory before: 4240695296 , after: 4240695296 , difference: 0

 Peer access available between 0 and 1

Timing on device 0
```

```
b_d=a_d transfer (GB/s): 10.8029337
cudaMemcpyPeer transfer (GB/s): 11.6984177
cudaMemcpyPeer transfer w/ P2P disabled (GB/s): 8.2490988

Timing on device 1

b_d=a_d transfer (GB/s): 11.3913746
cudaMemcpyPeer transfer (GB/s): 11.6451511
cudaMemcpyPeer transfer w/ P2P disabled (GB/s): 8.9019289
```

这里观察到更高的带宽，这是因为单块 K10 内的两个设备由一个 PCIe Gen 3 交换机相连接。可以利用环境变量 CUDA_VISIBLE_DEVICES 让位不同 K10 上的两个设备上进行传输。在这个特殊系统上，每个 K10 插在一个 PCIe Gen 3 槽位上：

```
% export CUDA_VISIBLE_DEVICES=0,2
% ./directTransfer
Number of CUDA -capable devices: 2

Allocation summary
  Device 0: Tesla K10.G1.8GB
    Free memory before: 4240695296 , after: 4223918080 , difference: 16777216

  Device 1: Tesla K10.G2.8GB
    Free memory before: 4240695296 , after: 4223918080 , difference: 16777216

 Peer access not available between 0 and 1

Timing on device 0

cudaMemcpyPeer transfer (GB/s): 8.3558540

Timing on device 1

cudaMemcpyPeer transfer (GB/s): 8.8945284
```

本章开头讲述一个代码，它将有端到端通信能力的设备对以矩阵的形式打印出来。上面的代码打印出来两个设备间的数据传输带宽。可以在一份代码里将这些特性结合起来，打印出来一个两两设备间带宽的矩阵：

```
1  program p2pBandwidth
2    use cudafor
3    implicit none
4    integer , parameter :: N = 4*1024*1024
5    type distributedArray
6       real , device , allocatable :: a_d(:)
7    end type distributedArray
```

```
 8    type (distributedArray), allocatable :: distArray(:)
 9
10    real , allocatable :: bandwidth(:,:)
11    real :: array(N), time
12    integer :: nDevices , access , i, j, istat
13    type (cudaDeviceProp) :: prop
14    type (cudaEvent) :: startEvent , stopEvent
15
16    istat = cudaGetDeviceCount(nDevices)
17    write(*,"('Number of CUDA -capable devices: ', i0 ,/)") &
18         nDevices
19
20    do i = 0, nDevices -1
21       istat = cudaGetDeviceProperties(prop , i)
22       write(*,"('Device ', i0, ': ', a)") i, trim(prop%name)
23    enddo
24    write(*,*)
25
26    allocate(distArray(0:nDevices -1))
27
28    do j = 0, nDevices -1
29       istat = cudaSetDevice(j)
30       allocate(distArray(j)%a_d(N))
31       distArray(j)%a_d = j
32       do i = j+1, nDevices -1
33          istat = cudaDeviceCanAccessPeer(access , j, i)
34          if (access == 1) then
35             istat = cudaSetDevice(j)
36             istat = cudaDeviceEnablePeerAccess(i, 0)
37             istat = cudaSetDevice(i)
38             istat = cudaDeviceEnablePeerAccess(j, 0)
39          endif
40       enddo
41    end do
42
43    allocate(bandwidth(0:nDevices -1, 0:nDevices -1))
44    bandwidth = 0.0
45
46    do j = 0, nDevices -1
47       istat = cudaSetDevice(j)
48       istat = cudaEventCreate(startEvent)
49       istat = cudaEventCreate(stopEvent)
50       do i = 0, nDevices -1
51          if (i == j) cycle
```

```
52          istat = cudaMemcpyPeer(distArray(j)%a_d , j, &
53               distArray(i)%a_d , i, N)
54          istat = cudaEventRecord(startEvent ,0)
55          istat = cudaMemcpyPeer(distArray(j)%a_d , j, &
56               distArray(i)%a_d , i, N)
57          istat = cudaEventRecord(stopEvent ,0)
58          istat = cudaEventSynchronize(stopEvent)
59          istat = cudaEventElapsedTime(time , &
60               startEvent , stopEvent)
61
62          array = distArray(j)%a_d
63          if (all(array == i)) bandwidth(j,i) = N*4/time/1.0E+6
64       end do
65       distArray(j)%a_d = j
66       istat = cudaEventDestroy(startEvent)
67       istat = cudaEventDestroy(stopEvent)
68    enddo
69
70    write(*,"('Bandwidth (GB/s) for transfer size (MB): ', &
71         f9.3,/)") N*4.0/1024**2
72    write(*,"(' S\\R 0')", advance='no')
73    do i = 1, nDevices -1
74       write(*,"(5x,i3)", advance='no') i
75    enddo
76    write(*,*)
77
78    do j = 0, nDevices -1
79       write(*,"(i3)", advance='no') j
80       do i = 0, nDevices -1
81          if (i == j) then
82             write(*,"(4x,'-',3x)", advance='no')
83          else
84             write(*,"(f8.2)",advance='no') bandwidth(j,i)
85          end if
86       end do
87       write(*,*)
88    end do
89
90    ! cleanup
91    do j = 0, nDevices -1
92       deallocate(distArray(j)%a_d)
93    end do
94    deallocate(distArray ,bandwidth)
95
```

```
96  end program p2pBandwidth
```

这里的所有传输都使用 cudaMemcpyPeer()，如果可能的话就打开到端访问。这份代码里的大部分内容都在前面提到的两个代码里出现过，只有设备数组的组织方式代码是新的。5-7 行定义的派生类型 distributedArray 包含一个可分配设备数组 a_d. 第 8 行声明一个该类型的可分配主机数组 distArray. 探明系统上的设备数量之后，第 26 行分配主机数组 distArray，该数组元素从零编号，以匹配 CUDA 枚举设备的方式。这里用到的派生类型是处理分布多个设备上数据的方便、通用做法。如果支持到端访问，36-40 行的循环打开它，46-68 行执行传输并计时，70-78 输出带宽矩阵。在带有两块 Tesla K10 的系统上运行这个代码，得到：

```
Number of CUDA -capable devices: 4

Device 0: Tesla K10.G1.8GB
Device 1: Tesla K10.G1.8GB
Device 2: Tesla K10.G2.8GB
Device 3: Tesla K10.G2.8GB

Bandwidth (GB/s) for transfer size (MB): 16.000

 S\R   0       1       2       3
  0    -      11.64    9.60    9.74
  1   11.57    -       9.83    9.67
  2    9.53    9.61    -      11.70
  3    9.95    9.71   11.64    -
```

输出的行对应传输的发送端设备，列是接收端设备。和前面显示的一样，这里观察到同一个 K10 里的两个设备间有更好的带宽。在带有两个 Tesla S2050 的系统上，得到：

```
Number of CUDA -capable devices: 8

Device 0: Tesla S2050
Device 1: Tesla S2050
Device 2: Tesla S2050
Device 3: Tesla S2050
Device 4: Tesla S2050
Device 5: Tesla S2050
Device 6: Tesla S2050
Device 7: Tesla S2050

Bandwidth (GB/s) for transfer size (MB):    16.000

 S\R   0       1       2       3       4       5       6       7
  0    -       6.61    6.61    6.61    5.25    5.25    5.25    5.25
  1    6.61    -       6.61    6.61    5.25    5.25    5.25    5.25
  2    6.61    6.61    -       6.61    5.24    5.25    5.25    5.25
  3    6.61    6.61    6.61    -       5.25    5.25    5.25    5.25
  4    5.25    5.25    5.25    5.25    -       6.61    6.61    6.61
```

```
  5    5.25    5.25    5.25    5.25    6.61    -       6.61    6.61
  6    5.25    5.25    5.25    5.25    6.61    6.61    -       6.61
  7    5.25    5.25    5.25    5.23    6.61    6.61    6.61    -
```

这里观察到，同一个 S2050 内的传输性能稍微好点。

4.1.3 端到端转置

本节将 3.4 节的矩阵转置例子扩展到操作一个分布在多块 GPU 上的矩阵。一个矩阵分布在四块设备上，图 4.2 是其 nx×ny = 1024×768 个元素的数据摆放方式。每个设备包含图中所示输入矩阵的一个水平条和输出矩阵的一个水平条。这些包含 1024×192 个元素的输入矩阵条都被分割为四个包含 256×192 个元素的小片，小片对应代码里的 p2pTile．正如它名字里暗示的那样，p2pTiles 用来进行端到端传输。 如果必要，将 p2pTile 传输到合适的设备（块对角线上的小片不必传输，因为输入和输出小片都在同一个设备上），然后启动一个 CUDA 转置内核将 p2pTile 中的元素转置，用线程块处理包含 32 × 32 个元素的更小片。

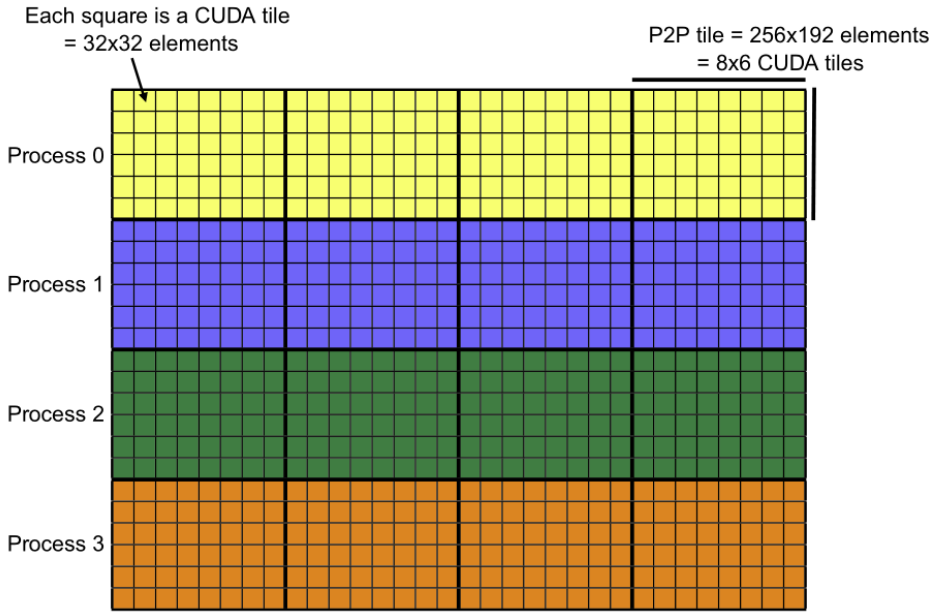


图 4.2

为在四个设备上转置，一个 nx × ny = 1024 × 768 矩阵的设备数据摆放情情况。每个设备保存输入矩阵的一个 1024 × 192 水平条(还有输出矩阵的一个 768 × 256 竖直条). 输入矩阵的每个条分解成四个包含 256 × 192 个元素的小片，小片被端到端传输使用。CUDA 内核使用 48 个线程块来转置这些小片，每个线程块处理一个 32 × 32 的小片。

完整代码在附录 D.4.1. 本节只摘取谈到的部分。论述由转置内核代码开始:

```
14    attributes(global) subroutine cudaTranspose( &
15        odata , ldo , idata , ldi)
16      implicit none
17      real , intent(out) :: odata(ldo ,*)
18      real , intent(in) :: idata(ldi ,*)
19      integer , value , intent(in) :: ldo , ldi
20      real , shared :: tile(cudaTileDim+1, cudaTileDim)
21      integer :: x, y, j
22
```

```
23      x = (blockIdx%x-1) * cudaTileDim + threadIdx%x
24      y = (blockIdx%y-1) * cudaTileDim + threadIdx%y
25
26      do j = 0, cudaTileDim -1, blockRows
27         tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
28      end do
29
30      call syncthreads()
31
32      x = (blockIdx%y-1) * cudaTileDim + threadIdx%x
33      y = (blockIdx%x-1) * cudaTileDim + threadIdx%y
34
35      do j = 0, cudaTileDim -1, blockRows
36         odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
37      end do
38    end subroutine cudaTranspose
```

这个转置基本上就是 3.4 节的单 GPU 转置，不同之处仅有传递给内核的两个额外参数 `ldi` 和 `ldo`，它们是输入矩阵和输出矩阵的第一个维度。因为每个内核调用都要将每个设备上的矩阵片的一个子矩阵转置，所以需要这些参数。对代码不做任何修改，将数据复制到一个临时数组然后复制出来也能实现转置，但这些中间数据传输会严重影响性能。注意，第一维参数只用在 17-18 行的输入矩阵和输出矩阵的声明中；剩余代码与单 GPU 代码完全一样。

大部分主机代码完成的都是常规任务，诸如获取设备数量和设备类型(85-94 行)、核对所有设备具有端到端能力和打开端到端通信(96-119 行)、验证矩阵能被平均分割成多种尺寸的小片(121-140 行)、打印出多种尺寸(146-165 行)、初始化主机数据和在主机上转置(169-170 行)。

因为想用 GPU 间的数据传输来掩盖转置内核的执行，所以端到端通信和内核执行应避开使用默认流。让每个设备都有 `nDevices` 个流，每个流对应一个转置调用。因为有 `nDevices` 个设备，每个设备都要求 `nDevices` 个流，所以使用一个二维变量来存放流的编号：

```
180    attributes(global) subroutine cudaTranspose( &
181         odata , ldo , idata , ldi)
182      implicit none
183      real , intent(out) :: odata(ldo ,*)
184      real , intent(in) :: idata(ldi ,*)
185      integer , value , intent(in) :: ldo , ldi
186      real , shared :: tile(cudaTileDim+1, cudaTileDim)
187      integer :: x, y, j
188
189      x = (blockIdx%x-1) * cudaTileDim + threadIdx%x
190      y = (blockIdx%y-1) * cudaTileDim + threadIdx%y
191
192      do j = 0, cudaTileDim -1, blockRows
193         tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
194      end do
195
```

```
196       call syncthreads()
197
198       x = (blockIdx%y-1) * cudaTileDim + threadIdx%x
199       y = (blockIdx%x-1) * cudaTileDim + threadIdx%y
200
201       do j = 0, cudaTileDim -1, blockRows
202          odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
203       end do
204    end subroutine cudaTranspose
```

其中，streamID 的第一个指标对应该流关联的特定设备，第二个指标表示计算的阶段。

转置的阶段从 0 数到 nDevices-1，组织如下：在第零阶段，每个设备将全局矩阵块对角线上的子矩阵转置，图 4.3 的顶端示意图显示了本阶段操作。先完成该操作是因为它没有调用端到端通信，且内核执行还能重叠第一阶段的数据传输。

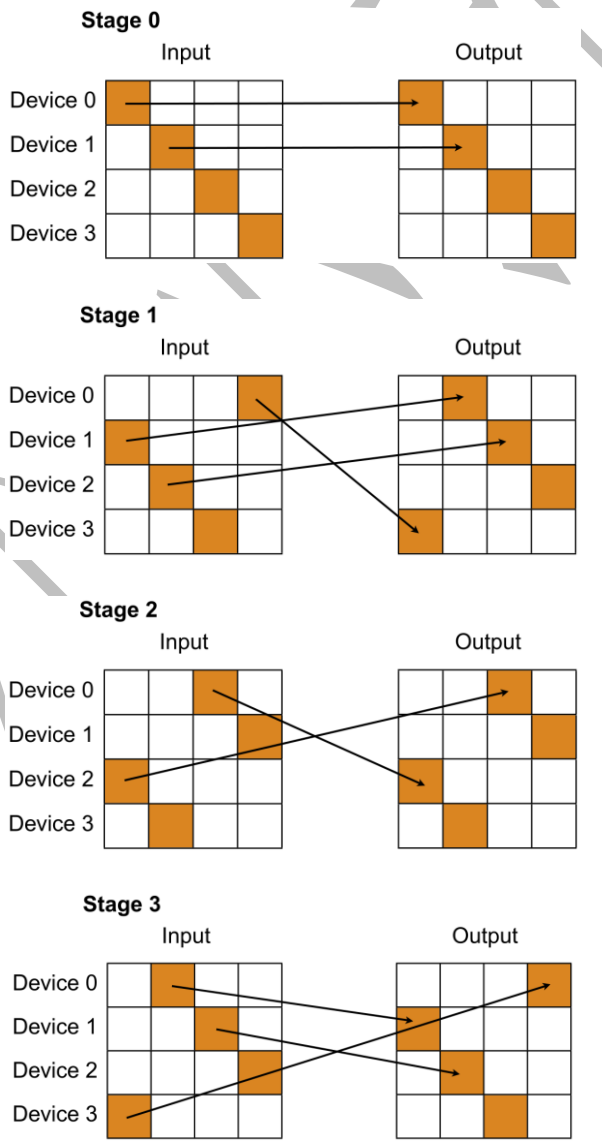


图 4.3

矩阵转置的各个阶段。阶段零，每个设备将全局矩阵对角线上的线转置，不需要端到端通信。阶段一，输入矩阵第一条下对角线上的块被传送到相应上对角线上块的所在设备，此后接收设备实施转置操作。后续阶段(例如阶段 2)对接下来的上、下对角线做同情的工作。对角线封装对后续阶段更好，最后一个阶段的通信模式是第一阶段的反转。

在第一阶段，数据从输入矩阵的第一条块下对角线出来，发送到对应的第一条块上对角线所在的矩阵，如图 4.3 所示。传输完成后，接收设备实施转置操作。注意，第一阶段传输的块中有一个不在下对角线上，因为模式封装要使所有设备在每阶段都要发送数据和接收数据。接下来的阶段在其它下对角线和上对角线上完成相似的操作，直到所有的块都被转置。这些阶段间中的封装更加明显，因此最终阶段的通信模式就是第一阶段的反转。这样安排，第零阶段除外，每个阶段中，每个设备都发送和接收一块数据，只要传输是异步的，这两个传输都能被重叠，因为设备具有相互分离发送复制引擎和接收复制引擎。

使用派生类型 deviceArray 来分布式存储全局矩阵：

```
68    ! distributed arrays
69    type deviceArray
70      real , device , allocatable :: v(:,:)
71    end type deviceArray
72
73    type (deviceArray), allocatable :: &
74         d_idata(:), d_tdata(:), d_rdata(:) ! (0:nDevices -1)
```

前一节的p2pBandwidth代码里用到了同样的技术。派生类型的实例将是主机数据，但成员v是设备数据。第 74 行是该派生类型的三个可分配数组的声明：d_idata 是输入数据；d_rdata 是传输中用到的接收缓冲区；d_tdata 保存转置后的最终数据。这些变量这样分配：

```
190    allocate(d_idata(0:nDevices -1),&
191         d_tdata(0:nDevices -1), d_rdata(0:nDevices -1))
```

它表示全局数组被分解成了图 4.2 所示的水平条。派生类型的成员存放水平条，它们的空间分配和初始化为：

```
193    do p = 0, nDevices -1
194       istat = cudaSetDevice(p)
195       allocate(d_idata(p)%v(nx,p2pTileDimY), &
196            d_rdata(p)%v(nx,p2pTileDimY), &
197            d_tdata(p)%v(ny,p2pTileDimX))
198
199       yOffset = p*p2pTileDimY
200       d_idata(p)%v(:,:) = h_idata(:, &
201            yOffset+1:yOffset+p2pTileDimY)
202       d_rdata(p)%v(:,:) = -1.0
203       d_tdata(p)%v(:,:) = -1.0
204    enddo
```

这里的 nx 和 ny 是全局矩阵尺寸，p2pTileDimY 和 p2pTileDimX 分别是输入矩阵和转置后矩阵水平片的寸尽。注意，在为每个成员 v 分配空间之前，第 194 行上已经将恰当的设备设定为当前设备。同样，因为主机上的矩阵存储在h_idata(nx,ny)中，所以200-201行上的偏移量yOffset用来初始化d_idata：

实施多个转置阶段的代码是：

```
216    ! Stage 0:
```

```
217    ! transpose diagonal blocks (local data) before kicking off
218    ! transfers and transposes of other blocks
219
220    do p = 0, nDevices -1
221       istat = cudaSetDevice(p)
222       if (asyncVersion) then
223          call cudaTranspose &
224               <<<dimGrid , dimBlock , 0, streamID(p,0)>>> &
225               (d_tdata(p)%v(p*p2pTileDimY+1,1), ny, &
226               d_idata(p)%v(p*p2pTileDimX+1,1), nx)
227       else
228          call cudaTranspose <<<dimGrid , dimBlock >>> &
229               (d_tdata(p)%v(p*p2pTileDimY+1,1), ny, &
230               d_idata(p)%v(p*p2pTileDimX+1,1), nx)
231       endif
232    enddo
233
234    ! now send data to blocks to the right of diagonal
235    ! (using mod for wrapping) and transpose
236
237    do stage = 1, nDevices -1 ! stages = offset diagonals
238       do rDev = 0, nDevices -1 ! device that receives
239          sDev = mod(stage+rDev , nDevices) ! dev that sends
240
241          if (asyncVersion) then
242             istat = cudaSetDevice(rDev)
243             istat = cudaMemcpy2DAsync( &
244                  d_rdata(rDev)%v(sDev*p2pTileDimX+1,1), nx, &
245                  d_idata(sDev)%v(rDev*p2pTileDimX+1,1), nx, &
246                  p2pTileDimX , p2pTileDimY , &
247                  stream=streamID(rDev ,stage))
248          else
249             istat = cudaMemcpy2D( &
250                  d_rdata(rDev)%v(sDev*p2pTileDimX+1,1), nx, &
251                  d_idata(sDev)%v(rDev*p2pTileDimX+1,1), nx, &
252                  p2pTileDimX , p2pTileDimY)
253          end if
254
255          istat = cudaSetDevice(rDev)
256          if (asyncVersion) then
257             call cudaTranspose &
258                  <<<dimGrid , dimBlock , 0, &
259                  streamID(rDev ,stage)>>> &
260                  (d_tdata(rDev)%v(sDev*p2pTileDimY+1,1), ny, &
```

```
261                 d_rdata(rDev)%v(sDev*p2pTileDimX+1,1), nx)
262         else
263            call cudaTranspose <<<dimGrid , dimBlock >>> &
264                 (d_tdata(rDev)%v(sDev*p2pTileDimY+1,1), ny, &
265                 d_rdata(rDev)%v(sDev*p2pTileDimX+1,1), nx)
266         endif
267       enddo
268    enddo
```

阶段 0 发生在 220-232 行上的循环里。在 221 行上设定设备之后，实施对角块转置，要么使用第 228 行上的默认阻塞流，要么使用第 223 行上非默认流。参数 asyncVersion 用来在异步执行和同步执行间切换。内核启动中采用的执行配置这样决定：

```
142    dimGrid = dim3(p2pTileDimX/cudaTileDim , &
143         p2pTileDimY/cudaTileDim , 1)
144    dimBlock = dim3(cudaTileDim , blockRows , 1)
```

这里的线程块与单 GPU 的情形相同，每个内核操作一个尺寸为 p2pTileDimX×p2pTileDimY 的子矩阵。

其它阶段由 237-268 行里的循环实施。第 238 行和第 239 行上决定发送设备和接收设备之后，就实施端到端传输，根据 asyncVersion 选择使用 cudaMemcpy2DAsync() 或 cudaMemcpy2D()。如果选用异步版本，那么在第 242 行上将接收设备设定为当前设备，相应地，传输使用的非默认流就是与接收设备关联的那个流。这里使用与接收数据的设备相关连的流，而不使用与发送数据的设备相关连的流，是因为想阻止接收设备启动转置内核直至传输完成。当传输与转置使用同一个流时，默认这样完成。对同步数据传输，不必通过 cudaSetDevice() 指定设备。注意，接收数据的数组是 d_rdata。接着，从 d_rdata 到 d_tdata 的移位转置由第 257 行或第 263 行启动的内核实施。不管是否使用默认流，第 255 行所做的设备设定必须执行。

剩余代码将数据传回主机，检查正确性，并报告实际带宽。这里的记时使用一个墙钟记时器完成。这个代码使用了 C 函数 gettimeofday()：

```
 1  #include <time.h>
 2  #include <sys/types.h>
 3  #include <sys/times.h>
 4  #include <sys/time.h>
 5
 6  double wallclock()
 7  {
 8    struct timeval tv;
 9    struct timezone tz;
10    double t;
11
12    gettimeofday(&tv, &tz);
13
14    t = (double)tv.tv_sec;
15    t += ((double)tv.tv_usec )/1000000.0;
16
```

```
17     return t;
18   }
```

Fortran 代码使用 timing 模块来访问计时代码：

```
1  module timing
2    interface wallclock
3       function wallclock() result(res) bind(C, name='wallclock')
4         use iso_c_binding
5         real (c_double) :: res
6       end function wallclock
7    end interface wallclock
8  end module timing
```

每当调用该例程，都要显式检测以确保设备上没有挂起的操作或正在执行的操作：

```
217    do p = 0, nDevices -1
218       istat = cudaSetDevice(p)
219       istat = cudaDeviceSynchronize()
220    enddo
221    timeStop = wallclock()
```

注意，这个多 GPU 代码的大部分都是声明和初始化数组以及开启端到端通信相关的开销。真正的数据传输和内核启动都包含在近 50 行代码中，这些代码还包含了异步执行和同步执行分支。转置内核本身也是对单 GPU 转置稍作修改以允许数组有任意的首维长度。

使用一个带有两块设备的计算节点来运行这个程序。为与 3.4 节使用 1024 × 1024 矩阵的单 GPU 转置对比结果，这里全部选择尺寸为 2048 × 2048 的矩阵。这种情况下，每个转置内核处理一个 1024 × 1024 的子矩阵，与单 GPU 情形相同。采用阻塞传输，得到结果：

```
Number of CUDA -capable devices: 2

  Device 0: Tesla M2050
  Device 1: Tesla M2050

Array size: 2048x2048

CUDA block size: 32x8, CUDA tile size: 32x32
dimGrid: 32x32x1 , dimBlock: 32x8x1

nDevices: 2, Local input array size: 2048x1024
p2pTileDim: 1024x1024

async mode: F


Bandwidth (GB/s): 16.43
```

当使用异步传输时，得到：

```
Number of CUDA -capable devices: 2
```

```
  Device 0: Tesla M2050
  Device 1: Tesla M2050

Array size: 2048x2048

CUDA block size: 32x8, CUDA tile size: 32x32
dimGrid: 32x32x1 , dimBlock: 32x8x1

nDevices: 2, Local input array size: 2048x1024
p2pTileDim: 1024x1024

async mode: T


Bandwidth (GB/s): 29.73
```

尽管这两个数字均不及单 GPU 情形达到有效带宽，但是必须考虑到一半的数据是在 PCIe 总线上传输的，它比一个 GPU 内部的全局内存带宽要慢一个数量级以上。根据这个事实，重叠内核执行的异步传输确实很有优势，这个优势可以在结果中看到。另外，转置通常用作其它可并行执行的操作的一个手段，那种情况下 PCIe 传输时间可以进一步摊薄。

4.2 用 MPI 多 GPU 编程

前面小节探究了从一个主机线程使用多个 GPU. 用 `cudaSetDevice()`在 GPU 间切换为在几个 GPU 上分布及处理数据提供了一个方便的途径。然而，随着问题规模增长，这种方法遇到了一个限制：单个节点能安装多少块 GPU？当达到这个上限时，就需要使用 MPI 为多个节点编程。MPI 可以与讲述过的多 GPU 技术配合使用，MPI 用来在节点间传输数据，而 CUDA 4.0 的多 GPU 特性用来在单个节点附属的 GPU 间分布与处理数据。这与 CPU 集群上使用 OpenMP 和 MPI 的方式类似。

这里简要介绍一个本节用到的 MPI 库的调用。对初次接触 MPI 的读者，接口例程的更详细讲述可以在这里找到：MPI: The Complete Reference (Snir, 1966) and Using MPI: Portable Parallel Programming with the Message-Passing Interface (Gropp et al., 1999). 话题转入 MPI 代码之前，应该说说 MPI 编程模块的几个高层次方面。正如一个内核里的所有设备线程都执行同一份设备代码，一个 MPI 应用里的所有主机线程都执行同一份主机代码。CUDA 中使用预定义变量来识别设备代码里的单个设备线程。MPI 中，单个 MPI 线程，或秩(rank)，通过库调用 `MPI_COMM_RANK()`来识别。CUDA 编程模块从细粒度并行(例如合并访问)中获益，而 MPI 通常从粗粒度并行中获益，每个 MPI 秩操作一大块数据。

MPI CUDA Fortran 代码的编译器由 MPI 封装 `mpif90` 完成，众多 MPI 发行版都提供这个封装。MPI 程序的执行通常由命令 `mpirun` 实施，在命令行上给它提供程序可执行文件及使用的 MPI 秩数量。因为 MVAPICH(可在 `http://mvapich.cse.ohio-state.edu` 找到)这个 MPI 实现的 CUDA 特性，本节例子选用 MVAPICH 软件包，它的 CUDA 特性本节稍后论述。

就将设备映射到 MPI 秩的方法而言，有多种方法可以联合使用 CUDA Fortran 与 MPI. 本节选取一个简单、通用的方法，据此每个 MPI 秩都被关联到单个 GPU. 这样的配置里，简单地在每个节点上使用多个 MPI 秩就能继续在一个节点使用多个 GPU， 单节点多 MPI 秩由应用的启动方式决定，而不是由代码内部决定。

If the nature of the application merits a different mapping of GPUs to MPI ranks, we can add this later using the techniques discussed earlier in this chapter, but in general the one-GPU-per-MPI rank model is a good first approach.

4.2.1 将设备指派给 MPI 秩

编写多 GPU MPI 程序使用的配置是每个 MPI 秩都拥有唯一的设备，此时遇到的第一个问题是如何保证没有设备被指派给多个 MPI 秩。设备关联到 CPU 进程和线程的方式依赖于系统是如何用 `nvidia-smi` 来配置的。英伟达的系统管理接口(`nvidia-smi`)是一个随驱动一起发布的工具，它允许用户显示系统附属设备[1]的设置，允许管理员修改这些设置。可以利用这个基础工具方便地打印出系统的附属设备：

```
% nvidia -smi -L
GPU 0: Tesla M2050 (S/N: 0322210101582)
GPU 1: Tesla M2050 (S/N: 0322210101238)
```

还可以得到关于温度、功耗及多种设置的详细信息。这里关注的设置是计算模式。计算模式决定多个进程或线程能否使用同一块 GPU. 四种模式为：

默认：0 这种模式下多个主机线程可以通过调用 `cudaSetDevice()` 来使用同一个设备。

互斥线程：1 这种模式下，整个系统内只能有一个进程创建仅一个上下文，在某一时刻这个上下文至多是该进程的一个线程的当前上下文。

禁止：2 这种模式下，设备上不能创建上下文。

互斥进程：3 这种模式下，整个系统内只允许一个进程创建一个上下文，这个上下文可以是该进程的所有线程的当前上下文。

可以像下面这样查询计算模式：

```
% nvidia -smi -q -d COMPUTE
============== NVSMI  LOG ==============
Timestamp                           : Wed Feb 1 17:06:23 2012
Driver Version                      : 285.05.32
Attached GPUs                       : 2
GPU 0000:02:00.0
   Compute Mode                     : Exclusive_Process
GPU 0000:03:00.0
   Compute Mode                     : Exclusive_Process
```

输出信息显示两个设备都处于互斥进程模式。

为展示这些模式的不同形为，使用下面的简单程序：

```
1  program mpiDevices
2    use cudafor
3    use mpi
4    implicit none
5
6    ! global array size
7    integer , parameter :: n = 1024*1024
8    ! MPI variables
```

[1] nvidia-smi 的更详细论述在附录 B.

```
 9    integer :: myrank , nprocs , ierr
10    ! device
11    type(cudaDeviceProp) :: prop
12    integer(int_ptr_kind ()) :: freeB , totalB , freeA , totalA
13    real , device , allocatable :: d(:)
14    integer :: i, j, istat
15
16    ! MPI initialization
17    call MPI_init(ierr)
18    call MPI_comm_rank(MPI_COMM_WORLD , myrank , ierr)
19    call MPI_comm_size(MPI_COMM_WORLD , nProcs , ierr)
20
21    ! print compute mode for device
22    istat = cudaGetDevice(j)
23    istat = cudaGetDeviceProperties(prop , j)
24    do i = 0, nprocs -1
25       call MPI_BARRIER(MPI_COMM_WORLD , ierr)
26       if (myrank == i) write(*,"('[',i0,'] using device: ', &
27            i0, ' in compute mode: ', i0)") &
28            myrank , j, prop%computeMode
29    enddo
30
31    ! get memory use before large allocations ,
32    call MPI_BARRIER(MPI_COMM_WORLD , ierr)
33    istat = cudaMemGetInfo(freeB , totalB)
34
35    ! now allocate arrays , one rank at a time
36    do j = 0, nProcs -1
37
38       ! allocate on device associated with rank j
39       call MPI_BARRIER(MPI_COMM_WORLD , ierr)
40       if (myrank == j) allocate(d(n))
41
42       ! Get free memory after allocation
43       call MPI_BARRIER(MPI_COMM_WORLD , ierr)
44       istat = cudaMemGetInfo(freeA , totalA)
45
46       write(*,"(' [',i0,'] after allocation on rank: ', i0, &
47            ', device arrays allocated: ', i0)") &
48            myrank , j, (freeB -freeA)/n/4
49
50    end do
51
52    deallocate(d)
```

```
53    call MPI_Finalize(ierr)
54  end program mpiDevices
```

这个代码简单地让每个秩都分配一个设备数组并在分配完成后报告每个设备上的内存使用情况。第 3 行包含进来的模块里含有所有的 MPI 接口和参数。典型的 MPI 初始化在 17-20 行。第 17 行调用 `MPI_init()` 来初始化 MPI，第 18 行上调用 `MPI_comm_rank()` 将 MPI 秩返回到变量 `myrank` 中，调用 `MPI_comm_size()` 返回应用启动的秩数量。每个秩使用的设备号及其计算模式在 22-29 行打印出来。第 24 行上用来输出的循环不是技术上必须的，但它与 `MPI_BARRIER()` 调用配合使用能够避免来自不同秩的输出的混杂。同步屏障 `MPI_BARRIER()` 阻塞所有 MPI 进程的执行，直到每个 MPI 进程都已到达代码中的这个点，与用在设备代码中的 CUDA `syncthreads()` 类似。打印出设备号和计算模式之后，第 33 行探测每个设备上空闲空间。36-50 行循环里的每次迭代，都在与特定秩 (第 40 行) 关联的设备上分配一个设备数组，并探测分配后的空闲空间(第 44 行)，然后打印出每个设备上已分配的数组个数(第 46 行)。

在一个带有两个处于互斥模式设备的节点上，当用两个进程运行这个代码时，得到：

```
[0] using device: 1 in compute mode: 3
[1] using device: 0 in compute mode: 3
  [0] after allocation on rank: 0, device arrays allocated: 1
  [1] after allocation on rank: 0, device arrays allocated: 0
  [0] after allocation on rank: 1, device arrays allocated: 1
  [1] after allocation on rank: 1, device arrays allocated: 1
```

输出内容显示，前两行列出的两个独立设备分别归两个秩使用，后面输出的内存使用情况也验证了这一点。

在一个处于默认计算模式的设备上，用两个 MPI 进程运行得到结果：

```
[0] using device: 0 in compute mode: 0
[1] using device: 0 in compute mode: 0
  [0] after allocation on rank: 0, device arrays allocated: 1
  [1] after allocation on rank: 0, device arrays allocated: 1
  [0] after allocation on rank: 1, device arrays allocated: 2
  [1] after allocation on rank: 1, device arrays allocated: 2
```

结果显示设备 0 同时被两个 MPI 秩使用，后面的分配摘要也验证了这一点，即每个分配阶段之后，所有秩上空闲内存都有所减少。

不管计算模式设置是什么都能保证每个 MPI 秩拥有唯一设备的一个方法是使用下列模块：

```
 1  module mpiDeviceUtil
 2    interface
 3       subroutine quicksort(base , nmemb , elemsize , compar) &
 4            bind(C,name='qsort')
 5         use iso_c_binding
 6         implicit none
 7         !pgi$ ignore_tkr base ,nmemb ,elemsize ,compar
 8         type(C_PTR), value :: base
 9         integer(C_SIZE_T), value :: nmemb , elemsize
10         type(C_FUNPTR), value :: compar
11       end subroutine quicksort
12
13       integer function strcmp(a,b) bind(C,name='strcmp')
```

```
14          use iso_c_binding
15          implicit none
16          !pgi$ ignore_tkr a,b
17          type(C_PTR), value :: a, b
18        end function strcmp
19    end interface
20  contains
21    subroutine assignDevice(dev)
22      use mpi
23      use cudafor
24      implicit none
25      integer :: dev
26      character (len=MPI_MAX_PROCESSOR_NAME), allocatable :: hosts(:)
27      character (len=MPI_MAX_PROCESSOR_NAME) :: hostname
28      integer :: namelength , color , i
29      integer :: nProcs , myrank , newComm , newRank , ierr
30
31      call MPI_COMM_SIZE(MPI_COMM_WORLD , nProcs , ierr)
32      call MPI_COMM_RANK(MPI_COMM_WORLD , myrank , ierr)
33
34      ! allocate array of hostnames
35      allocate(hosts(0:nProcs -1))
36
37      ! Every process collects the hostname of all the nodes
38      call MPI_GET_PROCESSOR_NAME(hostname , namelength , ierr)
39      hosts(myrank)=hostname(1:namelength)
40
41      do i=0,nProcs -1
42         call MPI_BCAST(hosts(i),MPI_MAX_PROCESSOR_NAME , &
43              MPI_CHARACTER ,i,MPI_COMM_WORLD ,ierr)
44      end do
45
46      ! sort the list of names
47      call quicksort(hosts ,nProcs ,MPI_MAX_PROCESSOR_NAME ,strcmp)
48
49      ! assign the same color to the same node
50      color=0
51      do i=0,nProcs -1
52         if (i > 0) then
53            if ( lne(hosts(i-1),hosts(i)) ) color=color+1
54         end if
55         if ( leq(hostname ,hosts(i)) ) exit
56      end do
57
```

```
58      call MPI_COMM_SPLIT(MPI_COMM_WORLD ,color ,0,newComm ,ierr)
59      call MPI_COMM_RANK(newComm , newRank , ierr)
60
61      dev = newRank
62      ierr = cudaSetDevice(dev)
63
64      deallocate(hosts)
65    end subroutine assignDevice
66
67    ! lexical .eq.
68    function leq(s1, s2) result(res)
69      implicit none
70      character (len=*) :: s1, s2
71      logical :: res
72      res = .false.
73      if (lle(s1,s2) .and. lge(s1,s2)) res = .true.
74    end function leq
75
76  ! lexical .ne.
77  function lne(s1, s2) result(res)
78  implicit none
79  character (len=*) :: s1, s2
80  logical :: res
81  res = .not. leq(s1, s2)
82  end function lne
83  end module mpiDeviceUtil
```

这里 21-65 行的子例程 assignDevice() 负责查找并设定唯一的设备。该子例程利用 MPI 例程 MPI_GET_PROCESSOR_NAME()(第38行)和MPI_BCAST()(第42行)来编制一个所有秩所用主机名的列表。每个秩一旦拥有一份完整的主机名列表，立该调用 C 函数 quicksort() 对列表排序，对比函数是 strcmp.(C例程通过3-18行定义的接口来访问。) 51-56行的循环内，每个节点关联一个颜色，在MPI_COMM_SPLIT()的调用中使用这个颜色来决定一组新的 MPI 通信域(communicator)。一个 MPI 通信域就是一组 MPI 进程。每个新通信域仅包含关联节点上的 MPI 秩，调用 MPI_COMM_RANK() 将返回在新通信域中的新秩。新秩用来枚举本节点上的 CUDA 设备(第 61 行)并设定当前设备(第 62 行)。再一次强调，无论计算模式是什么设置，这个例程都可以使用。通过第 62 行调用 cudaSetDevice() 之前添加更多的逻辑条件，这个代码可以修改成只选择具有某些特征的 GPU，例如有双精度能力的设备和拥有一定数量内存的设备。

接下来的代码演示如何使用这个模块：

```
1  program main
2    use mpi
3    use mpiDeviceUtil
4    use cudafor
5    implicit none
6
7    ! global array size
```

```
 8    integer , parameter :: n = 1024*1024
 9    ! mpi
10    character (len=MPI_MAX_PROCESSOR_NAME) :: hostname
11    integer :: myrank , nprocs , ierr , namelength
12    ! device
13    type(cudaDeviceProp) :: prop
14    integer(int_ptr_kind ()) :: freeB , totalB , freeA , totalA
15    real , device , allocatable :: d(:)
16    integer :: deviceID , i, istat
17
18    call MPI_INIT(ierr)
19    call MPI_COMM_RANK(MPI_COMM_WORLD , myrank , ierr)
20    call MPI_COMM_SIZE(MPI_COMM_WORLD , nProcs , ierr)
21
22    ! get and set unique device
23    call assignDevice(deviceID)
24
25    ! print hostname and device ID for each rank
26    call MPI_GET_PROCESSOR_NAME(hostname , namelength , ierr)
27    do i = 0, nProcs -1
28       call MPI_BARRIER(MPI_COMM_WORLD , ierr)
29       if (i == myrank) &
30            write(*,"('[',i0,'] host: ', a, ', device: ', i0)") &
31            myrank , trim(hostname), deviceID
32    enddo
33
34    ! get memory use before large allocations ,
35    call MPI_BARRIER(MPI_COMM_WORLD , ierr)
36    istat = cudaMemGetInfo(freeB , totalB)
37
38    ! allocate memory on each device
39    call MPI_BARRIER(MPI_COMM_WORLD , ierr)
40    allocate(d(n))
41
42    ! Get free memory after allocation
43    call MPI_BARRIER(MPI_COMM_WORLD , ierr)
44    istat = cudaMemGetInfo(freeA , totalA)
45
46    do i = 0, nProcs -1
47       call MPI_BARRIER(MPI_COMM_WORLD , ierr)
48       if (i == myrank) &
49          write(*,"(' [', i0, '] ', &
50                'device arrays allocated: ', i0)") &
51                myrank , (freeB -freeA)/n/4
```

```
52     end do
53
54     deallocate(d)
55     call MPI_FINALIZE(ierr)
56   end program main
```

调用 MPI_INIT()(第 8 行)之后，只需简单地 use mpiDeviceUtil (第 3 行)并调用 assignDevice()(第 23 行)。当跨节点运行五个 MPI 秩时，这个代码输出：

```
% mpirun -np 5 -host c0-7,c0-2,c0-7,c0-3,c0 -7 assignDevice
[0] host: compute -0-7.local , device: 0
[1] host: compute -0-7.local , device: 1
[2] host: compute -0-7.local , device: 2
[3] host: compute -0-2.local , device: 0
[4] host: compute -0-3.local , device: 0
  [0] device arrays allocated: 1
  [1] device arrays allocated: 1
  [2] device arrays allocated: 1
  [3] device arrays allocated: 1
  [4] device arrays allocated: 1
```

这里，为节省输出空间，仅在所有的分配都做完之后代码才打印出每个设备上分配的数组。这个代码成功能将不同的设备指派给这些 MPI 秩。

4.2.2 MPI 转置

MPI 转置代码，完整版在附录 D.4.2，与本章前面讲述的端到端转置有很多共同之处：区域分解、转置内核、执行配置、通信模式和通信阶段都是相同的。一个细小差别是这个代码需要初始化 MPI 并将设备指派给 MPI 秩，就像前一节显示的那样：

```
73     call MPI_init(ierr)
74     call MPI_comm_rank(MPI_COMM_WORLD , myrank , ierr)
75     call MPI_comm_size(MPI_COMM_WORLD , nProcs , ierr)
76
77     ! get and set device
78
79     call assignDevice(deviceID)
```

这里使用前一节介绍的模块 mpiDeviceUtil 给 MPI 秩指派一个唯一设备。两份代码的参数检查和初始化都相同。MPI 代码里的计时由调用 MPI_BARRIER() 的 MPI 函数 MPI_Wtime() 来完成。

端到端代码和 MPI 代码间的主要区别发生在对通信阶段的循环之中：

```
176     do stage = 1, nProcs -1
177        ! sRank = the rank to which myrank sends data
178        ! rRank = the rank from which myrank receives data
179        sRank = modulo(myrank -stage , nProcs)
180        rRank = modulo(myrank+stage , nProcs)
181
182        call MPI_BARRIER(MPI_COMM_WORLD , ierr)
183
184        ! D2H transfer - pack into contiguous host array
```

```
185       ierr = cudaMemcpy2D(sTile , mpiTileDimX , &
186            d_idata(sRank*mpiTileDimX+1,1), nx, &
187            mpiTileDimX , mpiTileDimY)
188
189       ! MPI transfer
190       call MPI_SENDRECV(sTile , mpiTileDimX*mpiTileDimY , &
191            MPI_REAL , sRank , myrank , &
192            rTile , mpiTileDimX*mpiTileDimY , MPI_REAL , &
193            rRank , rRank , MPI_COMM_WORLD , status , ierr)
194
195       ! H2D transfer
196       d_rTile = rTile
197
198       ! do transpose from receive tile into final array
199       call cudaTranspose <<<dimGrid , dimBlock >>> &
200            (d_tdata(rRank*mpiTileDimY+1,1), ny, &
201            d_rTile , mpiTileDimX)
202    end do
```

端到端代码里在设备间传输数据的调用 `cudaMemcpy2d()` 或 `cudaMemcpy2dAsync()` 被替换为一个设备到主机传输(第 185 行)、一个主机间的 MPI 传输(第 190 行)和一个主机到设备传输(第 196 行).

在一个 2048 × 2048 的矩阵上用两个 MPI 秩运行本代码，得到：

```
Array size: 2048x2048

CUDA block size: 32x8, CUDA tile size: 32x32
dimGrid: 32x32x1 , dimBlock: 32x8x1

nprocs: 2, Local input array size: 2048x1024
mpiTileDim: 1024x1024

Bandwidth (GB/s): 7.37
```

既使两个 MPI 秩及设备都在同一节点上，这个性能也明显严重低于同步端到端代码的性能。但这并不奇怪，因为传输由主机中转。在分布在多个节点上的设备上实施并行转置时，需要忍受主机与设备间的传输开销。然而，当 MPI 传输发生在附属于同一个节点且具有端到端能力的设备之间时，就能利用端到端能力了。幸运的是，诸如 MVAPICH、OpenMPI 和 Cray MPI 等 MPI 实现恰好有这些功能。下一节将演示如何在转置代码里利用 MVAPICH 识别 GPU 的能力加速。

4.2.3 识别 GPU 的 MPI 转置

MPI 的实现 MVAPICH[2]重载了一些 MPI 调用，从而这些调用不但可以处理设备数组也可以主机数组。如果数组参数是设备数组，而设备位于不同的节点或设备不具有端到端能力，那么主机与设备间的传输就

[2] 关于识别 GPU 的 MVAPICH 实现的细节，建议读者阅读 MVAPICH 的文档。因为这里将要使用 CUDA Fortran,所以必须选择 PGI 编译器作为默认 Fortran 编译器。

会得到后台的特别处理。当数组参数是来自同一个节点上的设备数组时，就利用端到端机制完成传输(在默认流中)。

为利用 MVAPICH，只需对代码做稍微修改，附录 D.4.3 列出了修改后的代码。首先，必须在调用任何 MPI 函数前设定设备，但不再像前面的做法那样使用 `assignDevice()`。幸运地，MVAPICH 设定了一个包含所需信息的环境变量，只需像下面这样简单地读取该变量：

```
70    ! for MVAPICH set device before MPI initialization
71
72    call get_environment_variable('MV2_COMM_WORLD_LOCAL_RANK', &
73         localRankStr)
74    read(localRankStr ,'(i10)') localRank
75    ierr = cudaSetDevice(localRank)
```

MVAPICH 代码里通信阶段的主循环是：

```
178    do stage = 1, nProcs -1
179       ! sRank = the rank to which myrank sends data
180       ! rRank = the rank from which myrank receives data
181       sRank = modulo(myrank -stage , nProcs)
182       rRank = modulo(myrank+stage , nProcs)
183
184       call MPI_BARRIER(MPI_COMM_WORLD , ierr)
185
186       ! pack tile so data to be sent is contiguous
187
188       !$cuf kernel do(2) <<<*,*>>>
189       do j = 1, mpiTileDimY
190          do i = 1, mpiTileDimX
191             d_sTile(i,j) = d_idata(sRank*mpiTileDimX+i,j)
192          enddo
193       enddo
194
195       call MPI_SENDRECV(d_sTile , mpiTileDimX*mpiTileDimY , &
196            MPI_REAL , sRank , myrank , &
197            d_rTile , mpiTileDimX*mpiTileDimY , MPI_REAL , &
198            rRank , rRank , MPI_COMM_WORLD , status , ierr)
199
200       ! do transpose from receive tile into final array
201       ! (no need to unpack)
202
203       call cudaTranspose <<<dimGrid , dimBlock >>> &
204            (d_tdata(rRank*mpiTileDimY+1,1), ny, &
205            d_rTile , mpiTileDimX)
206
207    end do ! stage
```

这里第 195 行的 `MPI_SENDRECV()` 使用两个设备数组 `d_sTile` 和 `d_rTile`. 为方便传输，利用 188-193

行的 CUF 内核将待发送数据打包成一个连续数组 `d_sTile`.

当这个代码运行在与前一个 MPI 转置代码一样的节点和设备上时，得到：

```
Array size: 2048x2048

CUDA block size: 32x8, CUDA tile size: 32x32
dimGrid: 32x32x1 , dimBlock: 32x8x1

nprocs: 2, Local input array size: 2048x1024
mpiTileDim: 1024x1024

Bandwidth (GB/s): 18.06
```

它显示的性能与端到端代码的同步版相似。

附录 B 系统管理和环境管理

B.1 环境变量

许多环境变量能各自控制 CUDA Fortran 编译和执行的某些方面。这里将它们分组为与命令行测绘器相关的一般环境变量，和那些与设备代码的及时编译相关的环境变量。

B.1.1 一般环境变量

`CUDA_LAUNCH_BLOCKING`，当设定为 1 时，强制内核同步执行。即，启动内核之后，只有内核完成后控制权才返回 CPU. 这个环境变量提供了一个有效的方法来验证意外的形为是否由主机与设备的同步错误导致。启动阻塞默认关闭。

`CUDA_VISIBLE_DEVICES` 可以用来让系统的某些设备不可见，并能更改设备的枚举顺序。用一个逗号分隔的整数列表来给这个变量赋值，列表包含可见的设备，它们的枚举顺序由后续 CUDA Fortran 程序的执行来展示。回想一下，设备枚举从 0 开始。(可以利用前面出现过的 `deviceQuery` 代码或者实用工具 `pgaccelinfo` 来获取设备的默认枚举顺序。)

B.1.2 命令行测绘器

`COMPUTE_PROFILE`， 当设定为 1 时，打开命令行测绘器。设定为 0 时，测绘关闭。默认测绘关闭。

`COMPUTE_PROFILE_LOG` 设定为希望测绘器输出的文件路径。在多个设备上运行时，必须在文件名里添加字符串`%d`，用来为每一个设备创建单独的测绘输出文件。同样地，有多个主机进程(例如 MPI)的情况下，字符串`%p` 必须出现在文件名里。测绘器默认输出到本地目录内文件 `cuda_profile_%d.log`.

`COMPUTE_PROFILE_CSV`， 当设定为 0 或 1 时，关闭或开启一个逗号分隔的测绘器输出。这个特性是为方便将文件导入一个电子表格。

`COMPUTE_PROFILE_CONFIG` 用来指定一个配置文件，文件里包含跟踪执行(收集时间线数据)以及收集硬件计数器的选项。跟踪执行的选项列表以及他们的释义都在随 CUDA 工具包一起提供的 CUDA 测绘器用户指南 (CUDA Profiler Users Guide) 里给出，CUDA 工具包也能在线获取。可以测绘的硬件计数器的列表可通过提交测绘工具命令 `nvprof--query-events` 获得。测绘工具 `nvprof` 随 CUDA 工具包一起发布。

B.1.3 及时编译

`CUDA_CACHE_DISABLE`, 当设定为 1 时，关闭缓存，这意味着没有二进制代码被添加到缓存，也没有二进制码被从缓存取出。

`CUDA_CACHE_MAXSIZE` 指定计算缓存的字节尺寸。默认是 32 MB，最大值是 4 GB.超过上限的二进制代码将不会被缓存，必要时会将旧的二进制代码清理出缓存。

`CUDA_CACHE_PATH` 控制计算缓存的位置。Linux 上缓存的默认位置，在 Linux 上是 `~/.nv/ComputeCache`, 在 MacOS 上是 `Support/NVIDIA/ComputeCache` ，在 Windows 上

是%APPDATA%\NVIDIA\ComputeCache.

CUDA_FORCE_PTX_JIT，当设定为 1 时，强制驱动忽略一个应用里内嵌的所有二进制代码，并及时编译内嵌的 PTX 代码。这个选项可以用来测试一个应用内是否内嵌 PTX 代码以及这些内嵌代码是否正常工作。如果这个环境变量设定为 1 且一个内核没有内嵌 PTX 代码，那么将载入失败。

B.2 nvidia-smi 系统管理接口

对系统上设备的其它控制能够通过系统管理接口工具 nvidia-smi 完成，所有 Linux 平台上的英伟达驱动都捆绑这个工具。nvidia-smi 的手册页包含所有选项的列表。本节只演示这个工具几个常见用法。

不带任何选项的 nvidia-smi 列出所有附属的英伟达设备的一些基本信息，如图 B.1 所示。尽管 nvidia-smi 列出所有设备，但它仅提供 Tesla 和高端 Quadro 设备的详细信息。表 B.1 中输出里列出的 Quadro NVS 285 不是一个高端卡，因此只提供了少量信息。

```
% nvidia-smi
Thu Apr 25 14:48:10 2013
+------------------------------------------------------+
| NVIDIA-SMI 4.304.52   Driver Version: 304.52         |
|-------------------------------+----------------------+----------------------+
| GPU  Name                     | Bus-Id        Disp.  | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage         | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K20                | 0000:80:00.0     Off |                    0 |
| 30%   30C    P8    16W / 225W |   0%   13MB / 4799MB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Quadro NVS 285           | 0000:60:00.0     N/A |                  N/A |
| N/A   60C  N/A     N/A /  N/A |  45%   55MB /  123MB |     N/A          N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Compute processes:                                               GPU Memory |
|  GPU       PID  Process name                                     Usage      |
|=============================================================================|
|    1            Not Supported                                               |
+-----------------------------------------------------------------------------+
% 
```

系统上设备的简单列表可以从 nvidia-smi -L 的输出中获得：

```
% nvidia -smi -L
GPU 0: Tesla K20 (S/N: 0324612033969)
GPU 1: Quadro NVS 285 (UUID: N/A)
```

B.2.1 打开和关闭 ECC

探测一个设备上的 ECC 是打开还是关闭的方法有好几个。可以用派生类型 cudaDeviceProp 的域 ECCEnabled 来查询当前的 ECC 状态，实用工具 pgiaccelinfo 也可显示所有附属设备的 ECC 是打开还是关闭。

从 nvidia-smi 可以获得有关 ECC 的更多详尽信息，也可以打开或关闭 ECC. 利用 nvidia-smi 查询 ECC 状态可以像下面这样做：

```
% nvidia -smi -i 0 -q -d ECC


==============  NVSMI LOG  ==============
```

```
Timestamp                           : Tue Apr 16 16:43:35 2013

Driver Version                      : 304.52

Attached GPUs                       : 2
GPU 0000:80:00.0
    Ecc Mode
        Current                     : Enabled
        Pending                     : Enabled
    ECC Errors
        Volatile
            Single Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total : 0
            Double Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total               : 0
        Aggregate
            Single Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total               : 0
            Double Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total               : 0
```

这里的设备 0 由选项`-i 0`指定，ECC 输出由选项`-d ECC`指定。这个命令的输出列出了不同内存类型的错误。1 位错误(single-bit errors)已被纠正；2 位错误(double-bit errors)不可纠正。临时错误计数器追踪自上次驱动加载以来的错误数量，并无限期保存这些错误数据。

这个输出的靠近顶部处显示了当前域和挂起域。重启或重置后，ECC 挂起模式当成为当前 ECC 模式。ECC

模式可像下面这样关闭(假设有 root 权限)：

```
% nvidia -smi -i 0 -e 0
Disabled ECC support for GPU 0000:80:00.0.
All done.
Reboot required.
```

此刻，nvidia-smi -i 0 -q -d ECC 打印出来的 ECC 状态是：

```
    Ecc Mode
        Current                     : Enabled
        Pending                     : Disabled
```

为使挂起的改变生效，需要重新启动机器，重启后 ECC 模式状态是：

```
    Ecc Mode
        Current                     : Disabled
        Pending                     : Disabled
```

B.2.2 计算模式

计算模式决定了多个主机进程或线程能否使用同一个 GPU. 四种计算模式 ,从最宽松到最严格 ,分别是：

默认：0 这种模式下多个主机线程可以使用同一个设备。

互斥线程：1 这种模式下，整个系统内只能有一个进程创建仅一个上下文，在某一时刻这个上下文至多是该进程的一个线程的当前上下文。

禁止：2 这种模式下，设备上不能创建上下文。

互斥进程：3 这种模式下，整个系统内只允许一个进程创建一个上下文，这个上下文可以是该进程的所有线程的当前上下文。

和 ECC 状态一样，可以使用派生类型 cudaDeviceProp 的域 computeMode 和实用工具 pgaccelinfo 来探明计算模式。利用 nvidia-smi，可以像下面这样查询计算模式：

```
% nvidia -smi -q -i 0 -d COMPUTE

============== NVSMI LOG ==============

Timestamp                           : Thu Apr 18 13:38:29 2013
Driver Version                      : 304.52

Attached GPUs                       : 2
GPU 0000:80:00.0
    Compute Mode                    : Default
```

输出表明设备 0 处于默认计算模式。可以选项-c 来改变计算模式(假设有 root 权限)：

```
% nvidia -smi -i 0 -c 1
Set compute mode to EXCLUSIVE_THREAD for GPU 0000:80:00.0.
All done.
```

计算模式的改变会立即生效：

```
% nvidia -smi -q -i 0 -d COMPUTE

============== NVSMI LOG ==============
```

```
Timestamp                           : Thu Apr 18 13:49:40 2013
Driver Version                      : 304.52

Attached GPUs                       : 2
GPU 0000:80:00.0
    Compute Mode : Exclusive_Thread
```

重启机器或重置设备之后，计算模式被重置为默认计算模式。

B.2.3 持续模式

当在一个 GPU 上打开持续模式，即使没有稍活动的客户，驱动依然保持初始化完成状态，从而当运行 CUDA 应用时驱动延时最小。在运行 X Window System 的系统上这不是个问题，因为 X Window 的客户总是活动的，但在不运行 X 的无头系统上，打开持续模式来避免启动 CUDA 应用时的重新初始化就变得非常重要。

持续模式默认关闭，并当设备重置或系统重启后恢复关闭状态。可以从 `nvidia-smi` 的一般查询输出中判定持续模式是否打开：

```
% nvidia-smi -q -i 0

============== NVSMI LOG ==============

Timestamp                           : Thu Apr 18 14:17:25 2013
Driver Version                      : 304.52

Attached GPUs                       : 2
GPU 0000:80:00.0
    Product Name                    : Tesla K20
    Display Mode                    : Disabled
    Persistence Mode                : Disabled
    ...
```

持续模式可以用 `nvidia-smi` 的选项`-pm` 打开(假设有 root 仅限)：

```
% nvidia-smi -i 0 -pm 1
Enabled persistence mode for GPU 0000:80:00.0.
All done.
```

附录 C 从 CUDA Fortran 中调用 CUDA C

There are several reasons one would want to call CUDA C code from CUDA Fortran: (1) to leverage code already written in CUDA C, especially libraries where an explicit CUDA Fortran interface is not available, and (2) to write CUDA C code that uses features that are not available in CUDA Fortran. We provide an example for each of these use cases in this appendix.

附录 D 源代码

CUDA Fortran source code that was deemed too long to include in its entirety in earlier chapters is listed in this appendix. Each section in this appendix contains all the relevant code, both host code and device code, for the particular application.

D.1 纹理内存

The following is the CUDA Fortran code used in Section 3.2.3 to discuss how textures can be advanta-geous in accessing neighboring data on a 2D mesh using four- and eight-point stencils:

D.2 矩阵转置

下在的完整矩阵转置代码在 3.4 节详细讲述。

```
!this program demonstates various memory optimzation techniques
!applied to a matrix transpose.

module dimensions_m

  implicit none

  integer , parameter :: TILE_DIM = 32
  integer , parameter :: BLOCK_ROWS = 8
  integer , parameter :: NUM_REPS = 100
  integer , parameter :: nx = 1024, ny = 1024
  integer , parameter :: mem_size = nx*ny*4

end module dimensions_m



module kernels_m

  use dimensions_m
  implicit none

contains

  ! copy kernel using shared memory
  !
  ! used as reference case

  attributes(global) subroutine copySharedMem(odata , idata)
```

```
30
31      real , intent(out) :: odata(nx,ny)
32      real , intent(in) :: idata(nx,ny)
33
34      real , shared :: tile(TILE_DIM , TILE_DIM)
35      integer :: x, y, j
36
37      x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
38      y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
39
40      do j = 0, TILE_DIM -1, BLOCK_ROWS
41         tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
42      end do
43
44      call syncthreads()
45
46      do j = 0, TILE_DIM -1, BLOCK_ROWS
47         odata(x,y+j) = tile(threadIdx%x, threadIdx%y+j)
48      end do
49    end subroutine copySharedMem
50
51    ! naive transpose
52    !
53    ! simplest transpose - doesn 't use shared memory
54    ! reads from global memory are coalesced but not writes
55
56    attributes(global) &
57         subroutine transposeNaive(odata , idata)
58
59      real , intent(out) :: odata(ny,nx)
60      real , intent(in) :: idata(nx,ny)
61
62      integer :: x, y, j
63
64      x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
65      y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
66
67      do j = 0, TILE_DIM -1, BLOCK_ROWS
68         odata(y+j,x) = idata(x,y+j)
69      end do
70    end subroutine transposeNaive
71
72    ! coalesced transpose
73    !
```

```
74    ! uses shared memory to achieve coalesing
75    ! in both reads and writes
76    !
77    ! tile size causes shared memory bank conflicts
78
79    attributes(global) &
80         subroutine transposeCoalesced(odata , idata)
81
82      real , intent(out) :: odata(ny,nx)
83      real , intent(in) :: idata(nx,ny)
84      real , shared :: tile(TILE_DIM , TILE_DIM)
85      integer :: x, y, j
86
87      x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
88      y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
89
90      do j = 0, TILE_DIM -1, BLOCK_ROWS
91         tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
92      end do
93
94      call syncthreads()
95
96      x = (blockIdx%y-1) * TILE_DIM + threadIdx%x
97      y = (blockIdx%x-1) * TILE_DIM + threadIdx%y
98
99      do j = 0, TILE_DIM -1, BLOCK_ROWS
100        odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
101     end do
102   end subroutine transposeCoalesced
103
104   ! no bank -conflict transpose
105   !
106   ! like transposeCoalesced except the first tile dim
107   ! is padded to avoid shared memory bank conflicts
108
109   attributes(global) &
110        subroutine transposeNoBankConflicts(odata , idata)
111
112     real , intent(out) :: odata(ny,nx)
113     real , intent(in) :: idata(nx,ny)
114     real , shared :: tile(TILE_DIM+1, TILE_DIM)
115     integer :: x, y, j
116
117     x = (blockIdx%x-1) * TILE_DIM + threadIdx%x
```

```
118       y = (blockIdx%y-1) * TILE_DIM + threadIdx%y
119
120       do j = 0, TILE_DIM -1, BLOCK_ROWS
121          tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
122       end do
123
124       call syncthreads()
125
126       x = (blockIdx%y-1) * TILE_DIM + threadIdx%x
127       y = (blockIdx%x-1) * TILE_DIM + threadIdx%y
128
129       do j = 0, TILE_DIM -1, BLOCK_ROWS
130          odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
131       end do
132    end subroutine transposeNoBankConflicts
133
134    ! Diagonal reordering
135    !
136    ! This version should be used on cards of CC 1.3
137    ! to avoid partition camping. It reschedules the
138    ! order in which blocks are executed so requests
139    ! for global memory access by active blocks are
140    ! spread evenly amongst partitions
141
142    attributes(global) &
143         subroutine transposeDiagonal(odata , idata)
144
145       real , intent(out) :: odata(ny,nx)
146       real , intent(in) :: idata(nx,ny)
147       real , shared :: tile(TILE_DIM+1, TILE_DIM)
148       integer :: x, y, j
149       integer :: blockIdx_x , blockIdx_y
150
151       if (nx==ny) then
152          blockIdx_y = blockIdx%x
153          blockIdx_x = &
154               mod(blockIdx%x+blockIdx%y-2,gridDim%x)+1
155       else
156          x = blockIdx%x + gridDim%x*(blockIdx%y-1)
157          blockIdx_y = mod(x-1,gridDim%y)+1
158          blockIdx_x = &
159               mod((x-1)/gridDim%y+blockIdx_y -1,gridDim%x)+1
160       endif
161
```

```
162       x = (blockIdx_x -1) * TILE_DIM + threadIdx%x
163       y = (blockIdx_y -1) * TILE_DIM + threadIdx%y
164
165       do j = 0, TILE_DIM -1, BLOCK_ROWS
166          tile(threadIdx%x, threadIdx%y+j) = idata(x,y+j)
167       end do
168
169       call syncthreads()
170
171       x = (blockIdx_y -1) * TILE_DIM + threadIdx%x
172       y = (blockIdx_x -1) * TILE_DIM + threadIdx%y
173
174       do j = 0, TILE_DIM -1, BLOCK_ROWS
175          odata(x,y+j) = tile(threadIdx%y+j, threadIdx%x)
176       end do
177     end subroutine transposeDiagonal
178
179   end module kernels_m
180
181
182
183   program transposeTest
184
185     use cudafor
186     use kernels_m
187     use dimensions_m
188
189     implicit none
190
191     type (dim3) :: grid , tBlock
192     type (cudaEvent) :: startEvent , stopEvent
193     type (cudaDeviceProp) :: prop
194     real :: time
195
196     real :: in_h(nx,ny), copy_h(nx,ny), trp_h(ny,nx)
197     real :: gold(ny,nx)
198     real , device :: in_d(nx,ny), copy_d(nx,ny), trp_d(ny,nx)
199
200     integer :: i, j, istat
201
202     ! check parameters and calculate execution configuration
203
204     if (mod(nx, TILE_DIM) /= 0 &
205          .or. mod(ny, TILE_DIM) /= 0) then
```

```
206         write(*,*) 'nx and ny must be a multiple of TILE_DIM'
207         stop
208      end if
209
210      if (mod(TILE_DIM , BLOCK_ROWS) /= 0) then
211         write(*,*) 'TILE_DIM must be a multiple of BLOCK_ROWS'
212         stop
213      end if
214
215      grid = dim3(nx/TILE_DIM , ny/TILE_DIM , 1)
216      tBlock = dim3(TILE_DIM , BLOCK_ROWS , 1)
217
218      ! write parameters
219
220      i = cudaGetDeviceProperties(prop , 0)
221      write(*,"(/,'Device Name: ',a)") trim(prop%name)
222      write(*,"('Compute Capability: ',i0,'.',i0)") &
223           prop%major , prop%minor
224
225
226      write(*,*)
227      write(*,"('Matrix size:', i5, i5, ', Block size:', &
228           i3, i3, ', Tile size:', i3, i3)") &
229           nx, ny, TILE_DIM , BLOCK_ROWS , TILE_DIM , TILE_DIM
230
231      write(*,"('grid:', i4,i4,i4, ', tBlock:', i4,i4,i4)") &
232           grid%x, grid%y, grid%z, tBlock%x, tBlock%y, tBlock%z
233
234      ! initialize data
235
236      ! host
237
238      do j = 1, ny
239         do i = 1, nx
240            in_h(i,j) = i+(j-1)*nx
241         enddo
242      enddo
243
244      gold = transpose(in_h)
245
246      ! device
247
248      in_d = in_h
249      trp_d = -1.0
```

```
250     copy_d = -1.0
251
252     ! events for timing
253
254     istat = cudaEventCreate(startEvent)
255     istat = cudaEventCreate(stopEvent)
256
257     ! ------------
258     ! time kernels
259     ! ------------
260
261     write(*,'(/,a25 ,a25)') 'Routine', 'Bandwidth (GB/s)'
262
263     ! -------------
264     ! copySharedMem
265     ! -------------
266
267     write(*,'(a25)', advance='NO') 'shared memory copy'
268
269     copy_d = -1.0
270     ! warmup
271     call copySharedMem <<<grid , tBlock >>>(copy_d , in_d)
272
273     istat = cudaEventRecord(startEvent , 0)
274     do i=1, NUM_REPS
275        call copySharedMem <<<grid , tBlock >>> (copy_d , in_d)
276     end do
277     istat = cudaEventRecord(stopEvent , 0)
278     istat = cudaEventSynchronize(stopEvent)
279     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
280
281     copy_h = copy_d
282     call postprocess(in_h , copy_h , time)
283
284     ! --------------
285     ! transposeNaive
286     ! --------------
287
288     write(*,'(a25)', advance='NO') 'naive transpose'
289
290     trp_d = -1.0
291     ! warmup
292     call transposeNaive <<<grid , tBlock >>>(trp_d , in_d)
293
```

```
294     istat = cudaEventRecord(startEvent , 0)
295     do i=1, NUM_REPS
296        call transposeNaive <<<grid , tBlock >>>(trp_d , in_d)
297     end do
298     istat = cudaEventRecord(stopEvent , 0)
299     istat = cudaEventSynchronize(stopEvent)
300     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
301
302     trp_h = trp_d
303     call postprocess(gold , trp_h , time)
304
305     ! ------------------
306     ! transposeCoalesced
307     ! ------------------
308
309     write(*,'(a25)', advance='NO') 'coalesced transpose'
310
311     trp_d = -1.0
312     ! warmup
313     call transposeCoalesced <<<grid , tBlock >>>(trp_d , in_d)
314
315     istat = cudaEventRecord(startEvent , 0)
316     do i=1, NUM_REPS
317        call transposeCoalesced <<<grid , tBlock >>>(trp_d , in_d)
318     end do
319     istat = cudaEventRecord(stopEvent , 0)
320     istat = cudaEventSynchronize(stopEvent)
321     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
322
323     trp_h = trp_d
324     call postprocess(gold , trp_h , time)
325
326     ! ------------------------
327     ! transposeNoBankConflicts
328     ! ------------------------
329
330     write(*,'(a25)', advance='NO') 'conflict -free transpose'
331
332     trp_d = -1.0
333     ! warmup
334     call transposeNoBankConflicts <<<grid , tBlock >>>(trp_d , in_d)
335
336     istat = cudaEventRecord(startEvent , 0)
337     do i=1, NUM_REPS
```

```
338       call transposeNoBankConflicts &
339            <<<grid , tBlock >>>(trp_d , in_d)
340     end do
341     istat = cudaEventRecord(stopEvent , 0)
342     istat = cudaEventSynchronize(stopEvent)
343     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
344
345     trp_h = trp_d
346     call postprocess(gold , trp_h , time)
347
348     ! ----------------
349     ! transposeDigonal
350     ! ----------------
351
352     write(*,'(a25)', advance='NO') 'diagonal transpose'
353
354     trp_d = -1.0
355     ! warmup
356     call transposeDiagonal <<<grid , tBlock >>>(trp_d , in_d)
357
358     istat = cudaEventRecord(startEvent , 0)
359     do i=1, NUM_REPS
360        call transposeDiagonal <<<grid , tBlock >>>(trp_d , in_d)
361     end do
362     istat = cudaEventRecord(stopEvent , 0)
363     istat = cudaEventSynchronize(stopEvent)
364     istat = cudaEventElapsedTime(time , startEvent , stopEvent)
365
366     trp_h = trp_d
367     call postprocess(gold , trp_h , time)
368
369     ! cleanup
370
371     write(*,*)
372
373     istat = cudaEventDestroy(startEvent)
374     istat = cudaEventDestroy(stopEvent)
375
376   contains
377
378     subroutine postprocess(ref , res , t)
379       real , intent(in) :: ref(:,:), res(:,:), t
380       if (all(res == ref)) then
381         write(*,'(f20.2)') 2.0* mem_size *1.0e-6/(t/NUM_REPS)
```

```
382       else
383          write(*,'(a20)') '*** Failed ***'
384       end if
385     end subroutine postprocess
386
387 end program transposeTest
```

D.3 线程级并行和指令级并行

下面的完整 CUDA Fortran 代码用来讲述 3.5.2 节中的线程级并行和指令级并行：

```
 1  ! This code demonstrates use of thread - and instruction -
 2  ! level parallelism and their effect on performance
 3
 4  module copy_m
 5    integer , parameter :: N = 1024*1024
 6    integer , parameter :: ILP=4
 7  contains
 8
 9    ! simple copy code that requires thread -level paralellism
10    ! to hide global memory latencies
11
12    attributes(global) subroutine copy(odata , idata)
13      use precision_m
14      implicit none
15      real(fp_kind) :: odata(*), idata(*), tmp
16      integer :: i
17
18      i = (blockIdx%x-1)* blockDim%x + threadIdx%x
19      tmp = idata(i)
20      odata(i) = tmp
21    end subroutine copy
22
23    ! copy code which uses instruction -level parallelism
24    ! in addition to thread -level parallelism to hide
25    ! global memory latencies
26
27    attributes(global) subroutine copy_ILP(odata , idata)
28      use precision_m
29      implicit none
30      real(fp_kind) :: odata(*), idata(*), tmp(ILP)
31      integer :: i,j
32
33      i = (blockIdx%x-1)* blockDim%x*ILP + threadIdx%x
34
```

```
 35      do j = 1, ILP
35      do j = 1, ILP
 36         tmp(j) = idata(i+(j-1)* blockDim%x)
 37      enddo
 38
 39      do j = 1, ILP
 40         odata(i+(j-1)* blockDim%x) = tmp(j)
 41      enddo
 42    end subroutine copy_ILP
 43
 44  end module copy_m
 45
 46  program parallelism
 47    use cudafor
 48    use precision_m
 49    use copy_m
 50
 51    implicit none
 52
 53    type(dim3) :: grid , threadBlock
 54    type(cudaEvent) :: startEvent , stopEvent
 55    type(cudaDeviceProp) :: prop
 56
 57    real(fp_kind) :: a(N), b(N)
 58    real(fp_kind), device :: a_d(N), b_d(N)
 59
 60    real :: time
 61    integer :: i, smBytes , istat
 62
 63
 64    istat = cudaGetDeviceProperties(prop , 0)
 65    write(*,"(/,'Device Name: ',a)") trim(prop%name)
 66    write(*,"('Compute Capability: ',i0,'.',i0)") &
 67         prop%major , prop%minor
 68    if (fp_kind == singlePrecision) then
 69       write(*,"('Single Precision ')")
 70    else
 71       write(*,"('Double Precision ')")
 72    end if
 73
 74    a = 1.0
 75    a_d = a
 76
 77    smBytes = prop%sharedMemPerBlock
```

```
78
79    istat = cudaEventCreate(startEvent)
80    istat = cudaEventCreate(stopEvent)
81
82    write(*,'(/,"Thread -level parallelism runs")')
83
84    write(*,'(/," Multiple Blocks per Multiprocessor")')
85    write(*,'(a20 ,a25)') 'Threads/Block', 'Bandwidth (GB/s)'
86
87    do i = prop%warpSize , prop%maxThreadsPerBlock , prop%warpSize
88       if (mod(N,i) /= 0) cycle
89
90       b_d = 0.0
91
92       grid = dim3(ceiling(real(N)/i),1,1)
93       threadBlock = dim3(i,1,1)
94
95       istat = cudaEventRecord(startEvent ,0)
96       call copy <<<grid , threadBlock >>>(b_d , a_d)
97       istat = cudaEventRecord(stopEvent ,0)
98       istat = cudaEventSynchronize(stopEvent)
99       istat = cudaEventElapsedTime(time , startEvent , stopEvent)
100
101      b = b_d
102      if (all(b==a)) then
103         write(*,'(i20 , f20.2)') &
104              i, 2.*1000* sizeof(a)/(1024**3*time)
105      else
106         write(*,'(a20)') '*** Failed ***'
107      end if
108   end do
109
110   write(*,'(/," Single Block per Multiprocessor")')
111   write(*,'(a20 ,a25)') 'Threads/Block', 'Bandwidth (GB/s)'
112
113   do i = prop%warpSize , prop%maxThreadsPerBlock , prop%warpSize
114      if (mod(N,i) /= 0) cycle
115
116      b_d = 0.0
117
118      grid = dim3(ceiling(real(N)/i),1,1)
119      threadBlock = dim3(i,1,1)
120
121      istat = cudaEventRecord(startEvent ,0)
```

```
122       call copy <<<grid , threadBlock , 0.9*smBytes >>>(b_d , a_d)
123       istat = cudaEventRecord(stopEvent ,0)
124       istat = cudaEventSynchronize(stopEvent)
125       istat = cudaEventElapsedTime(time , startEvent , stopEvent)
126
127       b = b_d
128       if (all(b==a)) then
129          write(*,'(i20 , f20.2)') i, 2.*sizeof(a)*1.0e-6/time
130       else
131          write(*,'(a20)') '*** Failed ***'
132       end if
133    end do
134
135    write(*,'(/,"Intruction -level parallelism runs")')
136
137    write(*,'(/," ILP=", i0, &
138          ", Single Block per Multiprocessor")') ILP
139    write(*,'(a20 ,a25)') 'Threads/Block', 'Bandwidth (GB/s)'
140
141    do i = prop%warpSize , prop%maxThreadsPerBlock , prop%warpSize
142       if (mod(N,i) /= 0) cycle
143
144       b_d = 0.0
145
146       grid = dim3(ceiling(real(N)/(i*ILP)),1,1)
147       threadBlock = dim3(i,1,1)
148
149       istat = cudaEventRecord(startEvent ,0)
150       call copy_ILP <<<grid , threadBlock , &
151                        0.9*smBytes >>>(b_d , a_d)
152       istat = cudaEventRecord(stopEvent ,0)
153       istat = cudaEventSynchronize(stopEvent)
154       istat = cudaEventElapsedTime(time , startEvent , stopEvent)
155
156       b = b_d
157       if (all(b==a)) then
158          write(*,'(i20 , f20.2)') i, 2.*sizeof(a)*1.0e-6/time
159       else
160          write(*,'(a20)') '*** Failed ***'
161       end if
162    end do
163
164 end program parallelism
```