CUDA 编程：基础与实践

樊哲勇 著

内容简介

CUDA 是目前最为流行的 GPU 高性能计算的开发工具之一。本书通过大量实例系统地讲述 CUDA 编程的重要方面。前 12 章通过一些简短的例子循序渐进地介绍 CUDA 编程的基础知识，主要包括 GPU 硬件和 CUDA 软件开发工具（第 1 章）、CUDA 中的线程组织（第 2 章）、CUDA 程序的基本框架与错误检测（第 3-4 章）、获得 GPU 加速的关键条件（第 5 章）、CUDA 中的内存组织与各种内存的合理使用（第 6-8 章）、原子函数（第 9 章）、线程束内的基本函数（第 10 章）、CUDA 流（第 11 章）、统一内存（第 12 章）等。后面三章是可选读的内容：第 13 章综合运用前述章节中的知识，用 CUDA 开发一个简单的分子动力学模拟程序；第 14 章介绍若干 CUDA 库（包括 Thrust、cuBLAS、cuSolver 和 cuRAND）的使用；第 15 章介绍 在 Python 语言中利用 PyCUDA 进行 GPU 计算的方法。

本书适合高等院校理工科专业的本科生和研究生以及其它任何对 CUDA 编程感兴趣的人士阅读。

作者简介

樊哲勇，1983 年生，2010 年获南京大学物理系理学博士学位，目前在芬兰的 Aalto 大学做博士后。主要研究方向为计算凝聚态物理。他从 2011 年开始学习 CUDA 编程，后用 CUDA 开发了高效的分子动力学模拟程 序 GPUMD、经验势拟合程序 GPUGA 以及量子输运模拟程序 GPUQT，在计算物理的主流期刊之一《Computer Physics Communications》发表多篇与计算物理问题的 GPU 加速有关的论文。

前　言

基于 CPU（Central Processing Unit；中央处理器）和 GPU（Graphics Processing Unit；图形处理器）的异构计算（heterogeneous computing）已逐步发展为高性能计算（high performance computing）领域的主流模式。很多超级计算机都大量使用了 GPU。CUDA（Compute Unified Device Architecture）作为 GPU 高性能计算的主要开发工具之一，已经在计算机、物理、化学、生物、材料等众多领域发挥了重要的作用。掌握 CUDA 编程也就意味着开辟了一条通往高性能计算的新道路。

本书通过大量实例循序渐进地介绍 CUDA 编程的语法知识、优化策略以及程序开发实践。除了第 15 章，本书所有源代码都可以通过作者为本书创建的 GitHub 仓库（https://github.com/brucefan1983/CUDA-Programming）获得。读者也可以针对该仓库提出问题（issues）与作者进行交流。第 15 章的源代码见由琪同学的 GitHub 仓库 https://github.com/YouQixiaowu/CUDA-Programming-with-Python。本书中的所有程序都在 Linux 平台通过测试，其中绝大部分的程序也能在 Windows 平台通过测试。我们会在适当的地方指出哪些程序无法在（作者的） Windows 平台通过测试。

本书是一本较理想的学习 CUDA 编程的入门读物。在计算机方面，读者需要掌握初步的 Linux 或 Windows 命令行操作技能，并具有一定的 C++ 语言编程基础。第 13 章的内容要求读者具有《大学物理》或《普通物理》基础。第 14 章的部分内容要求读者熟悉理工科大学本科的《线性代数》。第 15 章要求读者熟悉 Python 语言和其中的 NumPy 包。本书前 12 章需顺序阅读，后 3 章可选读，而且可以按任意次序阅读。最后要强调的是，本书不假定读者有并行编程的经验。

本书不是一个 CUDA 编程手册，不追求面面俱到，但力求做到由浅入深、循序渐进。截至作者交稿之日，最新版本（10.2）的《CUDA C++ Programming Guide》和《CUDA C++ Best Practices Guide》加起来有 400 多页，再加上 CUDA 工具箱中各种应用程序库和编程开发工具的文档，总页数可能上万。在本书两百多页的篇幅中想要做到面面俱到是不可能的。明确地说：

- 本书只涉及 CUDA C++ 编程和 Python 中的 PyCUDA 编程，不涉及其它异构编程语言，例如 OpenCL、OpenACC 和 CUDA Fortran。

- 关于 CUDA C++ 编程，本书不涉及动态并行（dynamic parallelism）、CUDA Graph、CUDA 与 OpenGL 和 Direct3D 的交互、纹理和表面内存的使用。
- 本书不涉及多 GPU 编程，只讨论单 GPU 编程，并且不涉及 OpenMP 和 MPI。
- 在众多性能分析器（profiler）中，我们将仅偶尔使用 nvprof，不使用其它可视化性能分析器。

本书的出版受到国家自然科学基金的支持，项目编号为 11974059，名称为《基于石墨烯及其它两维材料的柔性热电材料的多尺度模拟》。本书中相关程序的开发和测试使用了由 Aalto Science-IT project 和 Finland's IT Center for Science (CSC) 提供的计算资源与技术支持。

复旦大学的周麟祥教授于 2011 年在厦门大学开设的 CUDA 编程讲座让作者有幸较早地接触 CUDA 编程。厦门大学的博士后导师郑金成和王惠琼教授以及芬兰 Aalto 大学的博士后导师 Ari Harju 博士和 Tapio Ala-Nissila 教授在作者学习与使用 CUDA 的过程中给予了很大的支持。在此对以上老师表示由衷的感谢！

特别感谢苏州吉浦讯科技的技术团队。该团队的工程师们为本书的初稿指出了三百多个问题。该团队的 GPUSLady（网名）在《GPU 世界》论坛（https://bbs.gpuworld.cn/）和《GPU 编程开发技术》QQ 群（群号为 62833093）为作者解答了很多有关 CUDA 编程的问题。如果没有该团队的帮助，本书一定有很多错误。厦门大学的徐克同学和渤海大学的由琪同学先后为本书制作了若干插图。由琪同学还为本书贡献了第 15 章关于 PyCUDA 的内容。中国科学技术大学的黄翔同学、潍坊学院高性能计算中心的李延龙同学以及西安理工大学的范亚东同学帮助审阅了全部书稿。在此对以上同学一并表示感谢。虽然有众多朋友帮助审阅书稿，本书一定还有一些残留的错误，而它们都应归咎于作者。

本书从构思到完成大概花了一年半的时间。在这一年多的时间里，此书的写作占用了我很多本应该陪家人的时间。所以，我将此书献给妻子秦海霞、大女儿樊怀瑾和小女儿樊婉瑜。最后，感谢我的父母含辛茹苦地供我读书。

目　录

第 1 章 GPU 硬件与 CUDA 程序开发工具

1.1 GPU 硬件简介

GPU 是英文 Graphics Processing Unit 的首字母缩写，意为图形处理器。GPU 也常被称为显卡（Graphics Card）。与它对应的一个概念是 CPU，即 Central Processing Unit（中央处理器）的首字母缩写。

从十多年前起，GPU 的浮点数运算峰值就比同时期的 CPU 高一个量级；GPU 的内存带宽峰值也比同时期的 CPU 高一个量级。CPU 和 GPU 的显著区别是：一颗典型的 CPU 拥有少数几个快速的计算核心，而一颗典型的 GPU 拥有几百到几千个不那么快速的计算核心。所以，GPU 是靠众多的计算核心来获得相对较高的计算性能。GPU 中有更多的晶体管被用于算术逻辑单元，而不是数据缓存和流程控制。图 1.1 形象地说明了（非集成）GPU 和 CPU 在硬件架构上的显著区别。

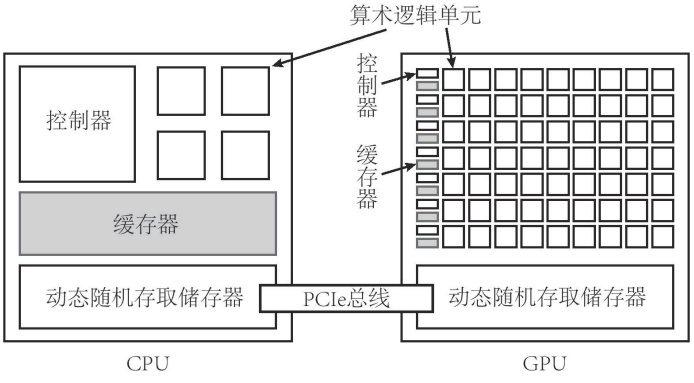

图 1.1: CPU（左）和 非集成 GPU（右）的硬件架构示意图。CPU 中有更多的晶体管用于数据缓存和流程控制，但 GPU 中有更多的晶体管用于算术逻辑单元。CPU 和 GPU 中都有 DRAM（动态随机存取内存），它们一般由 PCIe 总线连接。

GPU 计算不是指单独的 GPU 计算，而是指 CPU + GPU 的异构（heterogeneous）计算。一块单独的 GPU 是无法独立地完成所有计算任务的，它必须在 CPU 的调度下才能完成

特定任务。在由 CPU 和 GPU 构成的异构计算平台中，通常将起控制作用的 CPU 称为主机（host），将起加速作用的 GPU 称为设备（device）。主机和（非集成）设备都有自己的 DRAM（Dynamic Random-Access Memory；动态随机存取内存），它们之间一般由 PCIe 总线（Peripheral Component Interconnect express bus）连接，如图 1.1 所示。

本书中说的 GPU 都是指英伟达（Nvidia）公司推出的 GPU，因为 CUDA 编程目前只支持该公司的 GPU。以下几个系列的 GPU 都支持 CUDA 编程：

- Tesla 系列：其中的内存为纠错内存（Error-correcting code memory；简称为 ECC 内存），稳定性好，主要用于高性能、高强度的科学计算。
- Quadro 系列：支持高速 OpenGL（Open Graphics Library）渲染，主要用于专业绘图设计。
- GeForce 系列：主要用于游戏与娱乐，但也常用于科学计算。GeForce 系列的 GPU 没有纠错内存，用于科学计算时具有一定的风险。然而，GeForce 系列的 GPU 价格相对低廉、性价比高，用于学习 CUDA 编程是没有任何问题的。即使是笔记本电脑中 GeForce 系列的 GPU 也可以用来学习 CUDA 编程。
- Jetson 系列：嵌入式设备中的 GPU。作者对此无使用经验；本书也不专门讨论。

每一款 GPU 都有一个用以表示其“计算能力”（compute capability）的版本号。该版本号可以写为形如 X.Y 的形式。其中，X 表示主版本号，Y 表示次版本号。版本号决定了 GPU 硬件所支持的功能，为应用程序在运行时判断硬件特征提供依据。初学者往往误以为 GPU 的计算能力越高，性能就越高，但后面我们会看到，计算能力和性能没有简单的正比关系。

版本号越大的 GPU 架构（architecture）越新。主版本号与 GPU 的核心架构相关联。很有意思的是，英伟达选择用著名科学家（到目前为止，大部分是物理学家）的姓氏作为 GPU 核心架构的代号，见表 1.1。在主版本号相同时，具有较大次版本号的 GPU 的架构稍有更新。例如，同属于开普勒架构的 Tesla K40 和 Tesla K80 这两款 GPU 有相同的主版本号（X = 3），但有不同的次版本号：它们的计算能力分别是 3.5 和 3.7。注意：Tesla 既是第一代 GPU 架构的代号，也是科学计算系列 GPU 的统称，其具体含义要根据上下文确定。另外，计算能力 7.5 的架构虽然和伏特架构具有同样的主版本号（X = 7），但它一般被看作一个新的主要架构，代号为图灵（Turing）。据传，下一代 GPU 架构（X = 8）的代号为安培（Ampere）。表 1.2 列出了不同架构的各种 GPU 的名称。

特斯拉和费米架构的 GPU 已不再受到最近几个 CUDA 版本的支持。本书将忽略任何特定于这两个架构的硬件功能。可以预见，开普勒架构的 GPU 也将很快（比如一两年后）不受最新版 CUDA 的支持。为简洁起见，本书有时候也将忽略某些开普勒架构的特征。为

表 1.1: 各个 GPU 主计算能力的架构代号与发布年份。

主计算能力	架构代号	发布年份
X = 1	特斯拉（Tesla）	2006
X = 2	费米（Fermi）	2010
X = 3	开普勒（Kepler）	2012
X = 5	麦克斯韦（Maxwell）	2014
X = 6	帕斯卡（Pascal）	2016
X = 7	伏特（Volta）	2017
X.Y = 7.5	图灵（Turing）	2018

表 1.2: 当前常用的各种 GPU 的名称。特斯拉和费米架构的 GPU 已经不再受到最新 CUDA 的支持，故没有列出。

架构	Tesla 系列	Quadro 系列	GeForce 系列	Jetson 系列
开普勒	Tesla K 系列	Quadro K 系列	GeForce 600/700 系列	Tegra K1
麦克斯韦	Tesla M 系列	Quadro M 系列	GeForce 900 系列	Tegra X1
帕斯卡	Tesla P 系列	Quadro P 系列	GeForce 1000 系列	Tegra X2
伏特	Tesla V 系列	无	无	AGX Xavier
图灵	Tesla T 系列	Quadro RTX 系列	GeForce 2000 系列	AGX Xavier

简单起见，我们在表 1.2 中忽略了一类被称作 Titan 的 GPU。读者可以在如下网站查询任何一款支持 CUDA 的 GPU 的信息：http://developer.nvidia.com/cuda-gpus。

计算能力并不等价于计算性能。例如，GeForce RTX 2000 系列的计算能力高于 Tesla V100，但后者在很多方面性能更高（售价也高得多）。

表征计算性能的一个重要参数是浮点数运算峰值，即每秒最多能执行的浮点数运算次数，英文为 Floating-point operations per second，缩写为 FLOPS。GPU 的浮点数运算峰值在 10^{12} FLOPS，即 teraFLOPS（简写为 TFLOPS) 的量级。浮点数运算峰值有单精度和双精度之分。对 Tesla 系列的 GPU 来说，双精度浮点数运算峰值一般是单精度浮点数运算峰值的 1/2 左右（对计算能力为 3.5 和 3.7 的 GPU 来说，是 1/3 左右）。对 GeForce 系列的 GPU 来说，双精度浮点数运算峰值一般是单精度浮点数运算峰值的 1/32 左右。另一个影响计算性能的参数是 GPU 中的内存带宽（memory bandwidth）。GPU 中的内存常称为显存。最后，显存容量也是制约应用程序性能的一个因素。如果一个应用程序需要的显存数量超过了一个 GPU 的显存容量，在不使用统一内存（见第 12 章）的情况下程序就无法正确运行。表 1.3 列出了作者目前能够使用的几款 GPU 的主要性能指标。在浮点数运算峰值

一栏中，括号前和括号中的数字分别对应双精度和单精度的情形。

表 1.3: 若干 GPU 的主要性能指标。

GPU 型号	计算能力	显存容量	显存带宽	浮点数运算峰值
Tesla K40	3.5	12 GB	288 GB/s	1.4 (4.3) TFLOPS
Tesla P100	6.0	16 GB	732 GB/s	4.7 (9.3) TFLOPS
Tesla V100	7.0	32 GB	900 GB/s	7 (14) TFLOPS
GeForce RTX 2070 （笔记本）	7.5	8 GB	448 GB/s	0.2 (6.5) TFLOPS
GeForce RTX 2080ti	7.5	11 GB	616 GB/s	0.4 (13) TFLOPS

1.2 CUDA 程序开发工具

以下几种软件开发工具都可以用来进行 GPU 编程：

- CUDA（Compute Unified Device Architecture）。这是本书的主题。
- OpenCL（Open Computing Language）。这是一个更为通用的为各种异构平台编写并行程序的框架，也是 AMD 公司的 GPU 的主要程序开发工具。本书不涉及 OpenCL 编程，对此感兴趣的读者可参考刘文志、陈轶和吴长江合著的《OpenCL 异构并行计算：原理、机制与优化实践》（机械工业出版社）。
- OpenACC。这是一个由多个公司共同开发的异构并行编程标准。本书也不涉及 OpenACC 编程，对此感兴趣的读者可参考何沧平所著《OpenACC 并行编程实战》（机械工业出版社）。

CUDA 编程语言最初主要是基于 C 语言的，但目前越来越多地支持 C++ 语言。还有基于 Fortran 的 CUDA Fortran 版本以及由其它编程语言包装的 CUDA 版本，但本书只涉及基于 C++ 的 CUDA 编程和在 Python 中利用 PyCUDA 包的 CUDA 编程（第 15 章）。我们称基于 C++ 的 CUDA 编程语言为 CUDA C++。对 Fortran 版本感兴趣的读者可以参看网页 https://www.pgroup.com/。用户可以免费下载支持 CUDA Fortran 编程的 PGI 开发工具套装的社区版本（Community Edition）。对应的还有收费的专业版本（Professional Edition）。PGI 是高性能计算编译器公司 Portland Group, Inc. 的简称，已被英伟达公司收购。

CUDA 提供了两层 API（Application Programming Interface; 应用程序编程接口）给程序员使用：

- CUDA 驱动（driver）API
- CUDA 运行时（runtime）API

其中，CUDA 驱动 API 是更加底层的 API，它为程序员提供了更为灵活的编程接口；CUDA 运行时 API 是在 CUDA 驱动 API 的基础上构建的一个更为高级的 API，更容易使用。这两种 API 在性能上几乎没有差别。从程序的可读性来看，使用 CUDA 运行时 API 是更好的选择。在其它编程语言中使用 CUDA 的时候，驱动 API 很多时候是必需的。因为作者没有使用驱动 API 的经验，故本书只涉及 CUDA 运行时 API。

图 1.2 展示了 CUDA 开发环境的主要组件。开发的应用程序是以主机（CPU）为出发点的。应用程序可以调用 CUDA 运行时 API、CUDA 驱动 API、以及一些已有的 CUDA 库。所有这些调用都将利用设备（GPU）的硬件资源。对 CUDA 运行时 API 的介绍是本书大部分章节的重点内容；第 14 章将介绍若干常用的 CUDA 库。

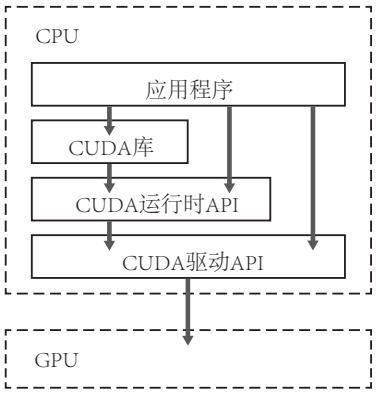

图 1.2: CUDA 编程开发环境概览。

CUDA 版本也由形如 X.Y 的两个数字表示，但它并不等同于 GPU 的计算能力。可以这样理解：CUDA 版本是 GPU 软件开发平台的版本，而计算能力对应着 GPU 硬件架构的版本。

最早的 CUDA 1.0 于 2007 年发布。当前（笔者交稿之日）最新的版本是 CUDA 10.2。CUDA 版本与 GPU 的最高计算能力都在逐年上升。虽然它们之间没有严格的对应关系，但一个具有较高计算能力的 GPU 通常需要一个较高的 CUDA 版本才能支持。最近的几个 CUDA 版本对计算能力的支持情况见表 1.4。一般来说，建议安装一个支持所用 GPU 的较新的 CUDA 工具箱。本书中的所有示例程序都可以在 CUDA 9.0-10.2 中进行测试。目前最新版本的 CUDA 10.2 有两个值得注意的地方。第一，它是最后一个支持 macOS 系统的 CUDA 版本。第二，它将 CUDA C 改名为 CUDA C++，用以强调 CUDA C++ 是基于 C++ 的扩展。

表 1.4: 最近的几个 CUDA 版本对 GPU 计算能力的支持情况。

CUDA 版本	所支持 GPU 的计算能力	架构
10.0-10.2	3.0-7.5	从开普勒到图灵
9.0-9.2	3.0-7.2	从开普勒到伏特
8.0	2.0-6.2	从费米到帕斯卡
7.0-7.5	2.0-5.3	从费米到麦克斯韦

1.3 CUDA 开发环境搭建示例

下面叙述作者最近在装有 GeForce RTX 2070 的笔记本中搭建 CUDA 开发环境的大致过程。因为作者的笔记本预装了 Windows 10 操作系统，所以我们以 Windows 10 操作系统为例进行讲解。因为 Linux 发行版有多种，故本书不列出在 Linux 安装 CUDA 开发环境的步骤。读者可参阅 Nvidia 的官方文档：https://docs.nvidia.com/cuda/cuda-installation-guide-linux。

我们说过，GPU 计算实际上是 CPU+GPU（主机 + 设备）的异构计算。在 CUDA C++ 程序中，既有运行于主机的代码，也有运行于设备的代码。其中，运行于主机的代码需要由主机的 C++ 编译器编译和链接。所以，除了安装 CUDA 工具箱，还需要安装一个主机的 C++ 编译器。在 Windows 中，最常用的 C++ 编译器是 Microsoft Visual C++ （MSVC），它目前集成在 Visual Studio 中。所以我们首先安装 Visual Studio。作者安装了最高版本的 Visual Studio 2019 16.x。因为这是个人使用的，故选择了免费的 Community 版本。下载地址为 https://visualstudio.microsoft.com/free-developer-offers/。对于 CUDA C++ 程序开发来说，只需要选择安装 Desktop development with C++ 即可。当然，读者也可以选择安装更多的组件。

关于 CUDA，作者选择安装 2019 年八月发布的 CUDA Toolkit 10.1 update2。首先，进入如下网址 https://developer.nvidia.com/cuda-10.1-download-archive-update2。然后根据提示，做如下选择：Operating System 选 Windows；Architecture 选 `x86_64`；Version 选操作系统版本，我们这里是 10；Installer Type 可以选 exe (network) 或者 exe (local)，分别代表一边下载一边安装和下载完毕后安装。接着，运行安装程序，根据提示一步一步安装即可。该版本的 CUDA 工具箱包含一个对应版本的 Nvidia driver，故不需要再单独安装 Nvidia driver。

安装好 Visual Studio 和 CUDA 后，进入到如下目录（读者如果找不到 C 盘下的 ProgramData 目录，可能是因为没有选择显示一些隐藏的文件）：

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.1\1_Utilities\deviceQuery
```

然后，用 Visual Studio 2019 打开文件 `deviceQuery_vs2019.sln`。接下来，编译（构建）、

运行。若输出内容的最后部分为有 `Result = PASS`，则说明已经搭建好 Windows 中的 CUDA 开发环境。若有疑问，请参阅 Nvidia 的官方文档：https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows。

在上面的测试中，我们是直接用 Visual Studio 打开一个已有的解决方案（solution），然后直接构建并运行。本书不介绍 Visual Studio 的使用，而是选择用命令行解释器编译与运行程序。这里的命令行解释器指的是 Linux 中的 terminal 或者 Windows 中的 command prompt 程序。在 Windows 中使用 MSVC 作为 C++ 程序的编译器时，需要单独设置相应的环境变量，或者从 Windows 的开始（start）菜单中找到 Visual Studio 2019 文件夹，然后点击其中的“x64 Native Tools Command Prompt for VS 2019”，而从打开一个加载了 MSVC 环境变量的命令行解释器。在本书的某些章节，需要有管理员的权限来使用 nvprof 性能分析器。此时，可以右键单击“x64 Native Tools Command Prompt for VS 2019”，然后选择“更多”，接着选择“以管理员身份运行”。

用命令行解释器编译与运行 CUDA 程序的方式在 Windows 和 Linux 系统几乎没有区别，但为简洁起见，本书后面主要以 Linux 开发环境为例进行讲解。虽然如此，Windows 和 Linux 中的 CUDA 编程功能还是稍有差别。我们将在后续章节中适当的地方指出这些差别。

1.4 用 nvidia-smi 检查与设置设备

可以通过 `nvidia-smi`（Nvidia’s system management interface）程序检查与设置设备。它包含在 CUDA 开发工具套装内。该程序最基本的用法就是在命令行解释器中使用不带任何参数的命令 `nvidia-smi`。在作者的笔记本中使用该命令，得到如下文本形式的输出：

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 426.00       Driver Version: 426.00       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce RTX 207... WDDM  | 00000000:01:00.0 Off |                  N/A |
| N/A   38C    P8    12W /  N/A |    161MiB /  8192MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

从中可以看出一些比较有用的信息：

- 第一行可以看到 Nvidia driver 的版本以及 CUDA 工具箱的版本。
- 作者所用计算机中有一型号为 GeForce RTX 2070 的 GPU。该 GPU 的设备号是 0。该计算机仅有一个 GPU。如果有多个 GPU，会将各个 GPU 从 0 开始编号。如果读者的系统中有多个 GPU，而且只需要使用某个特定的 GPU（比如两个之中更强大的那个），则可以通过设置环境变量 `CUDA_VISIBLE_DEVICES` 的值在运行 CUDA 程序之前选定一个 GPU。假如读者的系统中有编号为 0 和 1 的两个 GPU，而读者想在 1 号 GPU 运行 CUDA 程序，则可以用如下命令设置环境变量：

```
$ export CUDA_VISIBLE_DEVICES=1
```

  这样设置的环境变量在当前 shell session 及其子进程中有效。

- 该 GPU 处于 WDDM（Windows Display Driver Model）模式。另一个可能的模式是 TCC（Tesla Compute Cluster），但它仅在 Tesla、Quadro 和 Titan 系列的 GPU 中可选。可用如下方式选择（在 Windows 中需要用管理员身份打开 Command Prompt 并去掉命令中的 sudo）：

```
$ sudo nvidia-smi -g GPU_ID -dm 0 # 设置为 WDDM 模式
$ sudo nvidia-smi -g GPU_ID -dm 1 # 设置为 TCC 模式
```

  这里，`GPU_ID` 是 GPU 的编号。

- 该 GPU 当前的温度为 38 摄氏度。GPU 在满负荷运行时，温度会高一些。
- 这是 GeForce 系列的 GPU，没有 ECC 内存，故 Uncorr. ECC 为 N/A，代表不适用（not applicable）或者不存在（not available）。
- Compute M. 指计算模式（compute mode）。该 GPU 的计算模式是 Default。在默认模式中，同一个 GPU 中允许存在多个计算进程，但每个计算进程对应程序的运行速度一般来说会降低。还有一种模式为 E. Process，指的是独占进程模式（exclusive process mode），但不适用于处于 WDDM 模式的 GPU。在独占进程模式下，只能运行一个计算进程独占该 GPU。可以用如下命令设置计算模式（在 Windows 中需要用管理员身份打开 Command Prompt 并去掉命令中的 sudo）：

```
$ sudo nvidia-smi -i GPU_ID -c 0 # 默认模式
$ sudo nvidia-smi -i GPU_ID -c 1 # 独占进程模式
```

这里，-i GPU_ID 的意思是希望该设置仅仅作用于编号为 GPU_ID 的 GPU；如果去掉该选项，该设置将会作用于系统中所有的 GPU。

关于 nvidia-smi 程序更多的介绍，请参考如下官方文档：https://developer.nvidia.com/nvidia-system-management-interface。

1.5 其它学习资料

本书将循序渐进地带领读者学习 CUDA C++ 编程的基础知识。虽然本书力求自给自足，但读者在阅读本书的过程中同时参考一些其它的学习资料是有好处的。

任何关于 CUDA 编程的书籍都不可能替代官方提供的手册等资料。以下是几个重要的官方文档，请读者在有一定的基础之后务必查阅。限于作者水平，本书难免存在谬误。当读者觉得本书中的个别论断与官方资料有冲突时，当以官方资料为标准（官方手册的入口网址为：https://docs.nvidia.com/cuda）。在这个网站，包括但不限于以下几个方面的文档：

- 安装指南（Installation Guides）。读者遇到与 CUDA 安装有关的问题时，应该仔细阅读此处的文档。
- 编程指南（Programming Guides）。该部分有很多重要的文档：
  - 最重要的文档是《CUDA C++ Programming Guide》https://docs.nvidia.com/cuda/cuda-c-programming-guide。
  - 另一个值得一看的文档是《CUDA C++ Best Practices Guide》https://docs.nvidia.com/cuda/cuda-c-best-practices-guide。
  - 针对最近的几个 GPU 架构进行优化的指南，包括
    * https://docs.nvidia.com/cuda/kepler-tuning-guide
    * https://docs.nvidia.com/cuda/maxwell-tuning-guide
    * https://docs.nvidia.com/cuda/pascal-tuning-guide
    * https://docs.nvidia.com/cuda/volta-tuning-guide
    * https://docs.nvidia.com/cuda/turing-tuning-guide

    这几个简短的文档可以帮助有经验的用户迅速了解一个新的架构。
- CUDA API 手册（CUDA API References）。这里有
  - CUDA 运行时 API 手册：https://docs.nvidia.com/cuda/cuda-runtime-api
  - CUDA 驱动 API 手册：https://docs.nvidia.com/cuda/cuda-driver-api

  - CUDA 数学函数库 API 手册：https://docs.nvidia.com/cuda/cuda-math-api
  - 其它若干 CUDA 库的手册。

为明确起见，在撰写本书时，作者参考的是与 CUDA 10.2 对应的官方手册。

在学习 CUDA 编程的过程中如果遇到了某些难以解决的问题，可以考虑去论坛或者交流群求助。以下是几个比较有用的学习资源：

- 国内方面，作者推荐如下网站与交流群：
  - 《GPU 世界》论坛：https://bbs.gpuworld.cn
  - 《GPU 编程开发技术》QQ 群：62833093
  - 《CUDA 100%》QQ 群：195055206
  - 《CUDA Professional》QQ 群：45157483
- 国际方面，作者推荐如下网站：
  - 英伟达官方的开发者博客：https://devblogs.nvidia.com
  - Stack Overflow 问答网站：https://stackoverflow.com

第 2 章 CUDA 中的线程组织

我们以最简单的 CUDA 程序：从 GPU 中输出 Hello World! 字符串开始 CUDA 编程的学习。经典的 Hello World 程序几乎是学习任何一门新编程语言的出发点。学会了 Hello World 程序的开发过程，就对一个新的编程语言有了一个初步的认识。

本书的所有范例都是基于 Linux 操作系统开发的，但大部分也在 Windows 操作系统中使用 Command Prompt 命令行通过测试。因此，读者需要掌握基本的 Linux 或 Windows 命令行操作知识。

2.1 C++ 语言中的 Hello World 程序

学习 CUDA C++ 编程需要读者比较熟练地掌握 C++ 编程的基础。虽然 CUDA 支持很多 C++ 的特征，但作者写的 C++ 程序有很多 C 程序的痕迹，而且本书基本上不涉及 C++ 中的类和模板等编程特征。

我们先回顾一下 C++ 中 Hello World 程序的开发过程。在 C++ 语言中开发一个程序的大致过程是

1. 用文本编辑器写一个源代码（source code）。
2. 用编译器对源代码进行预处理、编译、汇编并链接必要的目标文件得到可执行文件（executable）。这些步骤往往可由一个命令完成。
3. 运行可执行文件得到结果。

Listing 2.1: 本章程序 hello.cpp 中的内容。

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World!\n");
```

```
6     return 0;
7 }
```

首先，让我们用编辑器写下 Listing 2.1 中的源代码。然后，将程序的文件命名为 hello.cpp，并用 g++ 编译（如上所述，此处以及后面所说的编译其实包含了预处理、编译、汇编、链接等步骤）：

```
$ g++ hello.cpp
```

编译通过后，将得到一个名为 a.out 的可执行文件。用如下命令执行该文件：

```
$ ./a.out
```

接着，就可以看到屏幕上打印出如下文字：

```
Hello World!
```

也可以在编译的时候指定二进制文件的名字。例如，用如下命令

```
$ g++ hello.cpp -o hello
```

将得到一个名为 hello 的可执行文件，可以用如下命令运行它

```
$ ./hello
```

以上假定使用了 GCC 编译器套装。如果使用 Windows 下的 MSVC 编译器套装，则可用 cl 编译程序：

```
$ cl hello.cpp
```

这将产生一个名为 hello.exe 的可执行文件。

2.2 CUDA 中的 Hello World 程序

在复习了 C++ 语言中的 Hello World 程序之后，我们接着介绍 CUDA 中的 Hello World 程序。

2.2.1 只有主机函数的 CUDA 程序

其实，我们已经写好了一个 CUDA 中的 Hello World 程序。这是因为，CUDA 程序的编译器驱动（compiler driver）nvcc 支持编译纯粹的 C++ 代码。一般来说，一个标准的 CUDA 程序中既有纯粹的 C++ 代码，也有不属于 C++ 的真正的 CUDA 代码。CUDA 程序的编译器

驱动 nvcc 在编译一个 CUDA 程序时，会将纯粹的 C++ 代码交给 C++ 的编译器（比如前面提到的 g++ 或 cl）去处理，它自己则负责编译剩下的部分。CUDA 程序源文件的后缀名默认是 .cu，所以我们可以将上面写好的源文件更名为 hello1.cu，然后用 nvcc 编译：

```
$ nvcc hello1.cu
```

编译好之后即可运行。运行结果与 C++ 程序的运行结果一样。关于 CUDA 程序的编译过程，将在本章最后一节以及后续的某些章节详细讨论，现在只要知道可以用 nvcc 编译 CUDA 程序即可。

2.2.2 使用核函数的 CUDA 程序

虽然上面的第一个版本是由 CUDA 的编译器编译的，但程序中根本没有使用 GPU。下面来介绍一个使用 GPU 的 Hello World 程序。

首先，我们要知道，GPU 只是一个设备，要它工作的话还需要有一个主机给它下达命令。这个主机就是 CPU。所以，一个真正利用了 GPU 的 CUDA 程序既有主机代码（在程序 hello1.cu 中的所有代码都是主机代码），也有设备代码（可以理解为需要设备执行的代码）。主机对设备的调用是通过核函数（kernel function）来实现的。所以，一个典型的、简单的 CUDA 程序的结构具有下面的形式：

```
int main(void)
{
    主机代码
    核函数的调用
    主机代码
    return 0;
}
```

CUDA 中的核函数与 C++ 中的函数是类似的，但一个显著的差别是：它必须被限定词（qualifier）__global__ 修饰。其中 global 前后是双下划线。另外，核函数的返回类型必须是空类型，即 void。这两个要求读者先记住即可。关于核函数的更多细节，以后再逐步深入介绍。遵循这两个要求，我们先写一个打印字符串的核函数：

```
__global__ void hello_from_gpu()
{
    printf("Hello World from the GPU!\n");
}
```

限定符 __global__ 和 void 的次序可随意。也就是说，上述核函数也可以写为：

```
void __global__ hello_from_gpu()
{
    printf("Hello World from the GPU!\n");
}
```

就像 C++ 语言中的函数要被调用才能发挥作用一样，这个核函数也要被调用才能发挥作用。下面，我们就写一个主函数来调用这个核函数，得到如 Listing 2.2 所示的完整 CUDA 程序。我们可以用如下命令编译：

```
$ nvcc hello2.cu
```

然后运行得到的可执行文件就可从屏幕上看到如下输出：

```
Hello World from the GPU!
```

Listing 2.2: 本章程序 hello2.cu 中的内容。

```
1  #include <stdio.h>
2
3  __global__ void hello_from_gpu()
4  {
5      printf("Hello World from the GPU!\n");
6  }
7
8  int main(void)
9  {
10     hello_from_gpu<<<1, 1>>>();
11     cudaDeviceSynchronize();
12     return 0;
13 }
```

上述程序有三个地方需要进一步解释：

- 先看看调用核函数的格式：

```
hello_from_gpu<<<1, 1>>>();
```

这个调用格式与普通的 C++ 函数的调用格式是有区别的。我们看到，在函数名 `hello_from_gpu` 和括号 () 之间有一对三括号 `<<<1, 1>>>`，里面还有用逗号隔开的两个数字。调用核函数时为什么需要这对三括号里面的信息呢？这是因为，一块 GPU 中有很多（例如，Tesla V100 中有 5120 个）计算核心，从而可以支持很多线程（thread）。主机在调用一个核函数时，必须指明需要在设备中指派多少个线程，不然设备不知道如何工作。三括号中的数就是用来指明核函数中的线程数目以及排列情况的。核函数中的线程常组织为若干线程块（thread block）：三括号中的第一个数字可以看做线程块的个数，第二个数字可以看做每个线程块中的线程数。一个核函数的全部线程块构成一个网格（grid），而线程块的个数就记为网格大小（grid size）。每个线程块中含有同样数目的线程，该数目叫做线程块大小（block size）。所以，核函数中总的线程数就等于网格大小乘以线程块大小，而三括号中的两个数字分别就是网格大小和线程块大小，即 `<<<网格大小，线程块大小>>>`。所以，在上述程序中，主机只指派了设备的一个线程，网格大小和线程块大小都是 1，即 $1 \times 1 = 1$。

- 核函数中的 `printf()` 函数的使用方式和 C++ 库（或者说 C++ 从 C 中继承的库）中的 `printf()` 函数的使用方式基本上是一样的。而且在核函数中使用 `printf()` 函数时也需要包含头文件 `<stdio.h>`（也可以写成 `<cstdio>`）。要注意的是，核函数中不支持 C++ 的 iostream（读者可亲自测试）。

- 我们注意到，在调用核函数之后，有如下一行：

  ```
  cudaDeviceSynchronize();
  ```

  这行语句调用了一个 CUDA 的运行时 API 函数。去掉这个函数就打印不出字符串了（请读者亲自尝试）。这是因为调用输出函数时，输出流是先存放在缓冲区的，而缓冲区不会自动刷新。只有程序遇到某种同步操作时缓冲区才会刷新。函数 `cudaDeviceSynchronize` 的作用是同步主机与设备，所以能够促使缓冲区刷新。读者现在不需要弄明白这个函数到底是什么，因为我们这里的主要目的是介绍 CUDA 中的线程组织。

2.3 CUDA 中的线程组织

2.3.1 使用多个线程的核函数

核函数中允许指派很多线程。这是一个必然的特征。这是因为，一个 GPU 往往有几千个计算核心，而总的线程数必须至少等于计算核心数时才有可能充分利用 GPU 中的全部计算资源。实际上，总的线程数大于计算核心数时才能更充分地利用 GPU 中的计算资源，因

为这会让计算和内存访问之间以及不同的计算之间合理地重叠，从而减小计算核心空闲的时间。

所以，根据需要，在调用核函数时可以指定使用多个线程。Listing 2.3 所示程序在调用核函数 `hello_from_gpu` 时指定了一个含有 2 个线程块的网格，而且每个线程块的大小是 4。

Listing 2.3: 本章程序 hello3.cu 中的内容。

```
#include <stdio.h>

__global__ void hello_from_gpu()
{
    printf("Hello World from the GPU!\n");
}

int main(void)
{
    hello_from_gpu<<<2, 4>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

因为网格大小是 2，线程块大小是 4，故总的线程数是 $2 \times 4 = 8$。也就是说，该程序中的核函数调用将指派 8 个线程。核函数中代码的执行方式是“单指令-多线程”，即每一个线程都执行同一串指令。既然核函数中的指令是打印一个字符串，那么编译、运行上述程序，将在屏幕打印如下 8 行同样的文字：

```
Hello World from the GPU!
```

其中，每一行对应一个指派的线程。读者也许要问，每一行分别是哪一个线程输出的呢？下面就来讨论这个问题。

2.3.2 使用线程索引

通过前面的介绍，我们知道，可以为一个核函数指派多个线程，而这些线程的组织结构是由执行配置（execution configuration）：

```
<<<grid_size, block_size>>>
```

决定的。这里的 grid_size（网格大小）和 block_size（线程块大小）一般来说是一个结构体类型的变量，但也可以是一个普通的整型变量。我们先考虑简单的整型变量，稍后再介绍更一般的情形。这两个整型变量的乘积就是被调用核函数中总的线程数。

我们强调过，本书不关心古老的特斯拉和费米架构。从开普勒架构开始，最大允许的线程块大小是 1024，而最大允许的网格大小是 $2^{31}-1$（针对这里的一维网格来说；后面介绍的多维网格能够定义更多的线程块）。所以，用上述简单的执行配置时最多可以指派大约两万亿个线程。这通常是远大于一般的编程问题中常用的线程数目的。一般来说，只要线程数比 GPU 中的计算核心数（几百至几千个）多几倍时，就有可能充分地利用 GPU 中的全部计算资源。总之，一个核函数允许指派的线程数目是巨大的，能够满足几乎所有应用程序的要求。需要指出的是，一个核函数中虽然可以指派如此巨大数目的线程数，但在执行时能够同时活跃（不活跃的线程处于等待状态）的线程数是由硬件（主要是 CUDA 核心数）和软件（即核函数中的代码）决定的。

每个线程在核函数中都有一个唯一的身份标识。由于我们用两个参数指定了线程数目，那么自然地，每个线程的身份可由两个参数确定。在核函数内部，程序是知道执行配置参数 grid_size 和 block_size 的值的。这两个值分别保存于如下两个内建变量（built-in variable）：

- gridDim.x：该变量的数值等于执行配置中变量 grid_size 的数值；
- blockDim.x：该变量的数值等于执行配置中变量 block_size 的数值。

类似地，在核函数中预定义了如下标识线程的内建变量：

- blockIdx.x：该变量指定一个线程在一个网格中的线程块指标，其取值范围是从 0 到 gridDim.x - 1。
- threadIdx.x：该变量指定一个线程在一个线程块中的线程指标，其取值范围是从 0 到 blockDim.x - 1 。

举一个具体的例子。假如某个核函数的执行配置是 <<<10000, 256>>>，那么网格大小 gridDim.x 的值为 10000，线程块大小 blockDim.x 的值为 256。线程块指标 blockIdx.x 可以取 0 到 9999 之间的值，而每一个线程块中的线程指标 threadIdx.x 可以取 0 到 255 之间的值。当 blockIdx.x 等于 0 时，所有 256 个 threadIdx.x 的值对应第 0 个线程块；当 blockIdx.x 等于 1 时，所有 256 个 threadIdx.x 的值对应于第 1 个线程块；依此类推。

再次回到 Hello World 程序。在程序 hello3.cu 中，我们指派了 8 个线程，每个线程输出了一行文字，但我们不知道哪一行是由哪个线程输出的。既然每一个线程都有一个唯一的身份标识，那么我们就可以利用该身份标识判断哪一行是由哪个线程输出的。为此，我们将程序改写为 Listing 2.4。

Listing 2.4: 本章程序 hello4.cu 中的内容。

```
#include <stdio.h>

__global__ void hello_from_gpu()
{
    const int bid = blockIdx.x;
    const int tid = threadIdx.x;
    printf("Hello World from block %d and thread %d!\n", bid, tid);
}

int main(void)
{
    hello_from_gpu<<<2, 4>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

编译、运行这个程序，有时候输出如下文字：

```
Hello World from block 1 and thread 0.
Hello World from block 1 and thread 1.
Hello World from block 1 and thread 2.
Hello World from block 1 and thread 3.
Hello World from block 0 and thread 0.
Hello World from block 0 and thread 1.
Hello World from block 0 and thread 2.
Hello World from block 0 and thread 3.
```

有时候输出如下文字：

```
Hello World from block 0 and thread 0.
Hello World from block 0 and thread 1.
Hello World from block 0 and thread 2.
Hello World from block 0 and thread 3.
Hello World from block 1 and thread 0.
Hello World from block 1 and thread 1.
```

```
Hello World from block 1 and thread 2.
Hello World from block 1 and thread 3.
```

也就是说，有时候是第 0 个线程块先完成计算，有时候是第 1 个线程块先完成。这反映了 CUDA 程序执行时的一个很重要的特征，即每个线程块的计算是相互独立的。不管完成计算的次序如何，每个线程块中的每个线程都进行一次计算。

2.3.3 推广至多维网格

细心的读者可能注意到，前面介绍的四个内建变量好像都用了 C++ 中的结构体（struct）或者类（class）的成员变量的语法。没错，它们都是结构体变量的成员。其中

- `blockIdx` 和 `threadIdx` 是类型为 `uint3` 的变量。该类型是一个结构体，具有 `x`、`y`、`z` 三个成员。所以，`blockIdx.x` 只是三个成员中的一个，另外两个成员分别是 `blockIdx.y` 和 `blockIdx.z`。类似地，`threadIdx.x` 只是三个成员中的一个，另外两个成员分别是 `threadIdx.y` 和 `threadIdx.z`。结构体 `uint3` 在头文 件 `vector_types.h` 中定义：

  ```
  struct __device_builtin__ uint3
  {
      unsigned int x, y, z;
  };
  typedef __device_builtin__ struct uint3 uint3;
  ```

  也就是说，该结构体由三个无符号整数类型的成员构成。

- `gridDim` 和 `blockDim` 是类型为 `dim3` 的变量。该类型是一个结构体，具有 `x`、`y`、`z` 三个成员。所以， `gridDim.x` 只是三个成员中的一个，另外两个成员分别是 `gridDim.y` 和 `gridDim.z`。类似地，`blockDim.x` 只是三个成员中的一个，另外两个成员分别是 `blockDim.y` 和 `blockDim.z`。结构体 `dim3` 也在头文件 `vector_types.h` 定义，除了和结构体 `uint3` 有同样的三个成员之外，还在使用 C++ 程序的情况下定义了一些成员函数，例如下面使用的构造函数。

这些内建变量都只在核函数中有效（可见），而且满足如下关系：

- `blockIdx.x` 的取值范围是从 0 到 `gridDim.x - 1`；
- `blockIdx.y` 的取值范围是从 0 到 `gridDim.y - 1`；
- `blockIdx.z` 的取值范围是从 0 到 `gridDim.z - 1`；

- threadIdx.x 的取值范围是从 0 到 blockDim.x - 1；
- threadIdx.y 的取值范围是从 0 到 blockDim.y - 1；
- threadIdx.z 的取值范围是从 0 到 blockDim.z - 1。

我们前面介绍过，网格大小和线程块大小是在调用核函数时通过执行配置指定的。在之前的例子中，我们用的执行配置仅仅用了两个整数：

```
<<<grid_size, block_size>>>
```

我们知道，这两个整数的值将分别赋给内建变量 gridDim.x 和 blockDim.x。此时，gridDim 和 blockDim 中没有被指定的成员取默认值 1。在这种情况下，网格和线程块实际上都是“一维”的。

也可以用结构体 dim3 定义“多维”的网格和线程块（这里用了 C++ 中构造函数的语法）：

```
dim3 grid_size(Gx, Gy, Gz);
dim3 block_size(Bx, By, Bz);
```

如果第三个维度的大小是 1，可以写

```
dim3 grid_size(Gx, Gy);
dim3 block_size(Bx, By);
```

例如，如果要定义一个 $2 \times 2 \times 1$ 的网格以及 $3 \times 2 \times 1$ 的线程块，可将执行配置中的 grid_size 和 block_size 分别定义为如下结构体变量：

```
dim3 grid_size(2, 2);  // 等价于 dim3 grid_size(2, 2, 1);
dim3 block_size(3, 2); // 等价于 dim3 block_size(3, 2, 1);
```

由此产生的核函数中的线程组织见图 2.1。

多维的网格和线程块本质上还是一维的，就像多维数组本质上也是一维数组一样。与一个多维线程指标 threadIdx.x、 threadIdx.y、 threadIdx.z 对应的一维指标为

```
int tid = threadIdx.z * blockDim.x * blockDim.y +
          threadIdx.y * blockDim.x + threadIdx.x;
```

也就是说，x 维度是最内层的（变化最快），而 z 维度是最外层的（变化最慢）。与一个多维线程块指标 blockIdx.x、blockIdx.y、blockIdx.z 对应的一维指标没有唯一的定义（主要是因为各个线程块的执行是相互独立的），但也可以类似地定义：

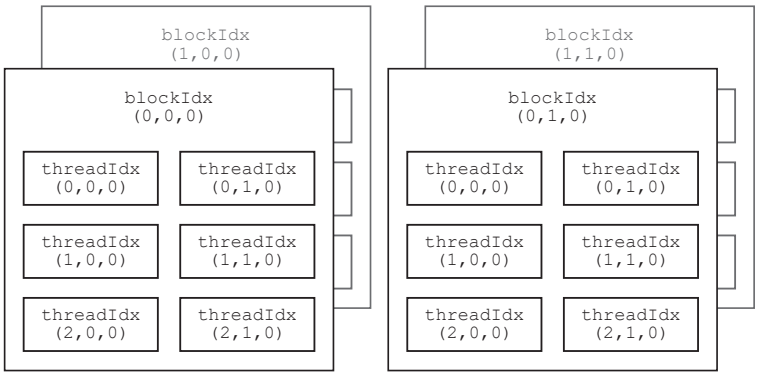

图 2.1: CUDA 核函数中的线程组织示意图。在执行一个核函数时，会产生一个网格，由多个相同大小的线程块构成。该图中展示的是有 $2 \times 2 \times 1$ 个线程块的网格，其中每个线程块包含 $3 \times 2 \times 1$ 个线程。

```
int bid = blockIdx.z * gridDim.x * gridDim.y +
          blockIdx.y * gridDim.x + blockIdx.x;
```

对于有些问题，例如第 7 章引入的矩阵转置问题，有时候使用如下复合线程索引更合适：

```
int nx = blockDim.x * blockIdx.x + threadIdx.x;
int ny = blockDim.y * blockIdx.y + threadIdx.y;
int nz = blockDim.z * blockIdx.z + threadIdx.z;
```

一个线程块中的线程还可以细分为不同的线程束（thread warp）。一个线程束（即一束线程）是同一个线程块中相邻的 `warpSize` 个线程。`warpSize` 也是一个内建变量，表示线程束大小，其值对于目前所有的 GPU 架构都是 32。所以，一个线程束就是连续的 32 个线程。具体地说，一个线程块中第 0 到第 31 个线程属于第 0 个线程束，第 32 到第 63 个线程属于第 1 个线程束，如此等等。图 2.2 中展示的每个线程块拥有两个线程束。

我们可以通过继续修改 Hello World 程序来展示使用多维线程块的核函数中的线程组织情况。Listing 2.5 是修改后的代码，在调用核函数时指定了一个 2×4 的两维线程块。程序的输出是

```
Hello World from block-0 and thread-(0, 0)!
Hello World from block-0 and thread-(1, 0)!
Hello World from block-0 and thread-(0, 1)!
Hello World from block-0 and thread-(1, 1)!
Hello World from block-0 and thread-(0, 2)!
Hello World from block-0 and thread-(1, 2)!
```

```
Hello World from block-0 and thread-(0, 3)!
Hello World from block-0 and thread-(1, 3)!
```

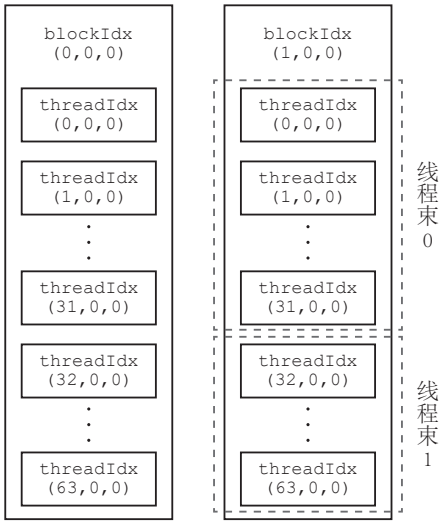

图 2.2: 线程块中相邻的 32 个线程构成一个线程束。该图展示了两个线程块，每个线程块拥有两个线程束。

Listing 2.5: 本章程序 hello5.cu 中的内容。

```
1  #include <stdio.h>
2
3  __global__ void hello_from_gpu()
4  {
5      const int b = blockIdx.x;
6      const int tx = threadIdx.x;
7      const int ty = threadIdx.y;
8      printf("Hello World from block-%d and thread-(%d, %d)!\n", b, tx,
           ty);
9  }
10
11 int main(void)
12 {
13     const dim3 block_size(2, 4);
```

```
14     hello_from_gpu<<<1, block_size>>>();
15     cudaDeviceSynchronize();
16     return 0;
17 }
```

因为线程块的大小是 2 × 4，所以我们知道在核函数中，blockDim.x 的值为 2，blockDim.y 的值为 4。可以看到，threadIdx.x 的取值范围是从 0 到 1，而 threadIdx.y 的取值范围是从 0 到 3。另外，因为网格大小 gridDim.x 是 1，故核函数中 blockIdx.x 的值只能为 0。最后，从输出结果可以确认，x 维度的线程指标 threadIdx.x 是最内层的（变化最快）。

2.3.4 网格与线程块大小的限制

CUDA 中对能够定义的网格大小和线程块大小做了限制。对任何从开普勒到图灵架构的 GPU 来说，网格大小在 x、y 和 z 三个方向的最大允许值分别为 $2^{31}-1$、65535 和 65535；线程块大小在 x、y 和 z 三个方向的最大允许值分别为 1024、1024 和 64。另外还要求线程块总的大小，即 blockDim.x、blockDim.y 和 blockDim.z 的乘积不能大于 1024。也就是说，不管如何定义，一个线程块顶多只能有 1024 个线程。这些限制是必须牢记的。

2.4 CUDA 中的头文件

我们知道，在编写 C++ 程序时，往往需要在源文件中包含一些标准的头文件。读者也许注意到了，本章程序包含了 C++ 的头文件 <stdio.h>，但并没有包含任何 CUDA 相关的头文件。CUDA 中也有一些头文件，但是在使用 nvcc 编译器驱动编译 .cu 文件时，将自动包含必要的 CUDA 头文件，如 <cuda.h> 和 <cuda_runtime.h>。因为 <cuda.h> 包含了 <stdlib.h>，故用 nvcc 编译 CUDA 程序时甚至不需要在 .cu 文件中包含 <stdlib.h>。当然，用户依然可以在 .cu 文件中包含 <stdlib.h>，因为（正确编写的）头文件不会在一个编译单元内被包含多次。本书会从第 4 章开始使用一个用户自定义头文件。

在本书第 14 章，我们将看到，在使用一些利用 CUDA 进行加速的应用程序库时，需要包含一些必要的头文件，并有可能还需要指定链接选项。

2.5 用 nvcc 编译 CUDA 程序

CUDA 的编译器驱动（compiler driver）nvcc 先将全部源代码分离为主机代码和设备代码。主机代码完整地支持 C++ 语法，但设备代码只部分地支持 C++。nvcc 先将设备代码编译为 PTX（Parallel Thread eXecution）伪汇编代码，再将 PTX 代码编译为二进制的 cubin 目

标代码。在将源代码编译为 PTX 代码时，需要用选项`-arch=compute_XY`指定一个虚拟架构的计算能力，用以确定代码中能够使用的 CUDA 功能。在将 PTX 代码编译为 cubin 代码时，需要用选项`-code=sm_ZW`指定一个真实架构的计算能力，用以确定可执行文件能够使用的 GPU。真实架构的计算能力必须等于或者大于虚拟架构的计算能力。例如，可以用选项

```
-arch=compute_35 -code=sm_60
```

编译，但不能用选项

```
-arch=compute_60 -code=sm_35
```

编译（编译器会报错）。如果仅仅针对一个 GPU 编译程序，一般情况下建议将以上两个计算能力都选为所用 GPU 的计算能力。

用以上的方式编译出来的可执行文件只能在少数几个 GPU 中才能运行。选项`-code=sm_ZW`指定了 GPU 的真实架构为`Z.W`。对应的可执行文件只能在主版本号为`Z`、次版本号大于或等于`W`的 GPU 中运行。举例来说，由编译选项

```
-arch=compute_35 -code=sm_35
```

编译出来的可执行文件只能在计算能力为 3.5 和 3.7 的 GPU 中执行，而由编译选项

```
-arch=compute_35 -code=sm_60
```

编译出来的可执行文件只能在所有帕斯卡架构的 GPU 中执行。

如果希望编译出来的可执行文件能够在更多的 GPU 中执行，可以同时指定多组计算能力，每一组用如下形式的编译选项：

```
-gencode arch=compute_XY,code=sm_ZW
```

例如，用选项

```
-gencode arch=compute_35,code=sm_35
-gencode arch=compute_50,code=sm_50
-gencode arch=compute_60,code=sm_60
-gencode arch=compute_70,code=sm_70
```

编译出来的可执行文件将包含四个二进制版本，分别对应开普勒（不包含比较老的 3.0 和 3.2 的计算能力）、麦克斯韦、帕斯卡和伏特架构。这样的可执行文件叫做胖二进制文件（fatbinary）。在不同架构的 GPU 中运行时会自动选择对应的二进制版本。要注意的是，上述编译选项假定所使用的 CUDA 版本支持 7.0 的计算能力，也就是说至少是 CUDA

9.0。如果在编译选项中指定了不被支持的计算能力，编译器会报错。另外要注意的是，过多地指定计算能力，会增加编译时间和可执行文件的大小。

nvcc 有一种叫做即时编译（just-in-time compilation）的机制，可以在运行可执行文件时从其中保留的 PTX 代码临时编译出一个 cubin 目标代码。要在可执行文件中保留（或者说嵌入）一个这样的 PTX 代码，就必须用如下方式指定所保留 PTX 代码的虚拟架构：

```
-gencode arch=compute_XY,code=compute_XY
```

这里的两个计算能力都是虚拟架构的计算能力，必须一模一样。例如，假如我们处于只有 CUDA 8.0 的年代（不支持伏特架构），但希望编译出的二进制版本适用于尽可能多的 GPU，则可以用如下的编译选项：

```
-gencode arch=compute_35,code=sm_35
-gencode arch=compute_50,code=sm_50
-gencode arch=compute_60,code=sm_60
-gencode arch=compute_60,code=compute_60
```

其中，前三行的选项分别对应三个真实架构的 cubin 目标代码，第四行的选项对应保留的 PTX 代码。这样编译出来的可执行文件可以直接在伏特架构的 GPU 中运行，只不过不一定能充分利用伏特架构的硬件功能。在伏特架构的 GPU 中运行时，会根据虚拟架构为 6.0 的 PTX 代码即时地编译出一个适用于当前 GPU 的目标代码。

在学习 CUDA 编程时，有一个简化的编译选项可以使用：

```
-arch=sm_XY
```

它等价于

```
-gencode arch=compute_XY,code=sm_XY
-gencode arch=compute_XY,code=compute_XY
```

例如，在作者的装有 GeForce RTX 2070 的笔记本中，可以用选项 `-arch=sm_75` 编译一个 CUDA 程序。

读者也许注意到了，本章的程序在编译时并没有通过编译选项指定计算能力。这是因为编译器有一个默认的计算能力。以下是各个 CUDA 版本中的编译器在编译 CUDA 代码时默认的计算能力：

- CUDA 6.0 以及更早的：默认的计算能力是 1.0。
- CUDA 6.5 到 CUDA 8.0：默认的计算能力是 2.0。
- CUDA 9.0 到 CUDA 10.2：默认的计算能力是 3.0。

作者用的 CUDA 版本是 10.1，故本章的程序在编译时实际上使用了 3.0 的计算能力。如果用 CUDA 6.0 进行编译，而且不指定一个计算能力，则会使用默认的 1.0 的计算能力。此时本章的程序将无法正确地编译，因为从 GPU 中直接向屏幕打印信息是从计算能力 2.0 才开始支持的功能。正如在上一章强调过的，本书中的所有示例程序都可以在 CUDA 9.0-10.2 中进行测试。

关于 nvcc 编译器驱动更多的介绍，请参考如下官方文档：https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc。

第 3 章 简单 CUDA 程序的基本框架

上一章通过经典的 Hello World 程序介绍了 CUDA 中的线程组织。学会了编写与运行 Hello World 程序，就对 CUDA 编程有了一个初步的认识。然而，用 `printf` 函数直接从核函数中输出数据只有在调试（debug）程序时才偶尔用到。从本章开始，我们将告别 Hello World 程序，学习编写更加有用的 CUDA 程序。本章将通过数组相加的计算讲解 CUDA 程序的基本框架。

3.1 例子：数组相加

我们考虑一个简单的计算：求两个具有同样长度的一维数组的对应元素之和。该计算是非常简单的。我们可以写出如 Listing 3.1 所示的 C++ 程序。

Listing 3.1: 本章程序 add.cpp 中的内容。

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

const double EPSILON = 1.0e-15;
const double a = 1.23;
const double b = 2.34;
const double c = 3.57;
void add(const double *x, const double *y, double *z, const int N);
void check(const double *z, const int N);

int main(void)
{
    const int N = 100000000;
    const int M = sizeof(double) * N;
```

```
16      double *x = (double*) malloc(M);
17      double *y = (double*) malloc(M);
18      double *z = (double*) malloc(M);
19
20      for (int n = 0; n < N; ++n)
21      {
22          x[n] = a;
23          y[n] = b;
24      }
25
26      add(x, y, z, N);
27      check(z, N);
28
29      free(x);
30      free(y);
31      free(z);
32      return 0;
33  }
34
35  void add(const double *x, const double *y, double *z, const int N)
36  {
37      for (int n = 0; n < N; ++n)
38      {
39          z[n] = x[n] + y[n];
40      }
41  }
42
43  void check(const double *z, const int N)
44  {
45      bool has_error = false;
46      for (int n = 0; n < N; ++n)
47      {
48          if (fabs(z[n] - c) > EPSILON)
49          {
50              has_error = true;
```

```
51        }
52    }
53    printf("%s\n", has_error ? "Has errors" : "No errors");
54}
```

用 g++ 编译运行该程序，将在屏幕打印 No errors，表示 add() 函数的计算结果正确。对该程序的解释如下：

- 在主函数中，第 16-18 行定义了三个双精度浮点数类型的指针变量，然后将他们指向由函数 malloc（在头文件 <stdlib.h> 中声明） 分配的内存，从而得到了三个长度为 $N = 10^8$ 的一维数组。这将需要约 2.4 GB 的主机内存。后面的 CUDA 程序也至少需要有这么多设备内存。如果读者的主机或者设备内存不够多，可以适当调整本书范例中相关数组的大小后再进行测试。
- 第 20-24 行将数组 x 和 y 中的每个元素分别初始化为 1.23 和 2.34。
- 接着，第 26 行调用自定义函数 add() 计算数组 x 与数组 y 的和，将结果存放在数组 z 中。
- 然后，第 27 行用一个自定义的 check() 函数检验数组 z 中的每个元素是不是都是正确值 3.57。注意：在判断两个浮点数是否相等时，不能用运算符 ==，而要将这两个数的差的绝对值与一个很小的数进行比较。在上述程序中，我们假定，当两个双精度浮点数的差的绝对值小于 10^{-15} 时它们就是相等的。求绝对值的函数在 C++ 头文件 <math.h> 中声明，故需要在程序开头包含此头文件。
- 最后，第 29-31 行释放分配的内存。

3.2 CUDA 程序的基本框架

在现实的中、大型程序中，往往使用多个源文件，每个源文件又包含多个函数。本书第二部分的例子就是这样。然而，在本书第一部分的例子中，我们只使用一个源文件，其中包含一个主函数和若干其它函数（包括 C++ 自定义函数和 CUDA 核函数）。在这种情况下，一个典型的 CUDA 程序的基本框架见 Listing 3.2：

Listing 3.2: 一个典型的 CUDA 程序的基本框架。

```
1 头文件包含
2 常量定义（或者宏定义）
```

```
C++ 自定义函数和 CUDA 核函数的声明（原型）
int main(void)
{
    分配主机与设备内存
    初始化主机中的数据
    将某些数据从主机复制到设备
    调用核函数在设备中进行计算
    将某些数据从设备复制到主机
    释放主机与设备内存
}
C++ 自定义函数和 CUDA 核函数的定义（实现）
```

在上述 CUDA 程序的基本框架中，有很多内容是还没有介绍的。但是，我们先把利用 CUDA 求数组之和的全部源代码列出来，之后再逐步讲解。Listing 3.3 给出了除 check 函数定义（该函数和前一个 C++ 程序中的同名函数具有相同的定义）之外的全部源代码。

Listing 3.3: 本章程序 add1.cu 中的大部分内容。

```
#include <math.h>
#include <stdio.h>

const double EPSILON = 1.0e-15;
const double a = 1.23;
const double b = 2.34;
const double c = 3.57;
void __global__ add(const double *x, const double *y, double *z);
void check(const double *z, const int N);

int main(void)
{
    const int N = 100000000;
    const int M = sizeof(double) * N;
    double *h_x = (double*) malloc(M);
    double *h_y = (double*) malloc(M);
    double *h_z = (double*) malloc(M);
```

```
18
19      for (int n = 0; n < N; ++n)
20      {
21          h_x[n] = a;
22          h_y[n] = b;
23      }
24
25      double *d_x, *d_y, *d_z;
26      cudaMalloc((void **)&d_x, M);
27      cudaMalloc((void **)&d_y, M);
28      cudaMalloc((void **)&d_z, M);
29      cudaMemcpy(d_x, h_x, M, cudaMemcpyHostToDevice);
30      cudaMemcpy(d_y, h_y, M, cudaMemcpyHostToDevice);
31
32      const int block_size = 128;
33      const int grid_size = N / block_size;
34      add<<<grid_size, block_size>>>(d_x, d_y, d_z);
35
36      cudaMemcpy(h_z, d_z, M, cudaMemcpyDeviceToHost);
37      check(h_z, N);
38
39      free(h_x);
40      free(h_y);
41      free(h_z);
42      cudaFree(d_x);
43      cudaFree(d_y);
44      cudaFree(d_z);
45      return 0;
46  }
47
48  void __global__ add(const double *x, const double *y, double *z)
49  {
50      const int n = blockDim.x * blockIdx.x + threadIdx.x;
51      z[n] = x[n] + y[n];
52  }
```

用 nvcc 编译该程序，并指定与 GeForce RTX 2070 对应的计算能力（读者可以选用自己所用 GPU 的计算能力）：

```
$ nvcc -arch=sm_75 add1.cu
```

将得到一个可执行文件 a.out。运行该程序得到的输出应该与前面 C++ 程序所得到的输出一样，说明得到了预期的结果。

值得注意的是，当使用较大的数据量时，网格大小往往很大。例如，本例中的网格大小为 $10^8/128 = 781250$。如果读者使用 CUDA 8.0，而在用 nvcc 编译程序时又忘了指定一个计算能力，那就会根据默认的 2.0 的计算能力编译程序。对于该计算能力，网格大小在 x 方向的上限为 65535，小于本例中所使用的值。这将导致程序无法正确地执行。这是初学者需要特别注意的一个问题。下面对该程序进行详细的讲解。

3.2.1 隐形的设备初始化

在 CUDA 运行时 API 中，没有明显地初始化设备（即 GPU）的函数。在第一次调用一个和设备管理与版本查询功能无关的运行时 API 函数时，设备将自动地初始化。

3.2.2 设备内存的分配与释放

在上述程序中，我们首先在主机中定义了三个数组并进行了初始化。这与之前 C++ 版本的相应部分是一样的。接着，第 25-28 行在设备中也定义了三个数组并分配了内存（显存）。第 25 行就是定义三个双精度类型变量的指针。如果不看后面的代码，我们并不知道这三个指针会指向哪些内存区域。只有通过第 26-28 行的 cudaMalloc() 函数才能确定它们将指向设备中的内存，而不是主机中的内存。该函数是一个 CUDA 运行时 API 函数。所有 CUDA 运行时 API 函数都以 cuda 开头。本书仅涉及极少数的 CUDA 运行时 API 函数。完整的列表见如下网页（一个几百页的手册）：https://docs.nvidia.com/cuda/cuda-runtime-api。

正如在 C++ 中可由 malloc() 函数动态分配内存，在 CUDA 中，设备内存的动态分配可由 cudaMalloc() 函数实现。该函数的原型如下：

```
cudaError_t cudaMalloc(void **address, size_t size);
```

其中，

- 第一个参数 address 是待分配设备内存的指针。注意：因为内存（地址）本身就是一个指针，所以待分配设备内存的指针就是指针的指针，即双重指针。
- 第二个参数 size 是待分配内存的字节数。
- 返回值是一个错误代号。如果调用成功，返回 cudaSuccess，否则返回一个代表某种错误的代号（下一章会进一步讨论）。

该函数为某个变量分配 size 字节的线性内存（linear memory）。初学者不必深究什么是线性内存，也暂时不用关心该函数的返回值。在 26-28 行，我们忽略了函数 cudaMalloc() 的返回值。这几行代码用到的参数 M 是所分配内存的字节数，即 sizeof(double) * N。注意：虽然在很多情况下 sizeof(double) 等于 8，但用 sizeof(double) 是更加通用、安全的做法。

调用函数 cudaMalloc() 时传入的第一个参数 (void **)&g_x 稍难理解。首先，我们知道 g_x 是一个 double 类型的指针，那么它的地址 &g_x 就是 double 类型的双重指针。而 (void **) 是一个强制类型转换操作，将一个某种类型的双重指针转换为一个 void 类型的双重指针。这种类型转换可以不明确地写出来，即对函数 cudaMalloc() 的调用可以简写为：

```
cudaMalloc(&g_x, M);
```

读者可以试一试。

读者也许会问，cudaMalloc() 函数为什么需要一个双重指针作为变量呢？这是因为（以第 26 行为例），该函数的功能是改变指针 d_x 本身的值（将一个指针赋值给 d_x），而不是改变 d_x 所指内存缓冲区中的变量值。在这种情况下，必须将 d_x 的地址 &d_x 传给函数 cudaMalloc() 才能达到此效果。这是 C++ 编程中非常重要的一点。如果读者对指针的概念比较模糊，请务必阅读相关资料，查漏补缺。从另一个角度来说，函数 cudaMalloc() 要求用传双重指针的方式改变一个指针的值，而不是直接返回一个指针，是因为该函数已经将返回值用于返回错误代号，而 C++ 又不支持多个返回值。

总之，用 cudaMalloc() 函数可以为不同类型的指针变量分配设备内存。注意，为了区分主机和设备中的变量，我们（遵循 CUDA 编程的传统）用 d_ 作为所有设备变量的前缀，而用 h_ 作为对应主机变量的前缀。

正如用 malloc() 函数分配的主机内存需要用 free() 函数释放，用 cudaMalloc() 函数分配的设备内存需要用 cudaFree() 函数释放。该函数的原型为：

```
cudaError_t cudaFree(void* address);
```

这里，参数 address 就是待释放的设备内存变量（不是双重指针）。返回值是一个错误代号。如果调用成功，返回 cudaSuccess。

主机内存也可由 C++ 中的 new 运算符动态分配，并由 delete 运算符释放。读者可以将程序 add1.cu 中的 malloc() 和 free() 语句分别换成用 new 和 delete 实现的等价的语句，看看是否能正确地编译、运行。

在分配与释放各种内存时，相应的操作一定要两两配对，否则将有可能出现内存错误。将程序 add1.cu 中的 cudaFree() 改成 free()，虽然能够正确地编译，而且能够在屏幕打印出 No errors 的结果，但在程序退出之前，还是会出现所谓的段错误（segmentation fault）。

读者可以试一试。主动尝试错误是编程学习中非常重要的技巧，因为通过它可以熟悉各种编译和运行错误，提高排错能力。

从计算能力 2.0 开始，CUDA 还允许在核函数内部用 `malloc()` 和 `free()` 动态地分配与释放一定数量的全局内存。一般情况下，这样容易导致较差的程序性能，不建议使用。如果发现有这样的需求，可能需要思考如何重构算法。

3.2.3 主机与设备之间数据的传递

在分配了设备内存之后，就可以将某些数据从主机传递到设备中去了。第 29-30 行将主机中存放在 `h_x` 和 `h_y` 中的数据复制到设备中的相应变量 `d_x` 和 `d_y` 所指向的缓冲区中去。这里用到了 CUDA 运行时 API 函数 `cudaMemcpy()`，其原型是：

```
cudaError_t cudaMemcpy
(
    void                *dst,
    const void          *src,
    size_t              count,
    enum cudaMemcpyKind kind
);
```

其中，

- 第一个参数 `dst` 是目标地址。
- 第二个参数 `src` 是源地址。
- 第三个参数 `count` 是复制数据的字节数。
- 第四个参数 `kind` 一个枚举类型的变量，标志数据传递方向。它只能取如下几个值：
  - `cudaMemcpyHostToHost`，表示从主机复制到主机；
  - `cudaMemcpyHostToDevice`，表示从主机复制到设备；
  - `cudaMemcpyDeviceToHost`，表示从设备复制到主机；
  - `cudaMemcpyDeviceToDevice`，表示从设备复制到设备；
  - `cudaMemcpyDefault`，表示根据指针 `dst` 和 `src` 所指地址自动判断数据传输的方向。这要求系统具有统一虚拟寻址（Unified Virtual Addressing）的功能（要求 64 位的主机）。CUDA 正在逐步放弃对 32 位主机的支持，故一般情况下用该选项自动确定数据传输方向是没有问题的。至于是明确地指定传输方向更好，还是利用自动判断更好，则是一个仁者见仁、智者见智的问题。

- 返回值是一个错误代号。如果调用成功，返回 cudaSuccess。
- 该函数的作用是将一定字节数的数据从源地址所指缓冲区复制到目标地址所指缓冲区。

我们回头看程序的第 29 行。它的作用就是将 h_x 指向的主机内存中 M 字节的数据复制到 d_x 指向的设备内存中去。因为这里的源地址是主机中的内存，目标地址是设备中的内存，所以第四个参数必须是 cudaMemcpyHostToDevice 或 cudaMemcpyDefault，否则将导致错误。

类似地，在调用核函数进行计算，得到需要的数据之后，我们需要将设备中的数据复制到主机，这正是第 36 行的代码所做的事情。该行代码的作用就是将 d_z 指向的设备内存中 M 字节的数据复制到 h_z 指向的主机内存中去。因为这里的源地址是设备中的内存，目标地址是主机中的内存，所以第四个参数必须是 cudaMemcpyDeviceToHost 或 cudaMemcpyDefault，否则将导致错误。

在本章的程序 add2wrong.cu 中，作者故意将第 29-30 行的传输方向参数写成了 cudaMemcpyDeviceToHost。请读者编译、运行该程序，看看会得到什么结果。

3.2.4 核函数中数据与线程的对应

将有关的数据从主机传至设备之后，就可以调用核函数在设备中进行计算了。第 32-34 行确定了核函数的执行配置：使用具有 128 个线程的一维线程块，一共有 $N/128$ 个这样的线程块。仔细比较程序 add.cpp 中的主机端函数（第 35-41 行）和程序 add1.cu 中的设备端函数（第 48-52 行），可以看出，将主机中的函数改为设备中的核函数是非常简单的：基本上就是去掉一层循环。在主机函数中，我们需要依次对数组的每一个元素进行操作，所以需要使用一个循环。在设备的核函数中，我们用“单指令-多线程”的方式编写代码，故可去掉该循环，只需将数组元素指标与线程指标一一对应即可。

例如，在上述核函数中，使用了语句

```
const int n = blockDim.x * blockIdx.x + threadIdx.x;
```

来确定对应方式。赋值号右边只出现标记线程的内建变量，左边的 n 是后面代码中将要用到的数组元素指标。在这种情况下，第 0 号线程块中的 blockDim.x 个线程对应于第 0 到第 blockDim.x-1 个数组元素，第 1 号线程块中的 blockDim.x 个线程对应于第 blockDim.x 到第 2*blockDim.x-1 个数组元素，第 2 号线程块中的 blockDim.x 个线程对应于第 2*blockDim.x 到第 3*blockDim.x-1 个数组元素。依此类推。这里的 blockDim.x 等于执行配置中指定的（一维）线程块大小。核函数中定义的线程数目与数组元素数目一样，都是 10^8。在将线程指标与数据指标一一对应之后，就可以对数组元素进行操作了。该操作的语句

```
z[n] = x[n] + y[n];
```

在主机和设备的核函数中是一样的。通常，在写出一个主机端的函数后，翻译成核函数是非常直接的。最后，值得一提的是，在调试程序时，也可以仅仅使用一个线程。为此，可以将核函数中的代码改成对应主机函数中的代码（即有 for 循环的代码），然后用执行配置 `<<<1, 1>>>` 调用核函数。

3.2.5 核函数的要求

核函数无疑是 CUDA 编程中最重要的方面。我们这里列出编写核函数时要注意的几点。

- 核函数的返回类型必须是 `void`。所以，在核函数中可以用 `return` 关键字，但不可返回任何值。
- 必须使用限定符 `__global__`。也可以加上一些其它 C++ 中的限定符。例如 `static`。限定符的次序可任意。
- 函数名无特殊要求，而且支持 C++ 中的重载（overload），即可以用同一个函数名表示具有不同参数列表的函数。
- 不支持可变数量的参数列表，即参数的个数必须确定。
- 可以向核函数传递非指针变量（如例子中的 `int N`），其内容对每个线程可见。
- 除非使用统一内存编程机制（将在第 12 章介绍），否则传给核函数的数组（指针）必须指向设备内存。
- 核函数不可成为一个类的成员。通常的做法是用一个包装函数调用核函数，而将包装函数定义为类的成员。
- 在计算能力 3.5 之前，核函数之间不能相互调用。从计算能力 3.5 开始，引入了动态并行（dynamic parallelism）机制，在核函数内部可以调用其它核函数，甚至可以调用自己（递归函数）。但本书不讨论动态并行，感兴趣的读者请参考《CUDA C++ Programming Guide》的附录 D。
- 无论是从主机调用，还是从设备调用，核函数都是在设备执行。调用核函数时必须指定执行配置，即三括号和它里面的参数。在本例中，选取的线程块大小为 128，网格大小为数组元素个数除以线程块大小，即 $10^8/128 = 781250$。

3.2.6 核函数中 if 语句的必要性

前面的核函数根本没有使用参数 N。当 N 是 blockDim.x（即 block_size）的整数倍时，这不会引起问题，因为核函数中的线程数目刚好等于数组元素个数。然而，当 N 不是 blockDim.x 的整数倍时，就有可能引发错误。

我们将 N 改为 10^8+1，而且依然取 block_size 等于 128。此时，我们首先面临的一个问题就是，grid_size 应该取多大？用 N 除以 block_size，商为 781250，余数为 1。显然，我们不能取 grid_size 为 781250，因为这样只能定义 10^8 个线程，在用一个线程对应一个数组元素的方案下无法处理剩下的 1 个元素。实际上，我们可以将 grid_size 取为 781251，使得定义的线程数为 10^8+128。虽然定义的总线程数多于元素个数，但我们可以通过条件语句规避不需要的线程操作。据此，我们可以写出如 Listing 3.4 所示的核函数。此时，在主机中调用该核函时所用的 grid_size 为

```
int grid_size  = (N - 1) / block_size + 1;
```

或者

```
int grid_size  = (N + block_size - 1) / block_size;
```

以上两个语句都等价于下述语句

```
int grid_size = (N % block_size == 0)
              ? (N / block_size)
              : (N / block_size + 1);
```

因为此时线程数（10^8+128）多于数组元素个数（10^8+1），所以如果去掉 if 语句，则会出现非法的设备内存操作，可能导致不可预料的错误。这是在 CUDA 编程中一定要避免的。另外，虽然核函数不允许有返回值，但还是可以使用 return 语句。上述核函数中的代码也可以写为如下等价的形式：

```
const int n = blockDim.x * blockIdx.x + threadIdx.x;
if (n >= N) return;
z[n] = x[n] + y[n];
```

Listing 3.4: 本章程序 add3if.cu 中的核函数定义。

```
1 void __global__ add(const double *x, const double *y, double *z, const
      int N)
2 {
```

```
3     const int n = blockDim.x * blockIdx.x + threadIdx.x;
4     if (n < N)
5     {
6         z[n] = x[n] + y[n];
7     }
8 }
```

3.3 自定义设备函数

核函数可以调用不带执行配置的自定义函数。这样的函数叫做设备函数（device function），是在设备中执行，并在设备中被调用的。与之相比，核函数是在设备中执行，但在主机端被调用的。现在也支持在一个核函数中调用其它核函数，甚至自己，但本书不涉及这方面的内容。设备函数的定义与使用涉及到 CUDA 中函数执行空间标识符的概念。我们先对此进行介绍，然后以数组相加的程序为例展示设备函数的定义与调用。

3.3.1 函数执行空间标识符

在 CUDA 程序中，由以下标识符确定一个函数在哪里被调用、在哪里执行：

- 用 `__global__` 修饰的函数叫做核函数，一般由主机调用，在设备执行。如果使用动态并行，则也可以在核函数中调用自己或其它核函数。
- 用 `__device__` 修饰的函数叫做设备函数，只能被核函数或其它设备函数调用，在设备执行。
- 用 `__host__` 修饰的函数就是主机端的普通 C++ 函数，在主机中被调用，在主机执行。对于主机端的函数，该修饰符可省略。之所以提供这样一个修饰符，是因为有时候可以用 `__host__` 和 `__device__` 同时修饰一个函数，使得该函数既是一个 C++ 中的普通函数，又是一个设备函数。这样做可以减少冗余代码。编译器将针对主机和设备分别编译该函数。
- 不能同时用 `__device__` 和 `__global__` 修饰一个函数，即不能将一个函数同时定义为设备函数和核函数。
- 也不能同时用 `__host__` 和 `__global__` 修饰一个函数，即不能将一个函数同时定义为主机函数和核函数。

- 编译器决定把设备函数当作内联函数（inline function）或非内联函数，但可以用修饰符 `__noinline__` 建议一个设备函数为非内联函数（编译器不一定接受），也可以用修饰符 `__forceinline__` 建议一个设备函数为内联函数。

3.3.2 例子：为数组相加的核函数定义一个设备函数

Listing 3.5 给出了三个版本的设备函数以及调用它们的核函数。这三个版本的设备函数分别利用返回值、指针和引用（reference）返回结果。这里涉及的语法和 C++ 中函数定义与调用的语法是一致的，故不多做解释。这几种定义设备函数的方式不会导致程序性能的差别，读者可选择自己喜欢的风格。

Listing 3.5: 本章程序 add4device.cu 中的核函数和设备函数的定义。

```
1  // 版本一：有返回值的设备函数
2  double __device__ add1_device(const double x, const double y)
3  {
4      return (x + y);
5  }
6
7  void __global__ add1(const double *x, const double *y, double *z,
       const int N)
8  {
9      const int n = blockDim.x * blockIdx.x + threadIdx.x;
10     if (n < N)
11     {
12         z[n] = add1_device(x[n], y[n]);
13     }
14 }
15
16 // 版本二：用指针的设备函数
17 void __device__ add2_device(const double x, const double y, double *z)
18 {
19     *z = x + y;
20 }
21
22 void __global__ add2(const double *x, const double *y, double *z,
```

```
    const int N)
{
    const int n = blockDim.x * blockIdx.x + threadIdx.x;
    if (n < N)
    {
        add2_device(x[n], y[n], &z[n]);
    }
}

// 版本三：用引用 (reference) 的设备函数
void __device__ add3_device(const double x, const double y, double &z)
{
    z = x + y;
}

void __global__ add3(const double *x, const double *y, double *z,
    const int N)
{
    const int n = blockDim.x * blockIdx.x + threadIdx.x;
    if (n < N)
    {
        add3_device(x[n], y[n], z[n]);
    }
}
```

第 4 章 CUDA 程序的错误检测

和编写 C++ 程序一样，编写 CUDA 程序时难免会出现各种各样的错误。有的错误在编译的过程中就可以被编译器捕捉，叫做编译错误。有的错误在编译期间没有被发现，但在运行的时候出现，叫做运行时刻的错误。一般来说，运行时刻的错误更难排错。本章讨论如何检测运行时刻的错误，包括使用一个检查 CUDA 运行时 API 函数返回值的宏函数以及使用 CUDA-MEMCHECK 工具。

4.1 一个检测 CUDA 运行时错误的宏函数

在上一章，我们学习了一些 CUDA 运行时 API 函数，例如分配设备内存的函数 `cudaMalloc()`、释放设备内存的函数 `cudaFree()` 以及传输数据的函数 `cudaMemcpy()`。所有 CUDA 运行时 API 函数都是以 `cuda` 为前缀的，而且都有一个类型为 `cudaError_t` 的返回值，代表了一种错误信息。只有返回值为 `cudaSuccess` 时才代表成功地调用了 API 函数。

Listing 4.1: 本书中使用的一个检测 CUDA 运行时错误的宏函数。

```
1  #pragma once
2  #include <stdio.h>
3
4  #define CHECK(call)                                                \
5  do                                                                 \
6  {                                                                  \
7      const cudaError_t error_code = call;                           \
8      if (error_code != cudaSuccess)                                 \
9      {                                                              \
10         printf("CUDA Error:\n");                                   \
11         printf("  File:     %s\n", __FILE__);                      \
12         printf("  Line:     %d\n", __LINE__);                      \
```

```
13         printf("   Error code: %d\n", error_code);             \
14         printf("   Error text: %s\n", cudaGetErrorString(error_code)); \
15         exit(1);                                                \
16     }                                                           \
17 } while (0)
```

根据这样的规则，我们可以写出一个头文件（error.cuh），它包含一个检测 CUDA 运行时错误的宏函数（macro function），见 Listing 4.1。对该宏函数的解释如下：

- 该文件开头一行的 `#pragma once` 是一个预处理指令，其作用是确保当前文件在一个编译单元中不被重复包含。该预处理指令和如下复合的预处理指令作用相当，但更加简洁：

```
#ifndef ERROR_CUH_
#define ERROR_CUH_
    头文件中的内容（即上述文件中第~2-17~行的内容）
#endif
```

- 该宏函数的名称是 `CHECK`，参数 `call` 是一个 CUDA 运行时 API 函数。
- 在定义宏时，如果一行写不下，需要在行末写 `\`，表示续行。
- 第 7 行定义了一个 `cudaError_t` 类型的变量 `error_code` 并初始化为函数 `call` 的返回值。
- 第 8 行判断该变量的值是否为 `cudaSuccess`。如果不是，在第 9-16 行报道相关文件、行数、错误代号以及错误的文字描述并退出程序。`cudaGetErrorString` 显然也是一个 CUDA 运行时 API 函数，作用是将错误代号转化为错误的文字描述。

在使用该宏函数时，只要将一个 CUDA 运行时 API 函数当作参数传入该宏函数即可。例如，如下宏函数的调用

```
CHECK(cudaFree(d_x));
```

将会被展开为 Listing 4.2 所示的代码段。

Listing 4.2: 宏函数调用的展开。

```
1 do
```

```
2  {
3      const cudaError_t error_code = cudaFree(d_x);
4      if (error_code != cudaSuccess)
5      {
6          printf("CUDA Error:\n");
7          printf("  File:      %s\n", __FILE__);
8          printf("  Line:      %d\n", __LINE__);
9          printf("  Error code: %d\n", error_code);
10         printf("  Error text: %s\n", cudaGetErrorString(error_code));
11         exit(1);
12     }
13 } while (0);
```

读者可能会问，宏函数的定义中为什么用了一个 do-while 语句？不用该语句在大部分情况下也是可以的，但在某些情况下不安全（这里不对此展开讨论；感兴趣的读者可自行研究）。也可以不用宏函数，而用普通的函数，但此时必须将宏 __FILE__ 和 __LINE__ 传给该函数，使得它用起来不如宏函数简洁。

4.1.1 检查运行时 API 函数

作为一个例子，我们将程序 add2wrong.cu 中的 CUDA 运行时 API 函数都用宏函数 CHECK 进行包装，得到 check1api.cu，部分代码见 Listing 4.3。在该文件的开头，包含了上述头文件：

```
#include "error.cuh"
```

第 27-29 行对分配设备内存的函数进行了检查；第 30-31 行以及第 37 行对数据传输的函数进行了检查；第 43-45 行对释放设备内存的函数进行了检查。用

```
$ nvcc -arch=sm_75 check1api.cu
```

编译该程序，然后运行得到的可执行文件，将得到如下输出：

```
CUDA Error:
    File:       check1api.cu
    Line:       30
    Error code: 11
    Error text: invalid argument
```

可见，宏函数正确地捕捉到了运行时刻的错误，告诉我们文件 check1api.cu 的第 30 行代码中出现了非法的参数。非法参数指的是 cudaMemcpy 函数的参数有问题，因为我们故意将 cudaMemcpyHostToDevice 写成了 cudaMemcpyDeviceToHost。可见，用了检查错误的宏函数之后，我们可以得到更有用的错误信息，而不仅仅是一个错误的运行结果。从这里开始，我们将坚持用这个宏函数包装大部分的 CUDA 运行时 API 函数。有一个例外是 cudaEventQuery 函数，因为它很有可能返回 cudaErrorNotReady，但又不代表程序出错了。

Listing 4.3: 本章程序 check1api.cu 中的部分代码。

```
1  #include "error.cuh"
2  #include <math.h>
3  #include <stdio.h>
4
5  const double EPSILON = 1.0e-15;
6  const double a = 1.23;
7  const double b = 2.34;
8  const double c = 3.57;
9  void __global__ add(const double *x, const double *y, double *z, const
       int N);
10 void check(const double *z, const int N);
11
12 int main(void)
13 {
14     const int N = 100000000;
15     const int M = sizeof(double) * N;
16     double *h_x = (double*) malloc(M);
17     double *h_y = (double*) malloc(M);
18     double *h_z = (double*) malloc(M);
19
20     for (int n = 0; n < N; ++n)
21     {
22         h_x[n] = a;
23         h_y[n] = b;
24     }
25
```

```
26     double *d_x, *d_y, *d_z;
27     CHECK(cudaMalloc((void **)&d_x, M));
28     CHECK(cudaMalloc((void **)&d_y, M));
29     CHECK(cudaMalloc((void **)&d_z, M));
30     CHECK(cudaMemcpy(d_x, h_x, M, cudaMemcpyDeviceToHost));
31     CHECK(cudaMemcpy(d_y, h_y, M, cudaMemcpyDeviceToHost));
32
33     const int block_size = 128;
34     const int grid_size = (N + block_size - 1) / block_size;
35     add<<<grid_size, block_size>>>(d_x, d_y, d_z, N);
36
37     CHECK(cudaMemcpy(h_z, d_z, M, cudaMemcpyDeviceToHost));
38     check(h_z, N);
39
40     free(h_x);
41     free(h_y);
42     free(h_z);
43     CHECK(cudaFree(d_x));
44     CHECK(cudaFree(d_y));
45     CHECK(cudaFree(d_z));
46     return 0;
47 }
```

4.1.2 检查核函数

用上述方法不能捕捉调用核函数的相关错误，因为核函数不返回任何值（回顾一下，核函必须用 `void` 修饰）。有一个方法可以捕捉调用核函数可能发生的错误，即在调用核函数之后加上如下两个语句：

```
CHECK(cudaGetLastError());
CHECK(cudaDeviceSynchronize());
```

其中，第一个语句的作用是捕捉第二个语句之前的最后一个错误，第二个语句的作用是同步主机与设备。之所以要同步主机与设备，是因为核函数的调用是异步的，即主机发出调用核函数的命令后会立即执行后面的语句，不会等待核函数执行完毕。关于核函数调用的

异步性，我们将在第 11 章中详细讨论。在这之前，我们无需对此深究。要注意的是，上述同步函数是比较耗时的，如果在程序的较内层循环调用的话，很可能会严重地降低程序性能。所以，一般不在程序的较内层循环调用上述同步函数。只要在核函数的调用之后还有对其它任何能返回错误值的 API 函数进行同步调用，都能够触发主机与设备的同步并捕捉到核函数调用中可能发生的错误。

为了展示对核函数调用的检查，我们在上一章的程序 `add1.cu` 的基础上写一个有错误的程序 `check2kernel.cu`，见 Listing 4.4。在上一章我们提到过，线程块大小的最大值是 1024（这对从开普勒到图灵的所有架构都成立）。假如我们不小心将核函数执行配置中的线程块大小写成了 1280，该核函数将不能被成功地调用。第 36 行的代码成功地捕获了该错误，告诉我们程序中核函数的执行配置参数有误：

```
CUDA Error:
    File:       check4kernel.cu
    Line:       36
    Error code: 9
    Error text: invalid configuration argument
```

如果不用宏函数检查（即去掉 第 36-37 行的代码），则很难知道错误的原因，只能看到程序给出 `Has errors` 的输出结果（因为执行配置错误，核函数无法正确执行，而从无法计算出正确的结果）。

Listing 4.4: 本章检查核函数调用的示例程序中的部分代码。

```
1  #include "error.cuh"
2  #include <math.h>
3  #include <stdio.h>
4
5  const double EPSILON = 1.0e-15;
6  const double a = 1.23;
7  const double b = 2.34;
8  const double c = 3.57;
9  void __global__ add(const double *x, const double *y, double *z, const
       int N);
10 void check(const double *z, const int N);
11
12 int main(void)
13 {
```

```
14      const int N = 100000000;
15      const int M = sizeof(double) * N;
16      double *h_x = (double*) malloc(M);
17      double *h_y = (double*) malloc(M);
18      double *h_z = (double*) malloc(M);
19
20      for (int n = 0; n < N; ++n)
21      {
22          h_x[n] = a;
23          h_y[n] = b;
24      }
25
26      double *d_x, *d_y, *d_z;
27      CHECK(cudaMalloc((void **)&d_x, M));
28      CHECK(cudaMalloc((void **)&d_y, M));
29      CHECK(cudaMalloc((void **)&d_z, M));
30      CHECK(cudaMemcpy(d_x, h_x, M, cudaMemcpyHostToDevice));
31      CHECK(cudaMemcpy(d_y, h_y, M, cudaMemcpyHostToDevice));
32
33      const int block_size = 1280;
34      const int grid_size = (N + block_size - 1) / block_size;
35      add<<<grid_size, block_size>>>(d_x, d_y, d_z, N);
36      CHECK(cudaGetLastError());
37      CHECK(cudaDeviceSynchronize());
38
39      CHECK(cudaMemcpy(h_z, d_z, M, cudaMemcpyDeviceToHost));
40      check(h_z, N);
41
42      free(h_x);
43      free(h_y);
44      free(h_z);
45      CHECK(cudaFree(d_x));
46      CHECK(cudaFree(d_y));
47      CHECK(cudaFree(d_z));
48      return 0;
```

```
49 }
```

在该例子中，去掉第 37 行对同步函数的调用也能成功地捕捉到上述错误信息。这是因为，第 39 行的数据传输函数起到了一种隐式的（implicit）同步主机与设备的作用。在一般情况下，如果要获得精确的出错位置，还是需要显式的（explicit）同步，例如调用 cudaDeviceSynchronize 函数，或者临时将环境变量 CUDA_LAUNCH_BLOCKING 的值设置为 1：

```
$ export CUDA_LAUNCH_BLOCKING=1
```

这样设置之后，所有核函数的调用都将不再是异步的，而是同步的。也就是说，主机调用一个核函数之后，必须等待核函数执行完毕，才能往下走。这样的设置一般来说仅适用于调试程序，因为它会影响程序的性能。

4.2 用 CUDA-MEMCHECK 检查内存错误

CUDA 提供了名为 CUDA-MEMCHECK 的工具集，具体包括 memcheck、 racecheck、initcheck、 synccheck 四个工具。它们可由可执行文件 cuda-memcheck 调用：

```
$ cuda-memcheck --tool memcheck [options] app_name [options]
$ cuda-memcheck --tool racecheck [options] app_name [options]
$ cuda-memcheck --tool initcheck [options] app_name [options]
$ cuda-memcheck --tool synccheck [options] app_name [options]
```

对于 memcheck 工具，可以简化为：

```
$ cuda-memcheck [options] app_name [options]
```

我们这里只给出一个使用 memcheck 工具的例子。如果将文件 add3if.cu 中的 if 语句去掉，编译后用

```
$ cuda-memcheck ./a.out
```

运行程序，作者得到一大串输出，其中最后一行为（读者得到的数字可能不一定是下面的 36）：

```
========= ERROR SUMMARY: 36 error
```

这说明程序有内存错误，与之前的讨论一致。将 if 语句加上，编译后再用

```
$ cuda-memcheck ./a.out
```

运行，将得到简单的输出，其中最后一行为

```
========= ERROR SUMMARY: 0 errors
```

在开发程序时，经常用 CUDA-MEMCHECK 工具集检测内存错误是一个好的习惯。关于 CUDA-MEMCHECK 的更多内容，参见 https://docs.nvidia.com/cuda/cuda-memcheck。最后要强调的是，最有效的防止出错的办法就是认真地写代码，而且在写好之后认真地检查。

第 5 章 获得 GPU 加速的关键

前几章主要关注程序的正确性，没有强调程序的性能（执行速度）。从本章起，我们开始关注 CUDA 程序的性能。在开发 CUDA 程序时往往要验证某些改变是否提高了程序性能，这就需要对程序进行比较精确的计时。所以，我们下面就从给主机和设备函数的计时讲起。

5.1 用 CUDA 事件计时

在 C++ 中，有多种可以对一段代码进行计时的方法，包括使用 GCC 和 MSVC 都有的 `clock` 函数和与头文件 `<chrono>` 对应的时间库、GCC 中的 `gettimeofday` 函数以及 MSVC 中的 `QueryPerformanceCounter` 和 `QueryPerformanceFrequency` 函数等。CUDA 提供了一种基于 CUDA 事件（CUDA Event）的计时方式，可用来给一段 CUDA 代码（可能包含了主机代码和设备代码）计时。为简单起见，我们这里仅介绍基于 CUDA 事件的计时方法。Listing 5.1 给出了使用 CUDA 事件对一段代码进行计时的方式。

Listing 5.1: 本书中常用的计时方式。

```
1  cudaEvent_t start, stop;
2  CHECK(cudaEventCreate(&start));
3  CHECK(cudaEventCreate(&stop));
4  CHECK(cudaEventRecord(start));
5  cudaEventQuery(start); // 此处不能用 CHECK 宏函数（见上一章的讨论）
6
7  需要计时的代码块
8
9  CHECK(cudaEventRecord(stop));
10 CHECK(cudaEventSynchronize(stop));
11 float elapsed_time;
12 CHECK(cudaEventElapsedTime(&elapsed_time, start, stop));
```

```
13 printf("Time = %g ms.\n", elapsed_time);
14
15 CHECK(cudaEventDestroy(start));
16 CHECK(cudaEventDestroy(stop));
```

下面是对该计时方式的解释：

- 第 1 行定义了两个 CUDA 事件类型（cudaEvent_t）的变量 start 和 stop，第 2 行和 3 行用 cudaEventCreate 函数初始化它们。
- 第 4 行将 start 传入 cudaEventRecord 函数，在需要计时的代码块之前记录一个代表开始的事件。
- 第 5 行对处于 TCC 驱动模式的 GPU 来说所可以省略，但对处于 WDDM 驱动模式的 GPU 来说必须保留。这是因为，在处于 WDDM 驱动模式的 GPU 中，一个 CUDA 流（CUDA Stream）中的操作（比如这里的 cudaEventRecord 函数）并不是直接提交给 GPU 执行，而是先提交到一个软件队列，需要添加一条对该流的 cudaEventQuery 操作（或者 cudaEventSynchronize）刷新队列，才能促使前面的操作在 GPU 执行。关于 CUDA 流，会在第 11 章详细讨论，读者暂时不必对此深究。
- 第 7 行代表一个需要计时的代码块，它可以是一段主机代码（例如对一个主机函数的调用），也可以是一段设备代码（例如对一个核函数的调用），还可以是一段混合代码。
- 第 9 行将 stop 传入 cudaEventRecord 函数，在需要计时的代码块之后记录一个代表结束的事件。
- 第 10 行的 cudaStreamSynchronize 函数让主机等待事件 stop 被记录完毕。
- 第 11-13 行调用 cudaEventElapsedTime 函数计算 start 和 stop 这两个事件之间的时间差（单位是毫秒）并输出到屏幕。
- 第 15-16 行调用 cudaEventDestroy 函数销毁 start 和 stop 这两个 CUDA 事件。这是本书中唯一使用 CUDA 事件的地方，故这里不对 CUDA 事件做进一步讨论。下面，我们对前两章讨论过的数组相加程序进行计时。

5.1.1 为 C++ 程序计时

先考虑 C++ 版本的程序。本章的程序 add1cpu.cu 是在第 3 章的程序 add.cpp 的基础上改写的。主要有如下三个方面的改动：

1. 即使该程序中没有使用核函数，我们也将源文件的后缀名改成了 .cu，这样就不用包含一些 CUDA 头文件。若用 .cpp 后缀，用 nvcc 编译时需要明确地增加一些头文件的包含，用 g++ 编译时还要明确地链接一些 CUDA 库。

2. 从本章起，我们用条件编译的方式选择程序中所用浮点数的精度。在程序的开头部分，有如下几行代码：

```
#ifdef USE_DP
    typedef double real;
    const real EPSILON = 1.0e-15;
#else
    typedef float real;
    const real EPSILON = 1.0e-6f;
#endif
```

   当宏 USE_DP 有定义时，程序中的 real 就代表 double，否则代表 float。该宏可以通过编译选项定义（具体见后面的编译命令）。

3. 我们用 CUDA 事件对该程序中函数 add 的调用进行了计时，而且重复了 11 次。我们将忽略第一次测得的时间，因为第一次计算时，机器（无论是 CPU 还是 GPU）都可能处于预热状态，测得的时间往往偏大。我们将根据后 10 次测试的时间计算一个平均值。具体细节见本章的程序 add1cpu.cu。

我们依然用 nvcc 编译程序。这里，有几个编译选项值得注意。首先，C++ 程序的性能显著地依赖于优化选项。我们将总是用 -O3 选项。然后，正如前面提到过的，我们可以用条件编译的方式来选择程序中浮点数的精度。具体地说，如果将 -DUSE_DP 加入编译选项，程序中的宏 USE_DP 将有定义，从而使用双精度浮点数，否则使用单精度浮点数。最后，对本例来说 GPU 架构的指定是无关紧要的，但还是可以指定一个具体的架构选项。

我们首先用如下命令编译程序：

```
$ nvcc -O3 -arch=sm_75 add1cpu.cu
```

这将得到一个使用单精度浮点数的可执行文件。运行该可执行文件，程序将输出其中的 add 函数所花的时间。在作者的笔记本中，该主机函数耗时约 60 毫秒。然后，我们用如下命令编译程序：

```
$ nvcc -O3 -arch=sm_75 -DUSE_DP add1cpu.cu
```

这将得到一个使用双精度浮点数的可执行文件。在该版本中，add 函数耗时约 120 毫秒。我们看到，双精度版本的 add 函数所用时间大概是单精度版本的 add 函数所用时间的两倍，这对于这种访存主导的函数来说是合理的。本章后面会继续讨论这一点。

5.1.2 为 CUDA 程序计时

类似地，我们在第 4 章的 `check1api.cu` 程序的基础上进行修改，用 CUDA 事件对其中的核函数 `add` 进行计时，从而得到本章的 `add2gpu.cu` 程序。我们用命令

```
$ nvcc -O3 -arch=sm_75 add2gpu.cu
```

编译出使用单精度浮点数的可执行文件，用命令

```
$ nvcc -O3 -arch=sm_75 -DUSE_DP add2gpu.cu
```

编译出使用双精度浮点数的可执行文件。在装有 GeForce RTX 2070 的笔记本中测试，使用单精度浮点数时核函数 `add` 所用时间约为 3.3 毫秒，使用双精度浮点数时核函数 `add` 所用时间约为 6.8 毫秒。这两个时间的比值也大概是 2。作者也用其它一些 GPU 进行了测试，结果见表 5.1。可以看到，这个时间比值对每一款 GPU 都是基本适用的。从表中可以看出，该比值与单、双精度浮点数运算峰值的比值没有关系。这是因为，对于数组相加的问题，其执行速度是由显存带宽决定的，而不是由浮点数运算峰值决定的。

我们还可以计算数组相加问题在 GPU 中达到的有效显存带宽（effective memory bandwidth），并与表 5.1 中的理论显存带宽（theoretical memory bandwidth）进行比较。有效显存带宽定义为 GPU 在单位时间内访问设备内存的字节数。以作者笔记本中的 GeForce RTX 2070 和使用单精度浮点数的情形为例，根据表中的数据，其有效显存带宽为：

$$\frac{3\times10^8\times4\ \mathrm{B}}{3.3\times10^{-3}\ \mathrm{s}}\approx360\ \mathrm{GB/s}. \tag{5.1}$$

可见，有效显存带宽略小于理论显存带宽，进一步说明该问题是访存主导的，即该问题中的浮点数运算所占比例可以忽略不计。

表 5.1: 数组相加程序中的核函数在若干 GPU 中的耗时。在“浮点数运算峰值”和“核函数耗时”这两栏，括号前的数字对应于双精度浮点数版本，括号中的数字对应于单精度浮点数版本。

GPU 型号	计算能力	显存带宽	浮点数运算峰值	核函数耗时
Tesla K40	3.5	288 GB/s	1.4 (4.3) TFLOPS	13 (6.5) ms
Tesla P100	6.0	732 GB/s	4.7 (9.3) TFLOPS	4.3 (2.2) ms
Tesla V100	7.0	900 GB/s	7 (14) TFLOPS	3.0 (1.5) ms
GeForce RTX 2070（笔记本）	7.5	448 GB/s	0.2 (6.5) TFLOPS	6.8 (3.3) ms
GeForce RTX 2080ti	7.5	616 GB/s	0.4 (13) TFLOPS	4.3 (2.1) ms

在程序 `add2gpu.cu` 中，我们仅仅对核函数进行了计时。因为我们的 CUDA 程序相对于 C++ 程序多了数据复制的操作，所以我们也尝试将数据复制的操作加入被计时的代码段。

由此得到的程序为 add3memcpy.cu。我们仅用 GeForce RTX 2070 进行测试：使用单精度时，数据复制和核函数调用共耗时 180 毫秒；使用双精度时，它们共耗时 360 毫秒。

从上述测试得到的数据可以看到一个令人惊讶的结果：核函数的运行时间只有数据复制时间的不到 2%。如果将 CPU 与 GPU 之间的数据传输时间也计入，CUDA 程序相对于 C++ 程序得到的不是性能提升，而是性能降低。总之，如果一个程序的计算任务仅仅是将来自主机端的两个数组相加，并且要将结果传回主机端，使用 GPU 就不是一个明智的选择。那么，什么样的计算任务能够用 GPU 获得加速呢？本章余下的部分将回答这个问题。

在 CUDA 工具箱中有一个叫做 nvprof 的可执行文件，可用于对 CUDA 程序进行更多的性能剖析。在使用 nvprof 时，将它置于程序执行命令之前即可：

```
$ nvprof ./a.out
```

如果用上述命令时遇到了类似如下的错误提示：

```
Unable to profile application. Unified Memory profiling failed
```

则可以尝试将运行命令换为：

```
$ nvprof --unified-memory-profiling off ./a.out
```

对程序 add3memcpy.cu 来说，在 GeForce RTX 2070 中使用上述命令，得到部分结果如下（单精度浮点数版本）：

```
Time(%) Time     Calls Avg       Min       Max       Name
47.00%  134.38ms 2     67.191ms 62.854ms 71.527ms [CUDA memcpy HtoD]
40.13%  114.74ms 1     114.74ms 114.74ms 114.74ms [CUDA memcpy DtoH]
12.86%  36.778ms 11    3.3435ms 3.3424ms 3.3501ms add()
```

为排版方便起见，我们将 add() 函数中的参数类型省去了；在原始的输出中函数的参数类型是保留的。这里的第一列是此处列出的每类操作所用时间的百分比，第二列是每类操作用的总时间，第三列是每类操作被调用的次数，第四列是每类操作单次调用所用时间的平均值，第五列是每类操作单次调用所用时间的最小值，第六列是每类操作单次调用所用时间的最大值，第七列是每类操作的名称。从这里的输出可以看出核函数的执行时间以及数据传输所用时间，它们和用 CUDA 事件获得的结果是一致的。

5.2 几个影响 GPU 加速的关键因素

5.2.1 数据传输的比重

从上一节的讨论我们知道，如果一个程序的目的仅仅是计算两个数组的和，那么用 GPU 可能比用 CPU 还要慢。这是因为，花在数据传输（CPU 与 GPU 之间）上的时间

比计算（求和）本身还要多很多。GPU 计算核心和设备内存之间数据传输的峰值理论带宽要远高于 GPU 和 CPU 之间数据传输的带宽。参看表 5.1，典型 GPU 的显存带宽理论值为几百 GB/s，而常用的连接 GPU 和 CPU 内存的 PCIe x16 Gen3 仅有 16 GB/s 的带宽。它们相差几十倍。要获得可观的 GPU 加速，就必须尽量缩减数据传输所花时间的比重。有时候，即使有些计算在 GPU 中的速度并不高，也要尽量在 GPU 中实现，避免过多的数据经由 PCIe 传递。这是 CUDA 编程中最重要的原则之一。

假设计算任务不是做一次数组相加的计算，而是做 10000 次数组相加的计算，而且只需要在程序的开始和结束部分进行数据传输，那么数据传输所占的比重将可以忽略不计。此时，整个 CUDA 程序的性能就大为提高。在本书第二部分的分子动力学模拟程序中，仅仅在程序的开始部分将一些数据从主机复制到设备，然后在程序的中间部分偶尔将一些在 GPU 中计算的数据复制到主机。对这样的计算，用 CUDA 就有可能获得可观的加速。本书第一部分的程序都是一些简短的例子，其中数据传输部分都可能占主导，但我们将主要关注核函数优化。

5.2.2 算术强度

从前面测试的数据我们可以看到，在作者的装有 GeForce RTX 2070 的笔记本中，数组相加的核函数比对应的 C++ 函数快 20 倍左右（这是在没有对 C++ 程序进行深度优化的情况下得到的结果，但本书不讨论对 C++ 程序的深度优化）。这是一个可观的加速比，但远远没有达到极限。其实，对于很多计算问题，能够得到的加速比是更高的。数组相加的问题之所以很难得到更高的加速比，是因为该问题的算术强度（arithmetic intensity）不高。一个计算问题的算术强度指的是其中算术操作的工作量与必要的内存操作的工作量之比。例如，在数组相加的问题中，在对每一对数进行求和时需要先将一对数据从设备内存（以后我们知道它叫做全局内存）中取出来，然后对它们实施求和计算，最后再将计算的结果存放到设备内存。这个问题的算术强度其实是不高的，因为在取两次数据、存一次数据的情况下只做了一次求和计算。在 CUDA 中，设备内存的读写都是代价高昂（比较耗时）的。

对设备内存的访问速度取决于 GPU 的显存带宽。以 GeForce RTX 2070 为例，其显存带宽理论值为 448 GB/s。相比之下，该 GPU 的单精度浮点数计算的峰值性能为 6.5 TFLOPS，意味着该 GPU 的理论寄存器带宽（只考虑浮点数运算，不考虑能同时进行的整数运算）为

$$\frac{4\ \mathrm{B} \times 4\ (\text{number of operands per FMA})}{2\ (\text{number of operations per FMA})} \times 6.5 \times 10^{12}/\mathrm{s} = 52\ \mathrm{TB/s}. \tag{5.2}$$

这里，FMA 指 fused multiply-add 指令，即涉及到四个操作数和两个浮点数操作的运算 $d = a \times b + c$。由此可见，对单精度浮点数来说，该 GPU 中的数据存取比浮点数计算慢一百多倍。如果考虑双精度浮点数，该比例将缩小 32 倍左右。对其它 GPU 也可以做类似的分析。

如果一个问题中需要的不仅仅是简单的单次求和操作，而是更为复杂的浮点数运算，那么就有可能得到更高的加速比。为了得到高的算术强度，我们将之前的程序（包

括 C++ 和 CUDA 的两个版本）中的数组相加函数进行修改。Listing 5.2 给出了修改后的主机函数和核函数。

Listing 5.2: 本章程序 arithmetic1cpu.cu 和 arithmetic2gpu.cu 中的 arithmetic 函数。

```
const real x0 = 100.0;

void arithmetic(real *x, const real x0, const int N)
{
    for (int n = 0; n < N; ++n)
    {
        real x_tmp = x[n];
        while (sqrt(x_tmp) < x0)
        {
            ++x_tmp;
        }
        x[n] = x_tmp;
    }
}

void __global__ arithmetic(real *d_x, const real x0, const int N)
{
    const int n = blockDim.x * blockIdx.x + threadIdx.x;
    if (n < N)
    {
        real x_tmp = d_x[n];
        while (sqrt(x_tmp) < x0)
        {
            ++x_tmp;
        }
        d_x[n] = x_tmp;
    }
}
```

也就是说，核函数中的计算不再是一次相加的计算，而是一个 10000 次的循环，而

且循环条件中还使用了数学函数 sqrt（本章最后一节将介绍 CUDA 中的数学函数）。程序 arithmetic1cpu.cu 可以用如下方式编译和运行（这是用单精度浮点数的情形；如果要用双精度浮点数，需要在编译选项中加上 -DUSE_DP）：

```
$ nvcc -O3 -arch=sm_75 -arithmetic1cpu.cu
$ ./a.out
```

程序 arithmetic2gpu.cu 可以用如下方式编译和运行：

```
$ nvcc -O3 -arch=sm_75 -arithmetic2gpu.cu
$ ./a.out N
```

注意，这里 CUDA 版本的可执行文件在运行时需要提供一个命令行参数 N。该参数将赋值给程序中的变量 N，相关代码如下：

```
if (argc != 2)
{
    printf("usage: %s N\n", argv[0]);
    exit(1);
}
const int N = atoi(argv[1]);
```

继续在装有 GeForce RTX 2070 的笔记本中测试：当数组长度为 10^4 时，主机函数的执行时间是 320 ms（单精度）和 450 ms（双精度）；当数组长度为 10^6 时核函数的执行时间是 28 ms（单精度）和 1000 ms（双精度）。因为核函数和主机函数处理的数组长度相差 100 倍，故在使用单精度和双精度浮点数时，GPU 相对于 CPU 的加速比分别为

$$\frac{320\ \text{ms} \times 100}{28\ \text{ms}} \approx 1100 \tag{5.3}$$

和

$$\frac{450\ \text{ms} \times 100}{1000\ \text{ms}} = 45. \tag{5.4}$$

可见，提高算术强度能够显著地提高 GPU 相对于 CPU 的加速比。另外值得注意的是，当算术强度很高时，GeForce 系列 GPU 的单精度浮点数运算能力就能更加充分地发挥出来。在我们的例子中，单精度版本的核函数是双精度版本的核函数 36 倍之快，接近于理论比值 32，进一步说明该问题是计算主导的，而不是访存主导的。用 GeForce RTX 2080ti 测试程序 arithmetic2gpu.cu，使用单精度和双精度浮点数时核函数的执行时间分别是 15 ms 和 450 ms，相差 30 倍。用 Tesla V100 测试，使用单精度和双精度浮点数时核函数的执行时间分别是 11 ms 和 28 ms，只相差两倍多。可见，对于算术强度很高的问题，在使用双精度浮点数时 Tesla 系列的 GPU 相对于 GeForce 系列的 GPU 有很大的优势，而在使用单精

度浮点数时前者没有显著的优势。对于算术强度不高的问题（例如前面的数组相加问题），Tesla 系列的 GPU 在使用单精度或双精度浮点数时都没有显著的优势。在使用单精度浮点数时，GeForce 系列的 GPU 具有更高的性价比。

5.2.3 并行规模

另一个影响 CUDA 程序性能的因素是并行规模。并行规模可用 GPU 中总的线程数目来衡量。从硬件的角度来看，一个 GPU 由多个流多处理器（Streaming Multiprocessor，简称为 SM）构成，而每个 SM 中有若干 CUDA 核心。每个 SM 是相对独立的。从开普勒到伏特架构，一个 SM 中最多能驻留（reside）的线程个数是 2048。对于图灵架构，该数目是 1024。一块 GPU 中一般有几个到几十个 SM（取决于具体的型号）。所以，一块 GPU 一共可以驻留几万到几十万个线程。如果一个核函数中定义的线程数目远小于这个数的话，就很难得到很高的加速比。

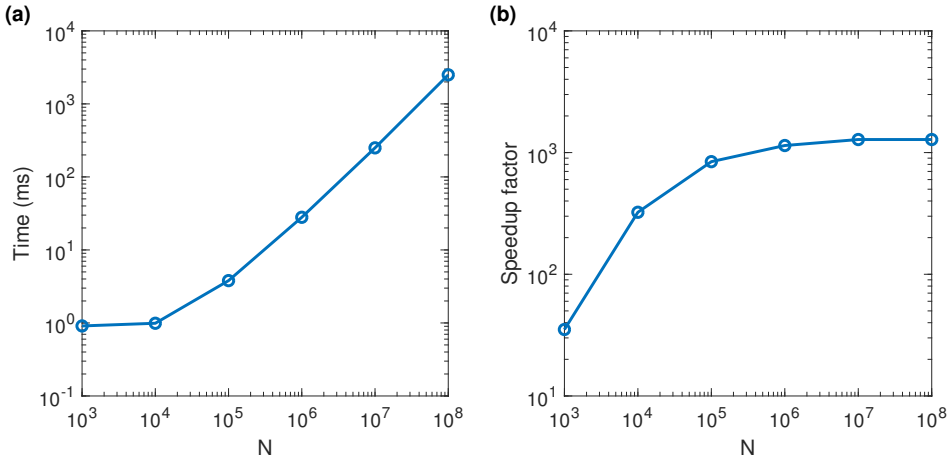

图 5.1: （a）核函数 arithmetic 的执行时间随数组元素个数（也就是线程数目）的变化关系。（b）核函数 arithmetic 相对于对应的主机函数的加速比随数组元素个数的变化关系。测试用的 GPU 为笔记本中的 GeForce RTX 2070。GPU 版本和 CPU 版本的程序都采用单精度浮点数。

为了验证这个论断，我们将 `arithmetic2gpu.cu` 程序中的数组元素个数 `N` 从 10^2 以十倍的间隔增加到 10^7，分别测试核函数的执行时间，结果展示在图 5.1（a) 中。因为 CPU 中的计算时间基本上与数据量成正比，我们可以根据之前的结果计算 `N` 取不同值时 GPU 程序相对于 CPU 程序的加速比。结果显示在图 5.1（b）中。

由图 5.1（a）可知，在数组元素个数 `N` 很大时，核函数的计算时间正比于 `N`；在 `N` 很小时，核函数的计算时间不依赖于 `N` 的值，保持为常数。这两个极限情况都是容易理解的。当 `N` 足够大时，GPU 是满负荷工作的，增加一倍的工作量就会增加一倍的计算时间。反之，当 `N` 不

够大时，GPU 中是有空闲的计算资源的，增加 N 的值并不会增加计算时间。要让 GPU 满负荷工作，核函数中定义的线程总数要不少于某个值，该值在一般情况下和 GPU 中能够驻留的线程总数相当，但也有可能更小。只有在 GPU 满负荷工作的情况下，GPU 中的计算资源才能充分地发挥作用，从而获得较高的加速比。

因为我们的 CPU 程序中的计算是串行的，其性能基本上与数组长度无关，所以 GPU 程序相对于 CPU 程序的加速比在小 N 的极限下几乎是正比于 N 的。在大 N 的极限下，GPU 程序相对于 CPU 程序的加速比接近饱和。总之，对于数据规模很小的问题，用 GPU 很难得到可观的加速。

5.2.4 总结

通过本节的例子，我们看到，一个 CUDA 程序能够获得高性能的必要（但不充分）条件是：

- 数据传输比重较小；
- 核函数的算术强度较高；
- 核函数中定义的线程数目较多。

所以，在编写与优化 CUDA 程序时，一定要想方设法（主要指仔细设计算法）做到以下几点：

- 减少主机与设备之间的数据传输；
- 提高核函数的算术强度；
- 增大核函数的并行规模。

5.3 CUDA 中的数学函数库

在前面的例子中，我们在核函数中使用了求平方根的数学函数。在 CUDA 数学库中，还有很多类似的数学函数，例如幂函数、三角函数、指数函数、对数函数等等。这些函数可以在如下网站查询：http://docs.nvidia.com/cuda/cuda-math-api。建议读者浏览该文档，了解 CUDA 的数学函数库都提供了哪些数学函数。这样，在需要使用的时候就容易想起来。

CUDA 数学库中的函数可以归纳如下：

1. 单精度浮点数内建函数和数学函数（Single Precision Intrinsics and Math functions）。使用该类函数时不需要包含任何额外的头文件。

2. 双精度浮点数内建函数和数学函数（Double Precision Intrinsics and Math functions）。使用该类函数时不需要包含任何额外的头文件。

3. 半精度浮点数内建函数和数学函数（Half Precision Intrinsics and Math functions）。使用该类函数时需要包含头文件 `<cuda_fp16.h>`。本书不涉及此类函数。

4. 整数类型的内建函数（Integer Intrinsics）。使用该类函数时不需要包含任何额外的头文件。本书不涉及此类函数。

5. 类型转换内建函数（Type Casting Intrinsics）。使用该类函数时不需要包含任何额外的头文件。本书不涉及此类函数。

6. 单指令-多数据内建函数（SIMD Intrinsics）。使用该类函数时不需要包含任何额外的头文件。本书不涉及此类函数。

本书将仅涉及单精度和双精度浮点数类型的数学函数和内建函数。其中数学函数（Math Functions）都是经过重载的。例如，求平方根的函数具有如下三种原型：

```
double sqrt(double x);
float sqrt(float x);
float sqrtf(float x);
```

所以，当 `x` 是双精度浮点数时，我们只可以用 `sqrt(x)`；当 `x` 是单精度浮点数时，我们可以用 `sqrt(x)`，也可以用 `sqrtf(x)`。那么综合起来，我们可统一地用双精度函数的版本处理单精度和双精度浮点数类型的变量。

内建函数（Intrinsics）指的是一些准确度较低，但效率较高的函数。例如，有如下版本的求平方根的内建函数：

```
__device__ float __fsqrt_rd (float  x); // round-down mode
__device__ float __fsqrt_rn (float  x); // round-to-nearest-even mode
__device__ float __fsqrt_ru (float  x); // round-up mode
__device__ float __fsqrt_rz (float  x); // round-towards-zero mode
__device__ double __fsqrt_rd (double  x); // round-down mode
__device__ double __fsqrt_rn (double  x); // round-to-nearest-even mode
__device__ double __fsqrt_ru (double  x); // round-up mode
__device__ double __fsqrt_rz (double  x); // round-towards-zero mode
```

在开发 CUDA 程序时，浮点数精度的选择以及数学函数和内建函数之间的选择都要视应用程序的要求而定。举两个例子。在作者开发的分子动力学模拟程序 GPUMD（https://github.com/brucefan1983/GPUMD）中，绝大部分的代码使用了双精度浮点数，只在极

个别的地方使用了单精度浮点数，而且没有使用内建函数。在作者开发的经验势拟合程序 GPUGA（https://github.com/brucefan1983/GPUGA）中，统一使用了单精度浮点数，而且使用了内建函数。之所以这样选择，是因为前者对计算精度要求较高，后者对计算精度要求较低。

第 6 章 CUDA 的内存组织

前一章讨论了几个获得 GPU 加速的必要但不充分条件。在满足那些条件之后，要获得尽可能高的性能，还有很多需要注意的方面，其中最重要的是合理地使用各种设备内存。本章从整体上介绍 CUDA 中的内存组织，为后续章节的讨论打好理论基础。

6.1 CUDA 的内存组织简介

现代计算机中的内存往往存在一种组织结构（hierarchy）。在这种结构中，含有多种类型的内存，分别具有不同的容量和延迟（latency，可以理解为处理器等待内存数据的时间）。一般来说，延迟低（速度高）的内存容量小；延迟高（速度低）的内存容量大。当前被处理的数据一般存放于低延迟、低容量的内存；当前没有被处理但之后将要被处理的大量数据一般存放于高延迟、高容量的内存。相对于不用分级的内存，用这种分级的内存可以降低延迟，提高计算效率。

CPU 和 GPU 中都有内存分级的设计。相对于 CPU 编程来说，CUDA 编程模型向程序员提供更多的控制权。因此，对 CUDA 编程来说，熟悉其内存的分级组织是非常重要的。

表 6.1: CUDA 中设备内存的分类与特征

内存类型	物理位置	访问权限	可见范围	生命周期
全局内存	在芯片外	可读可写	所有线程和主机端	由主机分配与释放
常量内存	在芯片外	仅可读	所有线程和主机端	由主机分配与释放
纹理和表面内存	在芯片外	一般仅可读	所有线程和主机端	由主机分配与释放
寄存器内存	在芯片内	可读可写	单个线程	所在线程
局部内存	在芯片外	可读可写	单个线程	所在线程
共享内存	在芯片内	可读可写	单个线程块	所在线程块

表 6.1 列出了 CUDA 中的几种内存和它们的主要特征。这些特征包括物理位置、设备的访问权限、可见范围、以及对应变量的生命周期。图 6.1 可以进一步帮助理解。本章仅简要介绍 CUDA 中的各种内存，更多细节在后续几章会有进一步讨论。本章会频繁引用前几章讨论过的数组相加的例子。

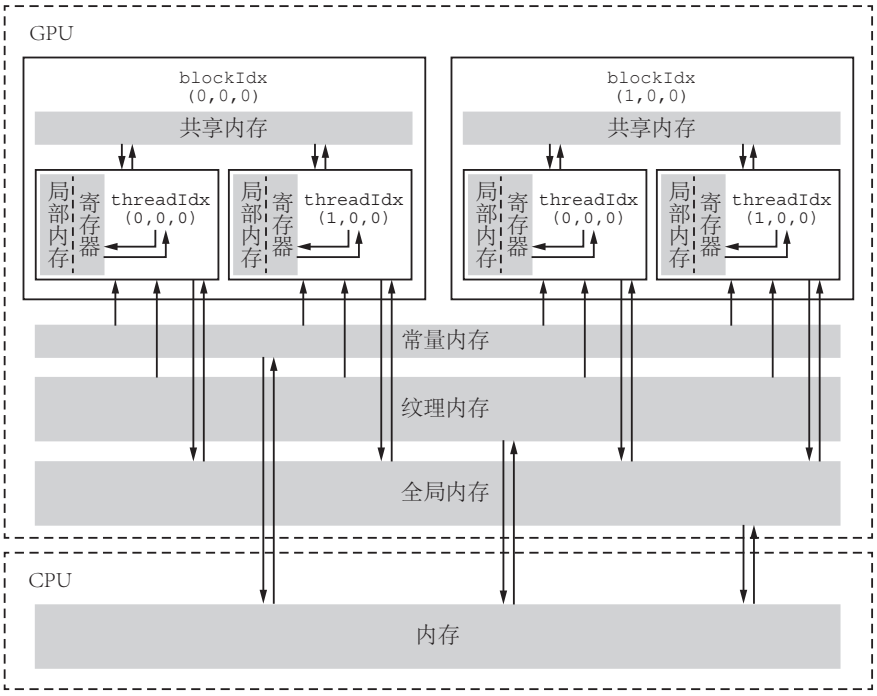

图 6.1: CUDA 中的内存组织示意图。请结合表 6.1 理解此图。图中箭头的方向表示数据可以移动的方向。

6.2 CUDA 中不同类型的内存

6.2.1 全局内存

这里“全局内存”（global memory）的含义是核函数中的所有线程都能够访问其中的数据，和 C++ 中的“全局变量”不是一回事。我们已经用过这种内存。在数组相加的例子中，指针 `d_x`、`d_y` 和 `d_z` 都是指向全局内存的。全局内存由于没有存放在 GPU 的芯片上，因此具有较高的延迟和较低的访问速度。然而，它的容量是所有设备内存中最大的。其容量基本上就是显存容量。第 1 章的表 1.3 列出了几款 GPU 的显存容量。

全局内存的主要角色是为核函数提供数据，并在主机与设备以及设备与设备之间传递数据。首先，我们用 `cudaMalloc` 函数为全局内存变量分配设备内存。然后，可以直接在核函数中访问分配的内存，改变其中的数据值。我们说过，要尽量减少主机与设备之间的数据传输，但有时是不可避免的。可以用 `cudaMemcpy` 函数将主机的数据复制到全局内存，或者反过来。在前几章数组相加的例子中，语句

```
cudaMemcpy(d_x, h_x, M, cudaMemcpyHostToDevice);
```

将中 M 字节的数据从主机复制到设备，而语句

```
cudaMemcpy(h_z, d_z, M, cudaMemcpyDeviceToHost);
```

将中 M 字节的数据从设备复制到主机。还可以将一段全局内存中的数据复制到另一段全局内存中去。例如：

```
cudaMemcpy(d_x, d_y, M, cudaMemcpyDeviceToDevice);
```

的作用就是将首地址为 d_y 的全局内存中 M 字节的数据复制到首地址为 d_x 的全局内存中去。注意，这里必须将数据传输的方向指定为 cudaMemcpyDeviceToDevice 或者 cudaMemcpyDefault。

全局内存可读可写。在数组相加的例子中，语句

```
d_z[n] = d_x[n] + d_y[n];
```

同时体现了全局内存的可读性和可写性。对于线程 n 来说，该语句将变量 d_x 和 d_y 所指全局内存缓冲区的第 n 个元素读出，相加后将结果写入变量 d_z 所指全局内存缓冲区的第 n 个元素。

全局内存对整个网格的所有线程可见。也就是说，一个网格的所有线程都可以访问（读或写）传入核函数的设备指针所指向的全局内存中的全部数据。在上面的语句中，第 n 个线程刚好访问全局内存缓冲区的第 n 个元素，但并不是非要这样。如有需要，第 n 个线程可以访问全局内存缓冲区中的任何一个元素。

全局内存的生命周期（lifetime）不是由核函数决定的，而是由主机端决定的。在数组相加的例子中，由指针 d_x、d_y 和 d_z 所指向的全局内存缓冲区的生命周期就是从主机端用 cudaMalloc 对它们分配内存开始，到主机端用 cudaFree 释放它们的内存结束。在这期间，可以在相同的或不同的核函数中多次访问这些全局内存中的数据。

在处理逻辑上的两维或三维问题时，可以用 cudaMallocPitch 和 cudaMalloc3D 函数分配内存，用 cudaMemcpy2D 和 cudaMemcpy3D 复制数据，释放时依然用 cudaFree 函数。本书不讨论这种内存分配函数以及相应的数据复制函数。

以上所有的全局内存都叫做线性内存（linear memory）。在 CUDA 中还有一种内部构造对用户不透明的（not transparent）全局内存，叫做 CUDA Array。CUDA Array 使用英伟达不对用户公开的数据排列方式，专为纹理拾取服务，本书不讨论。

我们前面介绍的全局内存变量都是动态地分配内存的。在 CUDA 中允许使用静态全局内存变量，其所占内存数量是在编译期间就确定的。而且，这样的静态全局内存变量必须在所有主机与设备函数外部定义，所以是一种“全局的静态全局内存变量”。这里，第一个“全局”的含义与 C++ 中全局变量的含义相同，指的是对应的变量对从其定义之处开始、一个翻译单元内的所有设备函数直接可见。如果采用所谓的分离编译 (separate compiling)，还可以将可见范围进一步扩大，但本书不讨论分离编译。

静态全局内存变量由以下方式在任何函数外部定义：

```
__device__ T x; // 单个变量
__device__ T y[N]; // 固定长度的数组
```

其中，修饰符 __device__ 说明该变量是设备中的变量，而不是主机中的变量，T 是变量的类型，N 是一个整型常数。Listing 6.1 展示了静态全局内存变量的使用方式。该程序将输出：

```
d_x = 1, d_y[0] = 11, d_y[1] = 21.
h_y[0] = 11, h_y[1] = 21.
```

在核函数中，可直接对静态全局内存变量进行访问，并不需要将它们以参数的形式传给核函数。不可在主机函数中直接访问静态全局内存变量，但可以用 cudaMemcpyToSymbol 函数和 cudaMemcpyFromSymbol 函数在静态全局内存与主机内存之间传输数据。这两个 CUDA 运行时 API 函数的原型如下：

```
cudaError_t cudaMemcpyToSymbol
(
    const void* symbol, // 静态全局内存变量名
    const void* src, // 主机内存缓冲区指针
    size_t count, // 复制的字节数
    size_t offset = 0, // 从 symbol 对应设备地址开始偏移的字节数
    cudaMemcpyKind kind = cudaMemcpyHostToDevice // 可选参数
);
cudaError_t cudaMemcpyFromSymbol
(
    void* dst, // 主机内存缓冲区指针
    const void* symbol, // 静态全局内存变量名
    size_t count, // 复制的字节数
    size_t offset = 0, // 从 symbol 对应设备地址开始偏移的字节数
    cudaMemcpyKind kind = cudaMemcpyDeviceToHost // 可选参数
);
```

这两个函数的参数 symbol 可以是静态全局内存变量的变量名，也可以是下面要介绍的常量内存变量的变量名。第 16 行调用 cudaMemcpyToSymbol 函数将主机数组 h_y 中的数据复制到静态全局内存数组 d_y，第 21 行调用 cudaMemcpyFromSymbol 函数将静态全局内存数组 d_y 中的数据复制到主机数组 h_y。这里只是展示静态全局内存的使用方法；我们将在第 10 章讨论一种利用静态全局内存加速程序的技巧。

Listing 6.1: 本章程序 static.cu 的全部代码。

```
#include "error.cuh"
#include <stdio.h>
__device__ int d_x = 1;
__device__ int d_y[2];

void __global__ my_kernel(void)
{
    d_y[0] += d_x;
    d_y[1] += d_x;
    printf("d_x = %d, d_y[0] = %d, d_y[1] = %d.\n", d_x, d_y[0], d_y
        [1]);
}

int main(void)
{
    int h_y[2] = {10, 20};
    CHECK(cudaMemcpyToSymbol(d_y, h_y, sizeof(int) * 2));

    my_kernel<<<1, 1>>>();
    CHECK(cudaDeviceSynchronize());

    CHECK(cudaMemcpyFromSymbol(h_y, d_y, sizeof(int) * 2));
    printf("h_y[0] = %d, h_y[1] = %d.\n", h_y[0], h_y[1]);

    return 0;
}
```

6.2.2 常量内存

常量内存（constant memory）是有常量缓存的全局内存，数量有限，一共仅有 64 KB。它的可见范围和生命周期与全局内存一样。不同的是，常量内存仅可读、不可写。由于有缓存，常量内存的访问速度比全局内存高，但得到高访问速度的前提是一个线程束中的线程（一个线程块中相邻的 32 个线程）要读取相同的常量内存数据。

一个使用常量内存的方法是在核函数外面用 `__constant__` 定义变量，并用前面介绍的 CUDA 运行时 API 函数 `cudaMemcpyToSymbol` 将数据从主机端复制到设备的常量内存后供核函数使用。当计算能力大于等于 2.0 时，给核函数传的参数（传值；不是像全局变量那样传指针）就存放在常量内存，但通过给核函数传参数顶多只能在一个核函数中使用 4 KB 常量内存。

所以，我们其实已经用过了常量内存。在数组相加的例子中，核函数的参数 `const int N` 就是在主机端定义的变量，并通过传值的方式送给核函数中的线程使用的。在核函数中的代码段 `if (n < N)` 中，这个参数 `N` 就被每一个线程使用了。所以，核函数中的每一个线程都知道该变量的值，而且对它的访问比对全局内存的访问要快。除了给核函数传单个的变量，还可以传结构体，同样也是使用常量内存。结构体中可以定义单个的变量，也可以定义固定长度的数组。我们将在第 13 章讨论常量内存的合理使用。

6.2.3 纹理内存和表面内存

纹理内存（Texture memory）和表面内存（Surface memory）类似于常量内存，也是一种具有缓存的全局内存，有相同的可见范围和生命周期，而且一般仅可读（表面内存也可写）。不同的是，纹理内存和表面内存容量更大，而且使用方式和常量内存也不一样。

对于计算能力大于等于 3.5 的 GPU 来说，将某些只读全局内存数据用 `__ldg()` 函数通过只读数据缓存（read-only data cache）读取，既可达到使用纹理内存的加速效果，又让代码简洁。该函数的原型为：

```
T __ldg(const T* address);
```

其中，`T` 是需要读取的数据的类型，`address` 是数据的地址。对帕斯卡和更高的架构来说，全局内存的读取在默认情况下就利用了 `__ldg()` 函数，所以不需要明显地使用它。我们在第 7 章就会讨论该函数的使用。

6.2.4 寄存器

在核函数中定义的不加任何限定符的变量一般来说就存放于寄存器（register）。核函数中定义的不加任何限定符的数组有可能存放于寄存器，但也有可能存放于局部内存（见下）。另外，以前提到过的各种内建变量，例如 `gridDim`、`blockDim`、`blockIdx`、`threadIdx` 以及 `warpSize` 都保存在特殊的寄存器中。在核函数中访问这些内建变量是很高效的。

我们已经使用过寄存器变量。在数组求和的例子中，我们在核函数中有如下语句：

```
const int n = blockDim.x * blockIdx.x + threadIdx.x;
```

这里的 `n` 就是一个寄存器变量。寄存器可读可写。上述语句的作用就是定义一个寄存器变量 `n` 并将赋值号右边计算出来的值赋给它（写入）。在稍后的语句

```
z[n] = x[n] + y[n];
```

中，寄存器变量 n 的值被使用（读出）。

寄存器变量仅仅被一个线程可见。也就是说，每一个线程都有一个变量 n 的副本。虽然在核函数的代码中用了这同一个变量名，但是不同的线程中该寄存器变量的值是可以不同的。每个线程都只能对它的副本进行读写。寄存器的生命周期也与所属线程的生命周期一致，从定义它开始，到线程消失时结束。

寄存器内存在芯片上 (on-chip)，是所有内存中访问速度最高的，但是其数量也很有限。几个不同计算能力的 GPU 中与寄存器和后面要介绍的共享内存有关的技术指标列于表 6.2。该表只包含了少数几个计算能力；更完整的列表见《CUDA C++ Programming Guide》的附录 H。要注意的是，表中列出的寄存器的单位是个，不是字节。例如，64 K 指的是 64×1024 个寄存器。一个寄存器占有 32 比特（4 字节）的内存。所以，一个双精度浮点数将使用两个寄存器。这是在估算寄存器使用量时要注意的。

表 6.2: 几个计算能力的技术指标

计算能力	3.5	6.0	7.0	7.5
GPU 代表	Tesla K40	Tesla P100	Tesla V100	Geforce RTX 2080
SM 寄存器数上限	64 K	64 K	64 K	64 K
单个线程块寄存器数上限	64 K	64 K	64 K	64 K
单个线程寄存器数上限	255	255	255	255
SM 共享内存上限	48 KB	64 KB	96 KB	64 KB
单个线程块共享内存上限	48 KB	48 KB	96 KB	64 KB

6.2.5 局部内存

我们还没有用过局部内存（local memory），但从用法上看，局部内存和寄存器几乎一样。核函数中定义的不加任何限定符的变量有可能在寄存器，也有可能在局部内存。寄存器中放不下的变量，以及索引值不能在编译时就确定的数组，都有可能放在局部内存。这种判断是由编译器自动做的。对于数组相加例子中的变量 n 来说，作者可以肯定它在寄存器，而不是局部内存，因为核函数所用寄存器数量还远远没有达到上限。

虽然局部内存在用法上类似于寄存器，但从硬件来看，局部内存只是全局内存的一部分。所以，局部内存的延迟也很高。每个线程最多能使用高达 512 KB 的局部内存，但使用过多会降低程序性能。

6.2.6 共享内存

我们还没有用过共享内存（shared memory）。共享内存和寄存器类似，存在于芯片上，具有仅次于寄存器的读写速度，数量也有限。表 6.2 列出了与几个计算能力对应的共享内存数量指标。

不同于寄存器的是，共享内存对整个线程块可见，其生命周期也与整个线程块一致。也就是说，每个线程块拥有一个共享内存变量的副本。共享内存变量的值在不同的线程块中可以不同。一个线程块中的所有线程都可以访问该线程块的共享内存变量副本，但是不能访问其它线程块的共享内存变量副本。共享内存的主要作用是减少对全局内存的访问，或者改善对全局内存的访问模式。这些将在第 8 章详细地讨论，所以读者若不明白上一句话的含义，也不要紧。

6.2.7 L1 和 L2 缓存

从费米架构开始，有了 SM 层次的 L1 缓存（一级缓存）和设备（一个设备有多个 SM）层次的 L2 缓存（二级缓存）。它们主要用来缓存全局内存和局部内存的访问，减少延迟。

从硬件的角度来看，开普勒架构中的 L1 缓存和共享内存使用同一块物理芯片；在麦克斯韦和帕斯卡架构中，L1 缓存和纹理缓存统一起来，而共享内存是独立的；在伏特和图灵架构中，L1 缓存、纹理缓存以及共享内存三者统一起来。从编程的角度来看，共享内存是可编程的缓存（共享内存的使用完全由用户操控），而 L1 和 L2 缓存是不可编程的缓存（用户顶多能引导编译器做一些选择）。从开普勒架构开始，在默认情况下全局内存仅使用 L2 缓存，但可以通过添加编译选项来改变默认行为（将在第 7 章具体介绍）。

对某些架构来说，还可以针对单个核函数或者整个程序改变 L1 缓存和共享内存的比例。具体地说：

- 计算能力 3.5：L1 缓存和共享内存一共有 64 KB，可以将共享内存上限设置成 16 KB、32 KB 和 48 KB，其余的归 L1 缓存。默认情况下有 48KB 共享内存。
- 计算能力 3.7：L1 缓存和共享内存一共有 128 KB，可以将共享内存上限设置成 80 KB、96 KB 和 112 KB，其余的归 L1 缓存。默认情况下有 112 KB 共享内存。
- 麦克斯韦架构和帕斯卡架构不允许调整共享内存的上限。
- 伏特架构：统一的（L1/纹理/共享内存）缓存共有 128 KB，共享内存上限可调整为 0、8、16、32、64 或 96 KB。
- 图灵架构：统一的（L1/纹理/共享内存）缓存共有 96 KB，共享内存上限可调整为 32 或 64 KB。

由于以上关于共享内存比例的设置不是很通用，本书不对它们做进一步讨论。感兴趣的读者可阅读《CUDA C++ Programming Guide》和其它资料进一步学习。

6.3 SM 及其占有率

6.3.1 SM 的构成

我们在第 5 章讨论并行规模对 CUDA 程序性能的影响时提到了流多处理器 SM（Streaming Multiprocessor)。一个 GPU 是由多个 SM 构成的。一个 SM 包含如下资源：

- 一定数量的寄存器（参见表 6.2）
- 一定数量的共享内存（参见表 6.2）
- 常量内存的缓存
- 纹理和表面内存的缓存
- L1 缓存
- 两个（计算能力 6.0）或四个（其它计算能力）线程束调度器（warp scheduler)：用于在不同线程的上下文之间迅速地切换，以及为准备就绪的线程束发出执行指令
- 执行核心，包括
  - 若干整型数运算的核心（INT32）
  - 若干单精度浮点数运算的核心（FP32）
  - 若干双精度浮点数运算的核心（FP64）
  - 若干单精度浮点数超越函数（transcendental functions）的特殊函数单元（SFUs, Special Function Units）
  - 若干混合精度的张量核心（tensor cores，由伏特架构引入，适用于机器学习中的低精度矩阵计算；本书不讨论）

6.3.2 SM 的占有率

因为一个 SM 中的各种计算资源是有限的，那么有些情况下一个 SM 中驻留的线程数目就有可能达不到理想的最大值。此时，我们说该 SM 的占有率小于 100%。获得 100% 的占有率并不是获得高性能的必要或充分条件，但一般来说，要尽量让 SM 的占有率不小于某个值，比如 25%，才有可能获得较高的性能。

在第 5 章，我们讨论了并行规模。当并行规模较小时，有些 SM 可能就没有被利用，占有率为零。这是导致程序性能低下的原因之一。当并行规模足够大时，也有可能得到非 100% 的占有率，这就是下面要讨论的情形。

在表 6.2 中，我们列举了一个 SM、一个线程块以及一个线程中能够使用的寄存器和共享内存的上限。在第 2 章，我们还提到了，一个线程块（无论几维的）中的线程数不能超过 1024。要分析 SM 的理论占有率（theoretical occupancy），还需要知道两个指标：

- 一个 SM 中最多能拥有的线程块个数为 $N_{\mathrm{b}} = 16$（开普勒架构和图灵架构）或者 $N_{\mathrm{b}} = 32$（麦克斯韦、帕斯卡和伏特架构）；
- 一个 SM 中最多能拥有的线程个数为 $N_{\mathrm{t}} = 2048$（从开普勒架构到伏特架构）或者 $N_{\mathrm{t}} = 1024$（图灵架构）。

下面在并行规模足够大（即核函数执行配置中定义的总线程数足够多）的前提下分几种情况来分析 SM 的理论占有率：

- 寄存器和共享内存使用量很少的情况。此时，SM 的占有率完全由执行配置中的线程块大小决定。关于线程块大小，读者也许注意到我们之前总是用 128。为什么用这样一个数呢？这是因为，SM 中线程的执行是以线程束为单位的。所以就最好将线程块大小取为线程束大小（32 个线程）的整数倍。举个例子，假设将线程块大小定义为 100，那么一个线程块中将有 3 个完整的线程束（一共 96 个线程）和一个不完整的线程束（只有 4 个线程）。在执行核函数中的指令时，不完整的线程束花的时间和完整的线程束花的时间一样，这就无形中浪费了计算资源。所以，建议将线程块大小取为 32 的整数倍。在该前提下，任何 $\geq N_{\mathrm{t}}/N_{\mathrm{b}}$ 而且能整除 N_{t} 的线程块大小都能得到 100% 的占有率。根据我们列出的数据，线程块大小 ≥ 128 时开普勒架构能获得 100% 的占有率；线程块大小 ≥ 64 时其它架构能获得 100% 的占有率。作者最近几年都用一块开普勒架构的 Tesla K40 开发程序，所以习惯了在一般情况下都用 128 的线程块大小。
- 现在考虑有限的寄存器资源对占有率的约束。我们只针对表中列出的几个计算能力进行分析；读者可类似地分析其它未列出的计算能力。对于表 6.2 中列出的所有计算能力，一个 SM 最多能使用的寄存器个数为 64 K（64×1024）。除了图灵架构，如果我们希望在一个 SM 中驻留最多的线程（2048 个），核函数中的每个线程顶多只能用 32 个寄存器。当每个线程所用寄存器个数大于 64 时，SM 的占有率将小于 50%；当每个线程所用寄存器个数大于 128 时，SM 的占有率将小于 25%。对于图灵架构，同样的占有率允许使用更多的寄存器。
- 最后，我们考虑有限的共享内存对占有率的约束。因为共享内存的数量随着计算能力的上升没有显著的变化规律，我们这里仅针对一个 3.5 的计算能力进行分析。对其它

计算能力可以类似地分析。如果线程块大小为 128，那么每个 SM 要激活 16 个线程块才能有 2048 个线程，达到 100% 的占有率。此时，一个线程块顶多能使用 3 KB 共享内存。在不改变线程块大小的情况下，要达到 50% 的占有率，一个线程块顶多能使用 6 KB 共享内存；要达到 25% 的占有率，一个线程块顶多能使用 12 KB 共享内存。最后，如果一个线程块使用了超过 48 KB 的共享内存，会直接导致核函数无法运行。对其它线程块大小可类似地分析。

- 以上单独分析了线程块大小、寄存器数量以及共享内存数量对 SM 占有率的影响。一般情况下，需要综合以上三点分析。在 CUDA 工具箱中，有一个名为 `CUDA_Occupancy_Calculator.xls` 的 Excel 文档，可用来计算各种情况下的 SM 占有率，感兴趣的读者可以去尝试使用。

最后值得一提的是，用编译器选项 `--ptxas-options=-v` 可以报道每个核函数的寄存器使用数量。CUDA 还提供了核函数的 `__launch_bounds__()` 修饰符和 `--maxrregcount=` 编译选项来让用户分别对一个核函数和所有核函数中寄存器的使用数量进行控制。本书第一部分的示例程序都很简单，使用的寄存器数量都不足以对 SM 的占有率产生影响。在本书第二部分讨论的分子动力学模拟程序中，有些核函数相对较复杂。所以，我们将在第 13 章讨论这些查询与限制寄存器使用数量的功能。

6.4 用 CUDA 运行时 API 函数查询设备

在第 1 章，我们介绍了如何用 nvidia-smi 程序对设备进行某些方面的查询与设置。本节介绍用 CUDA 运行时 API 函数查询所用 GPU 的规格。Listing 6.2 所示程序可用来查询一些到目前为止我们介绍过的 GPU 规格。

Listing 6.2: 本章程序 query.cu 的全部代码。

```
#include "error.cuh"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int device_id = 0;
    if (argc > 1) device_id = atoi(argv[1]);
    CHECK(cudaSetDevice(device_id));

    cudaDeviceProp prop;
```

```
11      CHECK(cudaGetDeviceProperties(&prop, device_id));
12
13      printf("Device id:                                %d\n",
14          device_id);
15      printf("Device name:                              %s\n",
16          prop.name);
17      printf("Compute capability:                       %d.%d\n",
18          prop.major, prop.minor);
19      printf("Amount of global memory:                  %g GB\n",
20          prop.totalGlobalMem / (1024.0 * 1024 * 1024));
21      printf("Amount of constant memory:                %g KB\n",
22          prop.totalConstMem / 1024.0);
23      printf("Maximum grid size:                        %d %d %d\n",
24          prop.maxGridSize[0],
25          prop.maxGridSize[1], prop.maxGridSize[2]);
26      printf("Maximum block size:                       %d %d %d\n",
27          prop.maxThreadsDim[0], prop.maxThreadsDim[1],
28          prop.maxThreadsDim[2]);
29      printf("Number of SMs:                            %d\n",
30          prop.multiProcessorCount);
31      printf("Maximum amount of shared memory per block: %g KB\n",
32          prop.sharedMemPerBlock / 1024.0);
33      printf("Maximum amount of shared memory per SM: %g KB\n",
34          prop.sharedMemPerMultiprocessor / 1024.0);
35      printf("Maximum number of registers per block: %d K\n",
36          prop.regsPerBlock / 1024);
37      printf("Maximum number of registers per SM:    %d K\n",
38          prop.regsPerMultiprocessor / 1024);
39      printf("Maximum number of threads per block: %d\n",
40          prop.maxThreadsPerBlock);
41      printf("Maximum number of threads per SM:   %d\n",
42          prop.maxThreadsPerMultiProcessor);
43
44      return 0;
45  }
```

程序的第 7 行定义了一个 CUDA 中定义好的结构体类型 cudaDeviceProp 的变量 prop。在第 8 行，利用 CUDA 运行时 API 函数 cudaGetDeviceProperties 得到了编号为 device_id = 0 的设备的性质，存放在结构体变量 prop 中。从第 10 行开始，将变量 prop 中的某些成员的值打印出来。在装有 GeForce RTX 2070 的笔记本中得到如下输出：

```
Device id:                                 0
Device name:                               GeForce RTX 2070 with Max-Q Design
Compute capability:                        7.5
Amount of global memory:                   8 GB
Amount of constant memory:                 64 KB
Maximum grid size:                         2147483647 65535 65535
Maximum block size:                        1024 1024 64
Number of SMs:                             36
Maximum amount of shared memory per block: 48 KB
Maximum amount of shared memory per SM:    64 KB
Maximum number of registers per block:     64 K
Maximum number of registers per SM:        64 K
Maximum number of threads per block:       1024
Maximum number of threads per SM:          1024
```

读者可以尝试在自己的系统中运行该程序，确保能够理解每一个规格的含义。在本例中，我们选择查询编号为 0 的设备。如果读者的系统中有不止一块 GPU，而且不想查询第 0 号设备，则可以修改第 6 行的设备编号。值得说明的是，如果读者想用编号为 1 的 GPU 执行程序，而不想用默认的编号为 0 的 GPU，则可以在调用任何 CUDA 运行时 API 函数之前写下如下语句：

```
CHECK(cudaSetDevice(1));
```

另外，读者还可以回顾一下在第 1 章介绍过的用 nvidia-smi 程序在命令行选择 GPU 的方法。在 CUDA 工具箱中，有一个名为 deviceQuery.cu 的程序，可以输出更多的信息。

第 7 章 全局内存的合理使用

在第 6 章，我们抽象地介绍了 CUDA 中的各种内存。从本章开始，我们将通过实例讲解各种内存的合理使用。在各种设备内存中，全局内存具有最低的访问速度（最高的延迟），往往是一个 CUDA 程序性能的瓶颈，所以值得特别地关注。本章讨论全局内存的合理使用。

7.1 全局内存的合并与非合并访问

对全局内存的访问将触发内存事务（memory transaction），也就是数据传输（data transfer）。在第 6 章我们提到，从费米架构开始，有了 SM 层次的 L1 缓存和设备层次的 L2 缓存，可以用于缓存全局内存的访问。其中，L1 缓存仅用于对全局内存的读取，不会用于对全局内存的写入。本书只关注开普勒和以上架构的 GPU，其中对全局内存的读取在默认情况下只经过 L2 缓存，但可以通过 `nvcc` 的编译选项 `-Xptxas -dlcm=ca` 选择同时使用 L1 缓存。在启用了 L1 缓存的情况下，对全局内存的读取将首先尝试经过 L1 缓存；如果未中，则接着尝试经过 L2 缓存；如果再次未中，则直从 DRAM 读取。一次数据传输处理的数据量在默认情况下是 32 字节。

关于全局内存的访问模式，有合并（coalesced）与非合并（uncoalesced）之分。合并访问指的是一个线程束对全局内存的一次访问请求（读或者写）导致最少数量的数据传输，否则称访问是非合并的。定量地说，可以定义一个合并度（degree of coalesing），它等于线程束请求的字节数除以由该请求导致的所有数据传输处理的字节数。如果所有数据传输中处理的数据都是线程束所需要的，那么合并度就是 100%，即对应合并访问。所以，也可以将合并度理解为一种资源利用率。利用率越高，核函数中与全局内存访问有关的部分的性能就更好；利用率低则意味着对显存带宽的浪费。本节后面主要探讨合并度与全局内存访问模式之间的关系。

为简单起见，我们主要以全局内存的读取和仅使用 L2 缓存的情况为例进行下述讨论。在此情况下，一次数据传输指的就是将 32 字节的数据从全局内存（DRAM）通过 32 字节的 L2 缓存片段（cache sector）传输到 SM。考虑一个线程束访问单精度浮点数类型的全局内存变量的情形。因为一个单精度浮点数占有四个字节，故该线程束将请求 128 字节的数据。在理想情况下（即合并度为 100% 的情况），这将仅触发 $128/32 = 4$ 次用 L2 缓存的数据传输。那么，在什么情况下会导致多于 4 次数据传输呢？

为了回答这个问题，我们首先需要了解数据传输对数据地址的要求：在一次数据传输中，从全局内存转移到 L2 缓存的一片内存的首地址一定是一个最小粒度（这里是 32 字节）的整数倍。例如，一次数据传输只能从全局内存读取地址从 0 到 31 字节、32 到 63 字节、64 到 95 字节、96 到 127 字节等片段的数据。如果线程束请求的全局内存数据的地址刚好为 0-127 字节，或者 128-255 字节等，就能与 4 次数据传输所处理的数据完全吻合。这种情况下的访问就是合并访问。

读者也许会问：如何保证一次数据传输中内存片段的首地址为最小粒度的整数倍呢？或者问：如何控制所使用的全局内存的地址呢？答案是：用 CUDA 运行时 API 函数（例如我们常用的 `cudaMalloc`）分配的内存的首地址至少是 256 字节的整数倍。

下面我们通过几个具体的核函数列举几种常见的内存访问模式及其合并度：

1. 顺序的合并访问。我们考察如下的核函数和相应的调用：

```
void __global__ add(float *x, float *y, float *z)
{
    int n = threadIdx.x + blockIdx.x * blockDim.x;
    z[n] = x[n] + y[n];
}
add<<<128, 32>>>(x, y, z);
```

   其中，`x`、`y` 和 `z` 是由 `cudaMalloc()` 分配全局内存的指针。很容易看出，核函数中对这几个指针所指内存区域的访问都是合并的。例如，第一个线程块中的线程束将访问数组 `x` 中第 0-31 个元素，对应 128 字节的连续内存，而且首地址一定是 256 字节的整数倍。这样的访问只需要 4 次数据传输即可完成，所以是合并访问，合并度为 100%。

2. 乱序的合并访问。将上述核函数稍作修改：

```
void __global__ add_permuted(float *x, float *y, float *z)
{
    int tid_permuted = threadIdx.x ^ 0x1;
    int n = tid_permuted + blockIdx.x * blockDim.x;
    z[n] = x[n] + y[n];
}
add_permuted<<<128, 32>>>(x, y, z);
```

   其中，`threadIdx.x ^ 0x1` 是某种置换操作，作用是将 0-31 之间的整数做某种置换（交换两个相邻的数）。第一个线程块中的线程束将依然访问数组 `x` 中第 0-31 个元素，

只不过线程号与数组元素指标不完全一致而已。这样的访问是乱序的（或者交叉的）合并访问，合并度也为 100%。

3. 不对齐的非合并访问。将第一个核函数稍作修改：

```
void __global__ add_offset(float *x, float *y, float *z)
{
    int n = threadIdx.x + blockIdx.x * blockDim.x + 1;
    z[n] = x[n] + y[n];
}
add_offset<<<128, 32>>>(x, y, z);
```

第一个线程块中的线程束将访问数组 x 中第 1-32 个元素。假如数组 x 的首地址为 256 字节（256 字节的整数倍），该线程束将访问数组 x 的 260-387 字节。这将触发 5 次数据传输，对应的内存地址分别是：256-287 字节、288-319 字节、320-351 字节、352-383 字节和 384-415 字节。这样的访问属于不对齐的非合并访问，合并度为 $4/5 = 80\%$。

4. 跨越式的非合并访问。将第一个核函数改写如下：

```
void __global__ add_stride(float *x, float *y, float *z)
{
    int n = blockIdx.x + threadIdx.x * gridDim.x;
    z[n] = x[n] + y[n];
}
add_stride<<<128, 32>>>(x, y, z);
```

第一个线程块中的线程束将访问数组 x 中指标为 0、128、256、384 等的元素。因为这里的每一对数据都不在一个连续的 32 字节的内存片段，故该线程束的访问将触发 32 次数据传输。这样的访问属于跨越式的非合并访问，合并度为 $4/32 = 12.5\%$。

5. 广播式的非合并访问。将第一个核函数改写如下：

```
void __global__ add_broadcast(float *x, float *y, float *z)
{
    int n = blockIdx.x + threadIdx.x * gridDim.x;
    z[n] = x[0] + y[n];
}
add_broadcast<<<128, 32>>>(x, y, z);
```

第一个线程块中的线程束将一致地访问数组 x 中的第 0 个元素。这只需要一次数据传输（处理 32 字节的数据），但由于整个线程束只请求了 4 字节的数据，故合并度为 $4/32 = 12.5\%$。这样的访问属于广播式的非合并访问。这样的访问（如果是读数据的话）适合采用前一章提到的常量内存。具体的例子见第 13 章。

7.2 例子：矩阵转置

本节将通过一个矩阵转置的例子讨论全局内存的合理使用。矩阵转置是线性代数中一个基本的操作。我们这里仅仅考虑行数与列数相等的矩阵，即方阵。学完本节后，读者可以思考如何在 CUDA 中对非方阵进行转置。

假设一个矩阵 A 的矩阵元为 A_{ij}，则其转置矩阵 $B = A^T$ 的矩阵元为

$$B_{ij} = \left(A^T\right)_{ij} = A_{ji}. \tag{7.1}$$

例如，取

$$A = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}, \tag{7.2}$$

则其转置矩阵为

$$B = A^T = \begin{pmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{pmatrix}. \tag{7.3}$$

7.2.1 矩阵复制

在讨论矩阵转置之前，我们先考虑一个更简单的问题：矩阵复制，即形如 $B = A$ 的计算。Listing 7.1 给出了矩阵复制核函数 copy 的定义和调用。

Listing 7.1: 本章程序 matrix.cu 中的 copy 函数及其调用。

```
1 __global__ void copy(const real *A, real *B, const int N)
2 {
3     const int nx = blockIdx.x * TILE_DIM + threadIdx.x;
4     const int ny = blockIdx.y * TILE_DIM + threadIdx.y;
5     const int index = ny * N + nx;
6     if (nx < N && ny < N)
```

```
7     {
8         B[index] = A[index];
9     }
10 }
11
12 const int grid_size_x = (N + TILE_DIM - 1) / TILE_DIM;
13 const int grid_size_y = grid_size_x;
14 const dim3 block_size(TILE_DIM, TILE_DIM);
15 const dim3 grid_size(grid_size_x, grid_size_y);
16 copy<<<grid_size, block_size>>>(d_A, d_B, N);
```

首先，我们说明一下，在核函数中可以直接使用在函数外部由 `#define` 或 `const` 定义的常量，包括整型和浮点型常量，但是在使用微软的编译器（MSVC）时有一个限制，即不能在核函数中使用在函数外部由 `const` 定义的浮点型常量。在本例中，`TILE_DIM` 是一个整型常量，在文件的开头定义：

```
const int TILE_DIM = 32; // C++ 风格
```

它可以等价地写为

```
#define TILE_DIM 32 // C 风格
```

可以在核函数中直接使用该常量的值，但要记住不能在核函数中使用这种常量的引用或者地址。

再看核函数 `copy` 的执行配置。在调用核函数 `copy` 时，我们用了两维的网格和两维的线程块。在该问题中，并不是一定要使用两维的网格和线程块，因为矩阵中的数据排列本质上依然是一维的。然而，在后面的矩阵转置问题中，使用两维的网格和线程块更为方便。为了保持一致（我们将对比程序中几个核函数的性能），我们这里也用两维的网格和线程块。如上所述，程序中的 `TILE_DIM` 是一个整型常量，取值为 32，指的是一片（tile）矩阵的维度（dimension，即行数）。我们将一片一片地处理一个大矩阵。其中的一片是一个 32×32 的矩阵。每一个两维的线程块将处理一片矩阵。线程块的维度和一片矩阵的维度一样大，如第 14 行所示。和线程块一致，网格也用两维的，维度为待处理矩阵的维度 `N` 除以线程块维度，如第 12-13 行和第 15 行所示。举个例子。假如 `N` 为 128，则 `grid_size_x` 和 `grid_size_y` 都是 $128/32 = 4$。也就是说，核函数所用网格维度为 4×4，线程块维度为 32×32。此时在核函数 `copy` 中的 `gridDim.x` 和 `gridDim.y` 都等于 4，而 `blockDim.x` 和 `blockDim.y` 都等于 32。读者应该注意到，一个线程块中总的线程数目为 1024，刚好为所允许的最大值。

最后看核函数 copy 的实现。第 3 行将矩阵的列指标 nx 与带 .x 后缀的内建变量联系起来，而第 4 行将矩阵的行指标 ny 与带 .y 后缀的内建变量联系起来。第 5 行将上述行指标与列指标结合起来转化成一维指标 index。第 6-9 行在行指标和列指标都不越界的情况下将矩阵 A 的第 index 个元素复制给矩阵 B 的第 index 个元素。

我们来分析一下核函数中对全局内存的访问模式。在第 2 章，我们介绍过，对于多维数组，x 维度的线程指标 threadIdx.x 是最内层的（变化最快），所以相邻的 threadIdx.x 对应相邻的线程。从核函数中的代码可知，相邻的 nx 对应相邻的线程，也对应相邻的数组元素（对 A 和 B 都成立）。所以，在核函数中，相邻的线程访问了相邻的数组元素，在没有内存不对齐的情况下属于前一节介绍的顺序的合并访问。我们取 N = 10000 并在 GeForce RTX 2080ti 中进行测试。采用单精度浮点数，核函数的执行时间为 1.6 ms。根据该执行时间，有效的显存带宽为 500 GB/s，略小于该 GPU 的理论显存带宽 616 GB/s。测试矩阵复制计算的性能是为后面讨论矩阵转置核函数的性能确定一个可以比较的基准。第 5 章讨论过有效显存带宽的计算，读者若忘记了可以去回顾一下。

7.2.2 使用全局内存进行矩阵转置

上一小节讨论了矩阵复制的计算。本小节讨论矩阵转置的计算。为此，我们回顾一下上一小节的矩阵复制核函数中的如下语句：

```
const int index = ny * N + nx;
if (nx < N && ny < N) B[index] = A[index];
```

为了便于理解，我们首先将这两条语句写成一条语句：

```
if (nx < N && ny < N) B[ny * N + nx] = A[ny * N + nx];
```

从数学的角度来看，这相当于做了 $B_{ij} = A_{ij}$ 的操作。如果要实现矩阵转置，即 $B_{ij} = A_{ji}$ 的操作，可以将上述代码换成

```
if (nx < N && ny < N) B[nx * N + ny] = A[ny * N + nx];
```

或者

```
if (nx < N && ny < N) B[ny * N + nx] = A[nx * N + ny];
```

以上两条语句都能实现矩阵转置，但是它们将带来不同的性能。与它们对应的核函数分别为 transpose1 和 transpose2，分别列于 Listing 7.2 和 Listing 7.3。可以看出，在核函数 transpose1 中，对矩阵 A 中数据的访问（读取）是顺序的，但对矩阵 B 中数据的访问（写入）不是顺序的。在核函数 transpose2 中，对矩阵 A 中数据的访问（读取）不是顺序的，但对矩阵 B 中数据的访问（写入）是顺序的。在不考虑数据是否对齐的情况下，我们可以说核

函数 transpose1 对矩阵 A 和 B 的访问分别是合并的和非合并的，而核函数 transpose2 对矩阵 A 和 B 的访问分别是非合并的和合并的。

Listing 7.2: 本章程序 matrix.cu 中的 transpose1 核函数。

```
__global__ void transpose1(const real *A, real *B, const int N)
{
    const int nx = blockIdx.x * blockDim.x + threadIdx.x;
    const int ny = blockIdx.y * blockDim.y + threadIdx.y;
    if (nx < N && ny < N)
    {
        B[nx * N + ny] = A[ny * N + nx];
    }
}
```

继续用 GeForce RTX 2080ti 测试相关核函数的执行时间（采用单精度浮点数计算）。核函数 transpose1 的执行时间为 5.3 ms，而核函数 transpose2 的执行时间为 2.8 ms。以上两个核函数中都有一个合并访问和一个非合并访问，但为什么性能差别那么大呢？这是因为，在核函数 transpose2 中，读取操作虽然是非合并的，但利用了第 6 章提到的只读数据缓存的加载函数 __ldg()。从帕斯卡架构开始，如果编译器能够判断一个全局内存变量在整个核函数的范围都只可读（例如这里的矩阵 A），则会自动用函数 __ldg() 读取全局内存，从而对数据的读取进行缓存，缓解非合并访问带来的影响。对于全局内存的写入，则没有类似的函数可用。这就是以上两个核函数性能差别的根源。所以，在不能同时满足读取和写入都是合并的情况下，一般来说应当尽量做到合并的写入。

Listing 7.3: 本章程序 matrix.cu 中的 transpose2 核函数。

```
__global__ void transpose2(const real *A, real *B, const int N)
{
    const int nx = blockIdx.x * blockDim.x + threadIdx.x;
    const int ny = blockIdx.y * blockDim.y + threadIdx.y;
    if (nx < N && ny < N)
    {
        B[ny * N + nx] = A[nx * N + ny];
    }
```

```
9 }
```

对于开普勒和麦克斯韦架构，默认情况下不会使用 `__ldg()` 函数。用 Tesla K40 测试（采用单精度浮点数计算），核函数 `transpose1` 的执行时间短一些，为 12 ms，而核函数 `transpose2` 的执行时间长一些，为 23 ms。这和用 GeForce RTX 2080ti 测试的结果是相反的。在使用开普勒和麦克斯韦架构的 GPU 时，需要明显地使用 `__ldg()` 函数。例如，可将核函数 `transpose2` 改写为核函数 `transpose3`，其中使用 `__ldg()` 函数的代码行为：

```
if (nx < N && ny < N) B[ny * N + nx] = __ldg(&A[nx * N + ny]);
```

完整的核函数见 Listing 7.4。该版本的核函数在 Tesla K40 中的执行时间为 8 ms，比核函数 `transpose1` 的执行时间短一些了。

Listing 7.4: 本章程序 matrix.cu 中的 transpose3 核函数。

```
1 __global__ void transpose3(const real *A, real *B, const int N)
2 {
3     const int nx = blockIdx.x * blockDim.x + threadIdx.x;
4     const int ny = blockIdx.y * blockDim.y + threadIdx.y;
5     if (nx < N && ny < N)
6     {
7         B[ny * N + nx] = __ldg(&A[nx * N + ny]);
8     }
9 }
```

除了利用只读数据缓存加速非合并的访问，有时还可以利用共享内存将非合并的全局内存访问转化为合并的。我们将在第 8 章讨论这个问题。

第 8 章 共享内存的合理使用

第 7 章讨论了全局内存的合理使用，本章接着讨论共享内存的合理使用。共享内存是一种可被程序员直接操控的缓存，主要作用有两个：一个是减少核函数中对全局内存的访问次数，实现高效的线程块内部的通信，另一个是提高全局内存访问的合并度。我们将通过两个具体的例子阐明共享内存的合理使用，包括一个数组规约的例子和第 7 章讨论过的矩阵转置的例子。其中，数组规约是一个非常适合学习 CUDA 编程的例子，通过它可以了解 CUDA 编程的很多方面，例如第 9 章要介绍的原子函数以及第 10 章要介绍的线程束内的函数和协作组。

8.1 例子：数组归约计算

考虑一个有 `N` 个元素的数组 `x`，假如我们需要计算该数组中所有元素的和，即 `sum = x[0] + x[1] + ... + x[N - 1]`。Listing 8.1 给出了一个实现该计算的 C++ 函数。

Listing 8.1: 本章程序 reduce1cpu.cu 中的 reduce 函数。

```
real reduce(const real *x, const int N)
{
    real sum = 0.0;
    for (int n = 0; n < N; ++n)
    {
        sum += x[n];
    }
    return sum;
}
```

在这个例子中，我们考虑一个长度为 $N = 10^8$ 的一维数组。在主函数中，我们将每个数组元素初始化为 1.23。接着，调用函数 `reduce` 并计时。在使用双精度浮点数时，该程序

输出：

```
sum = 123000000.110771.
```

该结果在前 9 位有效数字都正确，从第 10 位开始有错误。在使用单精度浮点数时，该程序输出：

```
sum = 33554432.000000.
```

该结果完全错误。这是因为，在累加计算中出现了所谓的“大数吃小数”的现象。单精度浮点数只有 6、7 位精确的有效数字。在上面的函数 `reduce` 中，将变量 `sum` 的值累加到三千多万后，再将它和 1.23 相加，其值就不再增加了（小数被大数“吃掉了”，但大数并没有变化）。已经发展出更加安全的求和算法，例如 Kahan 求和算法，但本书不讨论。后面会看到，我们的 CUDA 实现要比上述 C++ 实现稳健（robust）得多，使用单精度浮点数时结果也相当准确。这里，我们先关注计算效率。在作者的系统中测试，无论是使用单精度还是双精度浮点数，函数 `reduce` 的执行时间都是约 100 ms。下面我们讨论对应的 CUDA 程序的开发与优化。

8.1.1 仅使用全局内存

数组规约的并行计算显然比数组相加的并行计算复杂一些。对于数组相加的并行计算问题，我们只需要定义和数组元素一样多的线程，让一个线程去对两个数求和即可。对于数组规约的并行计算问题，我们是要从一个数组出发，最终得到一个数。所以，必须使用某种迭代方案。假如数组元素个数是 2 的次幂（我们稍后会去掉这个假设），我们可以将数组后半部分的各个元素与前半部分对应的数组元素相加。如果重复此过程，最后得到的第一个数组元素就是最初的数组中各个元素的和。这就是所谓的折半规约（binary reduction）法。假设使用一维网格和线程块，且将核函数的网格大小与线程块大小的乘积取为 `N`，初学者可能会写出如 8.2 所示的核函数，并认为核函数执行之后数组 `d_x` 的和就保存在 `d_x[0]` 中了。然而，用该核函数并不能得到正确的结果。这是因为，对于多线程的程序，两个不同线程中指令的执行次序可能和代码中所展现的次序不同。为了方便分析，我们将上述核函数中循环的前两次迭代明显地写出来：

```
if (n < N / 2) { d_x[n] += d_x[n + N / 2]; }
if (n < N / 4) { d_x[n] += d_x[n + N / 4]; }
```

考察对数组元素 `d_x[N / 4]` 的操作。在第一次迭代中（上述第一行）会有向数组元素 `d_x[N / 4]` 写入数据的操作（由线程 `n = N / 4` 执行）；在第二次迭代中（上述第二行）会有从 `d_x[N / 4]` 取出数据的操作（由线程 `n = 0` 执行）。有一种可能的情况是：在线程 `n = 0` 开始执行第二行语句时，线程 `n = N / 4` 还没执行完第一行语句。如果这种情况发生了，就有可能得到预料之外的结果。

Listing 8.2: 一个错误的规约核函数。

```
1 void __global__ reduce(real *d_x, int N)
2 {
3     int n = blockDim.x * blockIdx.x + threadIdx.x;
4     for (int offset = N / 2; offset > 0; offset /= 2)
5     {
6         if (n < offset) { d_x[n] += d_x[n + offset]; }
7     }
8 }
```

要保证核函数中语句的执行顺序与出现顺序一致，就必须使用某种同步机制。在 CUDA 中，提供了一个同步函数 `__syncthreads`。该函数只能用在核函数中，其最简单的用法是不带任何参数：

```
__syncthreads();
```

该函数保证一个线程块中的所有线程（或者说所有线程束）在执行该语句后面的语句之前都完全执行了该语句前面的语句。然而，该函数只是针对同一个线程块中的线程的；不同线程块中线程的执行次序依然是不确定的。

既然函数 `__syncthreads` 能够同步单个线程块中的线程，那么我们就利用该功能让每个线程块对其中的数组元素进行规约。Listing 8.3 给出了实现该方案的核函数。

Listing 8.3: 本章程序 reduce2gpu.cu 中仅使用全局内存的规约核函数。

```
1  void __global__ reduce_global(real *d_x, real *d_y)
2  {
3      const int tid = threadIdx.x;
4      real *x = d_x + blockDim.x * blockIdx.x;
5
6      for (int offset = blockDim.x >> 1; offset > 0; offset >>= 1)
7      {
8          if (tid < offset)
9          {
10             x[tid] += x[tid + offset];
11         }
12         __syncthreads();
```

```
13      }
14
15      if (tid == 0)
16      {
17          d_y[blockIdx.x] = x[0];
18      }
19  }
```

下面是该核函数中值得注意的地方：

- 核函数的第 4 行定义了一个指针 x。赋值符号的右边是（动态）数组 d_x 中第 blockDimx.x * blockIdx.x 个元素的地址。所以，第 4 行也可写成

  ```
  real *x = &d_x[blockDim.x * blockIdx.x];
  ```

  这样定义的 x 在不同的线程块中指向全局内存中不同的地址，使得我们可以在不同的线程块中对数组 d_x 中不同的部分进行规约。具体地说，每一个线程块处理 blockDim.x 个数据。我们这里不再假设 N 是 2 的整数次方，但假设 N 能够被 blockDim.x 整除，而且假设 blockDim.x 是 2 的整数次方（作者用的他最喜欢的线程块大小 128）。

- 第 6-13 行就是在各个线程块内对其中的数据独立地进行规约。第 12 行的同步语句保证了同一个线程块内的线程按照代码出现的顺序执行指令。至于两个不同线程块中的线程，则不一定按照代码出现的顺序执行指令，但这不影响程序的正确性。这是因为，在该核函数中，每个线程块都处理不同的数据，相互之间没有依赖。总结起来就是说：一个线程块内的线程需要合作，所以需要同步；两个线程块之间不需要合作，所以不需要同步。

- 核函数的第 7 行也值得注意。这里我们将 blockDim.x / 2 写成了 blockDim.x >> 1，并将 offset /= 2 写成了 offset >>= 1。这是利用了位操作。以上不同写法在结果上的等价性要求 blockDim.x 和 offset 都是 2 的整数次幂。在核函数中，位操作比对应的整数操作高效。当所涉及的变量在编译期间就知道其可能的取值时，编译器会自动用位操作取代相应的整数操作，但明显地使用位操作也是不错的做法。

- 该核函数仅仅将一个长度为 10^8 的数组 d_x 规约到一个长度为 $10^8/128$ 的数组 d_y。为了计算整个数组元素的和，我们将数组 d_y 从设备复制到主机，并在主机继续对数组 d_y 规约，得到最终的结果。这样做不是很高效，但我们暂时先这样做。

用如下命令编译（其中的 -O3 选项是针对主机端代码的）

```
$ nvcc -arch=sm_75 -O3 reduce2gpu.cu
```

用装有 GeForce RTX 2070 的笔记本测试，使用单精度浮点数时，全部计算（包括核函数执行、将数组 d_y 从设备复制到主机以及在主机中对数组 d_y 规约）所花时间为 5.8 毫秒，大概是 C++ 程序的 17 倍快。

8.1.2 使用共享内存

我们注意到，在前一个版本的核函数中，对全局内存的访问是很频繁的。我们介绍过，全局内存的访问速度是所有内存中最低的，应该尽量减少对它的使用。所有设备内存中，寄存器是最高效的，但在需要线程合作的问题中，用仅对单个线程可见的寄存器是不够的。我们需要使用对整个线程块可见的共享内存。

在核函数中，要将一个变量定义为共享内存变量，就要在定义语句中加上一个限定符 __shared__。一般情况下，我们需要的是一个长度等于线程块大小的数组。在当前问题中，我们可以定义如下共享内存数组变量：

```
__shared__ real s_y[128];
```

如果没有限定符 __shared__，该语句将极有可能定义一个长度为 128 的局部数组。注意：作者喜欢用前缀 s_ 给共享内存变量命名，而用前缀 d_ 给全局内存变量命名，虽然这并不是必须的。需要强调的是，在一个核函数中定义一个共享内存变量，就相当于在每一个线程块中有了一个该变量的副本。每个副本都不一样，虽然它们共用一个变量名。核函数中对共享内存变量的操作都是同时作用在所有的副本上的。这种并行的特征在使用共享内存时需要铭记在心。

Listing 8.4 给出了使用了共享内存的规约核函数。我们仔细分析该核函数：

- 第 6 行定义了共享内存数组 s_y[128]。
- 第 7 行将全局内存中的数据复制到共享内存中。这里用到了前面说过的共享内存的特征：每个线程块都有一个共享内存变量的副本。第 8 行的语句所实现的功能可以展开如下：
  - 当 bid 等于 0 时，将全局内存中第 0 到 blockDim.x - 1 个数组元素复制给第 0 个线程块的共享内存变量副本;
  - 当 bid 等于 1 时，将全局内存中第 blockDim.x 到 2 * blockDim.x - 1 个数组元素复制给第 1 个线程块的共享内存变量副本;

  - 因为这里有 `n < N` 的判断，所以该函数能够处理 `N` 不是线程块大小的整数倍的情形。此时，最后一个线程块中与条件 `n >= N` 对应的共享内存数组元素将被赋值为 0，不对规约（求和）的结果产生影响。

- 第 8 行调用函数 `__syncthreads` 进行线程块内的同步。在利用共享内存进行线程块之间的合作（通信）之前，都要进行同步，以确保共享内存变量中的数据对线程块内的所有线程来说都准备就绪。
- 第 10-18 行的规约计算就用共享内存变量替换了原来的全局内存变量。这里也要记住：每个线程块都对其中的共享内存变量副本进行操作。在规约过程结束后，每一个线程块中的 `s_y[0]` 副本就保存了若干数组元素的和。
- 因为共享内存变量的生命周期仅仅在核函数内，所以必须在核函数结束之前将共享内存中的某些结果保存到全局内存，如第 20-23 行所示。这里的判断 `if (tid == 0)` 保证其中的语句在一个线程块中仅被执行一次。该语句的作用可以展开如下：
  - 当 `bid` 等于 0 时，将第 0 个线程块中的 `s_y[0]` 副本复制给 `d_y[0]`;
  - 当 `bid` 等于 1 时，将第 1 个线程块中的 `s_y[0]` 副本复制给 `d_y[1]`;
  - 如此等等。

Listing 8.4: 本章程序 reduce2gpu.cu 中使用静态共享内存的规约核函数。

```
1  void __global__ reduce_shared(real *d_x, real *d_y)
2  {
3      const int tid = threadIdx.x;
4      const int bid = blockIdx.x;
5      const int n = bid * blockDim.x + tid;
6      __shared__ real s_y[128];
7      s_y[tid] = (n < N) ? d_x[n] : 0.0;
8      __syncthreads();
9
10     for (int offset = blockDim.x >> 1; offset > 0; offset >>= 1)
11     {
12
13         if (tid < offset)
14         {
15             s_y[tid] += s_y[tid + offset];
```

```
16          }
17          __syncthreads();
18      }
19
20      if (tid == 0)
21      {
22          d_y[bid] = s_y[0];
23      }
24  }
```

用装有 GeForce RTX 2070 的笔记本测试，使用单精度浮点数时，全部计算（包括核函数执行、将数组 d_y 从设备复制到主机以及在主机中对数组 d_y 规约）所花时间约为 5.8 毫秒，和不用共享内存的版本所用时间相当。用老一些的 Tesla K40（开普勒架构）测试，不使用共享内存时所用时间是 16.3 毫秒，使用共享内存时所用时间时 10.8 毫秒，后者更为高效。这说明使用共享内存减少全局内存的访问一般来说会带来性能提升，但也不是绝对如此。一般来说，在核函数中对共享内存访问的次数越多，则由使用共享内存带来的加速效果越明显。在我们的数组规约问题中，使用共享内存相对于仅使用全局内存还带来两个好处：一个是不再要求全局内存数组的长度 N 是线程块大小的整数倍，另一个是在规约的过程中不会改变全局内存数组中的数据（在仅使用全局内存时，数组 d_x 的中的部分元素被改变）。这两点在实际的应用中往往都是很重要的。

共享内存的另一个作用是改善全局内存的访问方式（将非合并的全局内存访问转化为合并的），这将在下一节通过矩阵转置的例子进行讨论。在此之前，我们进一步讨论共享内存的用法。

8.1.3 使用动态共享内存

在前面的核函数中，我们在定义共享内存数组时指定了一个固定的长度（128）。我们的程序假定了这个长度与核函数的执行配置参数 block_size （也就是核函数中的 blockDim.x）是一样的。如果在定义共享内存变量时不小心把数组长度写错了，就有可能引起错误或者降低核函数性能。

有一种方法可以减少这种错误发生的概率，那就是使用动态的共享内存。将前一个版本的静态共享内存改成动态共享内存，只需要做两处修改：

1. 调用核函数的执行配置中写下第三个参数：

```
<<<grid_size, block_size, sizeof(real) * block_size>>>
```

前两个参数分别是网格大小和线程块大小，第三个参数就是核函数中每个线程块需要定义的动态共享内存的字节数。在我们以前所有的执行配置中，这个参数都没有出现，其实是用了默认值零。

2. 要使用动态共享内存，还需要改变核函数中共享内存变量的声明方式。例如，

```
extern __shared__ real s_y[];
```

它与之前静态共享内存的声明方式

```
__shared__ real s_y[128];
```

有两点不同。第一，必须加上限定词 `extern`。第二，不能指定数组大小。读者也许觉得可以将动态共享内存数组声明为指针：

```
extern __shared__ real *s_y;
```

但这是错的（无法通过编译），因为数组并不等价于指针。

无论使用什么 GPU 进行测试，使用动态共享内存的核函数和使用静态共享内存的核函数在执行时间上都几乎没有差别。所以，使用动态共享内存不会影响程序性能，但有时候可提高程序的可维护性。我们将在第 9 章继续对数组规约的计算进行优化。

8.2 使用共享内存进行矩阵转置

在第 7 章，我们讨论了矩阵转置的计算，重点考察了全局内存的访问模式对核函数性能的影响。如果不利用共享内存的话，在矩阵转置问题中，对全局内存的读和写这两个操作中，总有一个是合并的，另一个是非合并的。在本节，我们将看到，利用共享内存可以改善全局内存的访问模式，使得对全局内存的读和写都是合并的。

Listing 8.5: 本章程序 bank.cu 中使用静态共享内存的规约核函数。

```
1 __global__ void transpose1(const real *A, real *B, const int N)
2 {
3     __shared__ real S[TILE_DIM][TILE_DIM];
4     int bx = blockIdx.x * TILE_DIM;
5     int by = blockIdx.y * TILE_DIM;
6
7     int nx1 = bx + threadIdx.x;
```

```
8     int ny1 = by + threadIdx.y;
9     if (nx1 < N && ny1 < N)
10    {
11        S[threadIdx.y][threadIdx.x] = A[ny1 * N + nx1];
12    }
13    __syncthreads();
14
15    int nx2 = bx + threadIdx.y;
16    int ny2 = by + threadIdx.x;
17    if (nx2 < N && ny2 < N)
18    {
19        B[nx2 * N + ny2] = S[threadIdx.x][threadIdx.y];
20    }
21 }
```

我们首先从上一章的核函数 `transpose1` 出发，写出如 Listing 8.5 所示的利用共享内存的矩阵转置核函数。下面是对该核函数详细的解释：

- 在矩阵转置的核函数中，最中心的思想是用一个线程块处理一片（tile）矩阵。这里，一片矩阵的行数和列数都是 `TILE_DIM = 32`。为了利用共享内存改善全局内存的访问方式，我们在第 3 行定义了一个两维的静态共享内存数组 `S`，其行、列数与一片矩阵的行、列数一致。

- 第 11 行，将一片矩阵数据从全局内存数组 `A` 中读出来，存放在共享内存数组。这里对全局内存的访问是合并的（不考虑内存对齐的因素），因为相邻的 `threadIdx.x` 与全局内存中相邻的数据对应。

- 第 13 行，在将共享内存中的数据写入全局内存数组 `B` 之前，进行一次线程块内的同步操作。一般来说，在利用共享内存中的数据之前，都要进行线程块内的同步操作，以确保共享内存数组中的所有元素都已经更新完毕。

- 接下来的几行极为关键。为了能更好第理解这几行代码，将第 15-20 行改写为：

```
int nx2 = bx + threadIdx.x;
int ny2 = by + threadIdx.y;
if (nx2 < N && ny2 < N)
{
```

```
        B[nx2 * N + ny2] = S[threadIdx.y][threadIdx.x];
    }
```

这样改写后的核函数与上一章的核函数 `transpose1` 相比，唯一的区别就是将数据从全局内存转移到共享内存，然后又原封不等地转移到全局内存，并没有改变对全局内存的访问方式。要改变对全局内存的访问方式很简单：只要调换这几行代码中的 `threadIdx.x` 和 `threadIdx.y` 即可。调换之后，对全局内存数组 `B` 的访问也是合并的了：因为相邻的 `threadIdx.x` 与全局内存数组 `B` 中相邻的数据对应。

所以，在本章的核函数 `transpose1` 中，对全局内存数组 `A` 和 `B` 的访问都是合并的。用 GeForce RTX 2080ti 测试（使用单精度浮点数），核函数的执行时间是 3.5 ms，这比上一章的 `transpose1` 的执行时间（5.3 ms）要短，但比上一章的 `transpose2` 的执行时间（2.8 ms）要长。本章的核函数 `transpose1` 还有优化的空间，见下一节的讨论。

8.3 避免共享内存的 bank 冲突

关于共享内存，有一个内存 bank 的概念值得注意。为了获得高的内存带宽，共享内存在物理上被分为 32 个（刚好等于一个线程束中的线程数目，即内建变量 warpSize 的值）同样宽度的、能被同时访问的内存 bank。我们可以将 32 个 bank 从 0 到 31 编号。在每一个 bank 中，又可以对其中的内存地址从 0 开始编号。为方便起见，我们将所有 bank 中编号为 0 的内存称为第一层内存；将所有 bank 中编号为 1 的内存称为第二层内存。在开普勒架构中，每个 bank 的宽度为 8 字节；在所有其它架构中，每个 bank 的宽度为 4 字节。我们这里不关注开普勒架构。

对于 bank 宽度为 4 字节的架构，共享内存数组是按如下方式线性地映射到内存 bank 的：共享内存数组中连续的 128 字节的内容分摊到 32 个 bank 的某一层中，每个 bank 管 4 字节的内容。例如，对一个长度为 128 的单精度浮点数变量的共享内存数组来说，第 0-31 个数组元素依次对应到 32 个 bank 的第一层；第 32-63 个数组元素依次对应到 32 个 bank 的第二层；第 64-95 个数组元素依次对应到 32 个 bank 的第三层；第 96-127 个数组元素依次对应到 32 个 bank 的第四层。也就是说，每个 bank 分摊四个在地址上相差 128 字节的数据。参看图 8.1。

只要同一线程束内的多个线程不同时访问同一个 bank 中不同层的数据，该线程束对共享内存的访问就只需要一次内存事务（memory transaction）。当同一线程束内的多个线程试图访问同一个 bank 中不同层的数据时，就会发生 bank 冲突。在一个线程束内对同一个 bank 中的 n 层数据同时访问将导致 n 次内存事务，称为发生了 n 路 bank 冲突。最坏的情况是线程束内的 32 个线程同时访问同一个 bank 中 32 个不同层的地址，这将导致 32 路 bank 冲突。这种 n 很大的 bank 冲突是要尽量避免的。

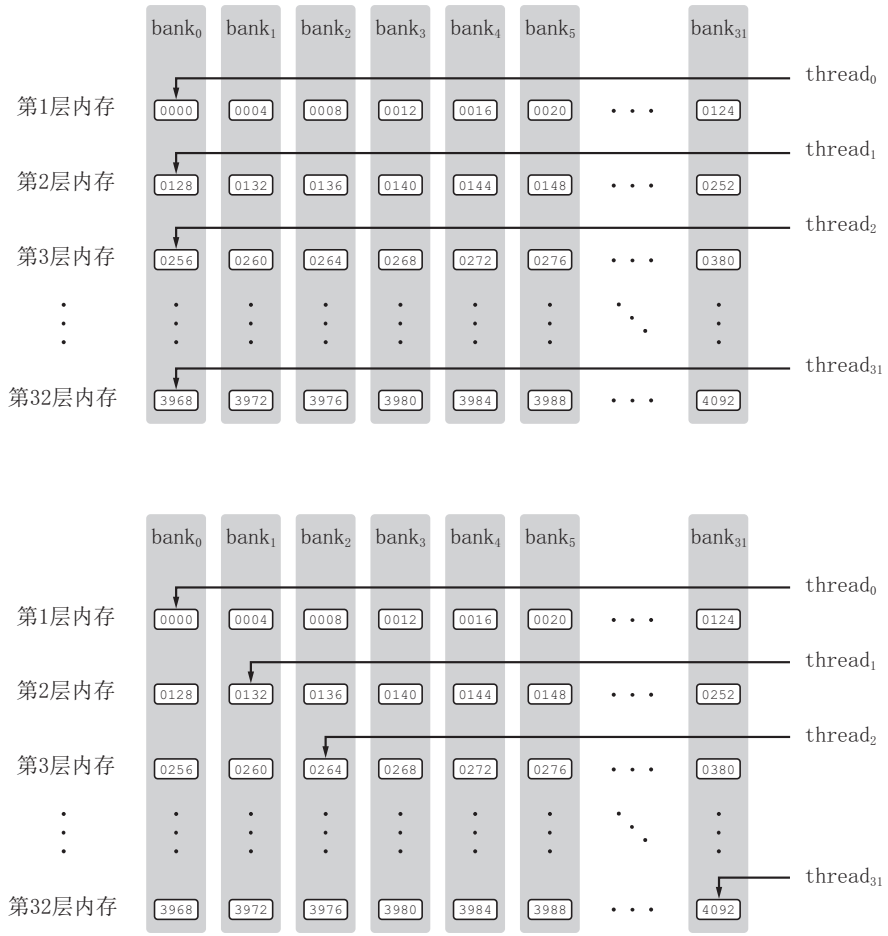

图 8.1: 共享内存 bank 以及有无 bank 冲突的示意图。

在前一节的核函数 `transpose1` 中，定义了一个长度为 $32 \times 32 = 1024$ 的单精度浮点型变量的共享内存数组。我们只讨论非开普勒架构的情形，其中每个共享内存 bank 的宽度为 4 字节。于是，每一层的 32 个 bank 将对应 32 个连续的数组元素；每个 bank 有 32 层数据。从前一节核函数 `transpose1` 的第 19 行可以看出，同一个线程束中的 32 个线程（连续的 32 个 threadIdx.x 值）将对应共享内存数组 `S` 中跨度为 32 的数据。也就是说，这 32 个线程将刚好访问同一个 bank 中的 32 个数据。这将导致 32 路 bank 冲突。参见图 8.1（上）。相比之下，第 11 行对共享内存的访问不导致 bank 冲突。

通常可以用改变共享内存数组大小的方式来消除或减轻共享内存的 bank 冲突。例如，将上述核函数中的共享内存定义修改为如下：

```
__shared__ real S[TILE_DIM][TILE_DIM + 1];
```

就可以完全消除第 19 行读取共享内存时的 bank 冲突。这是因为，这样改变共享内存数组的大小之后，同一个线程束中的 32 个线程（连续的 32 个 threadIdx.x 值）将对应共享内存数组 S 中跨度为 33 的数据。如果第一个线程访问第一个 bank 的第一层，第二个线程则会访问第二个 bank 的第二层（而不是第一个 bank 的第二层）；如此等等。于是，这 32 个线程将分别访问 32 个不同 bank 中的数据，所以没有 bank 冲突。参见图 8.1（下）。

用 GeForce RTX 2080ti 测试（使用单精度浮点数），消除了共享内存 bank 冲突的核函数的执行时间是 2.3 ms，这比第 7 章的 `transpose2` 的执行时间（2.8 ms）也要短了。所以，尽量消除共享内存的 bank 冲突是值得的。

表 8.1: 矩阵复制和转置问题（矩阵大小为 10000 × 10000）在两款 GPU 中的性能。

GPU（精度）	V100 (单)	V100 (双)	2080ti (单)	2080ti (双)
矩阵复制	1.1 ms	2.0 ms	1.6 ms	2.9 ms
合并读取的转置	4.5 ms	6.2 ms	5.3 ms	5.4 ms
合并写入的转置	1.6 ms	2.2 ms	2.8 ms	3.7 ms
有 bank 冲突的转置	1.8 ms	2.6 ms	3.5 ms	4.3 ms
无 bank 冲突的转置	1.4 ms	2.5 ms	2.3 ms	4.2 ms

最后，我们总结一下几个版本的数组转置核函数的性能，见表 8.1。对 Tesla V100 和 GeForce RTX 2080ti 这两款 GPU 来说，在使用单精度浮点数时，使用共享内存并消除了 bank 冲突的核函数最为高效，但在使用双精度浮点数时，仅使用全局内存（不使用共享内存）且保证全局内存的合并写入的（故导致全局内存的非合并读取；但此时会利用只读缓存加速）核函数最为高效。这说明，使用共享内存来改善全局内存的访问方式并不一定能够提高核函数的性能。所以，在优化 CUDA 程序时，一般需要对不同的优化方案进行测试与比较。