第 1 章 Chapter 1

# 基于 CUDA 的异构并行计算

本章内容：

- 了解异构计算架构
- 认识并行程序设计的范例转换
- 掌握 GPU 程序设计的基本要素
- 了解 CPU 和 GPU 编程的区别

随着新科技和处理方法的普及，高性能计算（HPC）领域也在不断变化，而 HPC 的定义也随之产生了相应的变化。一般来说，它涉及多个处理器或计算机的使用，以高吞吐量和高效率来完成一个复杂的任务。HPC 不仅可以认为是一个计算架构，还可以认为是包括硬件系统、软件工具、编程平台及并行编程范例的一组元素列表。

在过去的十几年中，高性能计算取得了极大的发展，尤其是 GPU-CPU 异构架构的出现，直接导致了在并行程序设计中一个基本的范例转变。将从本章开始学习异构并行程序设计。

## 1.1 并行计算

在过去的几十年间，人们对并行计算产生了越来越多的兴趣。并行计算的主要目标是提高运算速度。

从纯粹的计算视角来看，并行计算可以被定义为计算的一种形式，在这种形式下，计算机可以同时进行许多运算，计算原则是一个大的问题往往可以被划分为很多可以同时解决的小问题。

从程序员的角度来说，一个很自然的疑问，就是如何将并发计算映射到计算机上。假设你有许多计算资源，并行计算可以被定义为同时使用许多计算资源（核心或计算机）来执行并发计算，一个大的问题可以被分解成多个小问题，然后在不同的计算资源上并行处理这些小问题。并行计算的软件和硬件层面是紧密联系的。事实上，并行计算通常涉及两个不同的计算技术领域。

- 计算机架构（硬件方面）
- 并行程序设计（软件方面）

计算机架构关注的是在结构级别上支持并行性，而并行编程设计关注的是充分使用计算机架构的计算能力来并发地解决问题。为了在软件中实现并行执行，硬件必须提供一个支持并行执行多进程或多线程的平台。

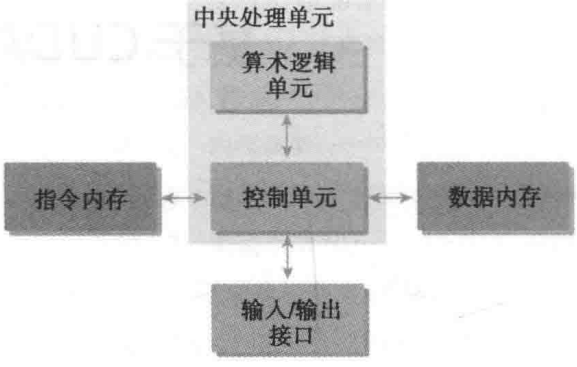

图 1-1

大多数现代处理器都应用了哈佛体系结构（Harvard architecture），如图 1-1 所示，它主要由 3 个部分组成。

- 内存（指令内存和数据内存）
- 中央处理单元（控制单元和算术逻辑单元）
- 输入 / 输出接口

高性能计算的关键部分是中央处理单元（CPU），通常被称为计算机的核心。在早期的计算机中，一个芯片上只有一个 CPU，这种结构被称为单核处理器。现在，芯片设计的趋势是将多个核心集成到一个单一的处理器上，以在体系结构级别支持并行性，这种形式通常被称为多核处理器。因此，并行程序设计可以看作是将一个问题的计算分配给可用的核心以实现并行的过程。

当实现一段串行算法时，你可能不需要为了编写一个程序而特意去理解计算机架构的细节。但是，当在多核计算机上执行算法时，对于程序员来说，了解基本的计算机架构的特点就显得非常重要了。要编写一个既正确又高效的并行程序需要对多核体系结构有一个基本的认识。

以下介绍了并行计算的一些基本概念，以及这些概念与 CUDA 编程设计的联系。

### 1.1.1 串行编程和并行编程

当用计算机程序解决一个问题时，我们会很自然地把这个问题划分成许多的运算块，每一个运算块执行一个指定的任务，如图 1-2 所示。这样的程序叫作串行程序。

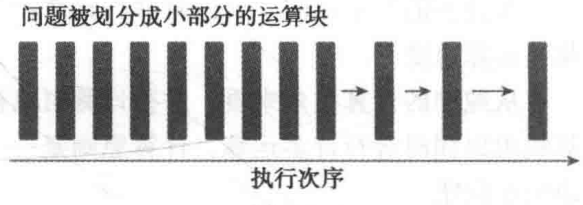

图 1-2

有两种方法可以区分两个计算单元之间

的关系：有些是有执行次序的，所以必须串行执行；其他的没有执行次序的约束，则可以并发执行。所有包含并发执行任务的程序都是并行程序。如图 1-3 所示，一个并行程序中可能会有一些串行部分。

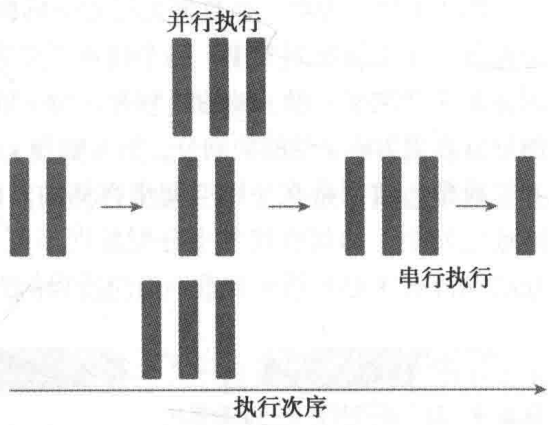

图　1-3

从程序员的角度来看，一个程序应包含两个基本的组成部分：指令和数据。当一个计算问题被划分成许多小的计算单元后，每个计算单元都是一个任务。在一个任务中，单独的指令负责处理输入和调用一个函数并产生输出。当一个指令处理前一个指令产生的数据时，就有了数据相关性的概念。因此，你可以区分任何两个任务之间的依赖关系，如果一个任务处理的是另一个任务的输出，那么它们就是相关的，否则就是独立的。

在并行算法的实现中，分析数据的相关性是最基本的内容，因为相关性是限制并行性的一个主要因素，而且在现代编程环境下，为了提高应用程序的运行速度，理解这些是很有必要的。在大多数情况下，具有依赖关系的任务之间的独立的关系链为并行化提供了很好的机会。

### 1.1.2　并行性

如今，并行性的应用非常广泛，在编程领域，并行编程设计正在成为主流。多层次的并行性设计是架构设计的驱动力。在应用程序中有两种基本的并行类型。

- 任务并行
- 数据并行

当许多任务或函数可以独立地、大规模地并行执行时，这就是任务并行。任务并行的重点在于利用多核系统对任务进行分配。

当可以同时处理许多数据时，这就是数据并行。数据并行的重点在于利用多核系统对数据进行分配。

CUDA 编程非常适合解决数据并行计算的问题。本书的重点便是如何使用 CUDA 编程解决数据并行问题。许多处理大数据集的应用可以使用数据并行模型来提高计算单元的速度。数据并行处理可以将数据映射给并行线程。

数据并行程序设计的第一步是把数据依据线程进行划分，以使每个线程处理一部分数据。通常来说，有两种方法可以对数据进行划分：块划分（block partitioning）和周期划分（cyclic partitioning）。在块划分中，一组连续的数据被分到一个块内。每个数据块以任意次序被安排给一个线程，线程通常在同一时间只处理一个数据块。在周期划分中，更少的数据被分到一个块内。相邻的线程处理相邻的数据块，每个线程可以处理多个数据块。为一

个待处理的线程选择一个新的块，就意味着要跳过和现有线程一样多的数据块。

图 1-4 所示为对一维数据进行划分的两个例子。在块划分中，每个线程仅需处理数据的一部分，而在周期划分中，每个线程要处理数据的多个部分。图 1-5 所示为对二维数据进行划分的 3 个例子：沿 $y$ 轴的块划分，沿 $x$ 轴和 $y$ 轴的块划分，以及沿 $x$ 轴的周期划分。其余的划分方式为沿 $x$ 轴的块划分，沿 $x$ 轴和 $y$ 轴的周期划分，以及沿 $y$ 轴的周期划分留作练习。

通常，数据是在一维空间中存储的。即便是多维逻辑数据，仍然要被映射到一维物理地址空间中。如何在线程中分配数据不仅与数据的物理储存方式密切相关，并且与每个线程的执行次序也有很大关系。组织线程的方式对程序的性能有很大的影响。

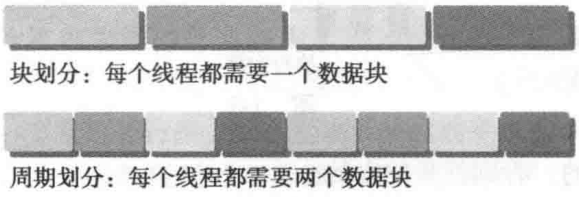


图 1-4

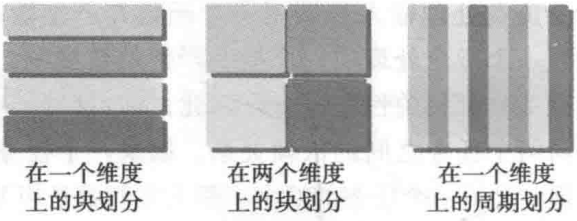


图 1-5

**数据划分**

对数据划分有两种基本的方法：

- 块划分：每个线程作用于一部分数据，通常这些数据具有相同大小。
- 周期划分：每个线程作用于数据的多部分。

程序性能通常对块的大小比较敏感。块划分与周期划分中划分方式的选择与计算机架构有密切关系。具体实例详见本书其他章节。

### 1.1.3 计算机架构

有多种不同的方法可以对计算机架构进行分类。一个广泛使用的分类方法是弗林分类法（Flynn's Taxonomy），它根据指令和数据进入 CPU 的方式，将计算机架构分为 4 种不同的类型（如图 1-6 所示）。

- 单指令单数据（SISD）
- 单指令多数据（SIMD）
- 多指令单数据（MISD）
- 多指令多数据（MIMD）


图 1-6

SISD 指的是传统计算机：一种串行架构。在这种计算机上只有一个核心。在任何时间点上只有一个指令流在处理一个数据流。

SIMD 是一种并行架构类型。在这种计算机上有多个核心。在任何时间点上所有的核心只有一个指令流处理不同的数据流。向量机是一种典型的 SIMD 类型的计算机，现在大多数计算机都采用了 SIMD 架构。SIMD 最大的优势或许就是，在 CPU 上编写代码时，程序员可以继续按串行逻辑思考但对并行数据操作实现并行加速，而其他细节则由编译器来负责。

MISD 类架构比较少见，在这种架构中，每个核心通过使用多个指令流处理同一个数据流。

MIMD 是一种并行架构，在这种架构中，多个核心使用多个指令流来异步处理多个数据流，从而实现空间上的并行性。许多 MIMD 架构还包括 SIMD 执行的子组件。

为了实现以下目的，在架构层次上已经取得了许多进展。

- 降低延迟
- 提高带宽
- 提高吞吐量

延迟是一个操作从开始到完成所需要的时间，常用微秒来表示。带宽是单位时间内可处理的数据量，通常表示为 MB/s 或 GB/s。吞吐量是单位时间内成功处理的运算数量，通常表示为 gflops（即每秒十亿次的浮点运算数量），特别是在重点使用浮点计算的科学计算领域经常用到。延迟用来衡量完成一次操作的时间，而吞吐量用来衡量在给定的单位时间内处理的操作量。

计算机架构也能根据内存组织方式进行进一步划分，一般可以分成下面两种类型。

- 分布式内存的多节点系统
- 共享内存的多处理器系统

在多节点系统中，大型计算引擎是由许多网络连接的处理器构成的。每个处理器有自己的本地内存，而且处理器之间可以通过网络进行通信。图 1-7 所示为一个典型的分布式内存的多节点系统，这种系统常被称作集群。

多处理器架构的大小通常是从双处理器到几十个或几百个处理器之间。这些处理器要么是与同一个物理内存相关联（如图 1-8 所示），要么共用一个低延迟的链路（如 PCI-Express 或 PCIe）。尽管共享内存意味着共享地址空间，但并不意味着它就是一个独立的物理内存。这样的多处理器不仅包括由多个核心组成的单片机系统，即所谓的多核系统，而且还包括由多个芯片组成的计算机系统，其中每一个芯片都可能是多核的。目

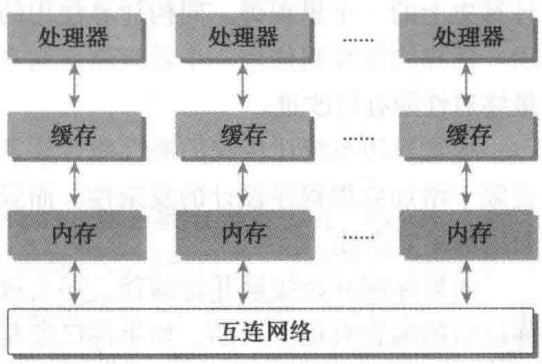


图　1-7

前，多核架构已经永久地取代了单核架构。

“众核”（many-core）通常是指有很多核心（几十或几百个）的多核架构。近年来，计算机架构正在从多核转向众核。

GPU 代表了一种众核架构，几乎包括了前文描述的所有并行结构：多线程、MIMD（多指令多数据）、SIMD（单指令多数据），以及指令级并行。NVIDIA 公司称这种架构为 SIMT（单指令多线程）。

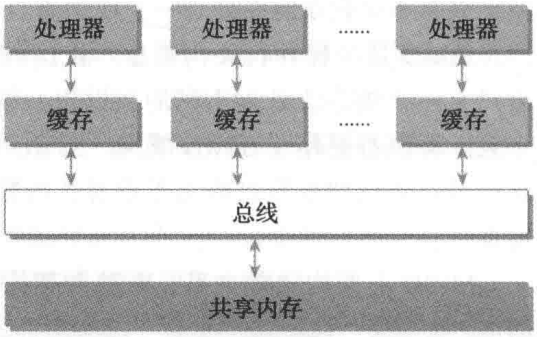


图 1-8

GPU 和 CPU 的来源并不相同。历史上，GPU 是图形加速器。直到最近，GPU 才演化成一个强大的、多用途的、完全可编程的，以及任务和数据并行的处理器，它非常适合解决大规模的并行计算问题。

**GPU 核心和 CPU 核心**

尽管可以使用多核和众核来区分 CPU 和 GPU 的架构，但这两种核心是完全不同的。

CPU 核心比较重，用来处理非常复杂的控制逻辑，以优化串行程序执行。

GPU 核心较轻，用于优化具有简单控制逻辑的数据并行任务，注重并行程序的吞吐量。

## 1.2 异构计算

最初，计算机只包含用来运行编程任务的中央处理器（CPU）。近年来，高性能计算领域中的主流计算机不断添加了其他处理元素，其中最主要的就是 GPU。GPU 最初是被设计用来专门处理并行图形计算问题的，随着时间的推移，GPU 已经成了更强大且更广义的处理器，在执行大规模并行计算中有着优越的性能和很高的效率。

CPU 和 GPU 是两个独立的处理器，它们通过单个计算节点中的 PCI-Express 总线相连。在这种典型的架构中，GPU 指的是离散的设备从同构系统到异构系统的转变是高性能计算史上的一个里程碑。同构计算使用的是同一架构下的一个或多个处理器来执行一个应用。而异构计算则使用一个处理器架构来执行一个应用，为任务选择适合它的架构，使其最终对性能有所改进。

尽管异构系统比传统的高性能计算系统有更大的优势，但目前对这种系统的有效利用受限于增加应用程序设计的复杂性。而且最近得到广泛关注的并行计算也因包含异构资源而增加了复杂性。

如果你刚开始接触并行编程，那么这些性能的改进和异构架构中可用的软件工具将对你以后的编程有很大帮助。如果你已经是一个很好的并行编程程序员了，那么适应并行异

构架构的并行编程是很简单的。

### 1.2.1　异构架构

一个典型的异构计算节点包括两个多核 CPU 插槽和两个或更多个的众核 GPU。GPU 不是一个独立运行的平台而是 CPU 的协处理器。因此，GPU 必须通过 PCIe 总线与基于 CPU 的主机相连来进行操作，如图 1-9 所示。这就是为什么 CPU 所在的位置被称作主机端而 GPU 所在的位置被称作设备端。

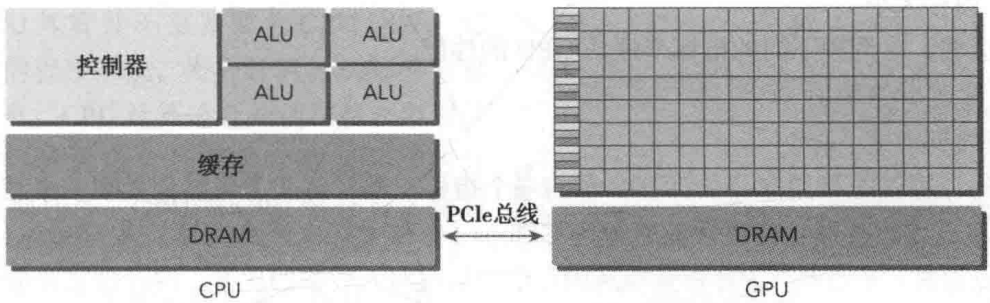


图　1-9

一个异构应用包括两个部分。

- 主机代码
- 设备代码

主机代码在 CPU 上运行，设备代码在 GPU 上运行。异构平台上执行的应用通常由 CPU 初始化。在设备端加载计算密集型任务之前，CPU 代码负责管理设备端的环境、代码和数据。

在计算密集型应用中，往往有很多并行数据的程序段。GPU 就是用来提高这些并行数据的执行速度的。当使用 CPU 上的一个与其物理上分离开的硬件组件来提高应用中的计算密集部分的执行速度时，这个组件就成为了一个硬件加速器。GPU 可以说是最为常见的硬件加速器。

以下产品应用了 NVIDIA 公司的 GPU 计算平台。

- Tegra
- GeForce
- Quadro
- Tesla

Tegra 系列产品是专为移动和嵌入式设备而设计的，如平板电脑和手机，GeForce 面向图形用户，Quadro 用于专业绘图设计，Tesla 用于大规模的并行计算。Fermi 是 Tesla 系列产品中的一种，用作 GPU 加速器，近来在高性能计算中获得了广泛应用。NVIDIA 于 2010 年发布的 Fermi 架构是世界上第一款完整的 GPU 计算架构。Fermi GPU 加速器的出现让

许多领域的高性能计算有了新的发展，如地震资料处理、生化模拟、天气和气候建模、信号处理、计算金融、计算机辅助工程、计算流体力学和数据分析等。Fermi 之后的新一代 GPU 计算架构 Kepler，于 2012 年秋季发布，其处理能力相比以往的 GPU 有很大提升，并且提供了新的方法来优化和提高 GPU 并行工作的执行，有望将高性能计算提升到新的高度。Tegra K1 包含一个 Kepler GPU，并能满足 GPU 在嵌入式应用中的一切要求。

以下是描述 GPU 容量的两个重要特征。

- CUDA 核心数量
- 内存大小

相应的，有两种不同的指标来评估 GPU 的性能。

- 峰值计算性能
- 内存带宽

峰值计算性能是用来评估计算容量的一个指标，通常定义为每秒能处理的单精度或双精度浮点运算的数量。峰值性能通常用 GFlops（每秒十亿次浮点运算）或 TFlops（每秒万亿次浮点运算）来表示。内存带宽是从内存中读取或写入数据的比率。内存带宽通常用 GB/s 表示。表 1-1 所示为 Fermi 架构和 Kepler 架构的一些性能指标。

表 1-1 Fermi 和 Kepler

| | Fermi (TESL A C2050) | Kepler (TESL A K10) | | Fermi (TESL A C2050) | Kepler (TESL A K10) |
|---|---|---|---|---|---|
| CUDA 核心 | 448 | 2×1536 | 峰值性能① | 1.03 TFlops | 4.58 TFlops |
| 内存 | 6 GB | 8 GB | 内存带宽 | 144 GB/s | 320 GB/s |

①单精度浮点性能的峰值。

本书中的大多数示例程序均可在 Fermi 和 Kepler 两种 GPU 上运行。一些示例需要在只包含 Kepler GPU 中特殊的架构上运行。

**计算能力**

NVIDIA 使用一个术语“计算能力”（compute capability）来描述整个 Tesla 系列的 GPU 加速器的硬件版本。表 1-2 给出了 Tesla 产品的各个版本及其计算能力。

具有相同主版本号的设备具有相同的核心架构。

- 主版本 NO.3 是 Kepler 类架构。
- 主版本 NO.2 是 Fermi 类架构。
- 主版本 NO.1 是 Tesla 类架构。

NVIDIA 发布的第一版 GPU 包含了与整个 Tesla GPU 加速器系列相同的名称“Tesla”。本书中的所有示例都需要版本 2 以上的计算能力。

表 1-2　Tesla GPU 计算产品的计算能力

| GPU | 计 算 能 力 | GPU | 计 算 能 力 |
|---|---|---|---|
| Tesla K40 | 3.5 | Tesla C2070 | 2.0 |
| Tesla K20 | 3.5 | Tesla C1060 | 1.3 |
| Tesla K10 | 3.0 | | |

## 1.2.2　异构计算范例

GPU 计算并不是要取代 CPU 计算。对于特定的程序来说，每种计算方法都有它自己的优点。CPU 计算适合处理控制密集型任务，GPU 计算适合处理包含数据并行的计算密集型任务。GPU 与 CPU 结合后，能有效提高大规模计算问题的处理速度与性能。CPU 针对动态工作负载进行了优化，这些动态工作负载是由短序列的计算操作和不可预测的控制流程标记的；而 GPU 在其他领域内的目的是：处理由计算任务主导的且带有简单控制流的工作负载。如图 1-10 所示，可以从两个方面来区分 CPU 和 GPU 应用的范围。

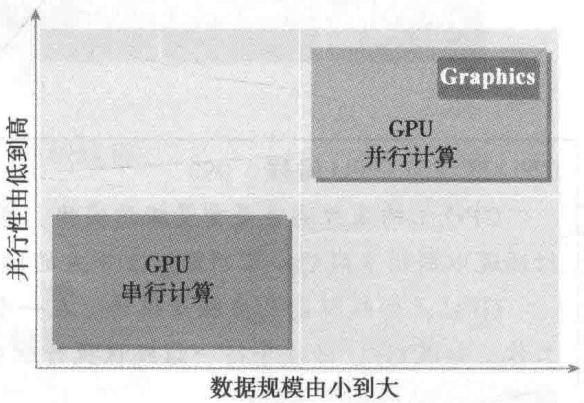


图　1-10

- 并行级
- 数据规模

如果一个问题有较小的数据规模、复杂的控制逻辑和 / 或很少的并行性，那么最好选择 CPU 处理该问题，因为它有处理复杂逻辑和指令级并行性的能力。相反，如果该问题包含较大规模的待处理数据并表现出大量的数据并行性，那么使用 GPU 是最好的选择。因为 GPU 中有大量可编程的核心，可以支持大规模多线程运算，而且相比 CPU 有较大的峰值带宽。

因为 CPU 和 GPU 的功能互补性导致了 CPU＋GPU 的异构并行计算架构的发展，这两种处理器的类型能使应用程序获得最佳的运行效果。因此，为获得最佳性能，你可以同时使用 CPU 和 GPU 来执行你的应用程序，在 CPU 上执行串行部分或任务并行部分，在 GPU 上执行数据密集型并行部分，如图 1-11 所示。

这种代码的编写方式能保证 GPU 与 CPU 相辅相成，从而使 CPU＋GPU 系统的计算能力得以充分利用。为了支持使用 CPU＋GPU 异构系统架构来执行应用程序，NVIDIA 设计了一个被称为 CUDA 的编程模型。这个新的编程模型是本书将要介绍的重点。

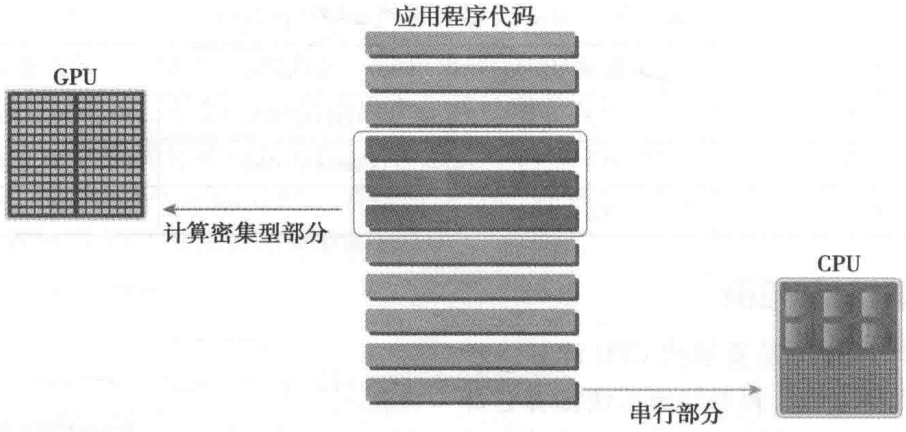


图 1-11

**CPU 线程与 GPU 线程**

CPU 上的线程通常是重量级的实体。操作系统必须交替线程使用启用或关闭 CPU 执行通道以提供多线程处理功能。上下文的切换缓慢且开销大。

GPU 上的线程是高度轻量级的。在一个典型的系统中会有成千上万的线程排队等待工作。如果 GPU 必须等待一组线程执行结束，那么它只要调用另一组线程执行其他任务即可。

CPU 的核被设计用来尽可能减少一个或两个线程运行时间的延迟，而 GPU 的核是用来处理大量并发的、轻量级的线程，以最大限度地提高吞吐量。

现在，四核 CPU 上可以同时运行 16 个线程，如果 CPU 支持超线程可支持多至 32 个线程。

现代的 NVIDIA GPU 在每个多处理器上最多可以并发支持 1 536 个同时活跃的线程。有 16 个多处理器的 GPU，可以并发支持超过 24 000 个同时活跃的线程。

### 1.2.3 CUDA：一种异构计算平台

CUDA 是一种通用的并行计算平台和编程模型，它利用 NVIDIA GPU 中的并行计算引擎能更有效地解决复杂的计算问题。通过使用 CUDA，你可以像在 CPU 上那样，通过 GPU 来进行计算。

CUDA 平台可以通过 CUDA 加速库、编译器指令、应用编程接口以及行业标准程序语言的扩展（包括 C、C++、Fortran、Python，如图 1-12 所示）来使用。本书重点介绍 CUDA C 的编程。

CUDA C 是标准 ANSI C 语言的一个扩展，它带有的少数语言扩展功能使异构编程成为可能，同时也能通过 API 来管理设备、内存和其他任务。CUDA 还是一个可扩展的编程模型，它使程序能对有不同数量核的 GPU 明显地扩展其并行性，同时对熟悉 C 编程语言的程

序员来说也比较容易上手。

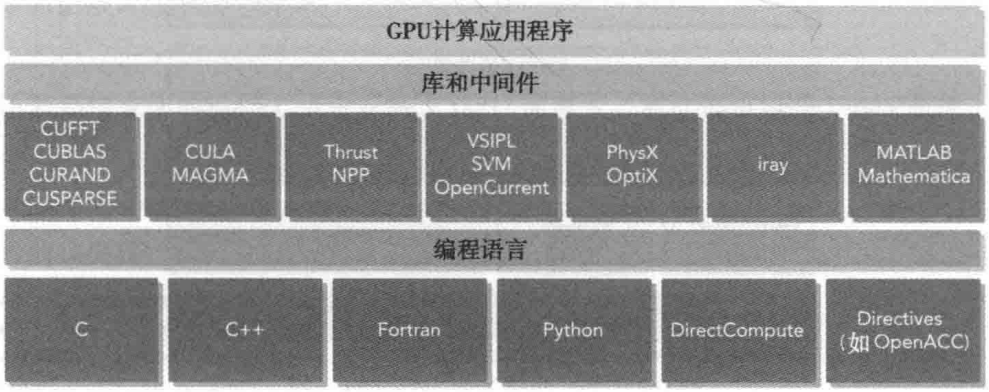


图　1-12

CUDA 提供了两层 API 来管理 GPU 设备和组织线程，如图 1-13 所示。

- CUDA 驱动 API
- CUDA 运行时 API

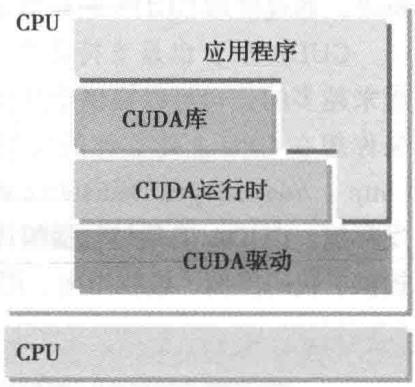


图　1-13

驱动 API 是一种低级 API，它相对来说较难编程，但是它对于在 GPU 设备使用上提供了更多的控制。运行时 API 是一个高级 API，它在驱动 API 的上层实现。每个运行时 API 函数都被分解为更多传给驱动 API 的基本运算。

**运行时 API 与驱动 API**

运行时 API 和驱动 API 之间没有明显的性能差异。在设备端，内核是如何使用内存以及你是如何组织线程的，对性能有更显著的影响。

这两种 API 是相互排斥的，你必须使用两者之一，从两者中混合函数调用是不可能的。本书中所有例子都使用运行时 API。

一个 CUDA 程序包含了以下两个部分的混合。

- 在 CPU 上运行的主机代码
- 在 GPU 上运行的设备代码

NVIDIA 的 CUDA nvcc 编译器在编译过程中将设备代码从主机代码中分离出来。如图 1-14 所示，主机代码是标准的 C 代码，使用 C 编译器进行编译。设备代码，也就是核函数，是用扩展的带有标记数据并行函数关键字的 CUDA C 语言编写的。设备代码通过 nvcc 进行编译。在链接阶段，在内核程序调用和显示 GPU 设备操作中添加 CUDA 运行时库。

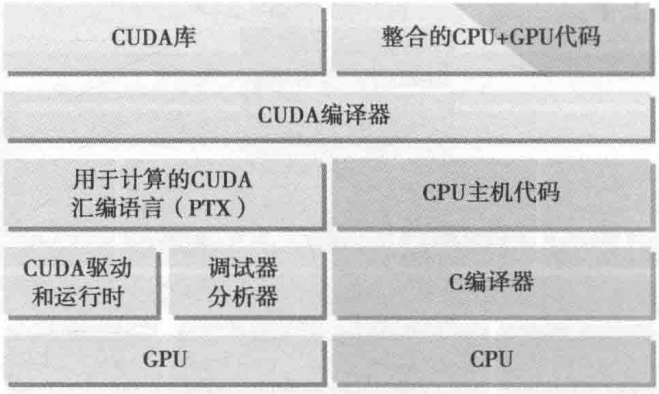


图 1-14

CUDA nvcc 编译器是以广泛使用 LLVM 开源编译系统为基础的。在 GPU 加速器的支持下，通过使用 CUDA 编译器 SDK，你可以创建或扩展编程语言，如图 1-15 所示。

CUDA 平台也是支持多样化并行计算生态系统的基础，如图 1-16 所示。现在，随着越来越多的公司可以提供全球性的工具、服务和解决方案，CUDA 生态系统迅速成长。如果你想在 GPU 上建立你的应用程序，强化 GPU 性能的最简单方法是使用 CUDA 工具包（http：//deve-loper.nvidia.com/cuda-toolkit），它为 C 和 C++ 开发人员提供了一个综合的开发环境。CUDA 工具包包括编译器、数学库，以及调试和优化应用程序性能的工具。同时提供了代码样例、编程指南、用户手册、API 参考文档和其他帮助你入门的文档。

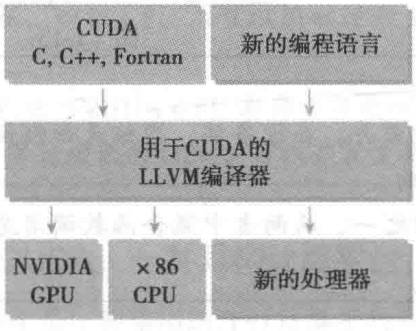


图 1-15


图 1-16

## 1.3 用 GPU 输出 Hello World

学习一个新编程语言的最好方法就是使用这种新语言来编写程序。在本节，你将开始编写在 GPU 上运行的第一个内核代码。像其他任何编程语言一样编写 GPU 上的第一个程序是输出字符串“Hello World”。

如果这是你第一次使用 CUDA，在 Linux 系统中，你可能想使用以下命令来检查 CUDA 编译器是否正确安装：

```
$ which nvcc
```

通常的结果可能是：

```
/usr/local/cuda/bin/nvcc
```

你还需要检查你的机器上是否安装了 GPU 加速卡。对此你可以在 Linux 系统上使用以下命令：

```
$ ls -l /dev/nv*
```

通常的结果是：

```
crw-rw-rw- 1 root root 195,   0 Jul  3 13:44 /dev/nvidia0
crw-rw-rw- 1 root root 195,   1 Jul  3 13:44 /dev/nvidia1
crw-rw-rw- 1 root root 195, 255 Jul  3 13:44 /dev/nvidiactl
crw-rw---- 1 root root  10, 144 Jul  3 13:39 /dev/nvram
```

在这个例子中，你安装了两个 GPU 卡（不同的用户配置可能有所不同，因此显示结果会有所差异）。现在你要准备好写你的第一个 CUDA C 程序。写一个 CUDA C 程序，你需要以下几个步骤：

1. 用专用扩展名 .cu 来创建一个源文件。
2. 使用 CUDA nvcc 编译器来编译程序。
3. 从命令行运行可执行文件，这个文件有可在 GPU 上运行的内核代码。

首先，我们编写一个 C 语言程序来输出“Hello World”，如下所示：

```
#include <stdio.h>
int main(void)
{
   printf("Hello World from CPU!\n");
}
```

把代码保存到 hello.cu 中，然后使用 nvcc 编译器来编译。CUDA nvcc 编译器和 gcc 编译器及其他编译器有相似的语义。

```
$ nvcc hello.cu -o hello
```

如果你运行可执行文件 hello，将会输出：

```
Hello World from CPU!
```

接下来，编写一个内核函数，命名为 helloFromGPU，用它来输出字符串“Hello World from GPU!”。

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

修饰符 __global__ 告诉编译器这个函数将会从 CPU 中调用，然后在 GPU 上执行。用下面的代码启动内核函数。

```
helloFromGPU <<<1,10>>>();
```

三重尖括号意味着从主线程到设备端代码的调用。一个内核函数通过一组线程来执行，所有线程执行相同的代码。三重尖括号里面的参数是执行配置，用来说明使用多少线程来执行内核函数。在这个例子中，有 10 个 GPU 线程被调用。综上所述，得到代码清单 1-1 所示的程序。

**代码清单1-1 Hello World from GPU（hello.cu）**

```
#include <stdio.h>

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

int main(void)
{
   // hello from cpu
   printf("Hello World from CPU!\n");

   helloFromGPU <<<1, 10>>>();
   cudaDeviceReset();
   return 0;
}
```

函数 cudaDeviceRest() 用来显式地释放和清空当前进程中与当前设备有关的所有资源。如下所示，在 nvcc 命令行中使用 -arch sm_20 进行编译：

```
$ nvcc  -arch sm_20 hello.cu -o hello
```

开关语句 -arch sm_20 使编译器为 Fermi 架构生成设备代码。运行这个可执行文件，它将输出 10 条字符串“Hello World from GPU”，每个线程输出 1 条。

```
$ ./hello
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

**CUDA 编程结构**

一个典型的 CUDA 编程结构包括 5 个主要步骤。

1. 分配 GPU 内存。
2. 从 CPU 内存中拷贝数据到 GPU 内存。

3. 调用 CUDA 内核函数来完成程序指定的运算。

4. 将数据从 GPU 拷回 CPU 内存。

5. 释放 GPU 内存空间。

在 hello.cu 中，你只看到了第三步：调用内核。本书其他部分的示例代码是完全按照 CUDA 编程结构来编写的。

## 1.4　使用 CUDA C 编程难吗

CPU 编程和 GPU 编程的主要区别是程序员对 GPU 架构的熟悉程度。用并行思维进行思考并对 GPU 架构有了基本的了解，会使你编写规模达到成百上千个核的并行程序，如同写串行程序一样简单。

如果你想编写一个像并行程序一样高效的代码，那么你需要对 CPU 架构有基本的了解。例如，数据局部性在并行编程中是一个非常重要的概念。数据局部性指的是数据重用，以降低内存访问的延迟。数据局部性有两种基本类型。时间局部性是指在相对较短的时间段内数据和 / 或资源的重用。空间局部性是指在相对较接近的存储空间内数据元素的重用。现代的 CPU 架构使用大容量缓存来优化具有良好空间局部性和时间局部性的应用程序。设计高效利用 CPU 缓存的算法是程序员的工作。程序员必须处理低层的缓存优化，但由于线程在底层架构中的安排是透明的，所以这一点程序员是没有办法优化的。

CUDA 中有内存层次和线程层次的概念，使用如下结构，有助于你对线程执行进行更高层次的控制和调度：

- 内存层次结构
- 线程层次结构

例如，在 CUDA 编程模型中使用的共享内存（一个特殊的内存）。共享内存可以视为一个被软件管理的高速缓存，通过为主内存节省带宽来大幅度提高运行速度。有了共享内存，你可以直接控制代码的数据局部性。

当用 ANSI C 语言编写一个并行程序时，你需要使用 pthreads 或者 OpenMP 来显式地组织线程，这两项技术使得在大多数处理器架构以及操作系统中支持并行编程。当用 CUDA C 编写程序时，实际上你只编写了被单个线程调用的一小段串行代码。GPU 处理这个内核函数，然后通过启动成千上万个线程来实现并行化，所有的线程都执行相同的计算。CUDA 编程模型提供了一个层次化地组织线程的方法，它直接影响到线程在 GPU 上的执行顺序。因为 CUDA C 是 C 语言的扩展，通常可以直接将 C 程序移植到 CUDA C 程序中。概念上，剥离代码中的循环后产生 CUDA C 实现的内核代码。

CUDA 抽象了硬件细节，且不需要将应用程序映射到传统图形 API 上。CUDA 核中有 3 个关键抽象：线程组的层次结构，内存的层次结构以及障碍同步。这 3 个抽象是最小的

一组语言扩展。随着 CUDA 版本的更新，NVIDIA 正在对并行编程进行不断简化。尽管一些人仍然认为 CUDA 的概念比较低级，但如果稍稍提高抽象级，对你控制应用程序和平台之间的互动关系来说会增加很大难度。如果那样的话，不管你掌握了多少底层架构的知识，你的应用程序的性能都将超出控制。

因此，你的目标应是学习 GPU 架构的基础及掌握 CUDA 开发工具和环境。

**CUDA 开发环境**

NVIDIA 为 C 和 C++ 开发人员提供了综合的开发环境以创建 GPU 加速应用程序，包括以下几种。

- NVIDIA Nsight 集成开发环境
- CUDA-GDB 命令行调试器
- 用于性能分析的可视化和命令行分析器
- CUDA-MEMCHECK 内存分析器
- GPU 设备管理工具

当你熟悉这些工具的使用之后，你会发现使用 CUDA C 语言进行编程是非常简单高效的。

## 1.5 总结

随着计算机架构和并行编程模型的发展，逐渐有了现在所用的异构系统。CUDA 平台帮助提高了异构架构的性能和程序员的工作效率。

CPU+GPU 的异构系统在高性能计算领域已经成为主流。这种变化使并行设计范例有了根本性转变：在 GPU 上执行数据并行工作，而在 CPU 上执行串行和任务并行工作。

作为完整的 GPU 计算架构，Fermi 和 Kepler GPU 加速器让许多领域的高性能计算水平有了提高。在阅读和理解本书中这些概念后，你会发现，在异构系统中编写一个具有成百上千个核的 CUDA 程序就像编写一个串行程序那样简单。

## 1.6 习题

1. 参考图 1-5，分析以下几种数据划分形式：
   （1）对于二维数据，沿 $x$ 轴进行块划分
   （2）对于二维数据，沿 $y$ 轴进行周期划分
   （3）对于三维数据，沿 $z$ 轴进行周期划分
2. 从 hello.cu 中移除 cudaDeviceReset 函数，然后编译运行，看看会发生什么。
3. 用 cudaDeviceSynchronize 函数来替换 hello.cu 中的 cudaDeviceReset 函数，然后编译运行，看看会

发生什么。

4. 参考 1.3 节，从编译器命令行中移除设备架构标志，然后按照下面的方式进行编译，看看会发生什么。

```
$ nvcc  hello.cu -o hello
```

5. 参阅 CUDA 在线文档（http://docs.nvidia.com/cuda/index.html）。基于“CUDA 编译器驱动 NVCC”一节，谈谈 nvcc 对带有哪些后缀的文件支持编译？

6. 为执行核函数的每个线程提供了一个唯一的线程 ID，通过内置变量 threadIdx.x 可以在内核中对线程进行访问。在 hello.cu 中修改核函数的线程索引，使输出如下：

```
$ ./hello
Hello World from CPU!
Hello World from GPU thread 5!
```