第1章 介绍

在过去四十年中，微处理器计算能力的持续提升主要得益于晶体管数量的增加和处理器主频的提高。这一发展长期符合摩尔定律和 Dennard 缩放定律，使得在功耗可控的前提下不断提升单核性能成为可能。然而，进入 21 世纪后，随着 Dennard 缩放定律逐渐失效，芯片在更高频率下运行所带来的功耗和散热问题日益突出，单纯依赖主频提升已难以持续推动性能提升。

为突破这一限制，处理器厂商开始通过在单个芯片上集成多个计算核心来提升性能，多核并行逐渐成为主流计算模式。尽管多核处理器显著提高了峰值算力，但大量现有软件仍基于顺序执行模型设计，难以直接利用硬件并行性。这一转变使并行算法设计和并行编程模型成为现代计算中的关键问题。

近年来，单个处理器中的核心数量不断增加。例如，AMD Ryzen Threadripper 系列处理器已支持数十个 CPU 核心，多核 CPU 在运行多个并发线程方面表现出良好的效率。然而，对于具有高度数据并行特征的应用，多核 CPU 仍难以充分发挥其计算潜力。

在此背景下，异构计算逐渐兴起。异构计算通过引入针对特定执行模型优化的处理器，加速特定类型的工作负载。图形处理单元（Graphics Processing Unit，GPU）最初用于图形渲染，但其高度并行的架构使其在数据并行和计算密集型任务中表现出显著优势。早期 GPU 主要依赖专有的图形编程接口，这在一定程度上限制了其作为通用计算加速器的应用范围。

随着可编程着色器和通用并行编程模型的发展，GPU 逐渐演化为可用于通用计算的并行处理器。现代 GPU 通常采用单指令多数据（Single Instruction Multiple Data，SIMD）或其变体的执行模型，通过大量并行执行单元同时处理不同数据元素。CUDA 和 OpenCL 等并行编程框架的出现，使开发者能够使用类 C/C++ 的语言在 GPU 上编写通用计算程序，从而显著降低了并行编程的门槛。

随着 CUDA 的广泛应用，其对特定硬件平台的依赖也引发了对可移植性的关注。OpenCL 通过支持多种硬件平台缓解了这一问题，但其设备端代码组织方式在工程实践中带来了较高的维护成本。基

于指令的并行编程模型（如 OpenMP 和 OpenACC）在一定程度上简化了编程过程，但其抽象层次和执行模型在 GPU 场景下存在性能和可扩展性方面的局限。

为在性能和可移植性之间取得平衡，AMD 于 2016 年推出了便携式异构接口（Heterogeneous-compute Interface for Portability，HIP）。HIP 在编程模型上与 CUDA 高度相似，同时支持在不同厂商的 GPU 平台上运行，并保持接近原生实现的性能。结合 AMD 的 Radeon Open Compute（ROCm）平台，HIP 已广泛应用于深度学习、科学计算和高性能计算等领域。

本书将系统介绍 HIP 编程语言及其相关生态系统与开发工具。由于 HIP 基于 C++ 语言，读者需要具备一定的 C++ 编程基础。书中示例主要以 AMD Instinct MI 系列 GPU 为目标平台，但相关代码具有良好的通用性，可在任何支持 ROCm 或 CUDA 的 GPU 平台上运行。本章旨在为读者建立使用 HIP 和 ROCm 进行并行计算的整体认识，为后续章节的深入学习奠定基础。

1.1 并行编程

许多科学计算和工程应用在其求解过程中天然具有并行性。一方面，不同任务之间往往可以并行执行，这种形式通常称为任务级并行性；另一方面，单个任务内部也可能存在数据级并行性，即对多个数据元素同时执行相同或相似的操作。数据级并行性在处理大规模数据集的应用中尤为常见，例如图像、视频和音频处理，以及神经网络计算等场景。

在过去的几十年中，围绕共享内存和分布式内存体系结构，人们提出并实现了多种并行编程模型和语言。其中，消息传递接口（Message Passing Interface，MPI）是面向分布式内存系统的代表性框架，广泛用于大规模集群计算。MPI 程序通常由多个进程组成，每个进程拥有独立的地址空间，不同节点之间通过显式的消息传递机制进行通信。相比之下，OpenMP 主要面向共享内存多处理器系统，采用基于编译指令的并行化方式，由编译器负责生成多线程并行代码。微软在其 C++ AMP 中采用了另一种思路，通过对 C++ 语言进行扩展，在编译器支持下表达并行执行。

在高性能并行程序设计中，一个常见的策略是重点加速程序中计算开销最大的部分。对于许多应用而言，这些热点通常集中在循环结构，尤其是嵌套循环中。代码清单 1.1 给出了一个典型的嵌套循环矩阵乘法示例，该示例将作为后续讨论并行化和加速技术的基础。

代码清单 1.1：简单嵌套循环的矩阵乘法示例

```
1  // 两个大小为 N×N 的矩阵相乘
2  for (i = 0; i < N; ++i)
3      for (j = 0; j < N; ++j)
4          C[i][j] += A[i][j] * B[j][i];
```

假设采用行优先（row-major）存储方式，将矩阵中每个元素 C[i][j] 的计算映射到

一个独立的并行线程中，如图 1.1 所示。通过这种方式，原本由嵌套循环顺序执行的计算被展开为大量相互独立的并行任务，并可同时在并行计算硬件上执行，从而显著缩短整体执行时间。该示例展示了一种典型的并行化思路：将计算密集型循环中的迭代映射到并行线程，充分利用并行硬件所提供的计算资源，实现对顺序代码的有效加速。

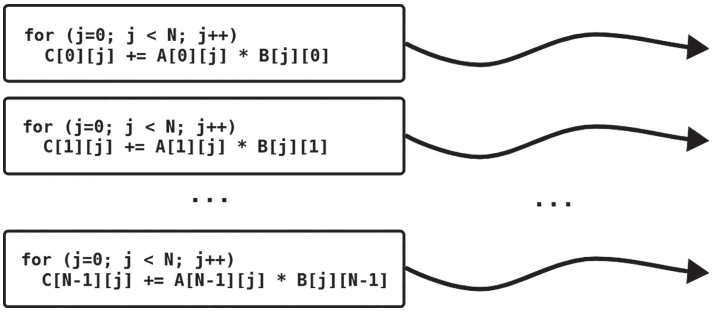

图 1.1　矩阵 C[i][j] 各行计算到各线程的并行执行映射

并行程序的实现通常具有较高的复杂性。若并行编程环境（如具备自动并行化能力的编译器或基于语言扩展的模型）能够自动识别程序中的并行机会，运行时系统便可在不改变程序语义的前提下，采用一组保守的策略对执行过程进行加速。然而，完全依赖自动并行化往往受到分析能力和适用范围的限制。

另一种常见做法是要求程序员显式标注并行区域并定义并行执行方式，例如在 OpenMP 等编程模型中。这种方式为程序员提供了更直接的控制能力，但同时也增加了程序正确性和可维护性方面的负担。

在实际工程中，更为可行的方案通常是在自动化与显式控制之间取得平衡：在关键计算路径上充分利用并行硬件的加速能力，而在其余部分保持程序结构的简洁性。ROCm 和 HIP 正是基于这一设计理念，提供了丰富的并行计算库，用于实现常见的通用并行操作（详见第 9 章）。借助这些高性能库，应用程序可以在不直接编写底层并行代码的情况下，高效利用并行硬件的计算能力。

1.2　GPU

GPU 最初被设计用于三维图形渲染，并且至今仍广泛应用于这一领域。自 2007 年以来，随着通用计算需求的兴起，硬件厂商和开发者对 GPU 的编程接口进行了重新设计，使程序员能够使用熟悉的 C/C++ 语义来开发并行应用程序。NVIDIA 推出的 CUDA 为 GPU 通用计算提供了基于 C/C++ 的编程模型，随后 OpenCL 及其他 GPU 编程接口也采用了类似的设计理念，从而推动了 GPU 在通用并行计算领域的广泛应用。

GPU 与 CPU 在体系结构设计目标上存在显著差异。CPU 的架构主要针对单线程性能和低延迟执行进行优化，通常采用较深的流水线结构、多级缓存体系以及复杂的分支预测和乱序执行机制。相比之下，GPU 的设计重点在于支持大规模线程并发，其

体系结构强调吞吐量优先，通过相对简化的控制逻辑，将更多的晶体管资源用于计算单元和内存带宽。

尽管近年来 CPU 在部分设计中显著增加了核心数量，但其整体架构特征仍与 GPU 存在本质区别。图 1.2 展示了 AMD Instinct MI100 GPU 的体系结构示意。GPU 通常由大量相对简单的顺序处理核心组成，这些核心以高度同步的方式执行指令。相对而言，现代 CPU 仍以多级缓存层次和复杂的控制流逻辑为核心，并通过非一致性内存访问（Non-Uniform Memory Access，NUMA）等机制优化多核环境下的内存访问，其设计目标依然侧重于提升单线程性能和缓存效率。

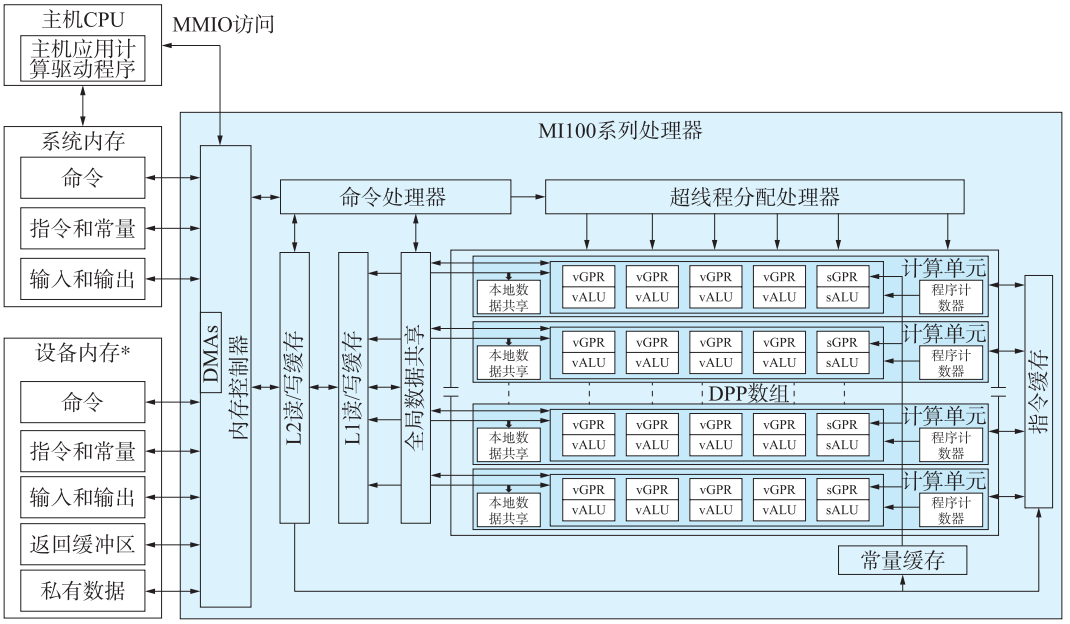

图 1.2　AMD MI100 微架构（由 AMD 提供）

GPU 的体系结构则更加关注内存吞吐量的利用效率。由于系统中可以同时调度和执行大量线程，GPU 通常以 wavefront 为基本执行单位来组织并行计算，从而在面对较高内存访问延迟时，通过快速切换执行上下文来隐藏延迟，提高整体吞吐率。

在多线程执行模型方面，CPU 与 GPU 同样存在明显差异。CPU 由多个核心组成，每个核心通常执行一个或少量线程，并由操作系统或运行时系统负责线程调度；在支持同时多线程的体系结构中，一个 CPU 核心可以同时执行多个硬件线程。相比之下，GPU 采用单指令多线程（SIMT）执行模型，大量线程按照相同的指令序列执行，但作用于不同的数据元素。GPU 中的线程调度主要由硬件完成，使系统能够以极低的开销在不同 wavefront 之间切换，从而实现高吞吐量的并行执行。

一个 wavefront 由固定数量的工作项（work item）组成。例如，在本书所使用的 MI100 GPU 上，一个 wavefront 包含 64 个工作项。GPU 通过线程级并行性来挖掘应用中的数据级并行性。在硬件层面，计算最终由 SIMD 执行单元完成，SIMD 单元以向量

指令的形式并行处理来自同一 wavefront 的多个工作项。

在实际编程中，程序员通常会在 GPU 上启动成千上万个线程。GPU 的硬件调度器能够高效管理如此规模的线程数量，并在执行过程中动态调度它们。在 AMD GPU 上，线程首先被组织为工作组，随后由运行时系统将这些工作组分配到具体的计算单元（Compute Unit，CU）上执行。每个工作组会被划分为一个或多个 wavefront，而 CU 中的硬件调度器负责从驻留的 wavefront 中选择可执行的 wavefront，并将其调度到执行单元上运行。

图 1.3 展示了工作项、wavefront 以及计算单元之间的层次关系。在第 6 章中，我们将对这些概念进行更为深入的讨论，因为它们构成了为 AMD GPU 编写高效并行程序的基础。

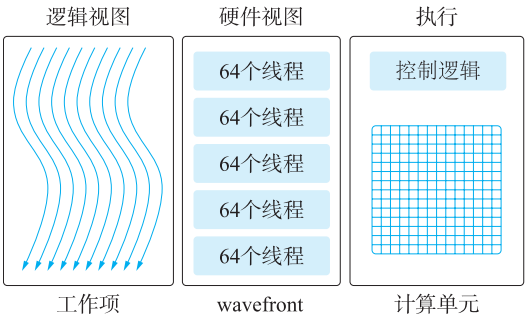

图 1.3　wavefront 的执行

1.3　ROCm

AMD 的 ROCm 是一个开源的软件平台，用于支持基于 GPU 的高性能计算应用。ROCm 的设计继承并吸收了早期异构系统架构（Heterogeneous System Architecture，HSA）相关工作的经验，这些工作致力于为不同体系结构和应用领域提供统一而灵活的编程支持。在此基础上，ROCm 被构建为一个面向 GPU 计算的开放软件栈，能够支持多种主流的开源框架和高性能计算应用。图 1.4 展示了 ROCm 软件栈。

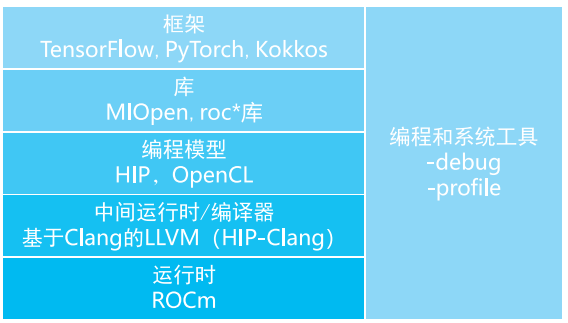

图 1.4　ROCm 软件栈

ROCm 软件栈在设计理念上借鉴了 UNIX 开源社区长期采用的原则，强调可移植性、模块化以及清晰的系统分层。其目标并非将所有功能固化在单一系统中，而是通过一组相对独立、可组合的软件组件，为 GPU 编程提供完整而灵活的支持。ROCm 以 AMD GPU 为主要目标平台，同时通过开放源代码的方式，为研究人员和开发者提供了扩展和演化该平台的可能性。

在实际的软件开发过程中，程序员通常会重用、移植或适配成熟的软件框架和库到新的硬件平台，以向应用开发者提供统一的 API 接口。通过在更高抽象层次上进行编程，可以显著降低应用在不同硬件平台之间迁移和维护的成本。ROCm 正是围绕这一目标构建其生态系统，使开发者能够在尽量保持应用代码结构稳定的前提下，充分利用 GPU 提供的高性能计算能力。

尽管 ROCm 于 2016 年正式发布，但其软件生态系统发展迅速，尤其是在高性能计算

和机器学习领域。目前，ROCm 已支持并集成了丰富的框架、库和开发工具，主要包括：

框架：MIOpen，以及对 TensorFlow、PyTorch 等主流机器学习框架的支持；此外，还支持 Kokkos 等面向性能可移植性的编程模型。

数学与通信库：rocBLAS、rocFFT、rocRAND、rocSPARSE、rocSOLVER、ROCm Collective Communications Library（RCCL）、rocPRIM、rocThrust、rocALUTION 等。

开发与调试工具：rocProfiler、rocTracer 和 rocgdb。

上述组件仅是 ROCm 生态系统中的一部分，实际可用的软件包还包括编译工具链、运行时组件以及面向特定应用领域的扩展库。ROCm 是支持 HIP 程序执行的主要运行时平台。当前，ROCm 主要面向 AMD GPU 提供支持，包括 Instinct MI50、MI100、MI200、MI250 系列以及部分 Radeon GPU。同时，ROCm 软件栈也可与 AMD Ryzen 和 EPYC 处理器协同工作，用于主机端程序执行和系统管理等功能。例如，HIP CPU Runtime 是一个仅包含头文件的实现，允许在 CPU 上执行未修改的 HIP 代码，主要用于功能验证和开发调试场景。随着 HIP 编程模型和 ROCm 软件栈被更多应用和平台采纳，其支持的硬件范围和软件生态预计将持续扩展。

1.4 HIP 框架

AMD 的 HIP 是一个开源并行编程框架，包含基于 C++ 的运行时 API、内核语言以及配套工具，支持通过单一源代码在 AMD 和 NVIDIA GPU 上构建可移植应用程序。对 CUDA 或 OpenCL 熟悉的开发者而言，HIP 提供了高度相似的 API 和库接口，降低了迁移成本。基于 clang 前端的 Hipify 工具可自动将 CUDA 代码转换为 HIP，如 7.6 节所述，大多数 CUDA API 调用可以一对一映射为对应的 HIP API。

在实际开发中，程序员通常受限于特定硬件平台所支持的编程模型。相比之下，跨平台模型能够为开发者提供更大的灵活性和硬件选择。CUDA 作为专有模型，仅支持 NVIDIA GPU，这在一定程度上限制了应用的可移植性。HIP 通过提供可同时在 AMD 和 NVIDIA 平台上编译的通用 C++ 编程接口，缓解了这一问题，使开发者在硬件选择上具有更高的自由度。

HIP 在设计上与 ROCm 运行时（ROCr）紧密协作，并采用与 CUDA 和 OpenCL 类似的主机—设备编程模型。HIP API 分为主机端和设备端两类：主机代码负责内存管理、数据传输、内核启动、同步以及流和事件管理；设备代码（内核）则在 GPU 上执行。关于 ROCr 的细节将在后续章节中介绍。

此外，HIP 还提供了一组封装库（如 hipBLAS、hipFFT、hipRAND、hipSPARSE），为常见计算模式提供可移植的高性能实现。这种供应商中立的 API 设计，使得在 ROCm 环境中编写的 HIP 代码能够较为直接地迁移到 CUDA 生态中复用。在实践中，HIP 程序在 NVIDIA GPU 上通常能够获得与原生 CUDA 实现相当的性能。

1.5 本书内容

本书旨在为读者提供编写高效 GPU 并行程序所需的核心知识与工具。前几章介绍

HIP 编程语言的基础，并结合 GPU 架构要点，帮助读者建立必要的背景理解。随后，内容将逐步深入，讲解如何利用 HIP 的语言特性以及 ROCm 提供的工具和库来开发和优化 GPU 并行程序。依托丰富的 ROCm 库生态，书中通过多个示例展示了如何高效实现常见计算模式，并重点讨论单 GPU 与多 GPU 系统上的性能优化方法。对于熟悉 CUDA 的读者，本书以现有 CUDA 应用为起点，演示如何借助 ROCm 工具将其迁移至 HIP。最后，书中还介绍了基于 ROCm 的机器学习框架，并说明了如何在 ROCm 平台上高效运行相关应用。