数据并行 C++，使用 C++ 和 SYCL 编程加速系统

杨丰

目录

1 序言

如果您是并行编程的新手，那也没关系。如果您从未听说过 SYCL 或 DPC++ 编译器，那也没关系

与 CUDA 中的编程相比，使用 SYCL 的 C++ 提供了超越 NVIDIA 的可移植性和超越 GPU 的可移植性，并且随着现代 C++ 的发展而紧密结合以增强它。带有 SYCL 的 C++ 在不牺牲性能的情况下提供了这些优势。

带有 SYCL 的 C++ 使我们能够利用 CPU、GPU、FPGA 和未来处理设备的组合功能来加速我们的应用程序，而无需依赖任何一家供应商。

SYCL 是行业驱动的 Khronos Group 标准，通过 C++ 添加了对数据并行性的高级支持，以利用加速（异构）系统。SYCL 为 C++ 编译器提供了与 C++ 和 C++ 构建系统高度协同的机制。DPC++ 是一个基于 LLVM 的开源编译器项目，添加了 SYCL 支持。本书中的所有示例都应适用于任何支持 SYCL 2020 的 C++ 编译器，包括 DPC++ 编译器。

如果您是一位不太精通 C++ 的 C 程序员，那么您有一个很好的伙伴。本书的几位作者很高兴地分享说，他们通过阅读像本书这样使用 C++ 的书籍，学到了很多 C++ 知识。只要有一点耐心，想要编写现代 C++ 程序的 C 程序员也应该可以理解这本书。

第二版

得益于不断增长的 SYCL 用户社区的反馈，我们能够添加内容来帮助比以往更好地学习 SYCL。

此版本使用 SYCL 2020 教授 C++。第一版早于 SYCL 2020 规范，与第一版所教授的内容仅略有不同（此版本中 SYCL 2020 最明显的变化是头文件位置、设备选择器语法和删除显式主机设备）。

注 1 *有关更新的 sYCl 信息(包括任何已知书籍勘误表)的重要资源,包括书籍 Github (https://github.com/Apress/data-parallel-CPP)、Khronos Group sYCl 标准网站 (www.khronos.org/sycl) ，以及一个重要的 sycl 教育网站 (https://sycl.tech)*。

第 20 章和第 21 章是受本书第一版读者鼓励而添加的内容。

我们添加了第 20 章来讨论后端互操作性。SYCL 2020 标准的主要目标之一是为具有多种架构的众多供应商的硬件提供广泛支持。这需要扩展到

SYCL 1.2.1 的仅 OpenCL 后端支持之外。虽然一般来说“它确实有效”，但第 20 章为那些认为在这个级别上理解和交互很有价值的人更详细地解释了这一点。

对于经验丰富的 CUDA 程序员，我们添加了第 21 章，以在方法和词汇方面将带有 SYCL 概念的 C++ 与 CUDA 概念明确连接起来。虽然表达异构并行性的核心问题在本质上是相似的，但带有 SYCL 的 C++ 由于其多供应商和多架构方法而提供了许多好处。第 21 章是我们唯一提到 CUDA 术语的地方；本书的其余部分教授如何使用 C++ 和 SYCL 术语及其开放的多供应商、多架构方法。在第 21 章中，我们强烈建议查看开源工具“SYCLomatic”（github.com/oneapi-src/SYCLomatic），它有助于自动迁移 CUDA 代码。因为它很有帮助，所以我们建议将其作为迁移代码的首选第一步。使用带有 SYCL 的 C++ 的开发人员报告称，在从 CUDA 移植的代码和带有 SYCL 的原始 C++ 代码上，在 NVIDIA、AMD 和 Intel GPU 上都取得了出色的结果。使用 SYCL 生成的 C++ 提供了 NVIDIA CUDA 无法实现的可移植性。

C++、SYCL 和编译器（包括 DPC++）的发展仍在继续。在我们一起学习如何使用 C++ 和 SYCL 为异构系统创建程序之后，尾声中讨论了对未来的展望。

我们希望本书能够支持和帮助 SYCL 社区的发展，并帮助促进使用 SYCL 进行 C++ 数据并行编程。

本书的结构

本书带领我们踏上一段旅程，了解如何使用 C++ 和 SYCL 成为一名高效的加速/异构系统程序员。

第 1-4 章：奠定基础

当第一次使用 SYCL 接触 C++ 时，按顺序阅读第 1-4 章非常重要。

第一章通过涵盖新的或值得我们刷新的核心概念奠定了第一个基础。

第 2-4 章为理解使用 SYCL 进行 C++ 数据并行编程奠定了基础。当我们读完第 1-4 章时，我们将为 C++ 数据并行编程打下坚实的基础。第 1 章至第 4 章相互关联，最好按顺序阅读。

第 5-12 章：构建基础

随着基础的建立，第 5 章至第 12 章通过在一定程度上相互借鉴来填补重要的细节，同时可以根据需要轻松地在之间跳转。所有这些章节对于所有使用 SYCL 的 C++ 用户都应该有价值。

第 13-21 章：SYCL 实践提示/建议

最后几章提供了针对特定需求的建议和详细信息。我们鼓励至少浏览所有内容以找到对您的需求重要的内容。

结语：对未来的推测

本书最后的尾声讨论了使用 SYCL 的 C++ 以及 SYCL 的数据并行 C++ 编译器可能和潜在的未来方向。

我们祝您在学习通过 SYCL 使用 C++ 时一切顺利。

2 前言

SYCL 2020 是并行计算领域的一个里程碑。我们第一次拥有了一个现代、稳定、功能齐全、可移植的开放标准，可以针对所有类型的硬件，而您手中的这本书是学习 SYCL 2020 的首要资源。

计算机硬件的发展是由我们解决更大、更复杂问题的需求驱动的，但是，除非像你我这样的程序员拥有允许我们实现我们的想法并通过合理的努力利用可用能力的语言，否则这些硬件进步在很大程度上是无用的。有许多令人惊叹的硬件的例子，并且第一个使用它们的解决方案通常是专有的，因为它可以节省时间，而不必费心与委员会就标准达成一致。然而，在计算的历史上，它们最终总是被供应商锁定——无法与允许开发人员瞄准任何硬件并共享代码的开放标准竞争——因为最终全球社区和生态系统的资源要大得多比任何单个供应商都高，更不用说开放软件标准如何推动硬件竞争了。

在过去的几年里，我的团队非常荣幸地通过开发 GROMACS（世界上使用最广泛的科学 HPC 代码之一）为塑造新兴的 SYCL 生态系统做出了贡献。我们需要我们的代码在世界上每台超级计算机以及我们的笔记本电脑上运行。虽然我们不能承受性能损失，但我们也依赖于成为更大社区的一部分，其他团队在我们依赖的库上投入精力，那里有可用的开放编译器，以及我们可以招募人才的地方。自本书第一版以来，SYCL 已发展成为这样一个社区；除了几个供应商提供的编译器之外，我们现在还有一个针对所有硬件的主要社区驱动的实现 1，并且全球有数千名开发人员分享经验、为培训活动做出贡献并参与论坛。开源的杰出力量——无论是应用程序、编译器还是开放标准——是我们可以深入了解、学习、借用和扩展。正如我们反复从 Intel 主导的 LLVM 实现中的代码、海德堡大学社区驱动的实现以及其他几个代码中学习一样，您可以使用我们的公共存储库 3 来比较大型生产代码库中的 CUDA 和 SYCL 实现，或者借用满足您需求的解决方案 - 因为当您这样做时，您正在帮助进一步扩展我们的社区。

也许令人惊讶的是，数据并行编程作为一种范式可以说比消息传递通信或显式多线程等经典解决方案容易得多，但它对我们这些在专注于硬件和显式的旧范式中花费了数十年时间的人提出了特殊的挑战。数据放置。在小规模上，我们明确决定如何在少数进程之间移动数据是微不足道的，但随着问题扩展到数千个单元，在不引入错误或让硬件闲置的情况下管理复杂性就变成了一场噩梦等待数据。使用 SYCL 进行数据并行编程通过取得平衡来解决这个问题，主要要求我们显式地表达算法的固有并行性，但是一旦

我们做到了这一点，编译器和驱动程序将主要处理数以万计的数据局部性和调度。功能单位。为了在数据并行编程中取得成功，重要的是不要将计算机视为执行一个程序的单个单元，而是将其视为独立工作以解决大问题的各个部分的单元的集合。只要我们可以将我们的问题表达为一种算法，其中每个部分不依赖于其他部分，理论上实现它就很简单，例如，作为通过设备队列在 GPU 上执行的并行 for 循环。然而，对于更实际的示例，我们的问题通常不足以有效地使用整个设备，或者我们依赖于每秒执行数万次迭代，其中设备驱动程序的延迟开始成为主要瓶颈。虽然本书是对高性能便携式 GPU 编程的出色介绍，但它远远超出了这一点，它展示了吞吐量和延迟对于实际应用程序的重要性，以及如何使用 SYCL 来开发 CPU、GPU、SIMD 单元的独特功能和 FPGA，但它也涵盖了一些注意事项，即为了获得良好的性能，我们需要了解并可能使代码适应每种类型硬件的特性。这样做，它不仅是关于数据并行编程的精彩教程，而且是任何对现代计算机硬件编程感兴趣的人都应该阅读的权威文本。

SYCL 的主要优势之一是与现代 C++ 的紧密结合。乍一看，这似乎令人望而生畏。C++ 不是一门容易完全掌握的语言（我当然还没有），但是 Reinders 和合著者牵着我们的手，带领我们走上了一条道路，我们只需要学习一些 C++ 概念就可以开始并在实际数据中发挥生产力 - 并行编程。然而，随着我们经验的积累，SYCL 2020 允许我们将其与 C++17 的极端通用性结合起来，编写可以动态针对不同设备的代码，或者依赖使用 CPU、GPU 和网络单元的异构并行性并行执行不同的任务。SYCL 并不是一个用于启用加速器的单独的固定解决方案，而是有望成为我们在 C++ 中表达数据并行性的通用方式。SYCL 2020 标准现在包含一些以前仅作为供应商扩展提供的功能，例如统一共享内存、子组、原子操作、归约、更简单的访问器以及许多其他概念，这些概念使代码更清晰，并促进开发和开发从标准 C++17 或 CUDA 移植，让您的代码面向更多样化的硬件。本书对所有这些内容进行了精彩且易于理解的介绍，您还将了解到 SYCL 将如何随着 C++ 的快速发展而发展。

这在理论上听起来不错，但 SYCL 在实践中的可移植性如何？我们的应用程序是一个代码库的示例，它的优化非常具有挑战性，因为数据访问模式是随机的，每个步骤中要处理的数据量是有限的，我们需要实现每秒数千次迭代，并且我们都受到内存的限制带宽、浮点和整数运算——它与简单的数据并行问题截然相反。我们花了二十多年的时间为多种 GPU 架构编写汇

编 SIMD 指令和本机实现，我们第一次接触 SYCL 时遇到了适应差异和向驱动程序和编译器开发人员报告性能回归的痛苦。然而，截至 2023 年春季，我们的 SYCL 内核不仅可以通过单个代码库，甚至可以通过单个预编译的二进制文件在所有 GPU 架构上实现 80-100% 的本机性能。

SYCL 还很年轻，并且拥有快速发展的生态系统。虽然还有一些东西尚未成为该语言的一部分，但 SYCL 是独一无二的，它是唯一可成功针对所有现代硬件的性能可移植标准。无论您是想要学习并行编程的初学者、对数据并行编程感兴趣的经验丰富的开发人员，还是需要将 100,000 行专有 API 代码移植到开放标准的维护者，这第二版都是您需要成为的唯一一本书这个社区的一部分。

致谢

我们很幸运地得到了社区对本书第二版的大量意见。许多灵感来自于与开发人员在生产、课程、教程、研讨会、会议和黑客马拉松中使用 SYCL 时的互动。特别是包含 NVIDIA 硬件的 SYCL 部署帮助我们增强了第二版 SYCL 教学的包容性和实用技巧。

SYCL 社区已经发展壮大，由实现编译器和工具的工程师以及更多采用 SYCL 来针对多种类型和供应商的硬件的用户组成。我们感谢他们的辛勤工作和分享的见解。

我们感谢 Khronos SYCL 工作组辛勤工作，制定了功能强大的规范。特别值得一提的是，Ronan Keryell 一直是 SYCL 规范的编辑者，也是 SYCL 的长期倡导者。

我们感谢无数以各种方式从 SYCL 社区向我们提供反馈的人们。我们还深深感谢几年前为第一版提供帮助的人们，我们在第一版致谢中提到了其中许多人的名字。

第一版通过 GitHub 收到了反馈，1 我们确实对其进行了审核，但我们并不总是及时予以确认（想象一下六位合著者都在想“你这样做了，对吗?”）。我们确实从这些反馈中受益匪浅，并且我们相信我们已经解决了本版本示例和文本中的所有反馈。Jay Norwood 是在评论和帮助我们方面最多产的人——所有作者都非常感谢 Jay! 其他反馈贡献者包括 Oscar Barenys、Marcel Breyer、Jeff Donner、Laurence Field、Michael Firth、Piotr Fusik、Vincent Mierlak 和 Jason Mooneyham。无论我们是否记得您的名字，我们都感谢所有提供反馈并帮助我们通过 SYCL 完善 C++ 教学的人。

对于这一版本，一些志愿者不知疲倦地阅读了手稿并提供了富有洞察力的反馈，对此我们深表感谢。这些审稿人包括 Aharon Abramson、Thomas Applencourt、Rod Burns、Joe Curley、Jessica Davies、Henry Gabb、Zheming Jin、Rakshith Krishnappa、Praveen Kundurthy、Tim Lewis、Eric Lindahl、Gregory Lueck、Tony Mongkolsmai、Ruyman Reyes Castro、Andrew Richards、Sanjiv 沙阿、尼尔 · 特雷维特和格奥尔格 · 维赫弗。

我们都享受家人和朋友的支持，我们对他们感激不尽。作为合著者，我们很享受作为一个团队工作，互相挑战并一起学习。我们感谢与整个 Apress 团队的合作，出版了这本书。

我们确信，有很多人对本书项目产生了积极的影响，但我们没有明确提及。我们感谢所有帮助过我们的人。

当您阅读第二版时，如果您发现任何改进方法，请提供反馈。通过 GitHub 提供的反馈可以打开对话，我们将根据需要更新在线勘误表和书籍示例。

谢谢大家，我们希望您发现这本书对您的努力非常有价值。

3 介绍

无可否认，我们已经进入了加速计算的时代。为了满足世界对更多计算的永不满足的需求，与早期解决方案相比，加速计算通过提供更高的性能和更高的能效来驱动复杂的模拟、人工智能等。

被誉为“计算机架构的新黄金时代”1，我们面临着计算设备丰富多样性带来的巨大机遇。我们需要不依赖于任何单一供应商或架构的便携式软件开发能力，以便充分发挥加速计算的潜力。

SYCL（发音为 sickle）是行业驱动的 Khronos Group 标准，通过 C++ 添加了对数据并行性的高级支持，以支持加速（异构）系统。SYCL 为 C++ 编译器提供了利用加速（异构）系统的机制，与现代 C++ 和 C++ 构建系统高度协同。SYCL 不是缩写词；SYCL 只是一个名称。

注 2 (加速 vs 异构) 这些术语是相辅相成的。异构是一种技术描述，承认以不同方式编程的计算设备的组合。加速是将这种复杂性添加到系统和编程中的动机。无法保证加速；只有当我们做得正确时，对异构系统进行编程才能加速我们的应用程序。这本书可以帮助我们教会我们如何正确地做事！

C++ 中的数据并行性与 SYCL 提供对现代加速（异构）系统中所有计算设备的访问。单个 C++ 应用程序可以使用适合当前问题的任意设备组合，包括 GPU、CPU、FPGA 和专用集成电路 (ASIC)。没有任何专有的单一供应商解决方案可以为我们提供同等水平的灵活性。

本书教我们如何使用带有 SYCL 的 C++ 进行数据并行编程来利用加速计算，并提供平衡应用程序性能、跨计算设备的可移植性以及我们作为程序员自己的生产力的实用建议。本章通过涵盖包括术语在内的核心概念奠定了基础，当我们学习如何使用数据并行性加速 C++ 程序时，这些概念对于我们保持新鲜感至关重要。

3.1 阅读书籍，而不是标准说明书

没有人愿意被告知“去阅读规范!”——规范很难阅读,SYCL 规范 (www.khronos.org/sycl/) 也不例外。就像每一个伟大的语言规范一样，它充满了精确性，但对动机、用法和教学却很淡薄。本书是使用 SYCL 教授 C++ 的“学习指南”。

没有一本书可以一次性解释所有事情。因此，本章所做的事情是其他章节所不会做的：代码示例包含一些编程结构，这些编程结构在后面的章节中

才会得到解释。我们不应该沉迷于完全理解第一章中的编码示例，并相信每一章都会变得更好。

3.2 SYCL 2020 和 DPC++

本书使用 SYCL 2020 教授 C++。本书的第一版早于 SYCL 2020 规范，因此该版本包含的更新包括头文件位置的调整（sycl 而不是 CL）、设备选择器语法以及删除显式主机设备。

DPC++ 是一个基于 LLVM 的开源编译器项目。我们希望 LLVM 社区最终能够默认支持 SYCL，并且 DPC++ 项目将帮助实现这一目标。DPC++ 编译器提供广泛的异构支持，包括 GPU、CPU 和 FPGA。本书中的所有示例均适用于 DPC++ 编译器，并且应适用于支持 SYCL 2020 的任何 C++ 编译器。

注 3 *有关更新的 SYCL 信息（包括任何已知书籍勘误表）的重要资源，包括书籍 Github (github.com/Apress/data-parallel-CPP)、Khronos Group SYCL 标准网站 (www.khronos.org/sycl) 以及重要的 SYCL 教育网站（sycl.tech）。*

截至发布时，尚无 C++ 编译器声称完全符合或符合 SYCL 2020 规范。尽管如此，本书中的代码适用于 DPC++ 编译器，并且应该适用于已实现大部分 SYCL 2020 的其他 C++ 编译器。我们仅在 SYCL 2020 中使用标准 C++，除了一些特定于 DPC++ 的扩展，这些扩展在第 17 章（FPGA 编程）、连接到零级后端时的第 20 章（后端互操作性）以及尾声中明确指出。当推测未来时。

3.3 为什么不使用 CUDA?

与 CUDA 不同，SYCL 支持所有供应商和所有类型的架构（不仅仅是 GPU）的 C++ 数据并行性。CUDA 仅专注于 NVIDIA GPU 支持，其他供应商将其重新用于 GPU 的努力（例如 HIP/ROCm）尽管取得了一些实实在在的成功和实用性，但成功的能力有限。随着加速器架构的爆炸式增长，只有 SYCL 能够为我们提供利用这种多样性所需的支持，并提供多供应商/多架构方法来帮助实现 CUDA 所不提供的可移植性。为了更深入地理解这一动机，我们强烈建议阅读（或观看他们精彩演讲的视频录制）行业传奇人物 John L. Hennessy 和 David A. Patterson 所著的《计算机架构的新黄金时代》。我们认为这是一篇必读的文章。

第 21 章除了讨论使用 SYCL 将代码从 CUDA 迁移到 C++ 有用的主题之外，对于那些有 CUDA 经验的人来说也很有价值，可以弥合一些术语和功能差异。CUDA 之外最重要的功能来自 SYCL 支持多个供应商、多个架构（不仅仅是 GPU）以及多个后端（甚至同一设备）的能力。这种灵活性回答了“为什么不使用 CUDA?”的问题。

与 CUDA 或 HIP 相比，SYCL 不涉及任何额外开销。它不是一个兼容层，而是一种通用方法，无论供应商和架构如何，都向所有设备开放，同时与现代 C++ 同步。与其他开放多供应商和多架构技术（例如 OpenMP 和 OpenCL）一样，最终的证明在于实现，包括在绝对需要时访问特定于硬件的优化的选项。

3.4 为什么使用带有 SYCL 的标准 C++?

正如我们将反复指出的，每个使用 SYCL 的程序首先都是 C++ 程序。SYCL 不依赖于对 C++ 的任何语言更改。SYCL 确实将 C++ 编程带到了没有 SYCL 就无法实现的地方。我们毫不怀疑所有用于加速计算的编程将继续影响包括 C++ 在内的语言标准，但我们不认为 C++ 标准应该（或将）很快发展以取代 SYCL 的需求。SYCL 具有一组丰富的功能，我们在本书中将介绍这些功能，这些功能通过类扩展 C++ 以及对新编译器功能的丰富支持，以满足多供应商和多体系结构支持的需求（目前已经存在）。

3.5 获取支持 SYCL 的 C++ 编译器

本书中的所有示例都可以与 DPC++ 编译器的所有不同发行版一起编译和使用，并且应该与支持 SYCL 的其他 C++ 编译器一起编译（请参阅 www.khronos.org/sycl 上的“SYCL 编译器开发”）。我们小心地注意到，在发布时，使用了 DPC++ 特定扩展的极少数地方。

作者推荐 DPC++ 编译器有多种原因，其中包括我们与 DPC++ 编译器的密切联系。DPC++ 是一个支持 SYCL 的开源编译器项目。通过使用 LLVM，DPC++ 编译器项目可以访问多种设备的后端。这已经导致对 Intel、NVIDIA 和 AMD GPU、众多 CPU 和 Intel FPGA 的支持。扩展和增强对多个供应商和多个架构的开放支持的能力使 LLVM 成为支持 SYCL 的开源工作的绝佳选择。

DPC++ 编译器有多个发行版，增加了额外的工具和库，可作为大型项目的一部分提供，为异构系统提供广泛的支持，其中包括库、调试器和其他

工具，称为 oneAPI 项目。oneAPI 工具（包括 DPC++ 编译器）可免费获取（www.oneapi.io/implements）。

3.6 你好世界！和 SYCL 程序剖析

图 1-1 显示了 SYCL 程序示例。编译并运行它会打印以下内容：

你好世界！（以及一些通过运行它来体验的附加文本）

到第 4 章结束时，我们将完全理解这个示例。在此之前，我们可以观察到定义所有 SYCL 结构所需的 <sycl/sycl.hpp>（第 2 行）的单个包含。所有 SYCL 构造都位于名为 sycl 的命名空间内。

- 第 3 行让我们避免一遍又一遍地编写 sycl::。
- 第 12 行实例化一个针对特定设备的工作请求队列（第 2 章）。
- 第 14 行为与设备共享的数据创建分配（第 3 章）。
- 第 15 行将秘密字符串复制到设备内存中，内核将在其中对其进行处理。
- 第 17 行将工作排队到设备（第 4 章）。
- 第 18 行是唯一将在设备上运行的代码行。所有其他代码都在主机（CPU）上运行。

第 18 行是我们要在设备上运行的内核代码。该内核代码减少一个字符。借助 parallel_for() 的强大功能，该内核会在秘密字符串中的每个字符上运行，以便将其解码为结果字符串。不需要对工作进行排序，一旦 parallel_for 将工作排队，它就会相对于主程序异步运行。在查看结果之前等待（第 19 行）以确保内核已完成是至关重要的，因为在本例中我们使用了一个方便的功能（统一共享内存，第 6 章）。如果没有等待，输出可能会在所有字符都被解密之前发生。还有更多内容需要讨论，但这是后面章节的工作。

3.7 队列和操作

第 2 章讨论队列和操作，但现在我们可以从简单的解释开始。队列是允许应用程序直接在设备上完成工作的唯一连接。可以将两种类型的操作放入队列中：(a) 要执行的代码和 (b) 内存操作。要执行的代码通过 single_task

或 parallel_for 表示（如图 1-1 中使用）。内存操作执行主机和设备之间的复制操作或填充操作以初始化内存。仅当我们寻求比自动为我们完成的控制更多的控制时，我们才需要使用内存操作。这些都将在本书后面从第 2 章开始讨论。现在，我们应该意识到队列是允许我们命令设备的连接，并且我们有一组可用于放入队列中以执行代码并执行操作的操作。移动数据。了解请求的操作无需等待即可放入队列中也非常重要。主机将操作提交到队列后，继续执行程序，而设备最终将异步执行通过队列请求的操作。

注 4 (队列将我们与设备连接起来) *我们将操作提交到队列中以请求计算工作和数据移动。*

操作异步发生。

3.8 一切都与并行性有关

由于数据并行性的 C++ 编程都是关于并行性的，所以让我们从这个关键概念开始。并行编程的目标是更快地计算。事实证明，这有两个方面：增加吞吐量和减少延迟。

3.8.1 吞吐量

当我们在规定的时间内完成更多的工作时，程序的吞吐量就会增加。像流水线这样的技术可能会延长完成单个工作项所需的时间，从而允许工作重叠，从而导致单位时间内完成更多的工作。人类在一起工作时经常会遇到这种情况。共享工作本身就涉及协调开销，这通常会减慢完成单个项目的时间。然而，多人的力量会带来更多的吞吐量。计算机也不例外——将工作分散到更多的处理核心会增加每个工作单元的开销，这可能会导致一些延迟，但目标是完成更多的总工作，因为我们有更多的处理核心一起工作。

3.8.2 延迟

如果我们想更快地完成一件事，例如分析语音命令并制定响应，该怎么办？如果我们只关心吞吐量，响应时间可能会变得难以忍受。减少延迟的概念要求我们将一项工作分解为可以并行处理的部分。对于吞吐量，图像处理可能会将整个图像分配给不同的处理单元 - 在这种情况下，我们的目标可能是优化每秒的图像。对于延迟，图像处理可能会将图像中的每个像素分配给

不同的处理核心 - 在这种情况下，我们的目标可能是最大化单个图像每秒的像素数。

3.8.3 并行思维

成功的并行程序员在编程中使用这两种技术。这是我们寻求并行思考的开始。

我们要调整思路，首先考虑在我们的算法和应用程序中可以在哪里找到并行性。我们还考虑了表达并行性的不同方式如何影响我们最终实现的性能。一下子要考虑的东西太多了。对并行思考的追求成为并行程序员的终生旅程。我们可以在这里学习一些技巧。

3.8.4 阿姆达尔和古斯塔夫森

阿姆达尔定律由超级计算机先驱 Gene Amdahl 在 1967 年提出，是一个预测使用多个处理器时理论上最大加速的公式。Amdahl 感叹并行性的最大增益仅限于 (1/(1-p))，其中 p 是并行运行的程序的比例。如果我们只并行运行三分之二的程序，那么该程序最多可以加速 3 倍。我们绝对需要深入理解这个概念！发生这种情况是因为无论我们使三分之二的程序运行得有多快，另外三分之一仍然需要相同的时间才能完成。即使我们添加 100 个 GPU，性能也只能提高 3 倍。

多年来，一些人认为这证明并行计算不会取得成果。1988 年，约翰 · 古斯塔夫森 (John Gustafson) 写了一篇题为“重新评估阿姆达尔定律”的文章。他观察到并行性并不是用来加速固定工作负载，而是用来扩展工作量。人类也会经历同样的事情。在更多人和卡车的帮助下，一名送货员无法更快地交付单个包裹。然而，一百个人和卡车可以比一个司机开一辆卡车运送一百个包裹更快。多个驱动程序肯定会增加吞吐量，并且通常还会减少包裹递送的延迟。阿姆达尔定律告诉我们，单个司机无法通过增加 99 名拥有自己卡车的司机来更快地交付一个包裹。古斯塔夫森注意到，通过这些额外的司机和卡车，可以更快地运送一百个包裹。

这强调了并行性是最有用的，因为我们解决的问题的规模逐年增长。如果我们只是想年复一年地更快地运行相同大小的问题，那么并行性的研究就不那么重要了。这种对解决越来越大问题的追求激发了我们对利用 C++ 和 SYCL 来开发数据并行性的兴趣，以实现计算机的未来（异构/加速系统）。

3.8.5 规模效应

“缩放”这个词出现在我们之前的讨论中。缩放是衡量当额外的计算可用时程序加速的程度（简称为“加速”）。如果一百个包裹与一个包裹同时交付，只需一百辆卡车配备司机而不是单一卡车和司机，就会实现完美的加速。当然，这种方式并不可靠。在某些时候，存在限制加速的瓶颈。配送中心可能没有一百个卡车停靠点。在计算机程序中，瓶颈通常涉及将数据移动到将要处理的位置。分发到一百辆卡车类似于必须将数据分发到一百个处理核心。分配行为不是瞬时的。第 3 章开始了我们探索如何将数据分发到异构系统中需要的地方的旅程。至关重要的是，我们知道数据分发是有成本的，而该成本会影响我们对应用程序的预期扩展程度。

3.8.6 异构系统

就我们的目的而言，异构系统是包含多种类型计算设备的任何系统。例如，同时具有中央处理单元（CPU）和图形处理单元（GPU）的系统是异构系统。CPU 通常简称为处理器，尽管当我们将异构系统中的所有处理单元称为计算处理器时，这可能会令人困惑。为了避免这种混淆，SYCL 将处理单元称为设备。应用程序始终在主机上运行，主机又将工作发送到设备。第 2 章开始讨论我们的主应用程序（主机代码）如何将工作（计算）引导到异构系统中的特定设备。

使用带有 SYCL 的 C++ 的程序在主机上运行并向设备发出工作内核。尽管这可能看起来令人困惑，但重要的是要知道主机通常能够充当设备。这有两个关键原因：(1) 主机通常是一个 CPU，如果不存在加速器，它将运行内核 - SYCL 对于应用程序可移植性的一个关键承诺是内核始终可以在任何系统上运行，即使是那些系统没有加速器 - (2) CPU 通常具有矢量、矩阵、张量和/或 AI 处理功能，这些功能是内核可以很好地映射以在其上运行的加速器。

注 5 主机代码调用设备上的代码。主机的功能通常也可以作为设备使用，以提供备份设备并提供主机具有的用于处理内核的任何加速功能。我们的主机通常是一个 *CPU*，因此它可以作为 *CPU* 设备使用。*SYCL* 不保证 *CPU* 设备，仅保证至少有一个设备可作为我们应用程序的默认设备。

虽然异构从技术角度描述了系统，但使我们的硬件和软件复杂化的原因是为了获得更高的性能。因此，加速计算一词在异构系统或其组件的营销

中很流行。我们想强调的是，不能保证加速。只有当我们做得正确时，异构系统的编程才会加速我们的应用程序。这本书可以帮助我们教会我们如何正确地做事!

GPU 已发展成为高性能计算 (HPC) 设备，因此有时被称为通用 GPU 或 GPGPU。出于异构编程的目的，我们可以简单地假设我们正在编程如此强大的 GPGPU，并将它们称为 GPU。

如今，异构系统中的设备集合可以包括 CPU、GPU、FPGA（现场可编程门阵列）、DSP（数字信号处理器）、ASIC（专用集成电路）和 AI 芯片（图形、神经形态等）.)。

此类设备的设计将涉及计算处理器（多处理器）的重复以及与内存等数据源的增加连接（增加带宽）。第一个是多处理，对于提高吞吐量特别有用。在我们的类比中，这是通过添加额外的司机和卡车来完成的。后者，更高的数据带宽，对于减少延迟特别有用。在我们的类比中，这是通过更多的装货码头来完成的，以使卡车能够并行满载。

拥有多种类型的设备，每种设备具有不同的架构，因此具有不同的特性，导致每种设备的编程和优化需求不同。这成为使用 SYCL 进行 C++ 以及本书所教授的大部分内容的动机。

注 6 *创建 SYCL 是为了解决异构（加速）系统的 C++ 数据并行编程挑战。*

3.8.7 数据并行编程

自从本书的标题出现以来，“数据并行编程”这个词就一直挥之不去，无法解释。数据并行编程侧重于并行性，可以将其想象为并行操作的一堆数据。这种焦点的转变就像古斯塔夫森与阿姆达尔的对比。我们需要运送一百个包裹（实际上是大量数据），以便将工作分配给一百辆配备司机的卡车。关键概念归结为我们应该划分什么。我们应该处理整个图像还是以较小的图块处理它们或逐像素处理它们? 我们应该将对象集合作为单个集合还是一组较小的对象分组或逐个对象进行分析?

选择正确的工作分工并将其有效地映射到计算资源上是任何使用带有 SYCL 的 C++ 的并行程序员的责任。第 4 章开始了这一讨论，并贯穿本书的其余部分。

3.9 带有 SYCL 的 C++ 的关键属性

每个使用 SYCL 的程序首先都是 C++ 程序。SYCL 不依赖于对 C++ 的任何语言更改。

具有 SYCL 支持的 C++ 编译器将根据 SYCL 规范的内置知识来优化代码，并实现支持，以便异构编译在传统 C++ 构建系统中“正常工作”。

接下来，我们将用 SYCL 解释 C++ 的关键属性：单源样式、主机、设备、内核代码和异步任务图。

3.9.1 单源

程序是单源的，这意味着同一个翻译单元 2 既包含定义要在设备上执行的计算内核的代码，也包含协调这些计算内核的执行的主机代码。第 2 章首先更详细地介绍此功能。如果我们愿意，我们仍然可以将程序源分为不同的文件和主机和设备代码的翻译单元，但关键是我们不必这样做！

3.9.2 主机

每个程序都是从在主机上运行开始的，程序中的大部分代码行通常都是针对主机的。到目前为止，主机一直是 CPU。标准没有这样的要求，所以我们小心地将其描述为主机。这似乎不可能是 CPU 以外的任何东西，因为主机需要完全支持 C++17 才能支持所有具有 SYCL 程序的 C++。正如我们稍后将看到的，设备（加速器）不需要支持所有 C++17。

3.9.3 设备

在一个程序中使用多个设备使得异构编程成为可能。这就是为什么自从几页前解释异构系统以来，设备这个词在本章中不断出现。我们已经了解到，异构系统中的设备集合可以包括 GPU、FPGA、DSP、ASIC、CPU 和 AI 芯片，但不限于任何固定列表。

设备是获得加速的目标。卸载计算的想法是将工作转移到可以加速工作完成的设备。我们必须担心如何弥补移动数据所损失的时间——这是一个需要时刻牢记在心的话题。

3.9.4 内核代码

在具有设备（例如 GPU）的系统上，我们可以设想运行两个或多个程序并希望使用单个设备。它们不需要是使用 SYCL 的程序。如果另一个程序当前正在使用该设备，则程序在设备处理过程中可能会出现延迟。这实际上与一般 CPU 的 C++ 程序中使用的原理相同。如果我们的 CPU 上同时运行太多活动程序（邮件、浏览器、病毒扫描、视频编辑、照片编辑等），任何系统都可能超载。

在超级计算机上，当节点（CPU + 所有连接的设备）被专门授予单个应用程序时，共享通常不是问题。在非超级计算机系统上，我们可以注意到，如果有多个应用程序同时使用相同的设备，程序的性能可能会受到影响。

一切仍然有效，并且我们不需要进行不同的编程。

3.9.5 异步执行

设备的代码被指定为内核。这个概念并不是带有 SYCL 的 C++ 独有的：它是其他卸载加速语言（包括 OpenCL 和 CUDA）的核心概念。虽然它与面向循环的方法（例如通常与 OpenMP 目标卸载一起使用）不同，但它可能类似于最内层循环中的代码主体，而不需要程序员显式编写循环嵌套。

内核代码具有某些限制，以允许更广泛的设备支持和大规模并行性。内核代码不支持的功能列表包括动态多态性、动态内存分配（因此不使用 new 或 delete 运算符进行对象管理）、静态变量、函数指针、运行时类型信息 (RTTI) 和异常处理。不允许从内核代码调用任何虚拟成员函数和可变参数函数。内核代码中不允许递归。

注 7 (虚函数) *虽然我们不会在本书中进一步讨论它，但 dpC++ 编译器项目确实有一个实验性扩展（当然，在开源项目中可见）来实现对内核中虚拟函数的一些支持。由于有效卸载到加速器的性质，如果没有一些限制，虚拟功能就无法得到很好的支持，但许多用户表示有兴趣看到 SYCL 即使有一些限制也能提供这种支持。开源和开放 SYCL 规范的美妙之处在于有机会参与可以为 C++ 和 SYCL 规范的未来提供信息的实验。请访问 dpC++ 项目 (github.com/intel/llvm) 了解更多信息。*

第 3 章描述了在调用内核之前和之后如何完成内存分配，从而确保内核始终专注于大规模并行计算。第 5 章描述了与设备相关的异常的处理。

C++ 的其余部分在内核中是公平的游戏，包括函子、lambda 表达式、运算符重载、模板、类和静态多态性。我们还可以与主机共享数据（参见第 3 章）并共享（非全局）主机变量的只读值（通过 lambda 表达式捕获）。

内核：矢量加法 (DAXPY)

对于任何处理过计算复杂代码的程序员来说，内核都应该感到熟悉。考虑实施 DAXPY，它代表“双精度 A 乘以 X 加 Y”。几十年来的经典。图 1-2 显示了用现代 Fortran、C/C++ 和 SYCL 实现的 DAXPY。令人惊讶的是，计算线（第 3 行）实际上是相同的。第 4 章和第 10 章详细解释了内核。图 1-2 应该有助于消除人们对内核难以理解的担忧——即使这些术语对我们来说是新的，它们也应该感到熟悉。

异步执行

使用 C++ 和 SYCL 进行编程的异步特性不容忽视。理解异步编程至关重要，原因有两个：(1) 正确使用可以为我们提供更好的性能（更好的扩展），(2) 错误会导致并行编程错误（通常是竞争条件），从而使我们的应用程序变得不可靠。

异步特性的产生是因为工作是通过请求操作的“队列”传输到设备的。主机程序将请求的操作提交到队列中，程序继续执行而不等待任何结果。这种无需等待很重要，这样我们就可以尝试让计算资源（设备和主机）始终保持忙碌。如果我们必须等待，就会占用主机而不是让主机做有用的工作。当设备完成时，它还会产生串行瓶颈，直到我们对新工作进行排队。正如前面所讨论的，阿姆达尔定律会因为我们花时间而不是并行工作而受到惩罚。我们需要构建我们的程序，以便在设备繁忙时将数据移入和移出设备，并在工作可用时保持设备和主机的所有计算能力繁忙。如果不这样做，我们就会受到阿姆达尔定律的全面诅咒。

第 3 章开始讨论将我们的程序视为异步任务图，第 8 章极大地扩展了这个概念。

3.9.6 当我们犯错误时的竞争条件

在我们的第一个代码示例（图 1-1）中，我们专门在第 19 行执行了“等待”，以防止第 21 行在结果可用之前写出结果中的值。我们必须牢记这种异步行为。在同一代码示例中还做了另一件微妙的事情 - 第 15 行使用 std memcpy 来加载输入。由于 std memcpy 在主机上运行，因此第 17 行及后续行在第 15 行完成之前不会执行。读完第 3 章后，我们可能会想将其

更改为使用 q.memcpy（使用 SYCL）。我们已经在图 1-3 的第 7 行中做到了这一点。由于这是一个队列提交，因此无法保证它将在第 9 行之前执行。这会产生竞争条件，这是一种并行编程错误。当程序的两个部分在没有协调的情况下访问相同的数据时，就会出现竞争条件。由于我们希望使用第 7 行写入数据，然后在第 9 行中读取数据，因此我们不希望在第 7 行完成之前执行第 9 行！这样的竞争条件将使我们的程序变得不可预测——我们的程序可能会在不同的运行和不同的系统上得到不同的结果。解决此问题的方法是在第 7 行末尾添加.wait() 来显式等待 q.memcpy 完成，然后再继续。这不是最佳解决方案。我们可以使用事件依赖来解决这个问题（第 8 章）。将队列创建为有序队列还会在 memcpy 和 parallel_for 之间添加隐式依赖关系。作为替代方案，在第 7 章中，我们将看到如何使用缓冲区和访问器编程风格来让 SYCL 管理依赖性并自动等待我们。

注 8 (竞争条件并不总是导致程序失败) 一位精明的读者注意到，图 *1-3* 中的代码在他们尝试过的每个系统上都没有失败。使用带有 *partition_max_sub_devices==0* 的 *Gpu* 并没有失败，因为它是一个小型 *Gpu*，在 *memcpy* 完成之前无法运行 *parallel_for*。不管怎样，代码是有缺陷的，因为竞争条件存在，即使它不会普遍导致运行时失败。我们称之为一场竞赛——有时我们赢，有时我们输。此类编码缺陷可能会一直处于休眠状态，直到编译和运行时环境的正确组合导致可观察到的故障为止。

添加 wait() 会强制 memcpy 和内核之间的主机同步，这违背了之前保持设备始终忙碌的建议。本书的大部分内容涵盖了不同的选项和权衡，以平衡程序的简单性和系统的有效使用。

注 9 (无序队列 VS 无序队列) 我们将在本书中使用无序队列，因为它们具有潜在的性能优势，但重要的是要知道对有序队列的支持确实存在。*In-order* 只是我们在创建队列时可以请求的一个属性。*Cuda* 程序员会知道 *Cuda* 流是无条件有序的。相反，*SYCL* 队列默认是无序的，但可以选择在创建 *SYCL* 队列时通过传递 *in_order* 队列属性来按顺序排列（请参阅第 *8* 章）。第 *21* 章为使用 *Cuda* 的程序员提供了有关此问题和其他注意事项的信息。

为了帮助检测程序（包括内核）中的数据竞争条件，Intel Inspector（可与前面“获取 DPC++ 编译器”中提到的 oneAPI 工具一起使用）等工具可能会有所帮助。此类工具使用的复杂方法通常不适用于所有设备。检测竞争

条件的最佳方法可能是让所有内核在 CPU 上运行，这可以在开发工作期间作为调试技术来完成。这个调试技巧在第 2 章中作为 Method#2 进行了讨论。

注 10 (为了教授死锁的概念，哲学家就餐问题是计算机科学中同步问题的经典例证) 想象一下一群哲学家围坐在一张圆桌旁，每个哲学家之间放着一根筷子。每个哲学家吃饭时都需要两根筷子，而且他们总是一次拿起一根筷子。遗憾的是，如果所有哲学家都先抓住左边的筷子，然后拿着它等待右边的筷子，那么如果他们同时饿了，我们就会遇到问题。具体来说，他们最终都会等待一根永远不会可用的筷子。

在这种情况下，糟糕的算法设计（向左抓取，然后等到向右抓取）可能会导致死锁，所有哲学家都饿死。那是可悲的。讨论设计一种算法的多种方法，该算法可以让更少的哲学家饿死，或者希望是公平的并养活所有人（没有人挨饿），这是一个值得思考的有趣话题，并且已经被写了很多次。

认识到犯此类编程错误是多么容易，在调试时查找它们，并了解如何避免它们，这些都是成为有效的并行程序员的过程中必不可少的经验。

3.9.7 死锁

死锁是不好的，我们将强调理解并发与并行（参见本章最后一节）对于理解如何避免死锁至关重要。

当两个或多个操作（进程、线程、内核等）被阻塞，每个操作都等待另一个操作释放资源或完成任务，从而导致停滞时，就会发生死锁。换句话说，我们的应用程序永远不会完成。每次我们使用等待、同步或锁时，都可能会造成死锁。缺乏同步可能会导致死锁，但更常见的是它表现为竞争条件（请参阅上一节）。

死锁可能很难调试。我们将在本章末尾的“并发与并行”部分重新讨论这一点。

注 11 *第 4 章将告诉我们“lambda 表达式不被认为是有害的”。我们应该熟悉 lambda 表达式，以便很好地使用 dpC++、SYCL 和现代 C++。*

3.9.8 C++ Lambda 表达式

现代 C++ 的一个被并行编程技术大量使用的功能是 lambda 表达式。内核（在设备上运行的代码）可以用多种方式表达，最常见的一种是 lambda

表达式。第 10 章讨论了内核可以采用的所有各种形式，包括 lambda 表达式。在这里，我们回顾了 C++ lambda 表达式以及有关用于定义内核的一些注释。在我们在中间的章节中了解了有关 SYCL 的更多信息之后，第 10 章将扩展内核方面的内容。

图 1-3 中的代码有一个 lambda 表达式。我们可以看到它，因为它以非常明确的 [=] 开头。在 C++ 中，lambda 以方括号开头，右方括号之前的信息表示如何捕获 lambda 中使用但未作为参数显式传递给它的变量。对于 SYCL 中的内核，捕获必须按值进行，该值通过在括号内包含等号来表示。

C++11 中引入了对 lambda 表达式的支持。它们用于创建匿名函数对象（尽管我们可以将它们分配给命名变量），这些对象可以从封闭范围捕获变量。C++ lambda 表达式的基本语法是

[capture-list] (params) -> ret body

其中

- capture-list 是一个以逗号分隔的捕获列表。我们通过在捕获列表中列出变量名称来按值捕获变量。我们通过在变量前面加上 & 符号来通过引用捕获变量，例如 &v。还有一些适用于所有作用域内自动变量的简写：[=] 用于捕获在正文中按值使用的所有自动变量和按引用捕获当前对象，[&] 用于捕获在正文中使用的所有自动变量 body 以及当前对象的引用，并且 [] 不捕获任何内容。对于 SYCL，始终使用 [=]，因为不允许通过引用捕获变量以在内核中使用。根据 C++ 标准，全局变量不会在 lambda 中捕获。非全局静态变量可以在内核中使用，但前提是它们是 const。这里提到的一些限制允许内核在不同的设备架构和实现中保持一致的行为。

- params 是函数参数的列表，就像命名函数一样。SYCL 提供参数来标识正在调用内核来处理的元素：这可以是唯一的 id（一维）或 2D 或 3D id。这些将在第 4 章中讨论。

- ret 是返回类型。如果未指定 ->ret，则从 return 语句推断。缺少 return 语句或返回没有值，意味着返回类型为 void。SYCL 内核必须始终具有 void 的返回类型，因此我们不应该使用此语法来指定内核的返回类型。

- body 是函数体。对于 SYCL 内核，该内核的内容有一些限制（请参阅本章前面的“内核代码”部分）。

我们可以将 lambda 表达式视为函数对象的实例，但编译器为我们创建了类定义。例如，我们在前面的示例中使用的 lambda 表达式类似于图 1-6 中所示的类实例。无论我们在哪里使用 C++ lambda 表达式，都可以将其替换为函数对象的实例，如图 1-6 所示。

每当我们定义一个函数对象时，我们都需要给它指定一个名称（图 1-6 中的 Functor）。内联表达的 Lambda 表达式（如图 1-4 所示）是匿名的，因为它们不需要名称。

3.9.9 功能可移植性和性能可移植性

可移植性是将 C++ 与 SYCL 结合使用的一个关键目标；然而，没有什么可以保证这一点。语言和编译器所能做的就是让我们在需要时更容易在应用程序中实现可移植性。确实，更高级别（更抽象）的编程（例如特定于领域的语言、库和框架）可以提供更多的可移植性，很大程度上是因为它们允许较少的规范性编程。由于我们在本书中重点关注 C++ 中的数据并行编程，因此我们假设希望拥有更多的控制权，并因此承担更多的责任来理解我们的编码如何影响可移植性。

可移植性是一个复杂的主题，包括功能可移植性和性能可移植性的概念。凭借功能的可移植性，我们希望我们的程序能够在各种平台上同等地编译和运行。凭借性能可移植性，我们希望我们的程序能够在各种平台上获得合理的性能。虽然这是一个相当软的定义，但反过来可能会更清楚——我们不想编写一个在一个平台上运行超快的程序，却发现它在另一个平台上运行得慢得不合理。事实上，我们希望它能够充分利用其运行的任何平台。鉴于异构系统中的设备种类繁多，性能可移植性需要我们作为程序员付出巨大的努力。

幸运的是，SYCL 定义了一种可以提高性能可移植性的编码方法。首先，通用内核可以在任何地方运行。在有限的情况下，这可能就足够了。更常见的是，可能会为不同类型的设备编写重要内核的多个版本。具体来说，内核可能具有通用 GPU 和通用 CPU 版本。有时，我们可能希望将内核专门用于特定设备，例如特定 GPU。当这种情况发生时，我们可以编写多个版本，并将每个版本专门用于不同的 GPU 模型。或者我们可以参数化一个版本以使用 GPU 的属性来修改 GPU 内核的运行方式以适应现有的 GPU。

虽然我们作为程序员自己负责设计有效的性能可移植性计划，但 SYCL 定义了允许我们实施计划的构造。如前所述，可以通过从适用于所有设备的

内核开始，然后根据需要逐步引入其他更专业的内核版本来对功能进行分层。这听起来不错，但程序的整体流程也会产生深远的影响，因为数据移动和整体算法选择很重要。了解这一点可以让我们深入了解为什么没有人应该声称带有 SYCL（或其他编程解决方案）的 C++ 解决了性能可移植性。然而，它是我们工具包中的一个工具，可以帮助我们应对这些挑战。

3.10 并发与并行

并发和并行这两个术语不一定是等价的，尽管它们有时会被误解。由于不同来源很少就相同的定义达成一致，因此对这些术语的任何讨论都变得更加复杂。

请考虑《Sun Microsystems 多线程编程指南》中的这些定义：

- 并发：当至少有两个线程正在进行时存在的条件
- 并行性：两个线程同时执行时存在的条件

为了充分理解这些概念之间的差异，我们需要对这里重要的内容有一个直观的理解。以下观察可以帮助我们获得这种理解：

- 可以伪造同时执行：即使没有硬件支持一次执行多件事情，软件也可以通过多路复用来伪造同时执行多件事情。多路复用是没有并行性的并发的一个很好的例子。
- 硬件资源是有限的：硬件永远不会无限“宽”，因为硬件始终具有有限数量的执行资源（例如处理器、内核、执行单元）。当硬件可以使用专用资源执行每个线程时，我们就拥有并发性和并行性。

当我们作为程序员说“同时执行 X、Y 和 Z”时，我们通常并不真正关心硬件是否提供并发性或并行性。我们可能不希望我们的程序（包含三个任务）无法在只能同时运行其中两个任务的机器上启动。我们希望并行处理尽可能多的任务，重复地逐步执行批量任务，直到它们全部完成。

但有时，我们确实关心。我们思维中的错误可能会产生灾难性的影响（例如“僵局”）。想象一下，我们对上一段的示例进行了修改，使得任务（X、Y 或 Z）执行的最后一件事是“等待所有任务完成”。如果任务数量永远不会超过硬件的限制，我们的程序就会运行得很好。但是，如果我们将任务分成

批次，那么第一批中的任务将永远等待。不幸的是，这意味着我们的应用程序永远不会完成。

这是一个很容易犯的常见错误，这就是我们强调这些概念的原因。即使是专家程序员也必须集中精力避免这种情况，而且我们都发现，当我们在思考中遗漏某些内容时，我们将需要调试问题。这些概念并不简单，C++ 规范包含一个很长的部分，详细说明了保证线程取得进展的精确条件。在这个介绍性部分中，我们所能做的就是强调尽可能多地理解这些概念的重要性。

直观地掌握这些概念对于异构和加速系统的有效编程非常重要。我们都需要给自己时间来获得这种直觉——它不会一下子发生。

3.11 总结

本章提供了通过 SYCL 理解 C++ 所需的术语，并复习了对 SYCL 至关重要的并行编程和 C++ 的关键方面。第 2、3 和 4 章详细介绍了使用 C++ 和 SYCL 进行数据并行编程的三个关键：需要为设备提供工作（发送代码以在其上运行）、提供数据（发送数据以在其上使用）），并且有编写代码的方法（内核）。

4 代码执行位置

并行编程并不是真正意义上的快车道行驶。它实际上是在所有车道上快速行驶。本章的主题是让我们能够将代码放在尽可能多的地方。只要有意义，我们就会选择启用异构系统中的所有计算资源。因此，我们需要知道这些计算资源隐藏在哪里（找到它们）并使它们发挥作用（在它们上执行我们的代码）。

我们可以控制代码的执行位置，换句话说，我们可以控制哪些设备用于哪些内核。带有 SYCL 的 C++ 提供了异构编程框架，其中代码可以在主机 CPU 和设备的混合上执行。确定代码执行位置的机制对于我们理解和使用非常重要。

本章描述代码可以在哪里执行、何时执行以及用于控制执行位置的机制。第 3 章将描述如何管理数据，以便数据到达我们执行代码的地方，然后第 4 章返回代码本身并讨论内核的编写。

4.1 单源

带有 SYCL 程序的 C++ 是单源的，这意味着相同的翻译单元（通常是源文件及其标头）既包含定义要在 SYCL 设备上执行的计算内核的代码，也包含协调这些内核执行的主机代码。图 2-1 以图形方式显示了这两个代码路径，图 2-2 提供了一个示例应用程序，其中标记了主机和设备代码区域。

将设备和主机代码组合到单个源文件（或翻译单元）中可以使异构应用程序更容易理解和维护。该组合还提供了改进的语言类型安全性，并且可以导致我们的代码的更多编译器优化。

4.1.1 主机代码

应用程序包含 C++ 主机代码，由操作系统在其上启动应用程序的 CPU 执行。主机代码是应用程序的主干，它定义和控制可用设备的工作分配。它也是我们定义应由 SYCL 运行时管理的数据和依赖项的接口。

主机代码是标准 C++，并添加了可作为 C++ 库实现的 SYCL 特定构造和类。这使得更容易推断主机代码中允许的内容（C++ 中允许的任何内容），并且可以简化与构建系统的集成。

应用程序中的主机代码协调数据移动和计算卸载到设备，但也可以自行执行计算密集型工作，并且可以像任何 C++ 应用程序一样使用库。

4.1.2 设备代码

设备对应于概念上独立于执行主机代码的 CPU 的加速器或处理器。实现也可以将主机处理器公开为设备，如本章后面所述，但主机处理器和设备应该被认为在逻辑上彼此独立。主机处理器运行本机 C++ 代码，而设备运行包含一些附加功能和限制的设备代码。

队列是一种将工作提交到设备以供将来执行的机制。需要了解设备代码的三个重要属性：

1. 它从主机代码异步执行。主机程序向设备提交设备代码，只有当所有执行依赖性都得到满足时，运行时才会跟踪并启动该工作（更多内容将在第 3 章中介绍）。主机程序执行在设备上启动提交的工作之前进行，从而提供了设备上的执行与主机程序执行异步的属性，除非我们明确地将两者绑定在一起。作为这种异步执行的副作用，只有主机程序通过我们在后面的章节中介绍的各种机制（例如主机访问器和阻塞队列等待操作）强制执行开始，才能保证设备上的工作开始。

2. 对设备代码进行限制，使其能够在加速器设备上编译并实现性能。例如，设备代码中不支持动态内存分配和运行时类型信息 (RTTI)，因为它们会导致许多加速器的性能下降。第 10 章详细介绍了一小部分设备代码限制。

3. SYCL 定义的一些函数和查询仅在设备代码中可用，因为它们只在那里有意义，例如，工作项标识符查询允许设备代码的执行实例查询其在更大的数据并行范围中的位置（描述第 4 章）。

一般来说，我们将提交到队列的工作称为操作。动作包括在设备上执行设备代码，但在第 3 章中我们将了解到动作还包括内存移动命令。在本章中，由于我们关注操作的设备代码方面，因此我们将在大部分时间中具体提及设备代码。

4.2 选择设备

为了探索让我们控制设备代码执行位置的机制，我们将看五个用例：

方法 #1：当我们不关心使用哪个设备时，在某个地方运行设备代码。这通常是开发的第一步，因为它是最简单的。

方法 #2：在 CPU 设备上显式运行设备代码，通常用于调试，因为大多数开发系统都有可访问的 CPU。CPU 调试器通常也具有非常丰富的功能。

方法 #3：将设备代码分派到 GPU 或其他加速器。

方法 #4：将设备代码分派到一组异构设备，例如 GPU 和 FPGA。

方法 #5：从更通用的设备类别中选择特定设备，例如从可用 FPGA 类型集合中选择特定类型的 FPGA。

注 12 *开发人员通常会使用 Method#2 尽可能多地调试代码，并且只有在使用 Method#2 对代码进行了尽可能多的测试后才转向方法 #3–#5。*

4.3 方法 #1：在任何类型的设备上运行

当我们不关心设备代码将在哪里运行时，很容易让运行时为我们选择。这种自动选择的目的是让我们在不关心选择什么设备时可以轻松地开始编写和运行代码。此设备选择没有考虑要运行的代码，因此应被视为任意选择，可能不是最佳选择。

在讨论设备的选择之前，即使是实现为我们选择的设备，我们应该首先介绍程序与设备交互的机制：队列。

4.3.1 队列

队列是一个抽象概念，操作被提交到该抽象概念以便在单个设备上执行。图 2-3 和 2-4 给出了队列类的简化定义。操作通常是数据并行计算的启动，尽管也可以使用其他命令，例如当我们需要比 SYCL 运行时提供的自动移动更多的控制时，手动控制数据移动。提交到队列的工作可以在满足运行时跟踪的先决条件（例如输入数据的可用性）后执行。第 3 章和第 8 章介绍了这些先决条件。

队列绑定到单个设备，并且该绑定发生在队列的构造上。重要的是要了解提交到队列的工作是在该队列绑定到的单个设备上执行的。队列无法映射到设备集合，因为这会导致哪个设备应执行工作不明确。同样，队列无法将提交给它的工作分散到多个设备上。相反，队列与执行提交到该队列的工作的设备之间存在明确的映射，如图 2-5 所示。

可以按照我们希望的应用程序架构或编程风格的任何方式在程序中创建多个队列。例如，可以创建多个队列以分别与不同的设备绑定或由主机程序中的不同线程使用。多个不同的队列可以绑定到单个设备（例如 GPU），

并且向这些不同队列的提交将导致在设备上执行组合工作。图 2-6 显示了一个示例。相反，正如我们之前提到的，一个队列不能绑定到多个设备，因为请求执行操作的位置不能有任何歧义。例如，如果我们想要一个能够跨多个设备负载平衡工作的队列，那么我们可以在代码中创建该抽象。

由于队列绑定到特定设备，因此队列构造是代码中选择将执行提交到队列的操作的设备的最常见方法。构造队列时设备的选择是通过设备选择器抽象来实现的。

4.3.2 当任何设备都可以时将队列绑定到设备

图 2-7 是未指定队列应绑定到的设备的示例。不带任何参数的默认队列构造函数（如图 2-7 所示）只是在幕后选择一些可用的设备。SYCL 保证至少有一个设备始终可用，因此这种默认选择机制将始终选择某个设备。在许多情况下，所选设备可能恰好是也正在执行主机程序的 CPU，尽管不能保证这一点。

使用简单的队列构造函数是开始应用程序开发以及启动和运行设备代码的简单方法。当它与我们的应用程序相关时，可以添加对绑定到队列的设备的选择的更多控制。

4.4 方法 #2：使用 CPU 设备进行开发、调试和部署

CPU 设备可以被认为使主机 CPU 能够像独立设备一样运行，从而允许我们的设备代码执行，而不管系统中是否有可用的加速器。我们总是有一些处理器运行主机程序，因此 CPU 设备通常可供我们的应用程序使用（极少数情况下，由于各种原因，CPU 可能不会通过实现公开为 SYCL 设备）。使用 CPU 设备进行代码开发有几个优点：

1. 在没有任何加速器的功能较差的系统上开发设备代码：一种常见用途是在本地系统上开发和测试设备代码，然后部署到 HPC 集群进行性能测试和优化。

2. 使用非加速器工具调试设备代码：加速器通常通过较低级别的 API 公开，这些 API 可能没有主机 CPU 可用的先进调试工具。考虑到这一点，CPU 设备通常支持使用开发人员熟悉的标准工具进行调试。

3. 如果没有其他设备可用，则进行备份，以保证设备代码可以正常执行：CPU 设备可能不以性能为主要目标，或者可能与内核代码优化的架构

不匹配，但通常可以考虑作为功能备份，以确保设备代码始终可以在任何应用程序中执行。

发现 SYCL 应用程序可以使用多个 CPU 设备应该不足为奇，其中一些旨在简化调试，而另一些则可能专注于执行性能。设备方面可用于区分这些不同的 CPU 设备，如本章后面所述。

当考虑使用 CPU 设备来开发和调试设备代码时，应考虑 CPU 和目标加速器架构（例如 GPU）之间的差异。特别是在优化代码性能时，特别是在使用更高级的功能（例如子组）时，跨架构的功能和性能可能存在一些差异。例如，当移动到新设备时，子组大小可能会发生变化。大多数开发和调试通常可以在 CPU 设备上进行，有时随后在目标设备架构上进行最终调整和调试。

CPU 设备在功能上类似于硬件加速器，队列可以与其绑定并且可以执行设备代码。图 2-8 显示了 CPU 设备如何与系统中可用的其他加速器对等。它可以执行设备代码，就像 GPU 或 FPGA 能够执行的方式一样，并且可以构建一个或多个与其绑定的队列。

应用程序可以通过将 cpu_selector_v 显式传递给队列构造函数来选择创建绑定到 CPU 设备的队列，如图 2-9 所示。

即使没有特别请求（例如，使用 cpu_selector_v），CPU 设备也可能恰好被默认选择器选择，如图 2-7 中的输出所示。

定义了设备选择器的一些变体，以便我们轻松地定位某种类型的设备。cpu_selector_v 是这些选择器的一个示例，我们将在接下来的部分中介绍其他选择器。

4.5 方法 #3：使用 GPU（或其他加速器）

下一个示例将展示 GPU，但任何类型的加速器都同样适用。为了轻松定位常见的加速器类别，设备被分为几个大类，并且 SYCL 为它们提供了内置选择器类别。要从广泛的设备类型（例如“系统中可用的任何 GPU”）中进行选择，相应的代码非常简短，如本节中所述。

4.5.1 加速器装置

在 SYCL 规范的术语中，有几组广泛的加速器类型：

1. CPU 设备。

2. GPU 设备。

3. 加速器，捕获不识别为 CPU 设备或 GPU 的设备。这包括 FPGA 和 DSP 设备。

来自任何这些类别的设备都可以使用内置选择器轻松绑定到队列，这些选择器可以传递给队列（和其他一些类）构造函数。

4.5.2 设备选择器

必须绑定到特定设备的类（例如队列类）具有可以接受 DeviceSelector 的构造函数。DeviceSelector 是一个可调用的设备，它采用常量引用设备，并按数字对其进行排名，以便运行时可以选择排名最高的设备。例如，接受 DeviceSelector 的队列构造函数是 queue(const DeviceSelector &deviceSelector, const property_list &propList =);

有四个内置选择器适用于各种常见的设备。

DPC++ 中包含的一个附加选择器（SYCL 中不可用）可通过包含标头“sycl ext intel fpga_extensions.hpp”来使用。

可以使用内置选择器之一构造队列，例如

队列 myQueue gpu_selector_v ; 图 2-10 显示了使用 GPU 选择器的完整示例，图 2-11 显示了队列与可用 GPU 设备的相应绑定。

图 2-12 显示了使用各种内置选择器的示例，并演示了设备选择器与另一个在构造时接受设备选择器的类（设备）的使用。

当设备选择失败时

如果在创建对象（例如队列）时使用 GPU 选择器，并且没有可供运行时使用的 GPU 设备，则选择器将引发 runtime_error 异常。对于所有设备选择器类都是如此，因为如果所需类的设备不可用，则会引发 runtime_error 异常。对于复杂的应用程序来说，捕获该错误并获取不太理想的（对于应用程序/算法）设备类作为替代方案是合理的。第 5 章更详细地讨论了异常和错误处理。

4.6 方法 #4：使用多个设备

如图 2-5 和 2-6 所示，我们可以在一个应用程序中构造多个队列。我们可以将这些队列绑定到单个设备（队列的工作总和集中到单个设备）、多

个设备或这些设备的某种组合。图 2-13 提供了一个示例，创建一个绑定到 GPU 的队列和另一个绑定到 FPGA 的队列。相应的映射如图 2-14 所示。

4.7 方法 #5：自定义（非常具体）的设备选择

现在我们将了解如何编写自定义选择器。除了本章中的示例之外，第 12 章中还显示了更多示例。内置设备选择器旨在让我们快速启动并运行代码。实际应用程序通常需要专门选择设备，例如从系统中可用的一组 GPU 类型中选择所需的 GPU。设备选择机制很容易扩展到任意复杂的逻辑，因此我们可以编写任何需要的代码来选择我们喜欢的设备。

4.7.1 根据设备方面进行选择

SYCL 定义了称为方面的设备属性。例如，设备可能展示的某些方面（在方面查询上返回 true）是 gpu、host_debuggable、fp64 和 online_compiler。请参阅 SYCL 规范的“设备方面”部分，了解标准方面及其定义的完整列表。

要使用 SYCL 中定义的方面来选择设备，可以使用 aspect_selector，如图 2-15 所示。以 aspect_selector 的形式，采用逗号分隔的 aspect 组，所有 aspect 都必须由要选择的设备显示。spect_ 选择器的另一种形式采用两个 std::vector。第一个向量包含设备中必须存在的方面，第二个向量包含设备中不得存在的方面（列出负面方面）。图 2-15 显示了使用这两种形式的 aspect_selector 的示例。

一些方面可用于推断设备的性能特征。例如，具有仿真方面的任何设备可能不如未仿真的相同类型的设备执行得那么好，而是可以表现出与改进的可调试性相关的其他方面。

4.7.2 通过自定义选择器进行选择

当现有方面不足以选择特定设备时，可以定义自定义设备选择器。这样的选择器只是一个 C++ 可调用的（例如，函数或 lambda），它接受 const Device& 作为参数，并返回特定设备的整数分数。SYCL 运行时在可以找到的所有可用根设备上调用选择器，并选择选择器返回最高分数的设备（该分数必须为非负数才能进行选择）。

如果最高分数出现平局，SYCL 运行时将选择平局设备之一。运行时不会选择选择器返回负数的任何设备，因此从选择器返回负数可保证该设备

不会被选择。

设备评分机制

我们有很多选项来创建与特定设备相对应的整数分数，例如：

1. 返回特定设备类别的正值。
2. 设备名称和/或设备供应商字符串的字符串匹配。
3. 根据设备或平台查询，计算我们可以想象得到的任何整数值。

例如，选择特定 Intel Arria FPGA 加速器板的一种可能方法如图 2-16 所示。

第 12 章有更多关于设备选择的讨论和示例,并更深入地讨论了 get_info 方法。

4.8 在设备上创建任务

应用程序通常包含主机代码和设备代码的组合。有一些类成员允许我们提交设备代码以供执行，并且由于这些工作调度构造是提交设备代码的唯一方法，因此它们使我们能够轻松区分设备代码和主机代码。

本章的其余部分介绍了一些工作调度结构，目的是帮助我们理解和识别设备代码和在主机处理器上本机执行的主机代码之间的划分。

4.8.1 任务图简介

SYCL 执行模型中的一个基本概念是节点图。该图中的每个节点（工作单元）都包含要在设备上执行的操作，最常见的操作是数据并行设备内核调用。图 2-17 显示了具有四个节点的示例图，其中每个节点都可以被视为设备内核调用。

图 2-17 中的节点具有依赖边，定义节点的工作何时开始执行是合法的。依赖边通常是根据数据依赖自动生成的，尽管我们可以通过一些方法在需要时手动添加额外的自定义依赖。例如，图中的节点 B 具有来自节点 A 的依赖边。该边意味着节点 A 必须完成执行，并且很可能（取决于依赖关系的具体情况）使生成的数据在节点 B 将执行的设备上可用在节点 B 的动作开始之前。运行时控制依赖关系的解析和节点执行的触发，与主机程序的执行完全异步。定义应用程序的节点图在本书中将称为任务图，并在第 3 章中进行更详细的介绍。

4.8.2 设备代码在哪里?

有多种机制可用于定义将在设备上执行的代码，但一个简单的示例展示了如何识别此类代码。即使示例中的模式乍一看很复杂，但该模式在所有设备代码定义中保持相同，因此很快就成为第二天性。

作为最后一个参数传递给 parallel_for 的代码（定义为图 2-18 中的 lambda 表达式）是要在设备上执行的设备代码。在这种情况下，parallel_for 是让我们区分设备代码和主机代码的构造。parallel_for 是一小组设备调度机制之一，所有成员都是处理程序类，定义要在设备上执行的代码。图 2-19 给出了处理程序类的简化定义。

除了调用处理程序类的成员来提交设备代码之外，还有队列类的成员允许提交工作。图 2-20 中所示的队列类成员是简化某些模式的快捷方式，我们将在以后的章节中看到这些快捷方式的使用。

4.8.3 行动

图 2-18 中的代码包含一个 parallel_for，它定义了要在设备上执行的工作。Parallel_for 位于提交给队列的命令组 (CG) 内，队列定义要在其上执行工作的设备。在命令组内，有两类代码:

1. 设置依赖关系的主机代码，定义运行时何时可以安全地开始执行 (2) 中定义的工作，例如创建缓冲区访问器（第 3 章中描述）
2. 最多调用一次对设备代码进行排队以供执行或执行手动内存操作（例如复制）的操作

处理程序类包含一小组成员函数，这些函数定义执行任务图节点时要执行的操作。图 2-21 总结了这些操作。

一个命令组内最多可以调用图 2-21 中的一个操作（调用多个操作是错误的），并且每个提交调用只能将一个命令组提交到队列中。其结果是，每个任务图节点都存在图 2-21 中的单个（或可能没有）操作，该操作将在满足节点依赖性并且运行时确定可以安全执行时执行。

代码在未来异步执行的想法是作为主机程序的一部分在 CPU 上运行的代码与将来在满足依赖性时运行的设备代码之间的关键区别。命令组通常包含每个类别的代码，其中定义依赖关系的代码作为主机程序的一部分运行（以便运行时知道依赖关系是什么），而设备代码则在满足依赖关系后运行。

图 2-22 中有三类代码：

1. 主机代码：驱动应用程序，包括创建和管理数据缓冲区以及将工作提交到队列以在任务图中形成新节点以进行异步执行。

2. 命令组内的主机代码：此代码在执行主机代码的处理器上运行，并在提交调用返回之前立即执行。例如，此代码通过创建访问器来设置节点依赖性。任何任意 CPU 代码都可以在这里执行，但最佳实践是将其限制为配置节点依赖项的代码。

3. 操作：图 2-21 中列出的任何操作都可以包含在命令组中，它定义了将来满足节点要求时异步执行的工作（由（2）设置）。

要了解应用程序中的代码何时运行，请注意，传递给图 2-21 中列出的启动设备代码执行的操作的任何内容，或图 2-21 中列出的显式内存操作，将来当 SYCL 任务图（稍后描述）节点依赖性已得到满足。所有其他代码立即作为主机程序的一部分运行，正如典型 C++ 代码中所预期的那样。

需要注意的是，虽然设备代码可以在满足任务图节点依赖性时开始（异步）运行，但不能保证设备代码在此时开始运行。确保设备代码开始执行的唯一方法是让主机程序通过主机访问器或队列等待操作等机制等待（阻塞）设备代码执行的结果，我们将在后面的章节中介绍这些机制。如果没有此类主机阻塞操作，SYCL 和较低级别的运行时将决定何时开始执行设备代码，可能会针对“尽快运行”以外的目标进行优化，例如针对功耗或拥塞进行优化。

4.8.4 主机任务

一般来说，提交到队列（例如通过 parallel_for）的操作执行的代码是设备代码，遵循一些语言限制，使其能够在许多体系结构上高效运行。不过，有一个重要的偏差是通过名为 host_task 的处理程序方法访问的。此方法允许将任意 C++ 代码作为任务图中的操作提交，并在满足任何任务图依赖性后在主机上执行。

宿主任务在某些程序中很重要，原因有二：

1. 可以包含任意 C++，甚至 std::cout 或 printf。这对于轻松调试、与 OpenCL 等较低级别 API 的互操作性或在现有代码中逐步启用加速器非常重要。

2. 主机任务作为任务图的一部分异步执行，而不是与主机程序同步执行。尽管主机程序可以启动附加线程或使用其他任务并行方法，但主机任务与 SYCL 运行时的依赖性跟踪机制集成。当设备和主机代码需要分散时，这非常方便，并且可能会带来更高的性能。

图 2-23 演示了一个简单的主机任务，当满足任务图依赖性时，它使用 std::cout 输出文本。请记住，主机任务是与主机程序的其余部分异步执行的。这是任务图机制的强大部分，其中 SYCL 运行时在安全时安排工作，而无需与主机程序交互，而主机程序可能会继续其他工作。

另请注意，主机任务的代码主体不需要遵循对设备代码施加的任何限制（如第 10 章所述）。

图 2-23 中的示例基于事件（在第 3 章中描述）来创建设备代码提交和后续主机任务之间的依赖关系，但是主机任务也可以通过以下方式与访问器（也在第 3 章中介绍）一起使用：target::host_task 的特殊访问器模板参数化（第 7 章）。

4.9 概括

在本章中，我们概述了队列、与队列关联的设备的选择以及如何创建自定义设备选择器。我们还概述了满足依赖性时在设备上异步执行的代码与作为 C++ 应用程序主机代码的一部分执行的代码。第 3 章介绍如何控制数据移动。

5 数据管理

超级计算机架构师经常感叹需要“喂养野兽”。“喂养野兽”一词指的是当我们使用大量并行性时我们创建的计算机的“野兽”，并向其提供数据成为需要解决的关键挑战。

在异构机器上提供 SYCL 程序需要小心，以确保数据在需要时位于需要的位置。在大型程序中，这可能需要大量工作。在现有的 C++ 程序中，仅仅弄清楚如何管理所需的所有数据移动就可能是一场噩梦。

我们将仔细解释管理数据的两种方式：统一共享内存（USM）和缓冲区。USM 是基于指针的，C++ 程序员对此很熟悉。缓冲区提供了更高级别的抽象。选择是好的。

我们需要控制数据的移动，本章将介绍实现这一目标的选项。

在第 2 章中，我们研究了如何控制代码的执行位置。我们的代码需要数据作为输入并生成数据作为输出。由于我们的代码可能在多个设备上运行，并且这些设备不一定共享内存，因此我们需要管理数据移动。即使数据是共享的（例如使用 USM），同步和一致性也是我们需要理解和管理的概念。

一个合乎逻辑的问题可能是“为什么编译器不自动为我们完成所有事情？”虽然可以自动为我们处理很多事情，但如果我们不宣称自己是程序员，那么性能通常不是最佳的。在实践中，为了获得最佳性能，我们在编写异构程序时需要关注代码放置（第 2 章）和数据移动（本章）。

本章概述了管理数据，包括控制数据使用的顺序。它是对前一章的补充，前一章向我们展示了如何控制代码的运行位置。本章帮助我们有效地使数据出现在我们要求代码运行的位置，这不仅对于正确执行应用程序很重要，而且对于最大限度地减少执行时间和功耗也很重要。

5.1 介绍

没有数据，计算就毫无意义。加速计算的全部目的是更快地产生答案。这意味着数据并行计算最重要的方面之一是它们如何访问数据，并将加速器设备引入机器使情况进一步复杂化。在传统的基于单插槽 CPU 的系统中，我们只有一个内存。加速器设备通常有自己的附加存储器，无法从主机直接访问。因此，支持分立设备的并行编程模型必须提供管理这些多个存储器并在它们之间移动数据的机制。

在本章中，我们概述了数据管理的各种机制。我们介绍了统一共享内存

和数据管理的缓冲区抽象，并描述了内核执行和数据移动之间的关系。

5.2 数据管理问题

从历史上看，用于并行编程的共享内存模型的优点之一是它们提供了单一的共享内存视图。拥有这种单一的内存视图可以简化生活。我们不需要做任何特殊的事情来从并行任务访问内存（除了适当的同步以避免数据竞争）。虽然某些类型的加速器设备（例如集成 GPU）与主机 CPU 共享内存，但许多离散加速器都有自己的本地内存，与 CPU 的内存分开，如图 3-1 所示。

5.3 本地设备与远程设备

在使用直接连接到设备的内存（而不是远程内存）读取和写入数据时，在设备上运行的程序通常性能更好。我们将对直接连接的存储器的访问称为本地访问。对另一台设备内存的访问是远程访问。远程访问往往比本地访问慢，因为它们必须通过带宽较低和/或延迟较高的数据链路进行传输。这意味着将计算和它将使用的数据放在一起通常是有利的。为了实现这一目标，我们必须以某种方式确保数据在不同内存之间复制或迁移，以便将其移至更靠近计算发生的位置。

5.4 管理多个内存

管理多个内存大致可以通过两种方式完成：显式地通过我们的程序或隐式地通过 SYCL 运行时库。每种方法都有其优点和缺点，我们可以根据情况或个人喜好选择其中一种。

5.4.1 显式数据移动

管理多个存储器的一种选择是在不同存储器之间显式复制数据。图 3-2 显示了一个具有离散加速器的系统，我们必须首先将内核所需的任何数据从主机内存复制到加速器内存。内核计算结果后，我们必须将这些结果复制回主机，然后主机程序才能使用该数据。

显式数据移动的主要优点是我们可以完全控制数据在不同内存之间传输的时间。这很重要，因为重叠计算与数据传输对于在某些硬件上获得最佳性能至关重要。

显式数据移动的缺点是指定所有数据移动可能很乏味且容易出错。传输不正确的数据量或不确保在内核开始计算之前已传输所有数据可能会导致不正确的结果。从一开始就确保所有数据移动正确可能是一项非常耗时的任务。

5.4.2 隐式数据

程序控制的显式数据移动的替代方案是由并行运行时或驱动程序控制的隐式数据移动。在这种情况下，并行运行时不需要在不同内存之间进行显式复制，而是负责确保数据在使用之前传输到适当的内存。

隐式数据移动的优点是，应用程序无需花费太多精力即可利用直接连接到设备的更快内存。所有繁重的工作都是由运行时自动完成的。这也减少了在程序中引入错误的机会，因为运行时将自动识别何时必须执行数据传输以及必须传输多少数据。

隐式数据移动的缺点是我们对运行时隐式机制的行为控制较少或无法控制。运行时将提供功能正确性，但可能无法以最佳方式移动数据，以确保计算与数据传输的最大重叠，这可能会对程序性能产生负面影响。

5.4.3 选择正确的策略

为项目选择最佳策略可能取决于许多不同的因素。不同的策略可能适合程序开发的不同阶段。我们甚至可以决定最好的解决方案是混合和匹配程序不同部分的显式和隐式方法。我们可能会选择开始使用隐式数据移动来简化将应用程序移植到新设备的过程。当我们开始调整应用程序的性能时，我们可能会开始在代码的性能关键部分用显式数据移动替换隐式数据移动。未来的章节将介绍如何将数据传输与计算重叠以优化性能。

5.5 USM、缓冲区和图像

管理内存有三个抽象：统一共享内存（USM）、缓冲区和图像。USM 是一种基于指针的方法，C/C++ 程序员应该熟悉。USM 的优点之一是更容易与现有的操作指针的 C++ 代码集成。缓冲区（由缓冲区模板类表示）描述一维、二维或三维数组。它们提供了可以在主机或设备上访问的内存的抽象视图。缓冲区不由程序直接访问，而是通过访问器对象使用。图像充当一种特殊类型的缓冲区，提供特定于图像处理的额外功能。此功能包括对特殊

图像格式的支持、使用采样器对象读取图像等等。缓冲区和图像是强大的抽象，可以解决许多问题，但重写现有代码中的所有接口以接受缓冲区或访问器可能非常耗时。由于缓冲区和图像的接口基本相同，因此本章的其余部分将仅关注 USM 和缓冲区。

5.6 统一共享内存

USM 是我们可用于数据管理的一种工具。USM 是一种基于指针的方法，使用 malloc 或 new 分配数据的 C 和 C++ 程序员应该熟悉它。USM 简化了移植大量使用指针的现有 C/C++ 代码的过程。支持 USM 的设备支持统一的虚拟地址空间。拥有统一的虚拟地址空间意味着主机上的 USM 分配例程返回的任何指针值都将是设备上的有效指针值。我们不必手动转换主机指针来获取“设备版本”——我们在主机和设备上看到相同的指针值。

USM 的更详细描述可以在第 6 章中找到。

5.6.1 通过指针访问内存

由于当系统同时包含主机内存和一定数量的设备连接本地内存时，并非所有内存都是平等创建的，因此 USM 定义了三种不同类型的分配：设备、主机和共享。所有类型的分配都在主机上执行。图 3-3 总结了每种分配类型的特征。

设备分配发生在设备附加内存中。这样的分配可以在设备上读取和写入，但不能从主机直接访问。我们必须使用显式复制操作在主机内存中的常规分配和设备分配之间移动数据。

主机分配发生在主机内存中，主机和设备上都可以访问该内存。这意味着相同的指针值在主机代码和设备内核中都有效。然而，当访问这样的指针时，数据总是来自主机存储器。如果在设备上访问，数据不会从主机迁移到设备本地内存。相反，数据通常通过总线发送，例如将设备连接到主机的 PCI Express (PCI-E)。

主机和设备上都可以访问共享分配。在这方面，它与主机分配非常相似，但不同之处在于数据现在可以在主机内存和设备本地内存之间迁移。这意味着迁移发生后，设备上的访问将从更快的设备本地内存中进行，而不是通过延迟较高的连接远程访问主机内存。通常，这是通过运行时内部的机制和对我们隐藏的较低级别驱动程序来完成的。

5.6.2 USM 和数据移动

USM 支持显式和隐式数据移动策略，不同的分配类型映射到不同的策略。设备分配要求我们在主机和设备之间显式移动数据，而主机和共享分配提供隐式数据移动。

USM 中的显式数据移动

USM 的显式数据移动是通过设备分配以及队列和处理程序类中的特殊 memcpy() 来完成的。我们将 memcpy() 操作（动作）排入队列，以将数据从主机传输到设备或从设备传输到主机。

图 3-4 包含一个在设备分配上运行的内核。在内核使用 memcpy() 操作执行之前和之后，数据会在 host_array 和 device_array 之间复制。调用队列上的 wait() 可确保在内核执行之前完成到设备的复制，并确保在数据复制回主机之前内核已完成。我们将在本章后面学习如何消除这些调用。

USM 中的隐式数据移动

USM 的隐式数据移动是通过主机和共享分配来完成的。通过这些类型的分配，我们不需要显式插入复制操作来在主机和设备之间移动数据。相反，我们只需访问内核内部的指针，任何所需的数据移动都会自动执行，无需程序员干预（只要您的设备支持这些分配）。这极大地简化了现有代码的移植：最多我们只需要简单地用适当的 USM 分配函数（以及调用 free 来释放内存）替换任何 malloc 或 new ，并且一切都应该正常工作。

在图 3-5 中，我们创建了两个数组：host_array 和 shared_array，分别是主机分配和共享分配。虽然主机和共享分配都可以在主机代码中直接访问，但我们在这里只初始化 host_array 。同样，可以在内核内部直接访问，进行数据的远程读取。运行时确保 shared_array 在内核访问它之前在设备上可用，并且当主机代码稍后读取它时将其移回，所有这些都无需程序员干预。

5.7 缓冲器

为数据管理提供的另一个抽象是缓冲区对象。缓冲区是一种数据抽象，表示给定 C++ 类型的一个或多个对象。缓冲区对象的元素可以是标量数据类型（例如 int、float 或 double）、向量数据类型（第 11 章）或用户定义的类或结构。SYCL 2020 定义了一个新概念“设备可复制”，它扩展了可简单复制的概念，并添加了允许类型集。特别是，如果常见 C++ 类（例如 std::array、std::pair、std::tuple 或 std::span）中的模板化类型本身是设备可复制的，那

么使用这些类型构建的那些 C++ 类特化也是设备可复制的可复制。在将数据类型与缓冲区一起使用之前，请注意您的数据类型是设备可复制的！

虽然缓冲区本身是单个对象，但缓冲区封装的 C++ 类型可以是包含多个对象的数组。缓冲区代表数据对象而不是特定的内存地址，因此不能像常规 C++ 数组一样直接访问。事实上，出于性能原因，缓冲区对象可能映射到多个不同设备上的多个不同内存位置，甚至映射到同一设备上。相反，我们使用访问器对象来读取和写入缓冲区。

缓冲区的更详细描述可以在第 7 章中找到。

5.7.1 创建缓冲区

可以通过多种方式创建缓冲区。最简单的方法是简单地构造一个新的缓冲区，其范围指定缓冲区的大小。然而，以这种方式创建缓冲区并不会初始化其数据，这意味着我们必须首先通过其他方式初始化缓冲区，然后才能尝试从中读取有用的数据。

还可以根据主机上的现有数据创建缓冲区。这是通过调用几个构造函数之一来完成的，这些构造函数采用指向现有主机分配的指针、一组 InputIterators 或具有某些属性的容器。在缓冲区构造期间，数据从现有主机分配复制到缓冲区对象的主机内存中。还可以使用 SYCL 互操作性功能（例如，从 OpenCL cl_mem 对象）从特定于后端的对象创建缓冲区。有关如何执行此操作的更多详细信息，请参阅有关互操作性的章节。

5.7.2 访问缓冲区

主机和设备可能无法直接访问缓冲区（除非通过此处未描述的高级且不常用的机制）。相反，我们必须创建访问器才能读取和写入缓冲区。访问器为运行时提供有关我们计划如何使用缓冲区中的数据的信息，使其能够正确安排数据移动。

5.7.3 接入方式

创建访问器时，我们可以通知运行时我们将如何使用它来提供更多优化信息。我们通过指定访问模式来做到这一点。访问模式在图 3-7 中描述的 access_mode 枚举类中定义。在图 3-6 所示的代码示例中，访问器 my_accessor 是使用默认访问模式 access_mode::read_write 创建的。这让运行时知道我们打算通过 my_accessor 读取和写入缓冲区。访问模式是运

行时优化隐式数据移动的方式。例如，access_mode::read 告诉运行时，在该内核开始执行之前，数据需要在设备上可用。如果内核仅通过访问器读取数据，则无需在内核完成后将数据复制回主机，因为我们没有修改它。同样，access_mode::write 让运行时知道我们将修改缓冲区的内容，并且可能需要在计算结束后将结果复制回来。

使用正确的模式创建访问器可以为运行时提供有关如何在程序中使用数据的更多信息。运行时使用访问器来排序数据的使用，但它也可以使用此数据来优化内核的调度和数据移动。第 7 章更详细地描述了访问模式和优化标签。

5.8 对数据的使用进行排序

内核可以被视为提交执行的异步任务。这些任务必须提交到队列，并安排它们在设备上执行。在许多情况下，内核必须按特定顺序执行，以便计算出正确的结果。如果要获得正确结果需要任务 A 先于任务 B 执行，则称任务 A 和任务 B 之间存在依赖关系 1。

然而，内核并不是必须调度的唯一任务形式。在内核开始执行之前，内核访问的任何数据都需要在设备上可用。这些数据依赖性可以以从一个设备到另一设备的数据传输的形式创建额外的任务。数据传输任务可以是显式编码的复制操作或更常见的由运行时执行的隐式数据移动。

如果我们获取程序中的所有任务以及它们之间存在的依赖关系，我们可以使用它来将信息可视化为图表。该任务图具体来说是有向无环图（DAG），其中节点是任务，边是依赖关系。该图是有向的，因为依赖关系是单向的：任务 A 必须在任务 B 之前发生。该图是非循环的，因为它不能包含从节点返回到自身的任何循环或路径。

在图 3-8 中，任务 A 必须在任务 B 和 C 之前执行。同样，B 和 C 必须在任务 D 之前执行。由于 B 和 C 之间没有依赖关系，因此运行时可以自由地以任何顺序执行它们（甚至并行）只要任务 A 已经执行。因此，该图可能的合法顺序是 A B C D、A C B D，如果 B 和 C 可以同时执行，甚至是 A B,C D。

任务可能与所有任务的子集具有依赖性。在这些情况下，我们只想指定对正确性重要的依赖关系。这种灵活性为运行时提供了优化任务图执行顺序的自由度。在图 3-9 中，我们扩展了图 3-8 中的早期任务图，添加了任务 E 和 F，其中 E 必须在 F 之前执行。但是，任务 E 和 F 与节点 A、B、C

和 D 没有依赖关系。这允许运行时从许多可能的合法顺序中进行选择来执行所有任务。

有两种不同的方法来对队列中任务的执行（例如启动内核）进行建模：队列可以按照提交的顺序执行任务，也可以按照我们指定的任何依赖项的任何顺序执行任务。定义。我们可以通过多种机制来定义正确排序所需的依赖关系。

5.8.1 有序队列

对任务进行排序的最简单选项是将它们提交到有序队列对象。有序队列按照任务提交的顺序执行任务，如图 3-10 所示。它们直观的任务排序意味着有序队列具有简单性的优点，但具有序列化任务的缺点，即使独立任务之间不存在依赖性。有序队列在启动应用程序时非常有用，因为它们简单、直观、执行顺序确定，并且适合许多代码。

5.8.2 无序队列

由于队列对象是无序队列（除非使用 inorder 队列属性创建），因此它们必须提供对提交给它们的任务进行排序的方法。队列通过让我们通知运行时任务之间的依赖关系来对任务进行排序。可以使用命令组显式或隐式地指定这些依赖性。我们将在以下部分中分别考虑它们。

命令组是指定任务及其依赖性的对象。命令组通常编写为 C++ lambda 表达式，作为参数传递给队列对象的 Submit() 方法。该 lambda 的唯一参数是对处理程序对象的引用。处理程序对象在命令组内部使用来指定操作、创建访问器并指定依赖关系。

与事件的显式依赖关系

任务之间的显式依赖关系类似于我们看到的示例（图 3-8），其中任务 A 必须在任务 B 之前执行。以这种方式表达依赖关系侧重于基于发生的计算而不是计算访问的数据的显式排序。请注意，表达计算之间的依赖关系主要与使用 USM 的代码相关，因为使用缓冲区的代码通过访问器表达大多数依赖关系。在图 3-4 和 3-5 中，我们只是告诉队列等待所有先前提交的任务完成，然后再继续。相反，我们可以通过事件对象来表达任务依赖性。将命令组提交到队列时，submit() 方法返回一个事件对象。这些事件可以通过两种方式使用。

首先，我们可以通过显式调用事件的 wait() 方法来通过主机进行同步。这会强制运行时等待生成事件的任务完成执行，然后主机程序才能继续执行。显式等待事件对于调试应用程序非常有用，但 wait() 可能会过度限制任务的异步执行，因为它会停止主机线程上的所有执行。类似地，我们还可以对队列对象调用 wait()，这将阻止主机上的执行，直到所有排队的任务完成为止。如果我们不想跟踪排队任务返回的所有事件，这可能是一个有用的工具。

这给我们带来了使用事件的第二种方式。处理程序类包含一个名为 depends_on() 的方法。此方法接受单个事件或事件向量，并通知运行时正在提交的命令组需要完成指定的事件，然后才能执行命令组内的操作。图 3-11 显示了如何使用 dependent_on() 来排序任务的示例。

与访问器的隐式依赖关系

任务之间的隐式依赖关系是根据数据依赖关系创建的。任务之间的数据依赖关系有三种形式，如图 3-12 所示。

数据依赖性以两种方式表达给运行时：访问器和程序顺序。运行时必须使用两者来正确计算数据依赖性。图 3-13 和 3-14 对此进行了说明。

在图 3-13 和 3-14 中，我们执行三个内核——computeB、readA 和 computeC——然后在主机上读回最终结果。内核 computeB 的命令组创建两个访问器 a 和 b。这些访问器使用访问标记 read_only 和 write_only 进行优化，以指定我们不使用默认访问模式 access_mode::read_write。我们将在第 7 章中了解有关访问标记的更多信息。内核 computeB 读取缓冲区 a_buf 并写入缓冲区 b_buf。在内核开始执行之前，必须将缓冲区 a_buf 从主机复制到设备。

内核 readA 还为缓冲区 a_buf 创建一个只读访问器。由于内核 readA 是在内核 computeB 之后提交的，因此这会创建 Read-afterRead (RAR) 场景。然而，RAR 不会对运行时施加额外的限制，并且内核可以自由地以任何顺序执行。事实上，运行时可能更喜欢在内核 computeB 之前执行内核 readA，甚至同时执行两者。两者都需要将缓冲区 a_buf 复制到设备，但内核 computeB 还需要复制缓冲区 b_buf，以防任何现有值不被 computeB 覆盖并且可能被以后的内核使用。这意味着运行时可以在缓冲区 b_buf 的数据传输发生时执行内核 readA，并且还表明即使内核仅写入缓冲区，缓冲区的原始内容仍可能被移动到设备，因为无法保证缓冲区中的所有值都将由内核写入（请参阅第 7 章了解允许我们在这些情况下进行优化的标签）。

内核 computeC 读取缓冲区 b_buf，这是我们在内核 computeB 中计算的。由于我们在提交内核 computeB 之后提交了内核 computeC，这意味着内核 computeC 对缓冲区 b_buf 有 RAW 数据依赖。RAW 依赖关系也称为真实依赖关系或流依赖关系，因为数据需要从一个计算流到另一个计算才能计算出正确的结果。最后，我们还在内核 computeC 和主机之间创建对缓冲区 c_buf 的 RAW 依赖，因为主机希望在内核完成后读取 C。这会强制运行时将缓冲区 c_buf 复制回主机。由于设备上没有对缓冲区 a_buf 进行写入，因此运行时不需要将该缓冲区复制回主机，因为主机已经拥有最新的副本。

在图 3-15 和 3-16 中，我们再次执行三个内核：computeB、rewriteA 和 rewriteB。内核 computeB 再次读取缓冲区 a_buf 并写入缓冲区 b_buf，内核 rewriteA 写入缓冲区 a_buf，内核 rewriteB 写入缓冲区 b_buf。理论上，内核 rewriteA 可以比内核 computeB 更早执行，因为在内核准备好之前需要传输的数据较少，但它必须等到内核 computeB 完成之后，因为对缓冲区 a_buf 存在 WAR 依赖性。

在这个例子中，内核 computeB 需要来自主机的 A 的原始值，如果内核 rewriteA 在内核 computeB 之前执行，它将读取错误的值。WAR 依赖也称为反依赖。RAW 依赖性确保数据正确地流向正确的方向，而 WAR 依赖性确保现有值在读取之前不会被覆盖。WAW 对缓冲区 b_buf 的依赖在内核重写函数中也类似。如果在内核 computeB 和 rewriteB 之间提交了对缓冲区 b_buf 的任何读取，它们将导致 RAW 和 WAR 依赖性，从而正确排序任务。然而，在此示例中，内核 rewriteB 和主机之间存在隐式依赖性，因为最终数据必须写回主机。我们将在第 7 章中详细了解导致此写回的原因。WAW 依赖性，也称为输出依赖性，可确保最终输出在主机上正确。

5.9 选择数据管理策略

为我们的应用程序选择正确的数据管理策略很大程度上取决于个人喜好。事实上，我们可能会从一种策略开始，随着我们的计划成熟而转向另一种策略。然而，有一些有用的指南可以帮助我们选择满足我们需求的策略。

首先要做的决定是我们是否要使用显式或隐式数据移动，因为这极大地影响了我们需要对程序执行的操作。隐式数据移动通常是一个更容易开始的地方，因为所有数据移动都为我们处理，让我们专注于计算的表达。

如果我们决定从一开始就完全控制所有数据移动，那么我们要从使用

USM 设备分配的显式数据移动开始。我们只需要确保在主机和设备之间添加所有必要的副本即可!

当选择隐式数据移动策略时，我们仍然可以选择是使用缓冲区还是 USM 主机或共享指针。同样，这种选择取决于个人喜好，但有几个问题可以帮助我们选择其中一个。如果我们要移植使用指针的现有 C/C++ 程序，USM 可能是一条更简单的路径，因为大多数代码不需要更改。如果数据表示没有引导我们做出偏好，我们可以问的另一个问题是我们希望如何表达内核之间的依赖关系。如果我们更愿意考虑内核之间的数据依赖性，请选择缓冲区。如果我们更愿意将依赖关系视为在另一项计算之前执行一项计算，并希望使用有序队列或显式事件或内核之间的等待来表达这一点，请选择 USM。

当使用 USM 指针（显式或隐式数据移动）时，我们可以选择要使用哪种类型的队列。中序队列简单直观，但它们限制了运行时间并可能限制性能。无序队列更复杂，但它们为运行时提供了更大的自由度来重新排序和重叠执行。如果我们的程序在内核之间具有复杂的依赖关系，那么无序队列类是正确的选择。如果我们的程序只是一个接一个地运行许多内核，那么有序队列对我们来说将是一个更好的选择。

5.10 处理程序类：关键成员

我们已经展示了多种使用处理程序类的方法。图 3-17 和 3-18 提供了这个非常重要的类的关键成员的更详细的解释。我们还没有使用所有这些成员，但稍后将在本书中使用它们。这是放置它们的好地方。

一个密切相关的类，队列类，在第 2 章末尾有类似的解释。

5.11 概括

在本章中，我们介绍了解决数据管理问题的机制以及如何排序数据的使用。使用加速器设备时，管理对不同内存的访问是一个关键挑战，我们有不同的选项来满足我们的需求。

我们概述了数据使用之间可能存在的不同类型的依赖关系，并描述了如何向队列提供有关这些依赖关系的信息，以便它们正确排序任务。

本章概述了统一共享内存和缓冲区。我们在第 6 章中更详细地探讨了 USM 的所有模式和行为。第 7 章更深入地探讨了缓冲区，包括创建缓冲区

和控制其行为的所有不同方法。第 8 章回顾了控制内核执行和数据移动顺序的队列调度机制。

6 表达并行性

我们已经知道如何在设备上放置代码（第 2 章）和数据（第 3 章）——我们现在要做的就是决定如何处理它。为此，我们现在转而填补一些迄今为止我们方便地遗漏或掩盖的事情。本章标志着从简单的教学示例到现实世界并行代码的转变，并扩展了我们在前面的章节中随意展示的代码示例的细节。

用一种新的并行语言编写我们的第一个程序似乎是一项艰巨的任务，特别是如果我们是并行编程的新手。语言规范不是为应用程序开发人员编写的，并且通常假设对术语有一定的熟悉；它们不包含以下问题的答案：

为什么有不止一种方式来表达并行性?

我应该使用哪种表达并行性的方法?

关于执行模型我到底需要了解多少?

本章旨在解决这些问题以及更多问题。我们介绍了数据并行内核的概念，使用工作代码示例讨论了不同内核形式的优点和缺点，并强调了内核执行模型的最重要方面。

6.1 内核内的并行性

近年来，并行内核作为表达数据并行性的强大手段而出现。基于内核的方法的主要设计目标是跨各种设备的可移植性和高程序员生产力。因此，内核通常不会被硬编码为与特定数量或配置的硬件资源（例如，核心、硬件线程、SIMD [单指令，多数据] 指令）一起工作。相反，内核根据抽象概念来描述并行性，然后实现（即编译器和运行时的组合）可以将其映射到特定目标设备上可用的硬件并行性。尽管此映射是实现定义的，但我们可以（并且应该）相信实现选择合理且能够有效利用硬件并行性的映射。

以与硬件无关的方式公开大量并行性可确保应用程序可以扩展（或缩小）以适应不同平台的功能，但是.....

支持的设备存在很大的多样性，我们必须记住，不同的架构是针对不同的用例设计和优化的。每当我们希望在特定设备上实现最高水平的性能时，无论我们使用哪种编程语言，我们都应该始终期望需要一些额外的手动优化工作！此类特定于设备的优化的示例包括针对特定缓存大小的阻塞、选择分摊调度开销的工作粒度大小、利用专用指令或硬件单元，以及最重要的是选择适当的算法。其中一些示例将在第 15、16 和 17 章中重新讨论。

在应用程序开发过程中在性能、可移植性和生产力之间取得适当的平衡是我们所有人都必须面对的挑战，也是本书无法完全解决的挑战。然而，我们希望表明，带有 SYCL 的 C++ 提供了使用单一高级编程语言维护通用可移植代码和优化的目标特定代码所需的所有工具。剩下的就留给读者作为练习了！

6.2 循环与内核

迭代循环本质上是串行构造：循环的每次迭代都是按顺序执行的（即按顺序）。优化编译器也许能够确定循环的部分或全部迭代可以并行执行，但它必须是保守的 - 如果编译器不够智能或没有足够的信息来证明并行执行始终是安全的，则它必须保留循环的顺序语义以确保正确性。

考虑图 4-1 中的循环，它描述了一个简单的向量加法。即使在这样的简单情况下，证明循环可以并行执行也不是微不足道的：只有当 c 不与 a 或 b 重叠时，并行执行才是安全的，而在一般情况下，如果没有运行时检查，就无法证明这一点！为了解决这样的情况，语言添加了一些功能，使我们能够为编译器提供额外的信息，这些信息可以简化分析（例如，断言指针不与限制重叠）或完全覆盖所有分析（例如，声明循环是独立的或准确定义如何将循环调度到并行资源）。

并行循环的确切含义有些模糊（由于不同并行编程语言和运行时对该术语的重载），但许多常见的并行循环结构表示应用于顺序循环的编译器转换。这种编程模型使我们能够编写顺序循环，然后才提供有关如何安全地并行执行不同迭代的信息。这些模型非常强大，与其他最先进的编译器优化集成良好，并极大地简化了并行编程，但并不总是鼓励我们在开发的早期阶段考虑并行性。

并行内核不是循环并且没有迭代。相反，内核描述了单个操作，该操作可以多次实例化并应用于不同的输入数据；当并行启动内核时，该操作的多个实例可能会同时执行。

图 4-2 显示了使用伪代码重写为内核的简单循环示例。该内核中的并行机会是清晰明确的：内核可以由任意数量的实例并行执行，并且每个实例独立地应用于单独的数据块。通过将此操作编写为内核，我们断言并行运行是安全的（并且理想情况下应该并行运行）。

简而言之，基于内核的编程不是一种将并行性改进到现有顺序代码中的方法，而是一种编写显式并行应用程序的方法。

6.3 多维内核

许多其他语言的并行结构是一维的，将工作直接映射到相应的一维硬件资源（例如，硬件线程的数量）。SYCL 中的并行内核是一个比这更高级别的概念，它们的维度更能反映我们的代码通常试图解决的问题（在一维、二维或三维空间中）。

然而，我们必须记住，并行内核提供的多维索引为程序员提供了便利，可以在底层一维空间之上实现。了解这种映射的行为方式可能是某些优化（例如，调整内存访问模式）的重要部分。

一个重要的考虑因素是哪个维度是连续的或单位步幅（即，多维空间中的哪些位置在一维映射中彼此相邻）。SYCL 中与并行性相关的所有多维量都使用相同的约定：维度从 0 到 N-1 进行编号，其中维度 N-1 对应于连续维度。无论多维数量被写为列表（例如，在构造函数中）或类支持多个下标运算符，此编号都从左到右应用（从左侧的维度 0 开始）。此约定与标准 C++ 中多维数组的行为一致。

使用 SYCL 约定将二维空间映射到线性索引的示例如图 4-3 所示。我们当然可以自由地打破这个约定并采用我们自己的线性化索引的方法，但必须小心行事——打破 SYCL 约定可能会对受益于 strideone 访问的设备产生负面的性能影响。

如果应用程序需要三个以上的维度，我们必须负责使用模算术或其他技术手动在多维和线性索引之间进行映射。

6.4 语言特性概述

一旦我们决定编写并行内核，我们必须决定要启动什么类型的内核以及如何在程序中表示它。表达并行内核的方法有很多种，如果我们想掌握这门语言，我们需要熟悉每一种方法。

6.4.1 将内核与主机代码分离

我们有几种分离主机和设备代码的替代方法，可以在应用程序中混合和匹配这些代码：C++ lambda 表达式或函数对象、通过互操作性接口定义的内核（例如 OpenCL C 源字符串）或二进制文件。其中一些选项已在第 2 章中介绍，其他选项将在第 10 章和第 20 章中详细介绍。

所有这些选项都共享表达并行性的基本概念。为了保持一致性和简洁性，本章中的所有代码示例都使用 C++ lambda 表达式来表达内核。

注 13 (Lambda 表达式不被认为是有害的) *为了开始使用 sYCl，无需完全理解 C++ 规范中有关 lambda 表达式的所有内容 - 我们需要知道的是 lambda 表达式的主体代表内核，并且（按值）捕获的变量将是作为参数传递给内核。*

使用 lambda 表达式而不是更详细的机制来定义内核不会对性能产生影响。支持 sYCl 的 C++ 编译器始终能够理解 lambda 表达式何时表示并行内核的主体，并可以相应地针对并行执行进行优化。

有关 C++ lambda 表达式的复习及其在 sYCl 中的使用说明，请参阅第 1 章。有关使用 lambda 表达式定义内核的更多具体细节，请参阅第 10 章。

6.5 不同形式的并行内核

SYCL 中有三种不同的内核形式，支持不同的执行模型和语法。可以使用任何内核形式编写可移植内核，并且可以调整以任何形式编写的内核以在各种设备类型上实现高性能。然而，有时我们可能希望使用特定的形式来使特定的并行算法更容易表达或利用其他无法访问的语言功能。

第一种形式用于基本数据并行内核，并提供编写内核的最温和的介绍。对于基本内核，我们牺牲了对调度等低级功能的控制，以使内核的表达尽可能简单。各个内核实例如何映射到硬件资源完全由实现控制，因此随着基本内核复杂性的增加，推断其性能变得越来越困难。

第二种形式扩展了基本内核以提供对低级性能调整功能的访问。由于历史原因，第二种形式被称为 ND 范围（N 维范围）数据并行，最重要的是要记住，它使某些内核实例能够分组在一起，从而允许我们对数据局部性和数据局部性进行一些控制。各个内核实例和用于执行它们的硬件资源之间的映射。

第三种形式提供了一种实验性的替代语法，用于使用类似于嵌套并行循环的语法来表达 ND 范围内核。第三种形式称为分层数据并行，指的是用户源代码中出现的嵌套结构的层次结构。编译器对此语法的支持仍然不成熟，并且许多 SYCL 实现不能像其他两种形式那样有效地实现分层数据并行内核。语法也不完整，因为 SYCL 有许多与分层内核不兼容或无法访问的性能支持功能。SYCL 中的分层并行性正在更新过程中，并且 SYCL 规

范包含一条注释，建议新代码在该功能准备就绪之前不要使用分层并行性；为了与本说明的精神保持一致，本书的其余部分仅教授基本的和 ND 范围的并行性。

在更详细地讨论了不同内核形式的特性后，我们将在本章末尾再次讨论如何在不同内核形式之间进行选择。

6.6 基础数据并行内核

并行内核的最基本形式适用于高度并行的操作（即可以完全独立且以任何顺序应用于每条数据的操作）。通过使用这种形式，我们可以实现对工作安排的完全控制。因此，它是描述性编程构造的一个示例——我们描述操作是极其并行的，并且所有调度决策都是由实现做出的。

基本数据并行内核以单程序、多数据 (SPMD) 风格编写——单个“程序”（内核）应用于多条数据。请注意，由于数据相关的分支，此编程模型仍然允许内核的每个实例在代码中采用不同的路径。

SPMD 编程模型的最大优势之一是它允许将相同的“程序”映射到多个级别和类型的并行性，而无需我们的任何明确指示。同一程序的实例可以通过管道传输、打包在一起并使用 SIMD 指令执行、分布在多个硬件线程上或三者的混合。

6.6.1 了解基本数据并行内核

基本并行内核的执行空间被称为它的执行范围，并且内核的每个实例被称为一个项目。图 4-4 对此进行了示意性表示。

基本数据并行内核的执行模型非常简单：它允许完全并行执行，但不保证或要求它。项目可以按任何顺序执行，包括在单个硬件线程上顺序执行（即，没有任何并行性）！因此，假设所有项目都将并行执行的内核（例如，通过尝试同步项目）可能很容易导致程序在某些设备上挂起。

然而，为了保证正确性，我们必须始终在假设内核可以并行执行的情况下编写内核。例如，我们有责任确保对内存的并发访问受到原子内存操作（参见第 19 章）的适当保护，以防止竞争条件。

6.6.2 编写基本数据并行内核

基本数据并行内核使用 parallel_for 函数表示。图 4-5 显示了如何使用这个函数来表达向量加法，这是我们对“Hello, world!”的看法。用于并行加速器编程。

该函数仅接受两个参数：第一个是范围（或整数），指定在每个维度中启动的项目数，第二个是要对该范围中的每个索引执行的内核函数。有几个不同的类可以被接受作为内核函数的参数，并且应该使用哪个类取决于哪个类公开所需的功能 - 我们稍后将重新讨论这一点。

图 4-6 显示了使用该函数非常类似地表达矩阵加法，除了二维数据之外，它与向量加法（在数学上）相同。这由内核反映出来——两个代码片段之间的唯一区别是所使用的 range 和 id 类的维度！可以用这种方式编写代码，因为 SYCL 访问器可以通过多维 id 进行索引。尽管看起来很奇怪，但它非常强大，使我们能够编写根据数据维度模板化的通用内核。

在 C/C++ 中更常见的是使用多个索引和多个下标运算符来索引多维数据结构，并且访问器也支持这种显式索引。当内核同时操作不同维度的数据时，或者当内核的内存访问模式比直接使用项目的 id 描述的更复杂时，以这种方式使用多个索引可以提高可读性。

例如，图 4-7 中的矩阵乘法内核必须提取索引的两个单独分量，以便能够描述两个矩阵的行和列之间的点积。作者认为，一致使用多个下标运算符（例如，[j][k]）比混合多种索引模式和构造二维 id 对象（例如 id(j,k)）更具可读性，但这是只是个人喜好问题。

本章其余部分的示例都使用多个下标运算符，以确保所访问的缓冲区的维数不存在歧义。

图 4-8 中的图表显示了矩阵乘法内核中的工作如何映射到各个项目。请注意，项目数是根据输出范围的大小得出的，并且多个项目可以读取相同的输入值：每个项目通过顺序迭代 C 矩阵的（连续）行来计算 C 矩阵的单个值。A 矩阵和 B 矩阵的一个（不连续）列。

6.6.3 基本数据并行内核的详细信息

基本数据并行内核的功能通过三个 C++ 类公开：range、id 和 item。我们已经在前面的章节中多次看到过 range 和 id 类，但我们在这里以不同的焦点重新审视它们。

Range 类

范围表示一维、二维或三维范围。范围的维度是模板参数，因此必须在编译时已知，但每个维度的大小是动态的，并在运行时传递给构造函数。range 类的实例用于描述并行构造的执行范围和缓冲区的大小。

图 4-9 显示了范围类的简化定义，显示了构造函数和查询其范围的各种方法。

id 类

id 表示一维、二维或三维范围的索引。id 的定义在许多方面与 range 相似：它的维数也必须在编译时已知，并且它可用于索引并行构造中内核的单个实例或缓冲区中的偏移量。

如图 4-10 中 id 类的简化定义所示，id 在概念上只不过是一个、两个或三个整数的容器。我们可用的操作也非常简单：我们可以查询每个维度中索引的组成部分，并且可以执行简单的算术来计算新的索引。

虽然我们可以构造一个 id 来表示任意索引，但要获取与特定内核实例关联的 id，我们必须接受它（或包含它的项）作为内核函数的参数。这个 id（或者它的成员函数返回的值）必须被转发到我们想要查询索引的任何函数——目前没有任何自由函数可以在程序中的任意点查询索引，但这可以简化为 SYCL 的未来版本。

每个接受 id 的内核实例只知道它被分配计算的范围内的索引，而对范围本身一无所知。如果我们希望内核实例知道它们自己的索引和范围，我们需要使用 item 类。

item 类

项代表内核函数的单个实例，封装了内核的执行范围和该范围内的实例索引（分别使用范围和 id）。与 range 和 id 一样，它的维数必须在编译时已知。

图 4-11 给出了项目类的简化定义。item 和 id 之间的主要区别在于 item 公开了额外的函数来查询执行范围的属性（例如，其大小）以及计算线性化索引的便利函数。与 id 一样，获取与特定内核实例关联的项的唯一方法是将其作为内核函数的参数接受。

6.7 显式 ND 范围内核

并行内核的第二种形式用项目属于组的执行范围替换基本数据并行内核的平坦执行范围。这种形式最适合我们想要在内核中表达某些局部性概念的情况。为不同类型的组定义和保证不同的行为，使我们能够更深入地了

解和/或控制如何将工作映射到特定的硬件平台。

因此，这些显式 ND 范围内核是更具规范性的并行构造的示例 - 我们规定了工作到每种类型组的映射，并且实现必须遵守该映射。然而，它并不完全是规定性的，因为组本身可以按任何顺序执行，并且实现对于每种类型的组如何映射到硬件资源保留了一定的自由度。这种规范性和描述性编程的结合使我们能够针对局部性设计和调整内核，而不会破坏其可移植性。

与基本数据并行内核一样，ND 范围内核以 SPMD 风格编写，其中所有工作项都执行应用于多个数据的相同内核“程序”。主要区别在于每个程序实例都可以查询其在包含它的组中的位置，并且可以访问特定于每种类型的组的附加功能（请参见第 9 章）。

6.7.1 了解显式 ND 范围并行内核

ND 范围内核的执行范围分为工作组、子组和工作项。ND-range 表示总的执行范围，它被划分为统一大小的工作组（即，工作组大小必须在每个维度上精确地除以 ND-range 大小）。每个工作组可以根据实施进一步划分为子组。了解为工作项和每种类型的组定义的执行模型是编写正确且可移植的程序的重要组成部分。

图 4-12 显示了大小为 (8, 8, 8) 的 ND 范围分为 8 个大小为 (4, 4, 4) 的工作组的示例。每个工作组包含 16 个一维子组，每组有 4 个工作项。请特别注意维度的编号：子组始终是一维的，因此 ND 范围和工作组的维度 2 成为子组的维度 0。

从每种类型的组到硬件资源的精确映射是实现定义的，正是这种灵活性使得程序能够在各种硬件上执行。例如，工作项可以完全顺序执行、由硬件线程和/或 SIMD 指令并行执行、或者甚至由专门为内核配置的硬件管道执行。

在本章中，我们仅关注 ND 范围执行模型在通用目标平台方面的语义保证，并且我们不会涵盖其到任何一个平台的映射。有关 GPU、CPU 和 FPGA 的硬件映射和性能建议的详细信息，请分别参阅第 15、16 和 17 章。

6.7.2 编写显式 ND 范围数据并行内核

工作项

工作项代表核函数的各个实例。在没有其他分组的情况下，工作项可以按任何顺序执行，并且不能相互通信或同步，除非通过对全局内存的原子内

存操作（参见第 19 章）。

工作组

ND 范围内的工作项被组织成工作组。工作组可以按任何顺序执行，不同工作组中的工作项不能相互通信，除非通过对全局内存的原子内存操作（参见第 19 章）。然而，当使用某些构造时，工作组内的工作项具有一些调度保证，并且该局部性提供了一些附加功能：

1. 工作组中的工作项可以访问工作组本地内存，该内存可能会映射到某些设备上的专用快速内存（请参阅第 9 章）。

2. 工作组中的工作项可以使用工作组屏障进行同步，并使用工作组内存栅栏保证内存一致性（参见第 9 章）。

3. 工作组中的工作项可以访问组功能，提供通用通信例程（参见第 9 章）和组算法的实现，提供通用并行模式的实现，例如归约和扫描（参见第 14 章）。

工作组中工作项的数量通常在运行时为每个内核配置，因为最佳分组将取决于可用并行度（即 ND 范围的大小）和目标设备的属性。我们可以使用设备类的查询函数确定特定设备支持的每个工作组的最大工作项数（参见第 12 章），并且我们有责任确保每个内核请求的工作组大小已验证。

工作组执行模型中有一些微妙之处值得强调。

首先，虽然工作组中的工作项被调度到单个计算单元，但是工作组的数量和计算单元的数量之间不需要有任何关系。事实上，ND 范围内的工作组数量可能比给定设备可以同时执行的工作组数量大很多倍！我们可能会尝试编写通过依赖非常聪明的设备特定调度来跨工作组同步的内核，但我们强烈建议不要这样做——这样的内核今天可能可以工作，但不能保证它们将来也能工作实现，并且当移动到不同的设备时很可能会中断。

其次，虽然工作组中的工作项目被安排为可以相互合作，但它们不需要提供任何具体的前进进度保证——在障碍和集体之间顺序执行工作组内的工作项目是一种有效实施。仅当使用提供的屏障和集合函数执行时，同一工作组中的工作项之间的通信和同步才能保证安全，并且手工编码的同步例程可能会死锁。

子组

在许多现代硬件平台上，工作组中的工作项子集（称为子组）在附加调度保证的情况下执行。例如，子组中的工作项可以由于编译器向量化而同时执行，和/或子组本身可以以强大的前进进度保证来执行，因为它们被映射

到独立的硬件线程。

当使用单一平台时，很容易将关于这些执行模型的假设融入到我们的代码中，但这使得它们本质上不安全且不可移植——在不同编译器之间移动时，甚至在不同代硬件之间移动时，它们可能会崩溃。同一个供应商!

将子组定义为语言的核心部分为我们提供了一种安全的替代方案，可以避免做出稍后可能被证明是特定于设备的假设。利用子组功能还允许我们在低级别（即接近硬件）推理工作项的执行，并且是跨许多平台实现非常高的性能水平的关键。

与工作组一样，子组内的工作项可以同步、保证内存一致性或通过组函数和组算法执行常见的并行模式。然而，子组没有工作组本地存储器的等价物（即，没有子组本地存储器）。相反，子组中的工作项可以使用组算法的子集（俗称"洗牌"操作）直接交换数据，无需显式内存操作（第 9 章）。

子组的某些方面是由实现定义的，不在我们的控制范围内。然而，对于给定的设备、内核和 ND 范围的组合，子组具有固定（一维）大小，我们可以使用内核类的查询函数来查询该大小（参见第 10 章和第 12 章）。默认情况下，每个子组的工作项数量也由实现选择 - 我们可以通过在编译时请求特定的子组大小来覆盖此行为，但必须确保我们请求的子组大小与设备兼容。

与工作组一样，子组中的工作项不需要提供任何特定的前进进度保证 - 实现可以自由地顺序执行子组中的每个工作项，并且仅在工作项发生变化时才在工作项之间切换。遇到子群集体函数。然而，在某些设备上，工作组内的所有子组都保证最终执行（取得进展），这是多种生产者-消费者模式的基石。这是当前实现定义的行为，因此如果我们希望内核保持可移植性，我们就不能依赖子组来取得进展。我们期望 SYCL 的未来版本能够提供描述子组进度保证的设备查询。

当为特定设备编写内核时，工作项到子组的映射是已知的，并且我们的代码通常可以利用此映射的属性来提高性能。然而，一个常见的错误是假设因为我们的代码可以在一台设备上运行，所以它也可以在所有设备上运行。图 4-13 和 4-14 仅显示了将范围为 4, 4 的多维内核中的工作项映射到子组（最大子组大小为 8）时的两种可能性。图 4-13 生成两个包含 8 个工作项的子组，而图 4-14 中的映射生成四个包含 4 个工作项的子组!

SYCL 当前不提供查询工作项如何映射到子组的方法，也不提供请求特定映射的机制。使用子组编写可移植代码的最佳方法是使用一维工作组或多维工作组，其中最高编号的维度可被内核所需的子组大小整除。

编写显式 ND 范围数据并行内核

图 4-15 使用 ND 范围并行内核语法重新实现了我们之前看到的矩阵乘法内核，图 4-16 中的图表显示了该内核中的工作如何映射到每个工作组中的工作项。以这种方式对工作项进行分组可确保访问的局部性，并有望提高缓存命中率：例如，图 4-16 中的工作组的本地范围为 (4, 4)，包含 16 个工作项，但仅访问 4 个工作项数据量是单个工作项的数据量的四倍——换句话说，我们从内存加载的每个值都可以重复使用四次。

到目前为止，我们的矩阵乘法示例依赖于硬件缓存来优化同一工作组中的工作项对 A 和 B 矩阵的重复访问。此类硬件缓存在传统 CPU 架构上很常见，并且在 GPU 架构上变得越来越常见，但一些架构已经明确管理可以提供更高性能（例如，通过更低延迟）的"暂存器"内存。ND 范围内核可以使用本地访问器来描述应放置在工作组本地内存中的分配，然后实现可以自由地将这些分配映射到特殊内存（如果存在）。该工作组本地内存的使用将在第 9 章中介绍。

6.7.3 显式 ND 范围数据并行内核的详细信息

与基本数据并行内核相比，ND 范围数据并行内核使用不同的类：range 被 nd_range 替换，item 被 nd_item 替换。还有两个新类，代表工作项可能所属的不同类型的组：与工作组相关的功能封装在组类中，与子组相关的功能封装在 sub_group 类中。

nd_range 类

nd_range 使用 range 类的两个实例表示分组执行范围：一个表示全局执行范围，另一个表示每个工作组的本地执行范围。图 4-17 给出了 nd_range 类的简化定义。

可能有点令人惊讶的是 nd_range 类根本没有提及子组：子组范围在构造过程中没有指定并且无法查询。造成这一遗漏的原因有两个。首先，子组是一个低级实现细节，对于许多内核来说可以忽略。其次，有多种设备恰好支持一种有效的子组大小，并且在任何地方指定该大小将是不必要的冗长。与子组相关的所有功能都封装在一个专用类中，稍后将讨论该类。

nd_item 类

nd_item 是项目的 ND 范围形式，再次封装了内核的执行范围以及该范围内项目的索引。nd_item 与 item 的不同之处在于如何查询和表示其在范围中的位置，如图 4-18 中简化的类定义所示。例如，我们可以使

用 get_global_id() 函数查询（全局）ND 范围中的项目索引，或者使用 get_local_id() 函数查询项目在其（本地）父工作组中的索引。

nd_item 类还提供了用于获取描述项目所属组和子组的类句柄的函数。这些类提供了用于查询 ND 范围中项目索引的替代接口。

group 类

组类封装了与工作组相关的所有功能，简化的定义如图 4-19 所示。

group 类提供的许多函数在 nd_item 类中都有等效的函数：例如，调用 group.get_group_id() 相当于调用 item.get_group_id()，调用 group.get_local_range() 相当于调用 item.get_local_range()。如果我们不使用任何群函数或算法，我们还应该使用群类吗？直接使用 nd_item 中的函数而不是创建中间组对象不是更简单吗？这里有一个权衡：使用 group 需要我们编写稍微多一些的代码，但该代码可能更容易阅读。例如，考虑图 4-20 中的代码片段：很明显，body 期望被组中的所有工作项调用，并且很明显，parallel_for 主体中的 get_local_range() 返回的范围是组的范围。仅使用 nd_item 可以很容易地编写相同的代码，但读者可能会更难理解。

组类启用的另一个强大选项是能够编写通过模板参数接受任何类型的组的通用组函数。尽管 SYCL（尚未）定义正式的 Group“概念”（在 C++20 意义上），但 group 和 sub_group 类公开了一个公共接口，允许使用 sycl::is_group_v 等特征来约束模板化 SYCL 函数。如今，这种通用编码形式的主要优点是能够支持具有任意维数的工作组，以及允许函数的调用者决定该函数是否应该在工作项之间划分工作的能力。工作组或子组中的工作项。然而，SYCL 组接口被设计为可扩展的，我们期望在 SYCL 的未来版本中出现更多代表不同工作项分组的类。

sub_group 类

sub_group 类封装了与子组相关的所有功能，简化的定义如图 4-21 所示。与工作组不同，sub_group 类是访问子组功能的唯一方法；它的功能在 nd_item 中没有重复。

请注意，有单独的函数用于查询当前子组中的工作项数以及工作组内任何子组中的最大工作项数。这些是否不同以及如何不同取决于具体设备的子组实现方式，但其目的是反映编译器目标子组大小与运行时子组大小之间的任何差异。例如，非常小的工作组可以包含比编译时子组大小更少的工作项，或者可以使用不同大小的子组来处理不能被子组大小整除的工作组和维度。

6.8 将计算映射到工作项

到目前为止，大多数代码示例都假设内核函数的每个实例对应于对单条数据的单个操作。这是一种编写内核的简单方法，但这种一对一的映射不是由 SYCL 或任何内核形式决定的——我们始终可以完全控制数据（和计算）分配给各个工作项，并且使该分配可参数化可以是提高性能可移植性的好方法。

6.8.1 一对一映射

当我们编写内核以实现工作到工作项的一对一映射时，这些内核必须始终使用范围或 nd_range 启动，其大小与需要完成的工作量完全匹配。这是编写内核的最明显的方法，在许多情况下，它工作得非常好——我们可以相信一个实现可以有效地将工作项映射到硬件。

然而，当调整系统和实现的特定组合的性能时，可能需要更加密切地关注低级调度行为。工作组对计算资源的调度是实现定义的，并且可能是动态的（即，当计算资源完成一个工作组时，它执行的下一个工作组可能来自共享队列）。动态调度对性能的影响并不是固定的，其重要性取决于内核函数每个实例的执行时间以及调度是在软件（例如在 CPU 上）还是硬件（例如在 CPU 上）中实现的因素。图形处理器）。

6.8.2 多对一映射

另一种方法是编写具有工作到工作项的多对一映射的内核。在这种情况下，范围的含义发生了微妙的变化：范围不再描述要完成的工作量，而是描述要使用的工人数量。通过更改工人数量和分配给每个工人的工作量，我们可以微调工作分配以最大限度地提高性能。

编写这种形式的内核需要进行两处更改：

1. 内核必须接受一个描述工作总量的参数。

2. 内核必须包含一个将工作分配给工作项的循环。

图 4-22 给出了此类内核的一个简单示例。请注意，内核内部的循环有一种稍微不寻常的形式 - 起始索引是工作项在全局范围内的索引，步幅是工作项的总数。这种数据到工作项的循环调度确保循环的所有 N 次迭代都将由工作项执行，而且线性工作项访问连续的内存位置（以改进缓存局部性和矢量化行为）。工作可以类似地跨组或各个组中的工作项进行分配，以进一

步改善局部性。

这些工作分配模式很常见，我们预计 SYCL 的未来版本将引入语法糖来简化 ND 范围内核中工作分配的表达。

6.9 选择内核形式

在不同的内核形式之间进行选择很大程度上取决于个人喜好，并且很大程度上受到其他并行编程模型和语言的先前经验的影响。

选择特定内核形式的另一个主要原因是它是公开内核所需的某些功能的唯一形式。不幸的是，在开发开始之前很难确定需要哪些功能，特别是当我们仍然不熟悉不同的内核形式及其与各种类的交互时。

我们根据自己的经验构建了两个指南来帮助我们驾驭这个复杂的空间。这些指南应被视为初步建议，绝对不是为了取代我们自己的实验 - 在不同内核形式之间进行选择的最佳方法始终是花一些时间编写每个内核形式，以便了解哪种形式是最好的适合我们的应用和开发风格。

第一个指南是图 4-23 所示的流程图，它根据以下条件选择内核形式：

1. 我们是否有并行编程的经验

2. 无论我们是从头开始编写新代码还是移植用不同语言编写的现有并行程序

3. 我们的内核是否是高度并行的，或者在内核函数的不同实例之间重用数据

4. 我们是否在 SYCL 中编写新内核是为了最大限度地提高性能、提高代码的可移植性，还是因为它提供了比低级语言更高效的表达并行性的方法

第二个指南是向每个内核形式公开的功能集。工作组、子组、组屏障、组本地内存、组函数（例如广播）和组算法（例如扫描、归约）仅适用于 ND-range 内核，因此我们应该更喜欢 NDrange 内核在我们有兴趣表达复杂算法或微调性能的情况下。

随着语言的发展，每种内核形式可用的功能预计会发生变化，但我们预计基本趋势保持不变：基本数据并行内核不会公开局部感知功能，显式 ND 范围内核将公开所有性能支持功能特征。

6.10 概括

本章介绍了使用 SYCL 在 C++ 中表达并行性的基础知识，并讨论了每种编写数据并行内核的方法的优点和缺点。

SYCL 提供对多种形式的并行性的支持，我们希望我们已经提供了足够的信息来帮助读者做好准备并开始编码!

我们只触及了表面，接下来将更深入地探讨本章中介绍的许多概念和类：第 9 章介绍了本地内存、屏障和通信例程的使用；除了使用 lambda 表达式之外定义内核的不同方法将在第 10 章和第 20 章中讨论；第 15、16 和 17 章探讨了 ND 范围执行模型到特定硬件的详细映射；第 14 章介绍了使用 SYCL 表达常见并行模式的最佳实践。

7 错误处理

错误处理是 C++ 的一项关键功能。本章讨论将工作卸载到设备（加速器）时遇到的独特错误处理挑战，以及 SYCL 如何使我们完全可以应对这些挑战。

检测和处理意外情况和错误在应用程序开发过程中很有帮助（想想：从事该项目的其他程序员确实会犯错误），但更重要的是在稳定和安全的生产应用程序和库中发挥关键作用。本章致力于描述 C++ 中可用的 SYCL 错误处理机制，以便我们能够了解我们的选项是什么，以及如果我们关心检测和管理错误，如何构建应用程序。

本章概述了 SYCL 中的同步和异步错误，描述了如果我们在代码中不执行任何操作来处理错误，应用程序的行为，并深入探讨允许我们处理异步错误的 SYCL 特定机制。

7.1 安全第一

C++ 错误处理的一个核心方面是，如果我们不采取任何措施来处理已检测到（引发）的错误，那么应用程序将终止并指示出现问题。这种行为使我们能够在编写应用程序时无需关注错误管理，并且仍然确信错误会以某种方式向开发人员或用户发出信号。当然，我们并不是建议我们应该忽略错误处理！生产应用程序的编写应将错误管理作为架构的核心部分，但应用程序在开始开发时通常没有这样的关注点。C++ 的目标是使不处理错误的代码仍然能够观察到许多错误，即使它们没有被显式处理。

由于 SYCL 是数据并行 C++，因此同样的理念成立：如果我们在代码中不采取任何措施来管理错误，并且检测到错误，则程序将发生异常终止，让我们知道发生了错误。生产应用程序当然应该将错误管理视为软件架构的核心部分，不仅要报告，而且通常还要从错误情况中恢复。

注 14 *如果我们不添加任何错误管理代码并且发生错误，我们仍然会看到程序异常终止，这表明需要进行更深入的研究。*

7.2 错误类型

C++ 通过其异常机制提供了一个用于通知和处理错误的框架。除此之外，异构编程还需要额外级别的错误管理，因为设备上或尝试在设备上启动

工作时会发生一些错误。这些错误通常与主机程序的执行及时分离，因此它们不能与常规 C++ 异常处理机制干净地集成。为了解决这个问题，有额外的机制可以使异步错误像典型的 C++ 异常一样易于管理和控制。

图 5-1 显示了典型应用程序的两个组件：(1) 主机代码按顺序运行并将工作提交到任务图以供将来执行；(2) 任务图与主机程序异步运行并执行内核或其他程序当满足必要的依赖性时对设备执行的操作。该示例显示了 parallel_for 作为任务图的一部分异步执行的操作，但其他操作也是可能的，并在第 3、4 和 8 章中讨论。

图 5-1 左右（主机和任务图）之间的区别是理解同步错误和异步错误之间差异的关键。

当主机程序执行操作（例如 API 调用或对象构造）时检测到错误条件时，就会发生同步错误。它们可以在图左侧的指令完成之前被检测到，并且导致错误的操作可以立即抛出错误。我们可以使用 try-catch 构造将特定指令包装在图的左侧，期望在 try 块结束之前检测到由于 try 内的操作而发生的错误（并因此捕获）。C++ 异常机制旨在准确处理这些类型的错误。

异步错误发生在图 5-1 右侧的部分，只有在执行任务图中的操作时才会检测到错误。当异步错误作为任务图执行的一部分被检测到时，主机程序通常已经继续执行，因此没有代码可以用 try-catch 结构包装来捕获这些错误。相反，SYCL 中有一个异步异常处理框架来处理这些相对于主机程序执行看似随机且不受控制的时间发生的错误。

7.3 让我们创建一些错误!

作为本章剩余部分的示例并允许我们进行实验，我们将在以下示例中创建同步和异步错误。

7.3.1 同步错误

在图 5-2 中，从缓冲区创建了一个子缓冲区，但其大小非法（大于原始缓冲区）。子缓冲区的构造函数检测到此错误并在构造函数执行完成之前抛出异常。这是一个同步错误，因为它是作为主机程序执行的一部分（同步）发生的。在构造函数返回之前可以检测到错误，因此可以在错误的起源点或在主机程序中检测到错误时立即对其进行处理。

我们的代码示例没有执行任何操作来捕获和处理 C++ 异常，因此默认的 C++ 未捕获异常处理程序为我们调用 std::terminate，表示出现了问题。

7.3.2 异步错误

生成异步错误有点棘手，因为实现会尽可能努力同步检测和报告错误。同步错误更容易调试，因为它们发生在主机程序中的特定起始点，因此只要有可能，它们都是实现的首选。出于演示目的，生成异步错误的一种方法是在主机任务内引发异常，该任务作为任务图的一部分异步执行。图 5-3 演示了此类异常。异步错误在许多情况下都可能发生并报告，因此请注意，图 5-3 中所示的主机任务示例只是一种可能性，而不是异步错误的要求。

7.4 应用程序错误处理策略

C++ 异常功能旨在将程序中检测到错误的点与可能处理错误的点清楚地分开，并且此概念非常适合 SYCL 中的同步错误和异步错误。通过抛出和捕获机制，可以定义处理程序的层次结构，这在生产应用程序中非常重要。

构建能够以一致且可靠的方式处理错误的应用程序需要预先制定策略以及为错误管理而构建的软件架构。C++ 提供了灵活的工具来实现许多替代策略，但这种架构超出了本章的范围。有许多书籍和其他参考文献专门讨论此主题，因此我们鼓励您查阅它们以全面了解 C++ 错误管理策略。

也就是说，错误检测和报告并不总是需要达到生产规模。如果目标只是在执行期间检测错误并报告错误（但不一定要从中恢复），则可以通过最少的代码可靠地检测和报告程序中的错误。以下各节首先介绍如果我们忽略错误处理并且不执行任何操作（默认行为并没有那么糟糕！）会发生什么，然后是在基本应用程序中易于实现的推荐错误报告。

7.4.1 忽略错误处理

C++ 和 SYCL 旨在告诉我们，即使我们没有显式处理错误，也会出现问题。未处理的同步或异步错误的默认结果是程序异常终止，操作系统应该告诉我们这一点。以下两个示例分别模拟了如果我们不处理同步错误和异步错误时将发生的行为。

图 5-4 显示了未处理的 C++ 异常的结果，例如，该异常可能是未处理的 SYCL 同步错误。我们可以使用此代码来测试特定操作系统在这种情况下会报告什么。

图 5-5 显示了调用 std::terminate 的示例输出，这将是我们的应用程序中未处理的 SYCL 异步错误的结果。我们可以使用此代码来测试特定操作

系统在这种情况下会报告什么。

尽管我们应该处理程序中的错误，但未捕获的异常最终会被捕获并终止程序，这比异常被默默地丢失要好!

7.4.2 同步错误处理

我们将本节保持得非常简短，因为 SYCL 同步错误只是 C++ 异常。SYCL 中添加的大多数附加错误机制与我们在下一节中介绍的异步错误相关，但同步错误很重要，因为实现尝试同步检测和报告尽可能多的错误，因为它们更容易推理和处理。

SYCL 定义的同步错误属于 sycl::exception 类型，它是一个从 std::exception 派生的类，它允许我们通过 try-catch 结构专门捕获 SYCL 错误，如图 5-6 所示。

在 C++ 错误处理机制之上，SYCL 为运行时抛出的异常添加了 sycl::exception 类型。其他一切都是标准 C++ 异常处理，因此大多数开发人员都会熟悉。

图 5-7 中提供了一个稍微更完整的示例，其中处理了其他类别的异常。

7.4.3 异步错误处理

异步错误由 SYCL 运行时（或底层后端）检测，并且错误的发生独立于主机程序中命令的执行。错误存储在 SYCL 运行时内部的列表中，并且仅在程序员可以控制的特定点释放以进行处理。我们需要讨论两个主题来讨论异步错误的处理:

1. 当处理未完成的异步错误时，处理程序应该做什么
2. 当异步处理程序被调用时

7.4.4 异步处理程序

异步处理程序是应用程序定义的函数，它注册到 SYCL 上下文和/或队列。在下一节定义的时间，如果有任何未处理的异步异常可供处理，则 SYCL 运行时将调用异步处理程序并传递这些异常的列表。

异步处理程序作为 std::function 传递到上下文或队列构造函数，并且可以根据我们的偏好以常规函数、lambda 表达式或函数对象等方式进行定义。该处理程序必须接受 sycl::exception_list 参数，如图 5-8 所示的示例处理程序所示。

在图 5-8 中，std::rethrow_exception 后跟特定异常类型的 catch 提供了异常类型的过滤，在本例中仅过滤 sycl::exception。我们还可以使用 C++ 中的替代过滤方法，或者只选择处理所有异常，无论其类型如何。

处理程序在构造时与队列或上下文相关联（第 6 章中详细介绍了低级细节）。例如，要将图 5-8 中定义的处理程序注册到我们正在创建的队列中，我们可以编写

queue my_queue gpu_selector_v, handle_async_error ;

同样，要将图 5-8 中定义的处理程序注册到我们正在创建的上下文中，我们可以编写

context my_contexthandle_async_error ;

大多数应用程序不需要显式创建或管理上下文（它们是在后台自动为我们创建的），因此如果要使用异步处理程序，大多数开发人员应该将此类处理程序与为特定设备构建的队列相关联（而不是明确的上下文）。

如果没有为队列或队列的父上下文定义异步处理程序，并且该队列（或上下文）发生必须处理的异步错误，则调用默认异步处理程序。默认处理程序的运行方式就好像它的编码如图 5-9 所示。

默认处理程序应向用户显示有关异常列表中任何错误的一些信息，然后通过 std::terminate 结束应用程序，这应导致操作系统报告终止异常。

我们在异步处理程序中放置的内容取决于我们。它的范围可以从记录错误到应用程序终止，再到恢复错误条件以便应用程序可以继续正常执行。常见情况是通过调用 sycl::exception::what() 报告可用错误的任何详细信息，然后终止应用程序。

虽然异步处理程序在内部做什么由我们决定，但一个常见的错误是打印一条错误消息（可能会在程序中的其他消息的噪音中错过），然后完成处理程序函数。除非我们制定了错误管理原则，允许我们恢复已知的程序状态并确信继续执行是安全的，否则我们应该考虑在异步处理程序函数中终止应用程序。这减少了检测到错误但无意中允许应用程序继续执行的程序出现错误结果的机会。在许多程序中，一旦我们检测到异步异常，异常终止是首选结果。

7.4.5 处理程序的调用

异步处理程序由运行时在特定时间调用。错误发生时不会立即报告，因为如果是这种情况，错误管理和安全应用程序编程（特别是多线程）将变得

更加困难和昂贵（例如，主机和设备之间的额外同步）。相反，异步处理程序会在以下特定时间被调用：

1. 当宿主程序对特定队列调用 queue::throw_asynchronous() 时
2. 当宿主程序对特定队列调用 queue::wait_and_throw() 时
3. 当主机程序对特定事件调用 event::wait_and_throw() 时
4. 当队列被销毁时
5. 当上下文被破坏时

方法 1-3 为主机程序提供了一种控制何时处理异步异常的机制，以便可以管理线程安全和特定于应用程序的其他细节。它们有效地提供了异步异常进入主机程序控制流的控制点，并且几乎可以像处理同步错误一样进行处理。

如果用户没有显式调用方法 1-3 之一，则在程序拆卸过程中，当队列和上下文被销毁时，通常会报告异步错误。这通常足以向用户发出信号，表明出现了问题并且程序结果不值得信任。

然而，依靠程序拆卸期间的错误检测并不适用于所有情况。例如，如果程序仅在达到某些算法收敛标准时才会终止，并且这些标准只能通过成功执行设备内核才能实现，则异步异常可能表明该算法永远不会收敛并开始拆卸（其中错误会被注意到）。在这些情况下，以及在制定了更完整的错误处理策略的生产应用程序中，在程序中的常规和受控点调用 throw_asynchronous() 或 wait_and_throw() 是有意义的（例如，在检查算法是否收敛之前）。

7.5 设备上的错误

本章讨论的错误检测和处理机制是基于主机的。它们是主机程序可以检测和处理主机程序中或在设备上执行内核期间可能出现问题的机制。我们没有讨论的是如何从我们编写的设备代码中发出信号，表明出现了问题。这种遗漏并不是错误，而是故意的。

SYCL 明确不允许在设备代码中使用 C++ 异常处理机制（例如 throw），因为某些类型的设备会产生我们通常不想支付的性能成本。如果我们检测到设备代码中出现问题，我们应该使用现有的非基于异常的技术来发出错误信号。例如，我们可以写入一个缓冲区来记录错误或从我们定义的数值计算中返回一些无效结果，这些结果意味着发生了错误。在这些情况下，正确的策略是非常具体的应用程序。

7.6 概括

在本章中，我们介绍了同步和异步错误，介绍了如果我们不采取任何措施来管理可能发生的错误时预期的默认行为，并介绍了用于在应用程序中的受控点处理异步错误的机制。错误管理策略是软件工程中的一个主要主题，并且在许多应用程序中编写的代码中占很大比例。SYCL 集成了我们在错误处理方面已有的 C++ 知识，并提供了灵活的机制来与我们首选的错误管理策略集成。

8 统一共享内存

接下来的两章将更深入地探讨如何管理数据。有两种不同的方法可以相互补充：统一共享内存 (USM) 和缓冲区。USM 公开了与缓冲区不同的内存抽象级别 - USM 使用指针，而缓冲区是更高级别的接口。本章重点介绍 USM。下一章将重点讨论缓冲区。

除非我们明确知道要使用缓冲区，否则 USM 是一个不错的起点。USM 是一种基于指针的模型，允许通过常规 C++ 指针读写内存。

8.1 为什么要使用 USM?

由于 USM 基于 C++ 指针，因此它是现有基于指针的 C++ 代码的自然起点。将指针作为参数的现有函数无需修改即可继续工作。在大多数情况下，唯一需要的更改是将现有的对 malloc 或 new 的调用替换为 USM 特定的分配例程，我们将在本章稍后讨论。

8.2 分配类型

虽然 USM 基于 C++ 指针，但并非所有指针都是一样的。USM 定义了三种不同类型的分配，每种类型都有独特的语义。

设备可能不支持所有类型（甚至任何类型）的 USM 分配。

稍后我们将学习如何查询设备支持的内容。图 6-1 总结了这三种类型的分配及其特点。

8.2.1 设备分配

为了获得指向设备附加内存（例如 (G)DDR 或 HBM）的指针，我们需要第一种类型的分配。设备分配可以由在特定设备上运行的内核读取或写入，但不能从主机上执行的代码直接访问它们（通常也不能由设备访问）。尝试访问主机上的设备分配可能会导致数据不正确或程序因错误而崩溃。我们必须使用显式 USM memcpy 机制在主机和设备之间复制数据，该机制指定必须在两个位置之间复制多少数据，这将在本章后面介绍。

8.2.2 主机分配

第二种类型的分配比设备分配更容易使用，因为我们不必在主机和设备之间手动复制数据。主机分配是主机内存中的分配，主机和设备均可访问。这些分配虽然可以在设备上访问，但无法迁移到设备的附加内存。相反，内核可以远程读取或写入该内存，通常通过 PCI Express 等较慢的总线（或者如果它是 CPU 设备或集成 GPU 设备，则实际上没有什么不同）。这种便利性和性能之间的权衡是我们必须考虑的。尽管主机分配可能会产生更高的访问成本，但仍然有充分的理由使用它们。示例包括很少访问的数据、无法放入设备附加内存的大型数据集，或者设备可能不支持共享分配等替代方案（如下所述）。

8.2.3 共享分配

最终类型的分配结合了设备和主机分配的属性，将主机分配的程序员便利性与设备分配提供的更高性能结合起来。与主机分配一样，共享分配可以在主机和设备上访问。它们之间的区别在于，共享分配可以自动在主机内存和设备附加内存之间自由迁移，无需我们干预。如果分配已迁移到设备，则在该设备上执行的任何访问该设备的内核都会比从主机远程访问该设备具有更高的性能。然而，共享分配并不能给我们带来所有好处而没有任何缺点。

自动迁移可以通过多种方式实现。无论运行时选择哪种方式实现共享分配，它们通常都会付出延迟增加的代价。通过设备分配，我们可以准确地知道需要复制多少内存，并可以安排复制尽早开始。自动迁移机制无法预见未来，并且在某些情况下，在内核尝试访问数据之前不会开始移动数据。然后，内核必须等待或阻塞，直到数据移动完成才能继续执行。在其他情况下，运行时可能无法确切知道内核将访问多少数据，并且可能会保守地移动比所需数量更大的数据，这也会增加内核的延迟。

我们还应该注意，虽然共享分配可以迁移，但这并不一定意味着 SYCL 的所有实现都会迁移它们。我们期望大多数实现通过迁移来实现共享分配，但某些设备可能更愿意以与主机分配相同的方式实现它们。在这样的实现中，分配在主机和设备上仍然可见，但我们可能看不到迁移实现可以提供的性能增益。

8.3 分配内存

USM 允许我们以各种不同的方式分配内存，以满足不同的需求和偏好。然而，在我们更详细地讨论所有方法之前，我们应该讨论 USM 分配与常规 C++ 分配有何不同。

8.3.1 我们需要知道什么?

常规 C++ 程序可以通过多种方式分配内存：new、malloc 或分配器。无论我们喜欢哪种语法，内存分配最终都是由主机操作系统中的系统分配器执行的。当我们在 C++ 中分配内存时，唯一关心的是“我们需要多少内存? ”和“有多少内存可供分配? ”然而，USM 在执行分配之前需要额外的信息。

首先，USM 分配需要指定所需的分配类型：设备、主机或共享。为了获得所需的行为，请求正确的分配类型非常重要。接下来，每个 USM 分配都必须指定一个将针对其进行分配的上下文对象。书中的大多数示例都传递队列对象（然后提供上下文）。到目前为止，上下文对象在本书中还没有进行太多讨论，因此值得在这里稍微讨论一下。上下文代表我们可以在其上执行内核的一个设备或一组设备。我们可以将上下文视为运行时存储有关其正在执行的操作的某些状态的方便位置。除了在大多数 SYCL 程序中传递上下文之外，程序员不太可能直接与上下文交互。我们确实在第 13 章中提供了一些有关上下文的提示。

不保证 USM 分配可在不同上下文中使用 - 所有 USM 分配、队列和内核共享相同的上下文对象非常重要。通常，我们可以从用于向设备提交工作的队列中获取此上下文。

最后，设备分配（以及一些共享分配）还要求我们指定哪个设备将为分配提供内存。这很重要，因为我们不想超额订阅设备的内存（除非设备能够支持这一点——我们将在本章后面讨论数据迁移时详细介绍这一点）。USM 分配例程可以通过添加这些额外参数来区别于它们的 C++ 类似例程。

8.3.2 多种风格

有时，试图用单一选项取悦所有人被证明是一项不可能完成的任务，就像有些人喜欢咖啡而不是茶，或者 emacs 而不是 vi 一样。如果我们问程序员分配接口应该是什么样子，我们会得到几个不同的答案。USM 拥抱这种

选择的多样性，并提供几种不同风格的分配接口。这些不同的风格是 C 风格、C++ 风格和 C++ 分配器风格。我们现在将讨论每一个并指出它们的相似点和不同点。

按 C 进行分配

第一种类型的分配函数（在图 6-2 中列出，稍后在图 6-6 和 6-7 所示的示例中使用）是根据 C 中的内存分配进行建模的：malloc 函数需要多个字节来分配并返回一个无效 * 指针。这种风格的函数与类型无关。我们必须指定要分配的总字节数，这意味着如果我们想要分配 N 个 X 类型的对象，则必须要求 N * sizeof(X) 总字节数。返回的指针是 void * 类型，这意味着我们必须将其转换为指向 X 类型的适当指针。这种样式非常简单，但由于所需的大小计算和类型转换，可能会很冗长。

我们可以将这种分配方式进一步分为两类：命名函数和单一函数。这两种风格之间的区别在于我们如何指定所需的 USM 分配类型。对于命名函数（malloc_device、malloc_host 和 malloc_shared），USM 分配的类型在函数名称中进行编码。单一函数 malloc 需要将 USM 分配的类型指定为附加参数。两种口味并不比另一种更好，选择哪种口味取决于我们的喜好。

如果不简要提及对齐，我们就无法继续前进。每个版本的 malloc 也有一个 aligned_alloc 对应项。malloc 函数返回与我们设备的默认行为一致的内存。成功时，它将返回一个具有有效对齐方式的合法指针，但在某些情况下，我们可能更愿意手动指定对齐方式。在这些情况下，我们应该使用 aligned_alloc 变体之一，它也要求我们指定所需的分配对齐方式。法律对齐是二的权力。值得注意的是，在许多设备上，分配最大程度地对齐以对应于硬件的功能，因此，虽然我们可能要求分配为 4、8、16 或 32 字节对齐，但实际上我们可能会看到更大的分配空间。对齐可以满足我们的要求，然后是一些。

C++ 的分配

USM 分配函数的下一个风格（图 6-3 中列出）与第一个风格非常相似，但更多的是 C++ 外观和感觉。我们再次拥有分配例程的命名版本和单函数版本，以及默认的和用户指定的对齐版本。不同之处在于，现在我们的函数是 C++ 模板函数，它分配 T 类型的 Count 对象并返回 T * 类型的指针。利用现代 C++ 可以简化事情，因为我们不再需要手动计算分配的总大小（以字节为单位）或将返回的指针转换为适当的类型。这也往往会在代码中产生更紧凑且不易出错的表达式。然而，我们应该注意到，与 C++ 中的

“new”不同，malloc 风格的接口不会为正在分配的对象调用构造函数——我们只是分配足够的字节来适合该类型。

对于考虑到 USM 编写的新代码来说，这种分配方式是一个很好的起点。对于已经大量使用 C 或 C++ malloc 的现有 C++ 代码，前面的 C 风格是一个很好的起点，我们将在其中添加 USM 的使用。

C++ 分配器

USM 分配的最终风格（图 6-4）甚至比以前的风格更多地拥抱现代 C++。这种风格基于 C++ 分配器接口，它定义了用于在容器（例如 std::vector）内直接或间接执行内存分配的对象。如果我们的代码大量使用容器对象，可以向用户隐藏内存分配和释放的详细信息，从而简化代码并减少出现错误的机会，则这种分配器风格非常有用。

8.3.3 释放内存

无论程序分配什么，最终都必须被释放。USM 定义了一种 free 方法来释放由 malloc 或 aligned_malloc 函数之一分配的内存。此 free 方法还将分配内存的上下文作为额外参数。队列也可以代替上下文。如果内存是使用 C++ 分配器对象分配的，则也应该使用该对象来释放内存。

8.3.4 分配示例

在图 6-5 中，我们展示了如何使用刚才描述的三种样式执行相同的分配。在此示例中，我们将 N 个单精度浮点数分配为共享分配。第一个分配 f1 使用 C 风格的 void * 返回 malloc 例程。对于此分配，我们显式传递从队列中获取的设备和上下文。我们还必须将结果转换回 float *。第二个分配 f2 执行相同的操作，但使用 C++ 样式模板化 malloc。由于我们将元素的类型 float 传递给分配例程，因此我们只需要指定要分配的浮点数量，并且不需要转换结果。我们还使用采用队列而不是设备和上下文的形式，产生一个非常简单和紧凑的语句。第三个分配 f3 使用 USM C++ 分配器类。我们实例化适当类型的分配器对象，然后使用该对象执行分配。最后，我们展示如何正确地释放每个分配。

8.4 数据管理

现在我们了解了如何使用 USM 分配内存，我们将讨论如何管理数据。我们可以将其分为两部分：数据初始化和数据移动。

8.4.1 初始化

数据初始化涉及在执行计算之前用值填充我们的内存。常见初始化模式的一个示例是在使用分配之前用零填充分配。如果我们要使用 USM 分配来做到这一点，我们可以通过多种方式来做到这一点。首先，我们可以编写一个内核来执行此操作。如果我们的数据集特别大或者初始化需要复杂的计算，这是一种合理的方法，因为初始化可以并行执行（并且它使初始化的数据准备好在设备上运行）。其次，我们可以将其实现为主机代码中对分配的所有元素进行循环，将每个元素设置为零。然而，这种方法可能存在一个问题。循环对于主机和共享分配来说效果很好，因为这些可以在主机上访问。但是，由于主机上无法访问设备分配，因此主机代码中的循环将无法写入它们。这给我们带来了第三种选择。

memset 函数旨在有效地实现此初始化模式。USM 提供了 memset 的一个版本，它是处理程序类和队列类的成员函数。它需要三个参数：表示我们要设置的内存基地址的指针、表示要设置的字节模式的字节值以及要设置到该模式的字节数。与主机上的循环不同，memset 并行发生，并且也适用于设备分配。

虽然 memset 是一个有用的操作，但它只允许我们指定一个字节模式来填充分配，这一事实是相当有限的。USM 还提供了一个 fill 方法（作为处理程序和队列类的成员），让我们可以用任意模式填充内存。fill 方法是一个以我们要写入分配的模式类型为模板的函数。使用 int 对其进行模板化，我们可以用 32 位整数“42”填充分配。与 memset 类似，fill 接受三个参数：指向要填充的分配基地址的指针、要填充的值以及我们想要将该值写入分配的次数。

8.4.2 数据移动

数据移动可能是 USM 需要理解的最重要的方面。如果正确的数据没有在正确的时间出现在正确的位置，我们的程序将产生错误的结果。USM 定义了两种可用于管理数据的策略：显式策略和隐式策略。我们想要使用哪种

策略的选择与我们的硬件支持或我们想要使用的 USM 分配类型有关。

显式的

USM 提供的第一个策略是显式数据移动（图 6-6）。

在这里，我们必须在主机和设备之间显式复制数据。我们可以通过调用处理程序类和队列类上的 memcpy 方法来做到这一点。memcpy 方法采用三个参数：指向目标内存的指针、指向源内存的指针以及要在主机和设备之间复制的字节数。我们不需要指定复制发生的方向——这隐含在源指针和目标指针中。

显式数据移动的最常见用法是在 USM 中的设备分配之间进行复制，因为它们在主机上无法访问。必须插入显式数据复制确实需要我们付出努力。此外，它可能是错误的来源：副本可能会被意外省略、复制的数据量可能不正确、或者源或目标指针可能不正确。

然而，显式数据移动不仅有缺点。它给我们带来了巨大的优势：完全控制数据移动。控制复制数据量和复制数据的时间对于在某些应用程序中实现最佳性能非常重要。理想情况下，我们可以尽可能将计算与数据移动重叠，确保硬件以高利用率运行。

其他类型的 USM 分配（主机分配和共享分配）都可以在主机和设备上访问，并且不需要显式复制到设备。这引出了 USM 中数据移动的另一种策略。

隐式的

USM 提供的第二种策略是隐式数据移动（示例用法如图 6-7 所示）。在这种策略中，数据移动是隐式发生的，也就是说，不需要我们的输入。通过隐式数据移动，我们不需要插入对 memcpy 的调用，因为我们可以在任何想要使用数据的地方通过 USM 指针直接访问数据。相反，系统的工作是确保数据在使用时在正确的位置可用。

对于主机分配，人们可能会争论它们是否真的会导致数据移动。由于根据定义，它们始终保留指向主机内存的指针，因此给定主机指针表示的内存无法存储在设备上。但是，当在设备上访问主机分配时，确实会发生数据移动。我们读取或写入的值不是通过适当的接口传入或传出内核，而是将内存迁移到设备。这对于数据不需要保留在设备上的流内核非常有用。

隐式数据移动主要涉及 USM 共享分配。这种类型的分配在主机和设备上都可以访问，更重要的是，可以在主机和设备之间迁移。关键点是，这种迁移只需访问不同位置的数据即可自动或隐式发生。接下来，我们将讨论共

享分配的数据迁移时需要考虑的几个问题。

迁移

通过显式数据移动，我们可以控制发生的数据移动量。通过隐式数据移动，系统可以为我们处理这个问题，但可能效率不高。SYCL 运行时不是预言机，它无法在应用程序执行操作之前预测将访问哪些数据。此外，指针分析对于编译器来说仍然是一个非常困难的问题，编译器可能无法准确地分析和识别内核内部可能使用的每个分配。因此，隐式数据移动机制的实现可能会根据支持 USM 的设备的功能做出不同的决策，这会影响共享分配的使用方式及其执行方式。

如果设备功能非常强大，它可能能够按需迁移内存。在这种情况下，数据移动将在主机或设备尝试访问当前不在所需位置的分配之后发生。按需数据极大地简化了编程，因为它提供了所需的语义，即 USM 共享指针可以在任何地方访问并且正常工作。如果设备不支持按需迁移（第 12 章解释了如何查询设备的功能），它可能仍然能够保证相同的语义，并对如何使用共享指针进行额外限制。

USM 共享分配的限制形式控制何时何地可以访问共享指针以及共享分配的大小。如果设备无法按需迁移内存，则意味着运行时必须保守，并假设内核可以访问其设备附加内存中的任何分配。这会带来一些后果。

首先，这意味着主机和设备不应尝试同时访问共享分配。应用程序应该分阶段交替访问。主机可以访问分配，然后内核可以使用该数据进行计算，最后主机可以读取结果。如果没有此限制，主机可以自由访问分配的不同部分，而不是内核当前正在访问的部分。这种并发访问通常发生在设备内存页面的粒度上。主机可以访问一个页面，而设备可以访问另一页面。以原子方式访问同一条数据将在第 19 章中介绍。程序员可以查询设备是否受到此限制，稍后我们将详细了解设备查询机制。

这种限制形式的共享分配的下一个后果是分配受到连接到设备的内存总量的限制。如果设备无法按需迁移内存，则它无法将数据迁移到主机以腾出空间来引入不同的数据。如果设备确实支持按需迁移，则可以超额订阅其连接的内存，从而允许内核计算比设备内存通常可以容纳的数据更多的数据，尽管这种灵活性可能会因额外的数据移动而带来性能损失。

细粒度控制

当设备支持共享分配的按需迁移时，在当前未驻留的位置访问内存后，会发生数据移动。但是，内核在等待数据移动完成时可能会停止。它执行的

下一条语句甚至可能会导致发生更多数据移动，并给内核执行带来额外的延迟。

SYCL 为我们提供了一种修改自动迁移机制性能的方法。它通过定义两个函数来实现这一点：prefetch 和 mem_advise。图 6-8 显示了每种方法的简单用法。这些函数让我们向运行时提供有关内核如何访问数据的提示，以便运行时可以选择在内核尝试访问数据之前开始移动数据。请注意，此示例使用队列快捷方式方法，直接在队列对象上调用 parallel_for，而不是在传递给 submit 方法（命令组）的 lambda 内部调用。

我们做到这一点的最简单方法是调用预取。该函数作为处理程序或队列类的成员函数进行调用，并采用基指针和字节数。这让我们可以通知运行时某些数据即将在设备上使用，以便它可以立即开始迁移它。理想情况下，我们会尽早发出这些预取提示，以便当内核接触数据时，它已经驻留在设备上，从而消除了我们之前描述的延迟。

SYCL 提供的另一个函数是 mem_advise。此函数允许我们提供有关如何在内核中使用内存的特定于设备的提示。我们可以指定的此类可能建议的一个示例是数据将仅在内核中读取，而不是写入。在这种情况下，系统可以意识到它可以复制设备上的数据，以便在内核完成后不需要更新主机的版本。但是，传递给 mem_advise 的建议特定于特定设备，因此在使用此函数之前请务必检查硬件文档。

8.5 查询

最后，并非所有设备都支持 USM 的所有功能。如果我们希望我们的程序可以跨不同设备移植，我们不应该假设所有 USM 功能都可用。USM 定义了一些我们可以查询的内容。这些查询可以分为两类：指针查询和设备能力查询。图 6-9 显示了每种方法的简单用法。

USM 中的指针查询回答两个问题。第一个问题是“这个指针指向什么类型的 USM 分配？”get_pointer_type 函数采用指针和 SYCL 上下文，并返回 usm::alloc 类型的结果，该结果可以有四个可能的值：主机、设备、共享或未知。第二个问题是“这个 USM 指针分配给什么设备？”我们可以将指针和上下文传递给函数 get_pointer_device 并获取设备对象。这主要用于设备或共享 USM 分配，因为它对于主机分配没有多大意义。SYCL 规范规定，当与主机分配一起使用时，将返回上下文中的第一个设备 - 除了避免引发异常之外，这没有任何特殊原因，这对于可能在 USM 分配上模板化的代码来

说似乎有点奇怪类型。

USM 提供的第二种类型的查询涉及设备的功能。USM 有自己的设备方面列表，可以通过调用设备对象上的 has 来查询。这些查询可用于测试设备支持哪些类型的 USM 分配。此外，我们可以查询主机和设备是否可以同时访问共享分配。完整的查询列表如图 6-10 所示。在第 12 章中，我们将更详细地了解查询机制。

8.6 还有一件事

我们还没有介绍另一种形式的 USM。我们在本章中描述的 USM 形式都需要使用特殊的分配函数。虽然不是一个巨大的负担，但这代表了传统 C++ 代码的变化，传统 C++ 代码使用 malloc 或 new 运算符形式的系统分配器。虽然当今的某些设备（例如 CPU）可能不需要此要求，但大多数加速器设备仍然需要它。因此，我们以更大的可移植性为名描述了如何使用 USM 分配函数。然而，我们相信我们很快就会看到更多支持使用系统分配器的加速器设计。此类设备将极大地简化程序，使程序员无需担心分配正确类型的 USM 内存或在适当的时间复制正确的数据。从某种意义上说，我们可以将最终的系统分配器支持视为 USM 的最终演进——它将提供共享 USM 分配的好处，而不需要使用特殊的分配函数。

8.7 概括

在本章中，我们描述了统一共享内存，这是一种基于指针的数据管理策略。我们介绍了 USM 定义的三种分配类型。我们讨论了使用 USM 分配和释放内存的所有不同方式，以及如何由我们（程序员）显式控制设备分配的数据移动或由系统隐式控制主机或共享分配的数据移动。最后，我们讨论了如何查询设备支持的不同 USM 能力以及如何在程序中查询 USM 指针信息。

由于我们还没有在本书中详细讨论同步，因此在后面的章节中，当我们讨论调度、通信和同步时，会有更多关于 USM 的内容。具体来说，我们在第 8、9 和 19 章中介绍了 USM 的这些额外注意事项。

在下一章中，我们将介绍数据管理的第二种策略：缓冲区。

9 Buffers

在本章中，我们将学习缓冲区抽象。我们在上一章中了解了统一共享内存（USM），这是一种基于指针的数据管理策略。USM 迫使我们思考内存存放在哪里以及什么应该在哪里访问。缓冲区抽象是一个更高级别的模型，它向程序员隐藏了这一点。缓冲区只是代表数据，运行时的工作就是管理数据在内存中的存储和移动方式。

本章介绍了管理数据的另一种方法。缓冲区和 USM 之间的选择通常取决于个人喜好和现有代码的风格，并且应用程序可以自由地混合和匹配这两种风格以表示应用程序中的不同数据。

USM 只是公开不同的内存抽象。USM 有指针，缓冲区是更高级别的抽象。缓冲区的抽象级别允许在应用程序内的任何设备上使用其中包含的数据，其中运行时管理使该数据可用所需的任何内容。USM 基于指针的模型可能更适合使用基于指针的数据结构（例如链表、树或其他结构）的应用程序。将缓冲区改造为已经使用指针的现有代码也可能更加棘手。但是，缓冲区保证可以在系统中的每个设备上工作，而某些设备可能不支持特定（或任何）USM 模式。选择很好，所以让我们深入研究缓冲区。

我们将更仔细地了解缓冲区是如何创建和使用的。如果不讨论访问器，对缓冲区的讨论就不完整。虽然缓冲区抽象了我们在程序中表示和存储数据的方式，但我们并不直接使用缓冲区访问数据。相反，我们使用访问器对象来通知运行时我们打算如何使用正在访问的数据，并且访问器与任务图中强大的数据依赖机制紧密耦合。在我们介绍了可以使用缓冲区执行的所有操作之后，我们还将探索如何在程序中创建和使用访问器。

9.1 缓冲器

缓冲区是数据的高级抽象。缓冲区不一定绑定到单个位置或虚拟内存地址。事实上，运行时可以自由地使用内存中的许多不同位置（甚至跨不同的设备）来表示缓冲区，但运行时必须确保始终为我们提供一致的数据视图。缓冲区可以在主机和任何设备上访问。

缓冲区类是一个具有三个模板参数的模板类，如图 7-1 所示。第一个模板参数是缓冲区将包含的对象的类型。此类型必须是设备可复制的，这扩展了 C++ 定义的普通可复制的概念。可简单复制的类型可以安全地逐字节复制，而无需使用任何特殊的复制或移动构造函数。设备可复制类型将此概念

递归地扩展到某些 C++ 类型，例如 std::pair 或 std::tuple。下一个模板参数是描述缓冲区维数的整数。最后的模板参数是可选的，默认值通常是使用的值。此参数指定一个 C++ 样式的分配器类，用于在主机上执行缓冲区所需的任何内存分配。首先，我们将研究创建缓冲区对象的多种方法。

9.1.1 缓冲区创建

9.1.2 我们可以用缓冲区做什么?

9.2 访问器

9.2.1 访问器创建

9.2.2 我们可以用访问器做什么?

9.3 概括