

Data Parallel C++

Mastering DPC++ for Programming of
Heterogeneous Systems using
C++ and SYCL

—
James Reinders
Ben Ashbaugh
James Brodman
Michael Kinsner
John Pennycook
Xinmin Tian



Apress
open

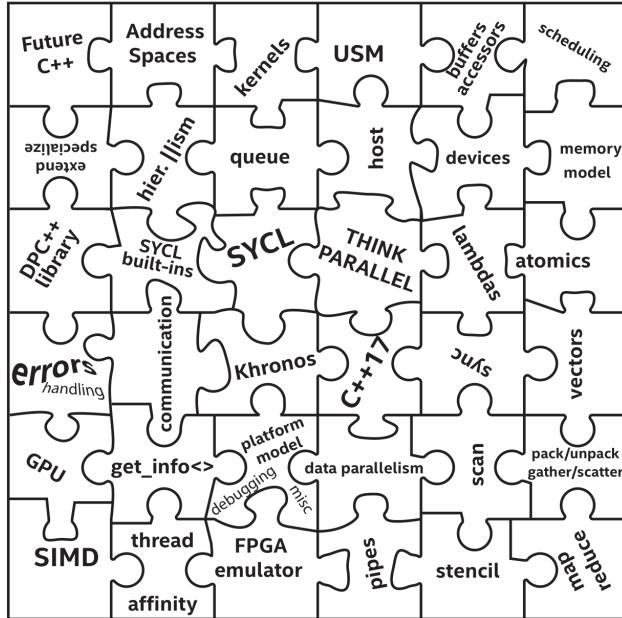
Data Parallel C++

Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

作者: James Reinders & Ben Ashbaugh & James Brodman & Michael Kinsner
John Pennycook & Xinmin Tian

译者: 陈晓伟

本书概述



本书是关于使用 C++ 编写数据并行程序的。如果你是并行编程的新手，也没关系。如果从未听说过 SYCL 或 DPC++ 编译器，也没有关系。

SYCL 是一个行业驱动的 Khronos 标准，在异构系统为 C++ 中添加原生的数据并行性。DPC++ 是一个开源编译器项目，它基于 SYCL、编译器扩展和异构支持组成，其中包括 GPU、CPU 和 FPGA 支持。本书中的所有例子都是用 DPC++ 编译器编译的。

如果你是一个不精通 C++ 的 C 开发者，不用太担心。本书的几位作者会告诉你，他们是通过阅读使用 C++ 的书籍来学习 C++ 的，就像这本书一样。只要有一点耐心，这本书对于想编写现代 C++ 代码的 C 开发者来说应该很容易。

本书项目始于 2019 年，对于完全支持 C++ 和数据并行的需要大量的扩展，超出当时的 SYCL 1.2.1 标准。DPC++ 编译器需要支持这些扩展，包括对统一共享内存 (USM) 的支持、通过 SYCL 完成三级层次结构的子工作组、匿名 Lambda 和许多编程简化。

本书出版的时候 (2020 年末)，会有一个临时的 SYCL 2020 规范可供阅览。临时规范包括对 USM、子工作组、匿名 Lambda 的支持，以及对编码的简化 (类似于 C++17 CTAD)。可以通过本书中 SYCL 的扩展，以大致了解 SYCL 的发展方向，并且这些扩展都会在 DPC++ 编译器项目中实现。我们希望与本书的内容相比，SYCL 的变化不会太大。但随着社区的发展，SYCL 将会有一些变化。更新

信息的重要资源包括本书 GitHub 和勘误表，可以从本书的网页 (www.apress.com/9781484255735) 找到，以及 oneAPI DPC++ 参考 (tinyurl.com/dpcppref)。

SYCL 和 DPC++ 的发展仍在继续。在学习了如何使用 DPC++ 为使用 SYCL 的异构系统创建程序之后，会在之后讨论对未来的展望。

希望我们的书能够支持和帮助 SYCL 社区的发展，并帮助推广 C++ 中的数据并行编程。

作者简介

James Reinders 是并行计算领域有 30 多年经验的专家，参与编纂了十余本与并行编程相关的技术书籍，以及为世界上最快的两台计算机（500 强中排名第一）以及许多其他超级计算机和软件开发工具做出了重要贡献。2016 年中期结束在 Intel 的任期（已经在 Intel 工作了 10,001 天（超过 27 年）），不过还继续在并行计算（高性能计算和人工智能）相关的领域进行写作、教学和编程。

Ben Ashbaugh 是 Intel 公司的软件架构师，他工作了 20 多年，为 Intel 图形产品开发软件驱动程序。过去的 10 年里，Ben 专注于并行编程模型，用于图形处理器上的通用计算，包括 SYCL 和 DPC++。Ben 活跃于 Khronos SYCL、OpenCL 和 SPIR 工作组，帮助定义并行编程的行业标准，也编写了许多扩展来展示 Intel GPU 的魅力。

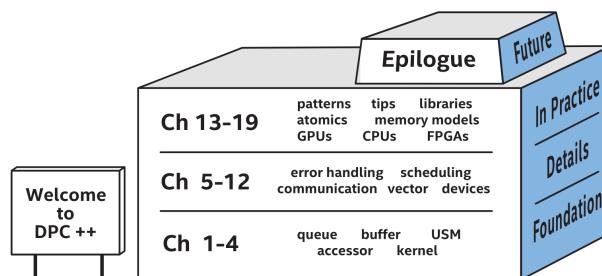
James Brodman 是 Intel 公司的软件工程师，专注于并行编程的运行时和编译器开发，并且是 DPC++ 的架构师之一。他拥有伊利诺伊大学厄巴纳-香槟分校的计算机博士学位。

Michael Kinsner 是 Intel 公司的首席工程师，为各种架构开发并行编程语言和模型，也是 DPC++ 的架构师之一。他对空间编程模型和编译器做出了重要的贡献，是 Khronos 组织中的 Intel 代表，他致力于制定 SYCL 和 OpenCL 并行编程行业标准。Mike 拥有麦克马斯特大学 (McMaster University) 的计算机工程博士学位，并且热衷于编写跨架构的编程模型（同时能够保证性能）。

John Pennycook 是 Intel 公司的一名 HPC 应用工程师，专注于让开发人员充分利用现代处理器中的并行性。他在一系列科学领域的应用程序优化和并行方面有丰富的经验，此前曾担任 Intel 极端性能用户组 (IXPUG) 指导委员会的代表。John 拥有华威大学计算机科学博士学位。他的研究点很多，主要在于跨不同硬件架构实现应用“性能可移植性”的能力。

Xinmin Tian 是 Intel 公司高级首席工程师和编译架构师，在 OpenMP 架构审查委员会 (ARB) 担任 Intel 代表。他负责为 Intel 架构驱动对 OpenMP 进行装载、向量化和并行化编译器技术。他目前的重点是基于 llvm 的 OpenMP 装载，使用 oneAPI 工具包的 DPC++ 编译器对 CPU 和 Xe 加速器进行优化，以及优化 HPC/AI 应用程序性能。他拥有计算机科学博士学位，拥有 27 项美国专利，发表了 60 多篇技术论文，被 1200 多次引用，并在其专业领域与他人合著了两本书。

章节架构



一下子解释一切是很困难的。因此，本书是一个旅程，通过需要知道什么，了解如何成为高效的数据并行 C++ 开发人员。

第 1 章通过介绍核心概念奠定基础，这些核心概念要么是新的，要么是需要更新的。

第 2-4 章可以理解为为 C++ 的数据并行编程奠定了基础。读完第 1-4 章后，我们将为 C++ 中的数据并行编程打下良好的基础。第 1-4 章是建立在彼此的基础上，最好按顺序阅读。

第 5-19 章在一定程度上相补了一些细节，同时也很容易在需要时进行切换。本书以结语结束，讨论了 C++ 数据并行未来可能的方向。

希望在学习使用 SYCL 和 DPC++ 时一切顺利！

本书相关

- github 翻译地址: <https://github.com/xiaoweiChen/Data-Paralle-Cpp>
- 英文原版 PDF: <https://www.apress.com/gp/book/9781484255735>
- 相关教程: <https://github.com/jeffhammond/dpcpp-tutorial>

目录

1 介绍	12
本书不是说明书	12
SYCL 1.2.1 vs SYCL 2020 和 DPC++	12
获取 DPC++ 编译器	13
本书代码库	13
Hello, World! SYCL 程序解析	14
队列和行动	14
关于并行	15
DPC++ 和 SYCL 的关键属性	17
并发和并行	22
总结	22
2 代码执行的位置	23
单个源文件	23
选择设备	25
方法 1: 在任何类型的设备上运行	25
方法 2: 使用主机设备进行开发和调试	29
方法 3: 使用 GPU(或其他加速器)	30
方法 4: 使用多个设备	33
方法 5: 选择自定义 (非常具体的) 设备	34
在 CPU 上执行设备端代码的三种方式	35
在设备上创建任务	36
总结	43
3 数据管理	44
介绍	44
数据管理的问题	44
本地设备和远程设备	45
管理多种内存	45
统一共享内存、内存和图像	46
统一共享内存	46
内存	49
对数据进行排序	51
选择管理策略	58
句柄类: 关键成员	59
总结	61

4 并发表示	62
内核间的并行	62
语言的特性	64
内核间的数据并行	65
显式 ND-Range 内核	69
内核间的分层并行	76
将计算映射到工作项中	79
选择内核形式	80
总结	81
5 错误处理	83
安全第一	83
错误类型	83
创建一些错误	84
错误的处理策略	86
设备上出现错误	91
总结	91
6 统一共享内存	92
为什么要使用统一共享内存	92
分配类型	92
分配内存	93
数据管理	98
查询	102
总结	103
7 内存	105
介绍	105
访存器	110
总结	114
8 调度内核和数据移动	116
什么是图调度?	116
DPC++ 中如何操作图	116
数据传送	122
与主机同步	123
总结	124
9 通信和同步	125
工作组和工作项	125
有效通讯的基础	126

使用工作组栅栏和本地内存	128
子工作组	134
通用功能	137
总结	140
10 定义内核函数	142
为什么用三种方法来表示内核函数?	142
使用 Lambda 表示内核函数	143
使用命名的函数对象表示内核函数	145
与其他 API 的互动性	147
程序对象中的内核函数	149
总结	150
11 向量	151
如何使用向量的方式思考	151
向量的类型	152
向量的接口	153
在并行内核中使用向量	157
向量并行	159
总结	159
12 设备信息	161
优化内核代码	161
如何枚举设备和功能	162
设备信息描述符	167
特定于设备的内核信息描述符	167
正确性	167
调优/优化	169
运行时和编译时属性	169
总结	170
13 实践技巧	171
获取 DPC++ 编译器和代码示例	171
在线论坛和文档	171
平台模型	171
将 SYCL 添加到现有 C++ 代码中	174
调试	175
初始化数据并访问内核输出	179
多个转译单元	184
需要为匿名的 Lambda 函数命名	185
将 CUDA 迁移到 SYCL	185

总结	185
14 常见的并行模式	187
了解模式	187
使用内置函数和库	192
直接编程	196
总结	203
15 GPU 编程	204
性能说明	204
GPU 的工作原理	204
将内核函数加载到 GPU	213
GPU 内核最佳实践	216
总结	221
16 CPU 编程	222
性能说明	222
通用 CPU 的基础知识	222
SIMD 硬件的基础知识	224
利用线程级别的并行性	227
CPU 上的 SIMD 向量化	232
总结	238
17 FPGA 编程	239
性能说明	239
FPGA 的工作原理	239
何时使用 FPGA	244
在 FPGA 上运行程序	246
为 FPGA 编写内核函数	250
一些相关的话题	266
总结	267
18 库	268
内置函数	268
DPC++ 库	273
总结	281
19 内存模型和原子操作	282
内存模型中有什么?	282
内存模型	288
实际中的原子操作	297
总结	300

20 结语：DPC++ 的未来方向	302
与 C++20 和 C++23 对齐	302
地址空间	303
扩展与更特化的机制	304
分层并行	305
总结	306

致谢

我们的贡献都是在他人工作的基础上取得新成就的，如同牛顿把他的成功归功于“站在巨人的肩膀上”一样。

写一本关于 SYCL 和 DPC++ 的开发新书不容易。幸运的是，一些人让这条路变得容易——在此感谢他们的帮助！

我们对为本书的出版做出贡献的所有人深表感谢。如果下面忘记了提您的名字，希望您能感受到我们的感激之情，也请接受我们的歉意。

部分人阅读了我们的早期手稿，并提供了深刻的反馈，我们非常感谢。他们是 Jefferson Amstutz, Thomas Applencourt, Alexey Bader, Gordon Brown, Konstantin Bobrovsky, Robert Cohn, Jessica Davies, Tom Deakin, Abhishek Deshmukh, Bill Dieter, Max Domeika, Todd Erdner, John Freeman, Joe Garvey, Nithin George, Milind Girkar, Sunny Gogar, Jeff Hammond, Tommy Hoffner, Zheming Jin, Paul Jurczak, Audrey Kertesz, Evgueny Khartchenko, Jeongnim Kim, Rakshith Krishnappa, Goutham Kalikrishna Reddy Kuncham, Victor Lomüller, Susan Meredith, Paul Petersen, Felipe De Azevedo Piovezan, Ruyman Reyes, Jason Sewall, Byron Sinclair, Philippe Thierry 和 Peter uek。

感谢 Intel 的开发团队，他们创建了 DPC++，包括它的库和文档，没有他们的工作，本书就不可能完成。

Khronos SYCL 工作组和 Codeplay 是我们的巨人。我们的共同目标是为 C++ 带来有效和可用的数据并行。感谢所有参与 SYCL 规范开发的人们，他们孜孜不倦地为整个行业提出了一个开放的标准。SYCL 团队一直忠于自己的使命，并保持这个标准的开放。我们也非常欣赏 Codeplay 的开拓性工作，在 DPC++ 还没有出现的时候，他们就已经开始推广和支持 SYCL 了。所以，他们仍然是整个社区的重要资源。

Intel 内部有很多人对 DPC++ 和 SYCL 做出了巨大的贡献。我们感谢所有人的工作，无论是在语言部分和 API 部分，还是在原型、编译器、库和工具的实现方面。虽然无法记得每个人的名字，但要特别感谢一些对 DPC++ 和 SYCL 做出变革性贡献的架构师：Roland Schulz, Alexey Bader, Jason Sewall, Alex Wells, Ilya Burylov, Greg Lueck, Alexey Kukanov, Ruslan Arutyunyan, Jeff Hammond, Erich Keane 和 Konstantin Bobrovsky。

感谢 DPC++ 用户社区的耐心和奉献。美国阿贡国家实验室的开发人员在我们与 DPC++ 的旅程中给予了极大的支持。

合著的队友们，我们对彼此感激不尽。我们在 2019 年初聚在一起，愿景是写一本书来教授 SYCL 和 DPC++。在接下来的一年里，我们成为了一个团队，并学会了如何一起教学。当然，我们也面临着来自许多方面的挑战，这些挑战把我们从写书和讨论的工作中拉出来，其中包括一些产品的最后期限和标准工作，同时 COVID-19 也席卷了全球。我们必须承认，“待在家里”的时光让我们有更多的注意力在这本书上。我们向所有受这一全球流行病影响的人表示慰问和祈祷。

James Reinders: 我要感谢 Jefferson Amstutz 对 C++ 并行性的看法，以及提供一些非常有用的帮助，通过使用 Jefferson 的超级功能作为 C++ 编译器错误消息耳语器可以直接获取一些 C++ 代码。感谢我的妻子 Susan Meredith，感谢她的爱、支持、建议和评论。并感谢 Intel 让我参与这个项目！非常感谢其他合著者对这个雄心勃勃的项目的耐心（对我）和辛勤工作。

Ben Ashbaugh: 很感激我妻子 Brenna 和儿子 Brenna 对我的支持和鼓励。谢谢你们给我不

受干扰的写作时间，也谢谢你在我需要休息的时候找借口出去走走或玩游戏！对于 Khronos SYCL 和 OpenCL 工作组的每一个人，感谢你们的讨论、合作和启发。DPC++ 和这本书你们也功不可没。

James Brodman: 感谢家人和朋友给我的支持。感谢在 Intel 和 Khronos 的所有同事。

Michael Kinsner: 我感谢我的妻子 Jasmine，以及我的孩子 Winston 和 Tilly，感谢他们在编写本书期间以及整个 DPC++ 项目过程中的支持。这两件事都需要大量的时间和精力，如果没有你们的支持，我也无法完成这两件事。也要感谢 Intel 和 Khronos 的许多人，他们为 SYCL 和 DPC++ 投入了大量的精力和时间，是所有人造就了 SYCL、OpenCL 和 DPC++，并参与了无数的讨论和实验，这些讨论和实验促成了 DPC++ 和这本书的诞生。

John Pennycook: 非常感谢我的妻子 Louise，感谢她的耐心、理解和支持，在写书的同时照顾我们刚出生的女儿 Tamsyn。也要感谢 Roland Schulz 和 Jason Sewall 在 DPC++ 所做的工作，以及对于“C++ 编译器错误”方面的帮助！

Xinmin Tian: 感谢 Alice S. Chan 和 Geoff Lowney，感谢他们在本书的编写过程和整个 DPC++ 性能工作过程中的大力支持。衷心感谢 Guei-Yuan Lueh, Konstantin Bobrovsky, Hideki Saito, Kaiyu Chen, Mikhail Loenko, Silvia Linares, Pavel Chupin, Oleg Maslov, Sergey Maslov, Vlad Romanov, Alexey Sotkin, Alexey Sachkov，以及整个 DPC++ 编译器、运行时和工具团队，感谢他们在创建 DPC++ 编译器和工具方面做出的巨大贡献和辛勤工作。

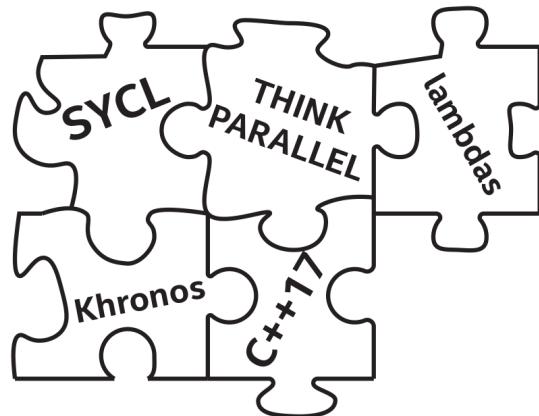
感谢整个记者团队的努力工作，包括与我们的合作人：Natalie Pao, Jessica Vakili, C Dulcy Nirmala 和 Krishnan Sathyamurthy。

我们有幸得到了经理们的支持和鼓励，包括 Herb Hinstorff, Bill Savage, Alice S. Chan, Victor Lee, Ann Bynum, John Kreatsoulas, Geoff Lowney, Zack Waters, Sanjiv Shah, John Freeman 和 Kevin Stevens。

许多同事提供了信息、建议和愿景。我们确信，有很多我们没有提到的人，也对本书项目产生了积极的影响。我们感谢所有为我们的图书计划提供帮助的人。我们向所有帮助过我们，但在这里没有提及的人道歉。

谢谢大家，希望你们会发现，在这本书上的努力是值得的。

1 介绍



本章通过介绍核心概念(包括术语)来奠定基础,这些概念在学习 C++ 数据并行加速时非常重要。

C++ 中的数据并行性,允许现代异构系统并行访问资源。C++ 应用程序可以使用任何设备的组合——包括 GPU、CPU、FPGA 和 AI 专用硬件(ASIC)——以便解决当前的问题。

在这本书中会学到如何使用 C++ 和 SYCL 进行数据并行编程。

SYCL(发音为 sickle)是一个行业驱动的 Khronos 标准,它为异构系统在 C++ 中增加了数据并行性。SYCL 程序与支持 SYCL 的 C++ 编译器(如本书中使用的开源数据并行 C++(DPC++)编译器)一起使用时,性能最好。SYCL 不是首字母缩写,SYCL 只是一个名字。

DPC++ 是一个开源编译器项目,最初由 Intel 创建,致力于支持 C++ 中的数据并行。DPC++ 编译器基于 SYCL、一些扩展和异构支持,包括 GPU、CPU 和 FPGA 设备。除了 DPC++ 的开源版本,还有 Intel oneAPI 工具包中的商业版本。

DPC++ 编译器的开源版本和商业版本都支持基于 SYCL 的实现特性。本书中的所有例子都可以使用 DPC++ 编译器的任何一个版本进行编译,而且所有的例子都可以使用最新的 SYCL 编译器进行编译。发布时,我们会特别注意使用 DPC++ 扩展的地方。

本书不是说明书

没有人喜欢听到“去阅读说明书吧!”的建议吧。规范很难阅读,SYCL 规范也不例外。就像所有伟大的语言规范一样,它在动机、用法和教学方面都具有足够的准确性。这本书可以作为 SYCL 和使用 DPC++ 编译器的“学习指南”。

正如前言中提到的,这本书不能把一切都解释清楚。因此,本章做了其他章节不会做的事情:代码示例包含的编程结构直到以后的章节再进行解释。我们试着不去完全理解第 1 章的代码示例,相信每一章都会让你更加的了解 SYCL。

SYCL 1.2.1 vs SYCL 2020 和 DPC++

本书出版时,临时的 SYCL 2020 规范可供公众阅览。随着时间的推移,SYCL 1.2.1 标准会有一个后续版本,预计会称为 SYCL 2020。虽然很高兴地说,这本书介绍的是 SYCL 2020,但这个标准目前还不存在。

本书介绍了 SYCL 的扩展，以大致了解 SYCL 将来的发展方向。这些扩展在 DPC++ 编译器项目中都有实现。几乎在 DPC++ 中实现的扩展都是暂定为 SYCL 2020 规范中的新特性。DPC++ 支持的新特性包括 USM、子工作组、C++17 支持的语法简化（称为 CTAD——class 模板参数推导），以及无需命名就可以使用匿名 Lambda 函数。

发布时，SYCL 编译器（包括 DPC++）没有实现 SYCL 2020 临时规范中的功能。

本书中使用的一些特性是 DPC++ 编译器特有的。这些特性中有许多是 Intel 对 SYCL 的扩展，后来被 SYCL 2020 临时规范所接受，其语法在标准化过程中发生了细微的变化。其他特性仍在开发或讨论中，可能会包含在未来的 SYCL 标准中，其语法也可能也会修改。语言开发过程中，非常需要这样的语法变化实际上，我们希望特性能够进化和改进，从而满足更广泛的开发群体和更广泛的功能需求。本书中的所有代码示例都使用了 DPC++ 语法，以确保与 DPC++ 编译器的兼容性。

在努力接近 SYCL 的发展方向的同时，需要对本书中的信息进行调整，以便与标准的发展保持一致。更新信息的重要资源包括本书 GitHub 和勘误表，可以从本书的网页 (www.apress.com/9781484255735) 找到，以及在线 oneAPI DPC++ 语言参考手册 (tinyurl.com/dpcppref)。

获取 DPC++ 编译器

DPC++ 可以从 GitHub 存储库 (github.com/intel/llvm) 获得。开始使用 DPC++ 指令，包括如何使用 GitHub 上的克隆编译器，可以在 intel.github.io/llvm-docs/GetStartedGuide.html 中找到。

也有打包版本的 DPC++ 编译器，增强了工具和库的 DPC++ 编程和支持，并作为一个 oneAPI 项目的一部分。该项目为异构系统带来了更广泛的支持，其中包括库、调试器和其他工具，称为 oneAPI。oneAPI 工具，包括 DPC++，可以免费获得 (oneapi.com/implementations)。官方的 oneAPI DPC++ 编译器文档，包括扩展列表，可以在 intel.github.io/llvirtual-docs 中找到。

本书的在线手册，[oneAPI DPC++ 语言在线手册](#)是很好的资源，对于本书介绍的内容会有更加详细的介绍。

本书代码库

下面的代码是最简单的应用。如果不手动输入代码，可以从 GitHub 库 (www.apress.com/9781484255735——查找本书的服务：源代码) 下载本书中的所有示例。库中包含了构建文件中的完整代码，大多数代码都省略了一些重复的或不必要的细节。库中有示例的最新版本，如果有更新，您就能获取到最新的例程。

图 1-1 简单的数据并行编程

```
1 #include <CL/sycl.hpp>
2 #include <iostream>
3 using namespace sycl;
4
5 const std::string secret {
6     "Ifmmp-!xpsme\"\\012J(n!tpssz-!Ebwf/!"
7     "J(n!bgsbje!J!dbo(u!ep!uibu/.!IBM\\01");
```

```

8 const auto sz = secret.size();
9
10 int main() {
11     queue Q;
12
13     char* result = malloc_shared<char>(sz, Q);
14     std::memcpy(result, secret.data(), sz);
15
16     Q.parallel_for(sz, [=](auto&i) {
17         result[i] -= 1;
18     }).wait();
19
20     std::cout << result << "\n";
21     return 0;
22 }

```

Hello, World! SYCL 程序解析

上面展示了 SYCL 的一个例程。使用 DPC++ 编译器编译，并运行结果如下：

Hello, world! (and some additional text left to experience by running it)

第 4 章结束时，我们就能完全理解这个例子了。在此之前，我们可以看到包含了 `<CL/sycl.hpp>`(第 1 行)，这是定义所有 SYCL 组件所需要的。所有的 SYCL 组件都存在名为 SYCL 的命名空间中：

- 第 3 行让我们避免反复写 `sycl::`。
- 第 11 行为指向特定设备的工作建立了一个队列 (第 2 章)。
- 第 13 行创建了设备共享的数据 (第 3 章)。
- 第 16 行将相应的工作加入到设备的工作队列中 (第 4 章)。
- 第 17 行是唯一在设备上运行的代码。所有其他代码都在主机 (CPU) 上运行。

第 17 行是我们想要在设备上运行的内核代码。该内核代码的功能是去掉一个字符。这里使用了 `parallel_for()`，内核对 `secret` 字符串中的每个字符上逐个进行处理，以便将其解码为 `result` 字符串。需要完成的工作没有顺序要求，并且当 `parallel_for` 将工作入队后，实际上是相对于主程序异步运行的。关键的等待 (第 18 行) 可以让我们确定，内核已经执行完成，因为在这个特定的示例中，我们使用一个特性 (统一共享内存，第 6 章)。如果不等待，`result` 字符串上可能会有还未解密的字符。还有很多细节需要讨论，我们会在之后的章节继续。

队列和行动

第 2 章将讨论队列和操作，但是我们可以从一个简单的开始了解。队列是允许应用程序在设备上完成工作的方式。有两种类型的操作可以放在队列中：(a) 要执行的代码和 (b) 内存操作。执行代码通过 `single_task`、`parallel_for` 或 `parallel_for_work_group` 表示。内存操作用来对主机与设备之间的复制或填充，从而初始化内存。只有需要做的更多的控制时，才需要使用内存操作。这些都会从本书第 2 章开始讨论。现在，我们知道到了队列是连接命令与设备的桥梁，我们有一组操作可

以放入队列来执行代码和/或移动数据。同样重要的是，要理解请求的操作放置在队列中而不需要等待。在将操作提交到队列后，主机继续执行程序，而设备最终将以异步执行的方式执行队列请求的操作。

队列将连接到设备。我们将操作提交到这些队列中，从而进行计算或数据移动。行为是异步发生的。

关于并行

在 C++ 中为数据并行编程与并行相关，并行编程的目标是更快地计算。结果表明，这有两个方面优点：增加吞吐量和减少延迟。

吞吐量

当在规定的时间内完成更多的工作时，吞吐量就会增加。像流水线这样的技术实际上可能会延长完成单个工作项目所需的时间，允许工作的重叠，从而让更多的工作在单位时间内完成。人们在一起工作时经常遇到这种情况，共享就涉及到协调的开销，这往往会减慢完成工作的时间。然而，多人的力量会带来更多的吞吐量。计算机也不例外——将工作扩展到更多的处理核芯中，会给每个工作单元增加开销，这可能会导致一些延迟，因为有更多的处理核芯可以一起工作，所以可以完成更多的工作。

延迟

如果想要更快地完成一件事，例如：分析语音命令并形成的反应，该怎么办？如果只关心吞吐量，那么响应时间可能会增加到无法忍受的程度。减少延迟要求我们将一项工作分解成可以并行处理的模式。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——这种情况下，目标可能是优化每秒的图像数。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芯——目标是最大化处理单个图像的每秒像素。

并行思维

并行开发者在编程中，会使用这两种技术。

想要调整思维，首先思考在算法和应用中，并行在哪里可行。还要考虑表达并行性的不同方式，以及如何影响最终性能。这么多东西需要去理解，所以并行思维是并行开发者一生的追求。这里可以学习一些技巧。

Amdahl 和 Gustafson

在 1967 年，由超级计算机先驱吉恩·阿姆达尔 (Gene Amdahl) 提出的阿姆达尔定律 (Amdahl's Law) 是一个公式，用来预测使用多处理器时理论上的最大速度。Amdahl 遗憾地说，并行的最大收益被限制在 $\frac{1}{1+p}$ ，其中 p 是并行运行的程序的部分。如果只并行运行程序的三分之二，那么最多可以加速 3 倍。当然要深入理解这个概念！这是因为不管让三分之二的程序以多快的速度运行，其他三分之一仍然需要相同的时间来完成。即使增加 100 个 GPU，也只能得到 3 倍的性能提升。

许多年来，有些人认为这证明了并行计算的不可行性。1988 年，John Gustafson 发表了一篇题

为“重新评估 Amdahl 定律”的文章。他注意到并行不是用来加速定量的工作，而是用来扩展的。大家也有同样的生活经历，1 个快递员无法在更多的人和卡车的帮助下更快地递送 1 个包裹。然而，100 个人和 100 辆卡车运送 100 件包裹比 1 个司机开着 1 辆卡车要快得多。多个司机肯定会增加吞吐量，通常也会减少交付的延迟。Amdahl 定律告诉我们，1 个司机不能通过增加 99 个司机，用 1 辆卡车更快地递送 1 个包裹。Gustafson 注意到，有了这些额外的司机和卡车，可以更快地递送 100 个包裹。

可扩展性

“可扩展性”一词在之前的讨论中出现过。可扩展性是衡量附加计算可用时程序加速的程度（简单地称为“加速”）。如果 100 个包裹与 1 个包裹同时运送，只需用 100 辆卡车和司机（而不是 1 辆卡车和 1 个司机），就可以实现加速。当然，存在限制加速的瓶颈。配送中心可能没有 100 个地方可以让卡车停靠。计算机程序中，瓶颈经常涉及到将数据移到将要处理的地方。分发到 100 个卡车类似于必须将数据分发到 100 个处理核心，且分发需要时间。第 3 章会探索如何将数据分发到异构系统需要的地方。数据分发是有成本的，而该成本会影响对应用可扩展性的预期。

异构系统

就我们的目的而言，异构系统是指包含多种类型计算设备的系统。例如：同时具有中央处理单元 (CPU) 和图形处理单元 (GPU) 的系统是一个异构系统。CPU 通常称为处理器，尽管我们将异构系统中的所有处理单元称为计算处理器时，可能会混淆。为了避免这种混淆，SYCL 将处理单元称为设备。第 2 章将开始讨论如何将工作（计算）放到异构系统中的特定设备上。

GPU 已经成为高性能计算设备，因此有时也称为通用 GPU 或 GPGPU。

现在，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(应用专用集成电路) 和 AI 芯片（图形、神经拟态等）。

这类设备通常重复涉及计算处理器（多处理器），以及增加诸如内存等数据源的连接（增加带宽）。第一个是多处理，对于提高吞吐量特别有用。我们的类比中，这通过添加额外的司机和卡车来实现。后者有更高的数据带宽，对于减少延迟特别有用。这些，可以通过使用更多的装载坞来实现，以使卡车能够并行装载。

拥有多种类型设备时，每种设备具有不同的架构，因此具有不同的特性，这就导致每种设备需要不同的编程和优化需求。这就成为了编写 SYCL(DPC++ 编译器) 的动机，也是本书的主要内容。

SYCL 是为了解决异构系统中数据并行编程的挑战。

数据并行编程

数据并行编程关注的是并行性，可以将其想象为一组并行操作的数据。我们需要运送 100 个包裹（有效地大量数据），以便将工作分配给 100 辆有司机的卡车。关键的是我们应该划分什么？处理整个图像，还是用小块，还是逐像素处理？应该将对象的集合分析为单个集合，还是将对象的一组更小的分组，或者一个对象一个对象地分析？

选择正确的工作分工并将其有效地映射到计算资源上，是使用 SYCL 和 DPC++ 的并行开发者的责任。这个主题会在第 4 章开始讨论。

DPC++ 和 SYCL 的关键属性

每个 DPC++(或 SYCL) 程序都是一个 C++ 程序。SYCL 和 DPC++ 都不依赖于对 C++，可以通过模板和 Lambda 函数实现。

SYCL 编译器要以依赖于 SYCL 规范的内置优化。缺乏 SYCL 内置优化的标准 C++ 编译器，无法达到支持 SYCL 的编译器的性能水平。

接下来，将了解 DPC++ 和 SYCL 的关键属性：单源模式、主机、设备、内核代码和异步任务图。

单源模式

程序可以是单源的，同一个翻译单元既包含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执行的主机代码。开发者仍然可以将程序源划分为不同的文件，分别用于表示主机和设备代码。

主机

每个程序都是从主机开始运行的，并且程序中的大部分代码通常为主机编写。目前为止，主机一直都是 CPU。标准不要求这样做，所以我们可以将其描述为主机。不过，主机似乎不太可能是 CPU 以外的任何东西，因为主机程序需要完全支持 C++17，以便支持所有 DPC++ 和 SYCL 程序。我们很快就会了解到，在设备端不需要支持所有的 C++17 特性。

设备

程序中使用多个设备是异构编程的原因。这就是为什么在之前对异构系统进行解释之后，“设备”这个词在本章中反复出现的原因。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。

SYCL 承诺的加速目标是设备端，加速计算的想法通常是将工作转移到可以加速工作完成的设备上进行。所以，需要了解如何消弭移动数据所消耗的时间，这需要不断的思考。

共享设备

一个有设备（如 GPU）的系统上，可以想象两个或更多的程序正在运行，并且希望使用一个设备。不必是使用 SYCL 或 DPC++ 的程序。如果另一个程序正在使用该设备，则其他程序需要等待，这与 C++ 程序中使用 CPU 的原理相同。如果在 CPU 上同时运行太多的活动程序（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。

超级计算机上，当为节点（CPU+ 附加设备）指定应用程序时，不需要考虑共享。非超级计算机系统上，如果有多个应用程序同时使用相同的设备，那么数据并行的程序性能会受到影响。

一切也都能正常工作，无需对此进行特别的编程。

内核代码

设备端执行的代码称为内核代码。这不是 SYCL 或 DPC++ 独有的概念：它是其他加速语言包括 OpenCL 和 CUDA 的内核概念。

内核代码有一定的限制，允许更广泛的设备支持和大量并行。内核代码中不支持的特性包括动

态多态性、动态内存分配 (因此没有使用 new 或 delete 操作符的对象管理)、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虚成员函数和变参数函数。内核代码不允许递归。

第 3 章将描述在调用内核前后，如何完成内存分配，从而确保内核代码专注于大规模并行计算。第 5 章将描述如何处理设备上代码的异常。

C++ 的其余部分在内核中是一样的，包括 Lambda、操作符重载、模板、类和静态多态性。我们还可以与主机共享数据 (见第 3 章)，并共享 (非全局) 主机变量的只读值 (通过 Lambda 捕获)。

内核代码：向量相加 (DAXPY)

任何编写过计算复杂代码的程序员都应该熟悉内核。考虑实现 DAXPY，代表“双精度 A 乘以 X 加 Y”。代码 1-2 展示了 DAXPY 在现代 Fortran, C/C++ 和 SYCL 中实现的，计算行 (第 3 行) 实际上是相同的。第 4 章和第 10 章将详细解释内核。代码 1-2 应该有助于消除内核难以理解的担忧。

图 1-2 DAXPY 计算在 Fortran, C++ 和 SYCL 的实现

```
1 ! Fortran loop
2 do i = 1, n
3   z(i) = alpha * x(i) + y(i)
4 end do
5
6 // C++ loop
7 for (int i=0;i<n;i++) {
8   z[i] = alpha * x[i] + y[i];
9 }
10
11 // SYCL kernel
12 myq.parallel_for(range{n}, [=](id<1> i) {
13   z[i] = alpha * x[i] + y[i];
14 }).wait();
```

异步任务图

SYCL/DPC++ 编程的异步特性是有必要了解的。理解异步编程至关重要，原因有二：(1) 正确使用可以获得更好的性能 (更好的扩展性)，(2) 错误会导致并行编程错误 (通常是竞争条件)，从而使程序变得不可靠。

之所以具有异步特性，是因为通过请求操作的“队列”传输到设备。主程序将请求操作提交到队列中，然后程序继续运行，不等待结果。这种不等待很重要，这样就可以让计算资源 (设备和主机) 处于忙碌状态。如果必须等待，这将占用主机资源。当设备完成时，还会产生串行瓶颈，直到新工作入队。Amdahl 定律惩罚的是花在不并行工作上的时间，我们需要构建我们的程序，以便在设备繁忙时，在设备和主机之间来回移动数据，并在执行任务时保持设备和主机的计算能力。如果不这样做，Amdahl 法则将会惩罚我们。

第 4 章将把程序看作异步任务图，第 8 章会扩展这个概念。

出错时的条件竞争

第一个代码示例(代码 1-1)中,在第 18 行做了 *wait*,以避免 *result* 可用前进行使用。相同的代码示例中,还有另一件事情——第 14 行使用 *std::memcpy* 加载输入。由于 *std::memcpy* 在主机上运行,所以直到第 15 行完成后,才执行第 16 行及以后的代码。阅读了第 3 章之后,可能会使用 *myQ.memcpy*(使用 SYCL),在代码 1-3 的第 8 行。因为这是队列提交,所以不能保证它会在第 10 行之前完成。这将引起条件竞争,是并行编程的 Bug。当程序的两部分同时访问相同的数据时,就存在条件竞争。我们希望使用第 8 行写入数据,然后在第 10 行读取数据,所以不希望在第 8 行完成之前执行第 17 行!这样的条件竞争会使程序的结果不可预测——在不同的系统和不同的运行情况下得到不同的结果。解决这个问题的方法是显式地等待 *myQ*,将 *.wait()* 添加到第 8 行末尾,但这不是最好的解决办法。可以使用事件来解决这个问题(第 8 章)。作为一种替代方案,第 7 章将看到如何使用缓冲区和访问器的编程方式,让 SYCL 管理依赖关系并自动等待。

图 1-3 添加条件竞争

```
1 // ...we are changing one line from Figure 1-1
2 char *result = malloc_shared<char>(sz, Q);
3
4 // Introduce potential data race!
5 // We don't define a dependence
6 // to ensure correct ordering with
7 // later operations.
8 Q memcpy(result, secret.data(), sz);
9
10 Q parallel_for(sz, [=](auto&i) {
11     result[i] -= 1;
12 }).wait();
13
14 // ...
```

添加 *wait()* 强制在 *memcpy* 和内核之间进行主机同步,这与让设备一直处于繁忙状态的建议相反。

为了检测程序中的数据条件竞争(包括内核),可以使用一些工具,如 Intel Inspector(可以使用前面“获得 DPC++ 编译器”中提到的 oneAPI 工具)。这些工具使用的方式并不适用于所有设备。检测条件竞争最好的方法是让所有内核都运行在一个 CPU 上,这可以作为开发期间的一种调试技巧。这个调试技巧会在第 2 章中以方法 2 的形式进行讨论。

第 4 章将告诉我们“Lambda 无害”,为了更好地使用 DPC++、SYCL 和现代 C++,我们应该去了解 Lambda 函数。

C++ Lambda 函数

并行编程技术大量使用的现代 C++ 的特性是 Lambda 表达式。内核(在设备上运行的代码)可以用多种方式表示,最常见的一种是 Lambda 函数。第 10 章讨论了内核可以采用的所有形式,包括 Lambda 函数。这里有一个关于 C++ Lambda 函数的复习,以及关于使用定义内核的注意事项。前面的章节中学习了更多关于 SYCL 的知识之后,第 10 章会对内核方面进行扩展。

图 1-3 中的代码有一个 Lambda 函数。在 C++ 中，Lambda 以一个方括号开始，在结束方括号之前的信息表示如何捕获在 Lambda 中使用，但没有显式地作为参数传递给它的变量。对于内核，必须捕获方括号中等号表示的值。

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赋值给命名变量），这些对象可以从外围作用域捕获变量。C++ Lambda 表达式的基本语法是：

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

- *capture-list* 是一个以逗号分隔的捕获列表，通过在捕获列表中列出变量名来按值捕获。通过引用来捕获变量，在它前面加上一个 & 号，例如：&v。还有适用于所有作用自动变量：[=] 是用来捕获所有变量和当前对象的引用，[&] 是用来捕获所有变量以及当前对象的引用，而 [] 就什么都不捕获。对于 SYCL，总使用 [=]，因为在内核中不允许通过引用捕获变量。根据 C++ 标准，Lambda 中不包含全局变量。非全局静态变量可以在内核中使用，但只有当它们是 const 时才可以。
- *params* 是函数参数的列表，就像命名函数一样。**SYCL 提供了一些参数来标识正在调用内核处理的元素：这可以是唯一的 id(一维的)，也可以是 2 维或 3 维的 id。** 这些将在第 4 章中讨论。
- *ret* 是返回类型。如果->ret 未指定，则从 return 语句推断。没有 return 语句或没有值的 return，意味着返回类型为 void。SYCL 内核必须总是有一个返回类型 void，所以可以不用这种方式来为内核指定返回类型。
- *body* 是函数体。对于 SYCL 内核的内容有一些限制（参见本章前面的“内核代码”部分）。

图 1-4 C++ 代码中的 Lambda 函数

```
1 int i = 1, j = 10, k = 100, l = 1000;
2
3 auto lambda = [i, &j] (int k0, int &l0) -> int {
4     j = 2* j;
5     k0 = 2* k0;
6     l0 = 2* l0;
7     return i + j + k0 + l0;
8 };
9
10 print_values( i, j, k, l );
11 std::cout << "First call returned " << lambda( k, l ) << "\n";
12 print_values( i, j, k, l );
13 std::cout << "Second call returned " << lambda( k, l ) << "\n";
14 print_values( i, j, k, l );
```

图 1-5 图 1-4 中的 Lambda 函数代码的输出

```

i == 1
j == 10
k == 100
l == 1000
First call returned 2221
i == 1
j == 20
k == 100
l == 2000
Second call returned 4241
i == 1
j == 40
k == 100
l == 4000

```

图 1-4 展示了一个 C++ Lambda 表达式，通过值捕获变量 i，通过引用捕获变量 j。还有一个参数 k0 和另一个通过引用接收的参数 l0。运行此示例将产生如代码 1-5 所示的输出结果。

可以把 Lambda 表达式看作函数对象的实例，编译器会创建类定义。例如，前面的示例中使用的 Lambda 表达式类似于代码 1-6 中所示的类实例。无论在哪里使用 C++ Lambda 表达式，都可以用函数对象的实例来替代它，如图 1-6 所示。

当定义函数对象时，需要给它分配一个名称（图 1-6 中的 Functor）。内联表示的 Lambda（如图 1-4 所示）是匿名的，不需要名称。

图 1-6 用函数对象代替 Lambda(详见第 10 章)

```

1 class Functor{
2 public:
3     Functor(int i, int &j) : my_i{i}, my_jRef{j} { }
4
5     int operator()(int k0, int &l0) {
6         my_jRef = 2 * my_jRef;
7         k0 = 2 * k0;
8         l0 = 2 * 10;
9         return my_i + my_jRef + k0 + 10;
10    }
11
12 private:
13     int my_i;
14     int &my_jRef;
15 };

```

可移植性和直接编程

可移植性是 SYCL 和 DPC++ 的目标，但两者都不能保证。一种语言和编译器所能做的，就是在需要的时候更容易在应用程序中实现可移植性。

可移植性比较复杂，包括功能和性能可移植性。功能可移植性，程序可以在各种平台上编译和运行。性能可移植性，程序能够在各种平台上获得合理的性能。虽然这是相当软的定义，但相反的情况可能会更清楚——我们不想写一个程序，其在一个平台上运行得非常快，但在另一个平台上运行得非常慢。事实上，我们更希望能充分利用运行它的任何平台。作为开发者，考虑到异构系统中设备的多样性，性能可移植性需要我们付出更多的努力。

幸运的是，SYCL 定义了可以提高性能可移植性的方式。首先，通用内核可以在任何地方运行。少数情况下，这可能就足够了。更常见的是，重要内核的几个版本可能是为不同类型的设备编写。具体来说，一个内核可能有一个通用 GPU 和通用 CPU 版本。有时候，可能想为特定的设备（特定的 GPU）专门设计内核。当这种情况发生时，可以编写多个版本，并针对不同的 GPU 模型对每个版本进行定制化。或者我们可以参数化一个版本，使用 GPU 的属性来修改现有的 GPU 内核代码，以适应当前的 GPU。

作为开发者，我需要设计有效的性能可移植性计划，SYCL 定义了一些结构来帮助我们实现这个计划。正如前面提到的，可以通过以下方式对功能进行分层：首先是针对所有设备的内核，然后根据需要逐步引入更特化的内核。这听起来很棒，但程序的整体流程也可能产生深远的影响，因为数据移动和算法选择也很重要。了解了这一点，我们就可以理解为什么 SYCL（或其他直接编程解决方案）没有解决性能可移植性。但作为工具，可以帮助我们应对这些挑战。

并发和并行

“并行”和“并发”并不等同，尽管有时会误解为等同。并发所需要的编程考虑对并行性也很重要。

“并发”是可以启动的代码，但不一定是在同一时刻。在我们的计算机上，如果打开了一个邮件程序和一个 Web 浏览器，那么它们是同时运行的。在只有一个处理器的系统上，通过一个时间切片（在运行每个程序之间快速来回切换）的过程可以发生并发。

Tip

对并发性的任何考虑，对并行性也很重要。

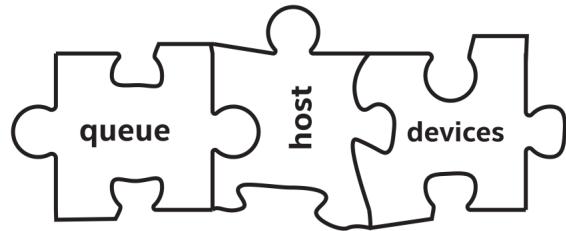
“并行”是可以在同一时刻启动的代码。并行要求系统可以同时做多件事。异构系统总是可以并行地做事，因为其至少有两个计算设备。当然，SYCL 程序不需要异构系统，因为它可以仅在主机系统上运行。

代码的并发执行通常面临着与代码并行执行相同的问题，因为任何特定的代码都不能假设是唯一的（数据位置、I/O 等）。

总结

本章提供了学习 SYCL 和 DPC++ 所需的术语，并提供了对学习 SYCL 和 DPC++ 至关重要的并行编程内容和 C++ 关键特性方面的复习。第 2 章、第 3 章和第 4 章会对 SYCL 编程的三个关键进行了扩展：给设备分配工作（发送代码在设备上运行），提供数据（发送数据在设备上使用），编写代码的方法（内核）。

2 代码执行的位置



并行编程并不是真的在快车道上行驶，而是在所有车道上快速行驶。这一章是关于能够把代码放到可以执行的地方执行，并且选择在合适的时候启用异构系统中的计算资源。因此，需要知道有哪些计算资源（找到它们），并让它们工作（执行代码）。

可以控制代码的执行位置——可以控制内核代码运行的设备。SYCL 为异构编程提供了框架，代码可以在主机 CPU 和设备的混合上执行。决定代码在何处执行的机制，对于我们理解和使用异构系统非常重要。

本章描述了代码可以在什么地方，什么时候执行，以及用来控制执行位置的机制。第 3 章将了解如何管理数据，使其在代码执行时使用，然后第 4 章返回到代码本身，讨论内核代码的编写。

单个源文件

SYCL 程序可以是单个源文件，意味着同一个源码单元（通常是源文件及其头文件）既包含定义要在 SYCL 设备上执行的计算内核的代码，也包含调度这些内核执行的主机代码。图 2-1 显示了这两个代码路径，图 2-2 给出了标有主机和设备代码区域的应用示例。

将设备代码和主机代码合并到单个源文件（或源码）可以更容易地理解和维护异构应用程序。这种组合还提供了语言的安全性，编译器可以对代码进行更多的优化。

图 2-1 单个源文件代码包含主机代码（CPU 上运行）和设备代码（SYCL 设备上运行）

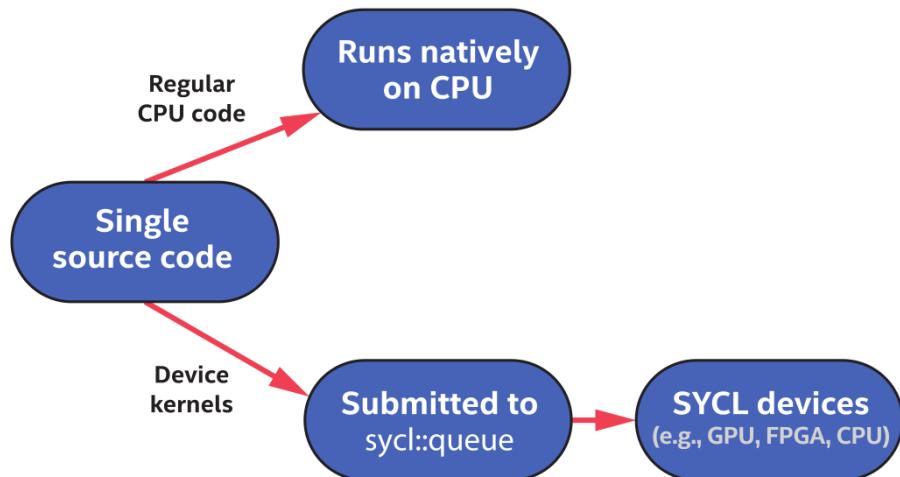


图 2-2 简单的 SYCL 程序

```

#include <CL/sycl.hpp>
#include <array>
#include <iostream>
using namespace sycl;

int main() {
    constexpr int size=16;
    std::array<int, size> data;

    // Create queue on implementation-chosen default device
    queue Q;

    // Create buffer using host allocated "data" array
    buffer B { data };

    Q.submit([&] (handler& h) {
        accessor A{B, h};
        h.parallel_for(size , [=](auto& idx) {
            A[idx] = idx;
        });
    });

    // Obtain access to buffer on the host
    // Will wait for device kernel to execute to generate data
    host_accessor A{B};
    for (int i = 0; i < size; i++)
        std::cout << "data[" << i << "] = " << A[i] << "\n";

    return 0;
}

```

The diagram illustrates the structure of SYCL code. It shows three main sections: Host code (the outermost braces), Device code (the dashed red rectangle), and another Host code section. The Device code section contains the parallel_for loop body.

主机端代码

应用程序包含 C++ 主机代码，由操作系统启动 CPU 执行应用。主机端代码是应用程序的主要部分，它定义和控制对可用设备的工作分配，也是定义运行时数据和依赖项的接口。

主机端代码特定于 SYCL 的构造和 C++ 标准的扩展类，这些构造和类设计可实现为库，使得主机端代码（只要 C++ 允许）更容易实现，并且可以简化与构建系统的集成。

SYCL 应用是用扩充的 C++ 标准，可以实现为 C++ 库。通过“理解”这些库，SYCL 编译器可以为程序提供更高的性能。

应用程序中的主机端代码会协调设备端数据移动和计算，也可以执行计算密集型工作，并可以使用任何 C++ 的库。

设备端代码

设备对应于加速器或处理器，在概念上独立于执行主机代码的 CPU。主机处理器也可以作为一个设备，但主机处理器和设备是逻辑上相互独立的。主机处理器运行本机 C++ 代码，而设备运行设备端代码。

队列是一种将工作提交给设备执行的机制。设备端代码有三个重要的属性：

- 主机端代码会**异步执行**。主机程序向设备提交设备代码，运行库只在满足所有的执行依赖关系时才跟踪和启动，并且主机程序在提交在设备上启动之前执行。提供设备上的执行与主机程序异步，除非显式地将两者结合在一起。
- 对设备代码有**限制**加速设备上编译和实现性能。例如，**设备代码不支持动态内存分配和运行时类型信息 (RTTI)**，这会导致许多加速器的性能下降。设备代码限制的小集合将在第 10 章中详细介绍。
- **SYCL 定义的一些函数和查询只能在设备代码中使用**，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数据范围内的位置（在第 4 章中描述）。

通常，我们将包括提交在内的工作称为操作。在第 3 章中，将学习执行设备代码，还包括内存移动命令。本章中，我们关注的是操作的设备代码方面，所以在大部分时间提及的是设备端代码。

选择设备

为了探索设备代码将在何处执行，我们来看五个用例：

方法 1：当**不关心使用哪个设备时**，**随意**在某个地方运行设备代码即可。这通常是开发的第一步。
没有设备，单纯为了调试

方法 2：在**主机设备上显式地运行设备代码**，通常用于**调试**。主机设备保证在任何系统上始终可用。

方法 3：将**设备代码分配到 GPU 或加速器上**。

方法 4：将**设备代码分配到异构设备集**，如同时分配到 GPU 和 FPGA 上。

方法 5：从**更一般的设备类别中选择特定的设备**，例如：从一组可用 FPGA 类型中选择特定类型的 FPGA。

☞ 开发人员通常会使用方法 2 调试代码，并且只有在使用方法 2 对代码进行了实际测试后，才会使用方法 3-5。

方法 1：在任何类型的设备上运行

不关心设备代码在哪里运行时，可以让运行时库帮忙选择。这种自动选择是不关心选择了什么设备时，可以很容易地开始编写和运行代码。这种设备选择没有考虑要运行的代码特性，所以是一个随意的选择，这个选择很可能不是最优的。

讨论设备的选择（即使是实现为选择的设备）之前，应该首先讨论与设备交互的机制：**队列**。

队列 *通过队列将工作提交给设备*

队列是一种抽象，将操作提交给它在单个设备上执行。图 2-3 和图 2-4 给出了 *queue* 类的定义。入队的通常是数据并行的计算，当需要更多的控制时，也可以使用其他命令，如手动控制数据移动。提交到队列的工作需要满足运行时跟踪的先决条件（例如输入数据的可用性）进行执行。这些先决条件在第 3 章和第 8 章中有介绍。

图 2-3 简化的 queue 类

```

1 class queue {
2 public:
3     // Create a queue associated with the default device
4     queue(const property_list = {});
5     queue(const async_handler&,
6           const property_list = {});
7
8     // Create a queue associated with an explicit device
9     // A device selector may be used in place of a device
10    queue(const device&, const property_list = {});
11    queue(const device&, const async_handler&,
12          const property_list = {});
13
14    // Create a queue associated with a device in a specific context
15    // A device selector may be used in place of a device
16    queue(const context&, const device&,
17          const property_list = {});
18    queue(const context&, const device&,
19          const async_handler&,
20          const property_list = {});
21};

```

图 2-4 简化了 queue 类中的关键成员函数

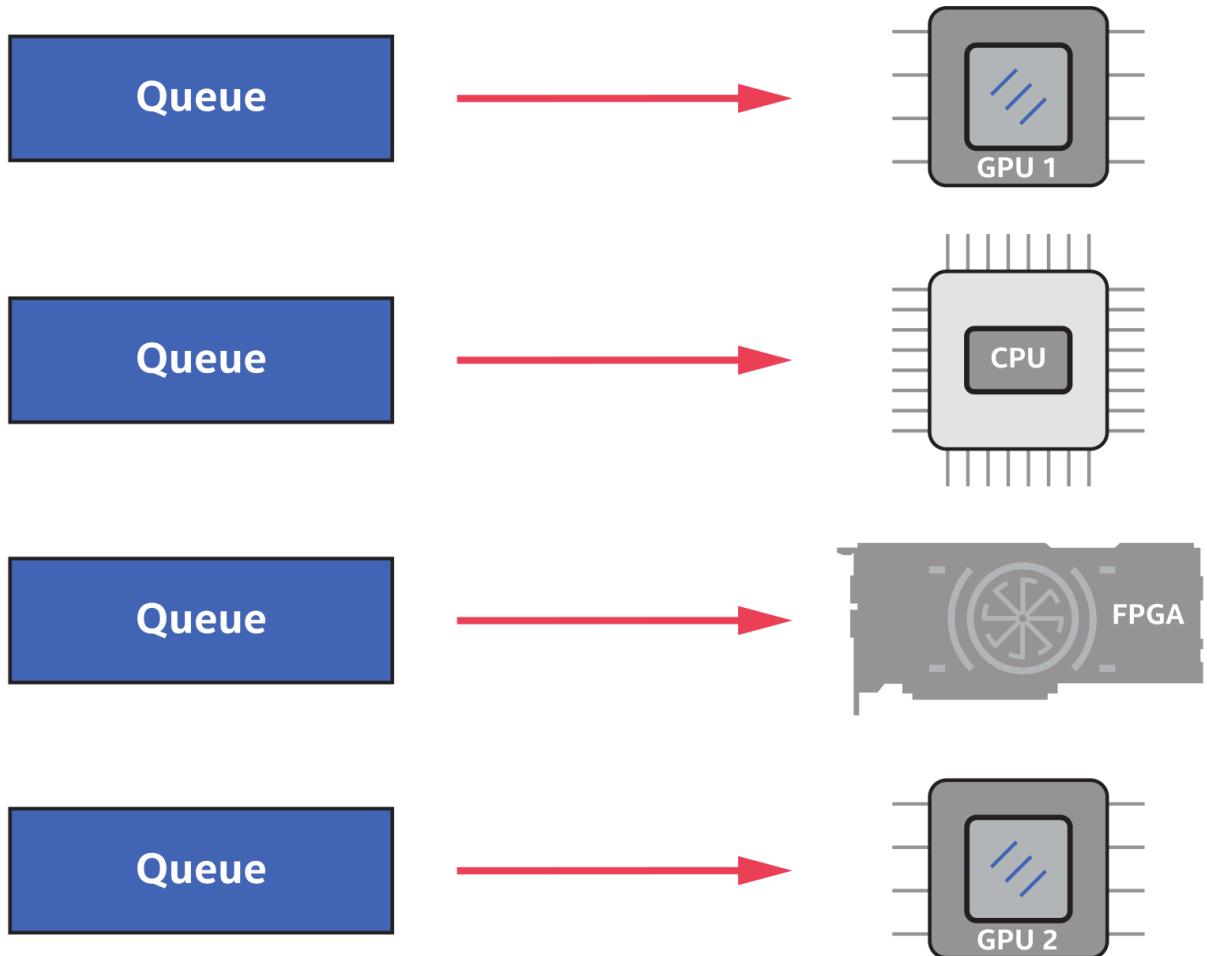
```

1 class queue {
2 public:
3     // Submit a command group to this queue.
4     // The command group may be a lambda or functor object.
5     // Returns an event representation the action
6     // performed in the command group.
7     template <typename T>
8     event submit(T);
9
10    // Wait for all previously submitted actions to finish executing.
11    void wait();
12
13    // Wait for all previously submitted actions to finish executing.
14    // Pass asynchronous exceptions to an async_handler if one was provided.
15    void wait_and_throw();
16};

```

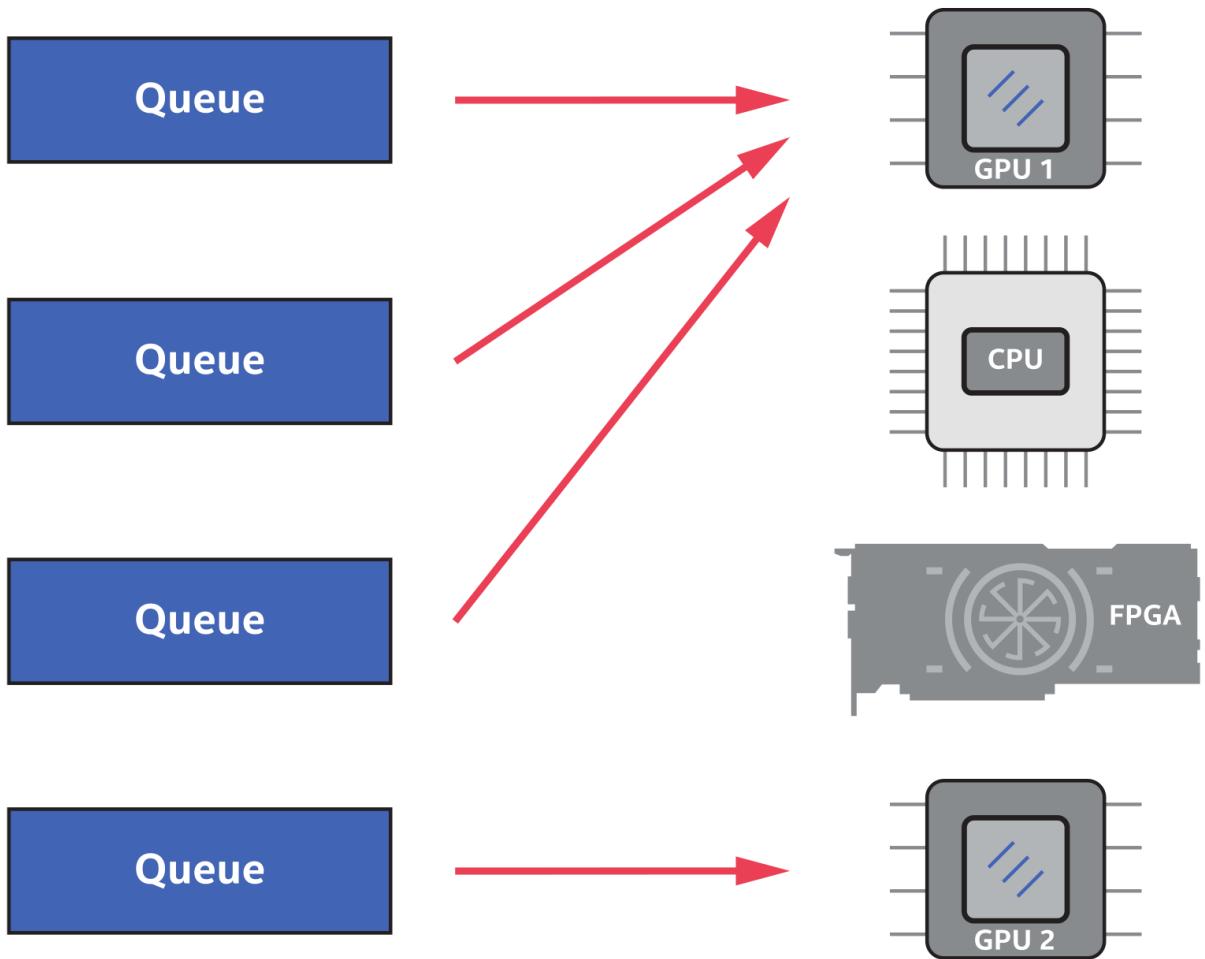
队列绑定到单个设备，绑定发生在队列的构造过程中。提交给队列的工作是在该队列绑定的设备上执行的。队列不能映射到设备集合，因为这会不明确应该在哪个设备上执行工作。同样，队列也不能将提交的工作分散到多个设备上。相反，队列和提交给队列的工作将在执行设备之间有一个明确的映射，如图 2-5 所示。

图 2-5 队列绑定到单个设备，提交到队列中的工作在相应的设备上执行



程序中可以创建多个队列，让每个队列与不同的设备绑定，或者让主机中的不同线程使用。**多个不同的队列可以绑定到单个设备上**，提交到这些不同队列的工作将组合在设备上执行，如图 2-6 所示。相反，正如我们前面提到的，队列不能绑定到多个设备，因为在执行操作的位置上不能有任何歧义。例如，想要一个能够跨多个设备加载平衡工作的队列，需要先创建这个对象。

图 2-6 多个队列可以绑定到单个设备



因为队列绑定到特定的设备，所以提交给队列的操作是将相应工作在设备上执行的常用方法。构造队列时选择设备是通过设备选择器和 `device_selector` 类实现的。

绑定队列与设备 (任何设备都可以)

图 2-7 是没有指定队列绑定设备的示例。queue 的构造函数没有任何参数 (如图 2-7 所示)，只是选择一些可用的设备。SYCL 保证至少有一个设备可用，即主机。主机也可以运行内核代码，它是执行主机程序的处理器，所以总是存在。

图 2-7 通过队列的隐式构造，选择默认设备

```

1 #include <CL/sycl.hpp>
2 #include <iostream>
3 using namespace sycl;
4
5 int main() {
6     // Create queue on whatever default device that the implementation
7     // chooses. Implicit use of the default_selector.
8     queue Q;
9
10    std::cout << "Selected device: " <<
11    Q.get_device().get_info<info::device::name>() << "\n";

```

```

12
13     return 0;
14 }
15
16 Possible Output:
17 Device: SYCL host device

```

实验室服务器默认3090。

使用 queue 的构造函数是启动设备代码的最简单方法。对于绑定到队列的设备的选择，因为与我们的应用程序相关，可以添加更多的控制。

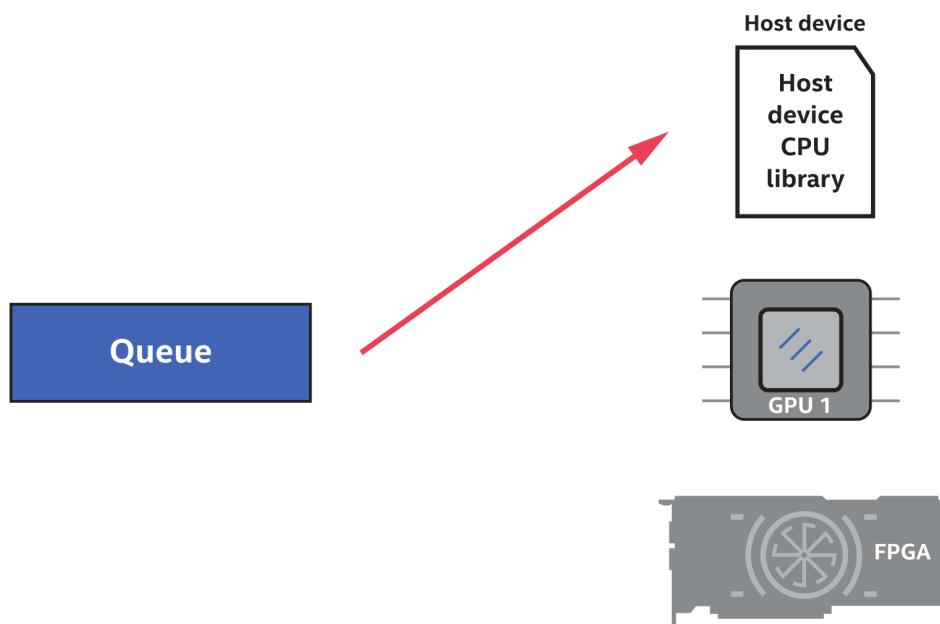
方法 2: 使用主机设备进行开发和调试

可以将主机看作是独立设备，可以执行设备代码。总是有一些处理器运行主机程序，因此主机对我们的应用程序总是可用的。主机设备提供了保证，设备代码可以始终运行（不依赖加速器硬件），并有以下几个用途：

1. **设备代码的开发**在没有任何加速器的系统上：常用的方式是在部署到 HPC 集群进行性能测试和优化之前，在**本地系统上开发和测试设备代码**。
2. **使用非加速工具调试设备代码**：加速工具通常通过公开底层 API 让开发者使用，这些 API 可能没有像主机端那样的调试工具。考虑到这一点，主机设备应该支持使用 CPU 开发，以便开发人员熟悉调试工具。
3. **备份**如果没有其他设备保证代码可以执行：主机端运行设备代码可能不以性能作为主要目标，因此需要考虑作一个**功能性的备份**，确保设备代码可以执行。

主机在功能上类似于硬件加速器设备，可以绑定队列，可以执行设备代码。图 2-8 展示了主机设备中与其他加速器是对等的。可以执行设备代码，就像 CPU、GPU 或 FPGA 一样，并且可以绑定一个或多个队列。

图 2-8 主机可以像任何加速器一样执行设备代码



应用程序可以显式地将 host_selector 传递给队列构造，来选择创建一个绑定到主机设备的队列，如图 2-9 所示。

图 2-9 使用 host_selector 选择主机设备

```
1 #include <CL/sycl.hpp>
2 #include <iostream>
3 using namespace sycl;
4
5 int main() {
6     // Create queue to use the host device explicitly
7     queue Q{ host_selector{} };
8
9     std::cout << "Selected device: " <<
10    Q.get_device().get_info<info::device::name>() << "\n";
11    std::cout << " -> Device vendor: " <<
12    Q.get_device().get_info<info::device::vendor>() << "\n";
13
14    return 0;
15 }
16
17 Possible Output:
18 Device: SYCL host device
```

即使没有特别的要求（例如，使用 host_selector），主机也会作为默认选项，如图 2-7 中的输出所示。

一些设备选择器类，可以很容易地指定一种类型的设备。host_selector 是这些选择器类的一个例子，我们将在接下来的几节中讨论其他的选择器类。

方法 3: 使用 GPU(或其他加速器)

GPU 会在下一个例子中进行展示，不过对于任何类型的加速器都一样。为了针对常见的加速器类，设备分为几个大类，SYCL 提供了内置的选择器类。要从设备类型类别中进行选择，例如“系统中可用的 GPU”，相应的代码非常短。

设备类型

队列可以绑定的设备主要有两类：

1. 主机。
2. 加速设备，如 GPU、FPGA 或 CPU 设备，用于加速程序中的负载。

加速器设备

加速器类型主要有以下几种：

1. CPU 设备
2. GPU 设备

3. 加速器，捕捉既不为 CPU，也不是 GPU 的设备。这包括 FPGA 和 DSP。

使用内置的选择器类，这些类别中的任何设备都可以很容易地绑定到队列，这些选择器类可以传递到队列（和其他一些类）的构造中。

设备选择器

必须绑定到特定设备的类，如：queue 类。其构造函数可以接受从 device_selector 派生的类。queue 构造函数是

```
queue( const device_selector &deviceSelector, const property_list &propList = {});
```

有 5 种常用的内置选择器：

default_selector	默认选择设备
host_selector	选择主机（始终可用）
cpu_selector	选择一个标识为 CPU 的设备
gpu_selector	选择一个标识为 GPU 的设备
accelerator_selector	选择一个标识为“加速器”的设备，其中包括 FPGA。

DPC++ 中包含的另一个选择器（SYCL 中不可用）可以通过包含头文件 "CL/sycl/intel/fpga_extensions.hpp" 来使用：

INTEL::fpga_selector 选择一个标识为 FPGA 的设备

可以使用内置选择器来构造队列，例如：

```
queue myQueue{ cpu_selector{} };
```

图 2-10 给出了使用 cpu_selector 的完整示例，图 2-11 给出了队列与可用 CPU 设备的对应绑定。

图 2-12 展示了使用各种内置选择器类的例子，演示了设备选择器与另一个类 (device) 的使用，该类在构造时接受一个 device_selector。

图 2-10 CPU 设备选择器示例

```
1 #include <CL/sycl.hpp>
2 #include <iostream>
3 using namespace sycl;
4
5 int main() {
6     // Create queue to use the CPU device explicitly
7     queue Q{ cpu_selector{} };
8
9     std::cout << "Selected device: " <<
```

```

10     Q.get_device().get_info<info::device::name>() << "\n";
11     std::cout << " -> Device vendor: " <<
12     Q.get_device().get_info<info::device::vendor>() << "\n";
13
14     return 0;
15 }
16 // Possible Output:
17 // Selected device: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
18 // -> Device vendor: Intel(R) Corporation

```

图 2-11 将 CPU 设备绑定到程序队列

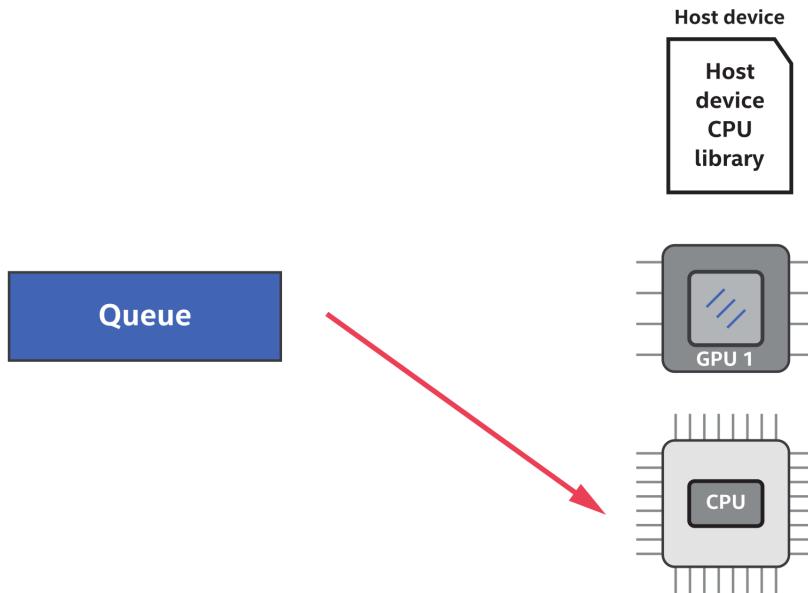


图 2-12 各种设备选择器类的标识输出，设备选择器可以用于构造多个队列 (本例中，构造了类实例)

```

1 #include <CL/sycl.hpp>
2 #include <CL/sycl/INTEL/fpga_extensions.hpp> // For fpga_selector
3 #include <iostream>
4 #include <string>
5 using namespace sycl;
6
7 void output_dev_info( const device& dev,
8     const std::string& selector_name ) {
9     std::cout << selector_name << ": Selected device: " <<
10     dev.get_info<info::device::name>() << "\n";
11     std::cout << " -> Device vendor: " <<
12     dev.get_info<info::device::vendor>() << "\n";
13 }
14
15 int main() {
16     output_dev_info( device{ default_selector{} },

```

```

17         " default_selector" );
18     output_dev_info( device{ host_selector{}},
19                     " host_selector" );
20     output_dev_info( device{ cpu_selector{}},
21                     " cpu_selector" );
22     output_dev_info( device{ gpu_selector{}},
23                     " gpu_selector" );
24     output_dev_info( device{ accelerator_selector{}},
25                     " accelerator_selector" );
26     output_dev_info( device{ INTEL::fpga_selector{}},
27                     " fpga_selector" );
28
29     return 0;
30 }
31
32 //Possible Output:
33 //default_selector: Selected device: Intel(R) Gen9 HD Graphics NEO
34 //          -> Device vendor: Intel(R) Corporation
35 //host_selector: Selected device: SYCL host device
36 //          -> Device vendor:
37 //cpu_selector: Selected device: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
38 //          -> Device vendor: Intel(R) Corporation
39 //gpu_selector: Selected device: Intel(R) Gen9 HD Graphics NEO
40 //          -> Device vendor: Intel(R) Corporation
41 //accelerator_selector: Selected device: Intel(R) FPGA Emulation Device
42 //          -> Device vendor: Intel(R) Corporation
43 //fpga_selector: Selected device: pac_a10 : PAC Arria 10 Platform
44 //          -> Device vendor: Intel Corp

```

设备选择失败时

gpu_selector 创建的对象 (比如: 队列) 使用时, 但没有可供运行时使用的 GPU 设备, 那么这个选择器会抛出 runtime_error 异常。对于所有的设备选择器类都是如此, 没有所需的设备可用, 则抛出 runtime_error 异常。对于复杂的应用程序来说, 捕获这个错误并使用不太理想的 (对于应用程序/算法) 设备类作为替代是合理的。异常和错误处理将在第 5 章中详细讨论。

方法 4: 使用多个设备

如图 2-5 和 2-6 所示, 可以在程序中构造多个队列。可以将这些队列绑定到单个设备上 (队列的工作汇集到单个设备上), 绑定到多个设备上, 或者绑定到这些设备的组合上。以 GPU 绑定队列和 FPGA 绑定队列为例, 如图 2-13 所示。对应的映射如图 2-14 所示。

图 2-13 创建 GPU 和 FPGA 设备的队列

```

1 #include <CL/sycl.hpp>
2 #include <CL/sycl/INTEL/fpga_extensions.hpp> // For fpga_selector
3 #include <iostream>

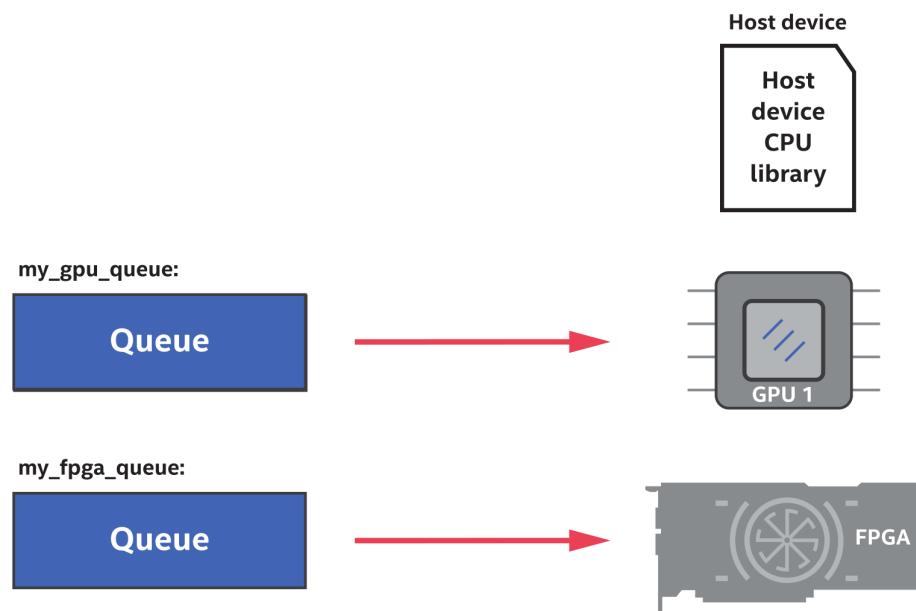
```

```

4 using namespace sycl;
5
6 int main() {
7     queue my_gpu_queue( gpu_selector{} );
8     queue my_fpga_queue( INTEL::fpga_selector{} );
9
10    std :: cout << " Selected device 1: " <<
11        my_gpu_queue.get_device().get_info<info::device::name>() << "\n";
12
13    std :: cout << " Selected device 2: " <<
14        my_fpga_queue.get_device().get_info<info::device::name>() << "\n";
15
16    return 0;
17}
18
19//Possible Output:
20//Selected device 1: Intel(R) Gen9 HD Graphics NEO
21//Selected device 2: pac_a10 : PAC Arria 10 Platform

```

图 2-14 GPU+FPGA 设备选择器示例: 一个队列绑定 GPU, 另一个队列绑定 FPGA



方法 5: 选择自定义 (非常具体的) 设备

现在来看看如何编写自定义选择器。除了本章的例子, 第 12 章还有一些例子。内置的设备选择器旨在快速启动代码并运行。真正的程序通常需要对设备进行选择, 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为复杂的逻辑, 因此可以编写代码来选择设备。

device_selector 基类

所有的设备选择器都从 device_selector 基类派生，在派生类中定义函数操作符：

```
1 virtual int operator()(const device &dev) const {  
2     /* User logic */  
3 }
```

从 device_selector 派生的类中定义这个操作符，是定义复杂选择逻辑所必须的，需要了解以下三件事：

1. 当运行时发现程序可访问的设备（包括主机设备），函数调用操作符会自动调用。
2. 操作符每次调用时都返回一个分数，可用设备中得分最高的会是选择器所选择的设备。
3. 函数操作符返回负整数表示不能选择相应的设备。

设备评分机制

有很多机制为设备来给定一个分数，例如：

1. 返回特定设备类的正值。
2. 匹配设备名称和/或设备供应商字符串。
3. 基于设备或平台查询，可以在代码中指向整数值。

例如，选择 Intel Arria 族中 FPGA 设备的一种方式如图 2-15 所示。

图 2-15 Intel Arria FPGA 设备的自定义选择器

```
1 class my_selector : public device_selector {  
2 public:  
3     int operator()(const device &dev) const override {  
4         if (  
5             dev.get_info<info::device::name>().find("Arria")  
6             != std::string::npos &&  
7             dev.get_info<info::device::vendor>().find("Intel")  
8             != std::string::npos)  
9             return 1;  
10        }  
11        return -1;  
12    }  
13};
```

第 12 章有更多关于设备选择的讨论和例子（图 12-2 和 12-3），并会更加深入地讨论 get_info。

在 CPU 上执行设备端代码的三种方式

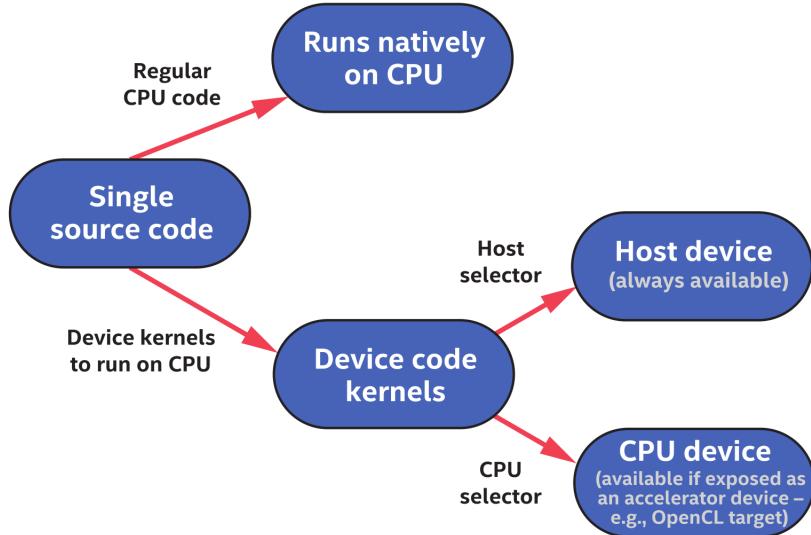
由于 CPU 上有多种代码执行的机制，所以会造成一些混乱，如图 2-16 所示。

CPU 最直接执行的还是主机代码，要么是单源应用程序的一部分（主机代码区域），要么是主机代码对其他主机代码或库（如库函数）的调用。

其他两个设备用于执行设备代码。第一段设备代码在 CPU 上是通过主机执行，这种方式在本章前面已经了解过。

SYCL 在 CPU 上执行设备代码的第二种可选方式是，使用 CPU 加速器对性能进行优化。该设备通常由 OpenCL 等底层运行时库实现，因此其可用性需要依赖于系统上安装的驱动程序和其他运行时库。SYCL 中，主机设备旨在通过本机 CPU 工具对设备代码进行调试，而 CPU 作为设备时，设备代码会运行在对性能有优化的实现上。

图 2-16 SYCL 在 CPU 上的执行机制



有一种机制本书没有涉及，可以在满足任务图中的先决条件时，将常规 CPU 代码（图 2-16 的顶部部分）入队。这个特性可以用来执行任务图中的 CPU 代码和设备代码，也称为主机任务。

在设备上创建任务

应用程序通常是包含主机代码和设备代码的组合。有一些类成员允许提交设备代码，并且这些工作分发机制是提交到设备执行的唯一方式，以便我们轻松地区分设备代码和主机代码。

本章的其余部分将介绍工作分发机制，目的是帮助了解和辨别在主机处理器上本机执行的设备代码和主机代码之间的区别。

任务图

SYCL 执行模型中的基本概念是节点图。图中的每个节点（工作单元）都包含在设备上执行的操作，其中最常见的操作是数据并行设备的内核调用。图 2-17 展示了有四个节点的示例图，其中每个节点都可以看作是一个设备内核。

图 2-17 中的节点具有依赖边，定义了节点的工作何时开始。依赖边最常见的是数据依赖项自动生成的，尽管也可以在需要时手动添加额外的依赖项，例如：图中的节点 B 与节点 A 有一条依赖边，这条边意味着节点 A 必须在 B 开始前完成执行，然后将结果数据放在节点 B 的设备上。运行时控制依赖项的解析和节点执行的触发，与主机程序的完全异步。定义应用程序的节点图在本书中称为任务图，并在第 3 章中进行更详细的介绍。

图 2-17 任务图定义了要在一个或多个设备上执行的操作 (与主机程序异步执行), 还定义了何时可以安全执行某个操作的依赖项

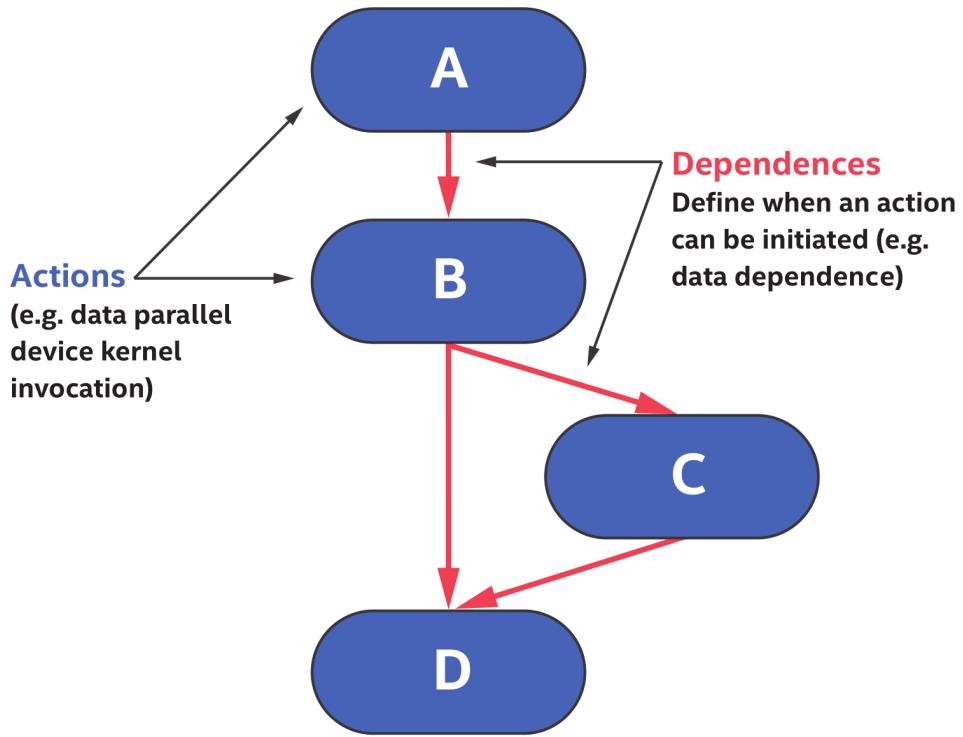


图 2-18 提交设备代码

```

Q.submit([&] (handler& h) {
    accessor acc{B, h};

    h.parallel_for(size, [=](auto& idx) { } );
    });
}
  
```

} Device code } Command group

设备代码在哪里?

七口一斤
捕获形参函数体

有多种机制可以在设备上执行代码, 但需要例子展示了如何识别这些代码。即使示例中的模式看上去很复杂, 但其模式在所有设备代码定义中都是相同的, 因此就成为了第二特性。

图 2-18 中定义为 Lambda, 作为最后一个参数传递给 parallel_for 的代码, 这个 Lambda 表达式就是要在设备上执行的设备代码。并行的 parallel_for 中可以区分设备代码和主机代码。parallel_for 是设备调度机制的一个小集合, 都是 handler 类的成员, 其定义了要在设备上执行的代码。图 2-19 给出了 handler 类的定义。

图 2-19 handler 类中成员函数的定义

```
1 class handler {
```

```

2 public:
3     // Specify event(s) that must be complete before the action
4     // defined in this command group executes.
5     void depends_on(std::vector<event>& events);
6
7     // Guarantee that the memory object accessed by the accessor
8     // is updated on the host after this action executes.
9     template <typename AccessorT>
10    void update_host(AccessorT acc);
11
12    // Submit a memset operation writing to the specified pointer.
13    // Return an event representing this operation.
14    event memset(void *ptr, int value, size_t count);
15
16    // Submit a memcpy operation copying from src to dest.
17    // Return an event representing this operation.
18    event memcpy(void *dest, const void *src, size_t count);
19
20    // Copy to/from an accessor and host memory.
21    // Accessors are required to have appropriate correct permissions.
22    // Pointer can be a raw pointer or shared_ptr.
23    template <typename SrcAccessorT, typename DestPointerT>
24        void copy(SrcAccessorT src, DestPointerT dest);
25
26    template <typename SrcPointerT, typename DestAccessorT>
27        void copy(SrcPointerT src, DestAccessorT dest);
28
29    // Copy between accessors.
30    // Accessors are required to have appropriate correct permissions.
31    template <typename SrcAccessorT, typename DestAccessorT>
32        void copy(SrcAccessorT src, DestAccessorT dest);
33
34    // Submit different forms of kernel for execution.
35    template <typename KernelName, typename KernelType>
36        void single_task(KernelType kernel);
37
38    template <typename KernelName, typename KernelType, int Dims>
39        void parallel_for(range<Dims> num_work_items,
40                          KernelType kernel);
41
42    template <typename KernelName, typename KernelType, int Dims>
43        void parallel_for(nd_range<Dims> execution_range,
44                          KernelType kernel);
45
46    template <typename KernelName, typename KernelType, int Dims>
47        void parallel_for_work_group(range<Dims> num_groups,
48                                      KernelType kernel);
49
50    template <typename KernelName, typename KernelType, int Dims>

```

```

51 void parallel_for_work_group(range<Dims> num_groups,
52                             range<Dims> group_size,
53                             KernelType kernel);
54 };

```

除了调用 handler 类的成员来提交设备代码之外，还有允许提交工作的 queue 类成员。图 2-20 表示 queue 类成员是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。

图 2-20 queue 类中成员函数的定义，该成员函数充当 handler 类中等效函数的快捷方式

```

1 class queue {
2 public:
3     // Submit a memset operation writing to the specified pointer.
4     // Return an event representing this operation.
5     event memset(void *ptr, int value, size_t count)
6
7     // Submit a memcpy operation copying from src to dest.
8     // Return an event representing this operation.
9     event memcpy(void *dest, const void *src, size_t count);
10
11    // Submit different forms of kernel for execution.
12    // Return an event representing the kernel operation.
13    template <typename KernelName, typename KernelType>
14        event single_task(KernelType kernel);
15
16    template <typename KernelName, typename KernelType, int Dims>
17        event parallel_for(range<Dims> num_work_items,
18                           KernelType kernel);
19
20    template <typename KernelName, typename KernelType, int Dims>
21        event parallel_for(nd_range<Dims> execution_range,
22                           KernelType kernel);
23
24    // Submit different forms of kernel for execution.
25    // Wait for the specified event(s) to complete
26    // before executing the kernel.
27    // Return an event representing the kernel operation.
28    template <typename KernelName, typename KernelType>
29        event single_task(const std::vector<event>& events,
30                           KernelType kernel);
31
32    template <typename KernelName, typename KernelType, int Dims>
33        event parallel_for(range<Dims> num_work_items,
34                           const std::vector<event>& events,
35                           KernelType kernel);
36
37    template <typename KernelName, typename KernelType, int Dims>
38        event parallel_for(nd_range<Dims> execution_range,
39                           const std::vector<event>& events,

```

```
40     KernelType kernel);  
41 };
```

执行机制

图 2-18 中的代码包含一个 parallel_for，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) 中，队列定义了要在其上执行工作的设备。在命令组中，有两类代码：

1. 一项行动只有一次调用，设备代码将排队等待执行，或执行手动内存操作 (如复制)。
2. 主机代码会设置依赖项，这些依赖项定义了代码时何时开始执行 (1) 是安全的，例如：创建缓冲区的访问器 (在第 3 章介绍)。

handler 类包含一组成员函数，这些函数定义了在执行任务图节点时要执行的操作。图 2-21 对这些操作进行了总结。

图 2-21 中只有一个动作可以在命令组中调用 (调用多个是错误的)，而且每个提交调用只能将一个命令组提交到队列中。这样，图 2-21 中的单个操作存在于每个任务图节点上，当节点依赖关系满足，且运行时安全时，就可以执行了。

命令组中只能有一个操作，例如：内核启动或显式内存操作。

代码异步执行，这就是主机程序一部分运行在 CPU 上和满足依赖关系时运行的设备代码之间的关键区别。命令组通常包含来自每种类别的代码，其中的代码定义了作为主机程序运行的依赖项 (以便运行时知道这些依赖项是什么)，以及在满足这些依赖项后在将要运行的设备代码。

图 2-21 调用设备代码或执行显式内存的操作

工作类型	行为 (handler 类的成员函数)	总结
设备代码执行	single_task	执行一个设备函数的单例
	parallel_for	有多种形式可用来启动不同组合的设备代码
	parallel_for_work_group	使用层次并行启动一个内核，在第 4 章介绍
显式内存操作	copy	访问器、指针和/或 shared_ptr 指定的位置之间复制数据。复制作为 DAG 的一部分发生，包括依赖性跟踪。
	update_host	触发缓存对象在主机端的备份更新。
	fill	将内存中的数据初始化为指定的值。

图 2-22 提交设备代码

```

#include <CL/sycl.hpp>
#include <array>
#include <iostream>
using namespace sycl;

int main() {
    constexpr int size = 16;
    std::array<int, size> data;
    buffer B{ data };

    queue Q{}; // Select any device for this queue

    std::cout << "Selected device is: " <<
        Q.get_device().get_info<info::device::name>() << "\n";

    Q.submit([&] (handler& h) {
        accessor acc{B, h};

        h.parallel_for(size, [=] (auto& idx) {
            acc[idx] = idx;
        });
    });

    return 0;
}

```

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，并向队列提交任务，在任务图中形成新的节点，以便异步执行。
2. 命令组中的主机代码：代码在执行主机代码的处理器上运行，并在 submit 操作返回前执行，例如：代码通过创建访问器来设置节点依赖关系。任何的 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关系的代码。
3. 执行机制：图 2-21 中列出的任何命令都可以包含在命令组中，它定义了在节点满足要求（由 (2) 设置）时要异步执行的工作。

为了理解应用程序中的代码何时会运行，注意图 2-21 中列出的启动设备代码的动作，以及图 2-21 中列出的显式内存操作，都会在 DAG 节点依赖满足时异步执行，所有代码会作为主程序的一部分立即运行。

后备队列机制

通常，命令组在提交的命令队列上执行。某些情况下，命令组提交到队列会失败（例如，当请求的工作大小超过设备的限制），或者成功提交的操作无法执行（例如，当硬件设备发生故障）。要处理这种情况，可以为执行的命令组指定后备队列。不过，不推荐这种错误管理方式，因为可控性太低，更建议捕捉和管理这些问题，会在第 5 章中进行介绍。这里简要介绍下后备队列，因为其在 SYCL 中比较有名，所以有些人喜欢这种方式。

这种类型的回退，适用于设备队列会提交失败的机器，但这不是解决加速器不存在问题的机制。在没有 GPU 设备的系统上，图 2-23 中的程序会在 Q 声明（尝试构造）处抛出一个错误，指示“请求的设备类型不可用”。

基于现有设备的后备队列将在第 12 章中进行介绍。

图 2-23 后备队列的例子

```
1 #include <CL/sycl.hpp>
2 #include <array>
3 #include <iostream>
4 using namespace sycl;
5
6 int main() {
7     constexpr int global_size = 16;
8     constexpr int local_size = 16;
9     buffer<int,2> B{ range{ global_size , global_size } };
10
11    queue gpu_Q{ gpu_selector{} };
12    queue host_Q{ host_selector{} };
13
14    nd_range NDR {
15        range{ global_size , global_size },
16        range{ local_size , local_size } ;
17
18    gpu_Q.submit([&](handler& h){
19        accessor acc{B, h};
20        h.parallel_for( NDR , [=](auto id) {
21            auto ind = id.get_global_id();
22            acc[ind] = ind[0] + ind[1];
23        });
24    }, host_Q); /** <== Fallback Queue Specified **/
25
26    host_accessor acc{B};
27    for(int i=0; i < global_size; i++){
28        for(int j = 0; j < global_size; j++){
29            if( acc[i][j] != i+j ) {
30                std::cout<<"Wrong result\n";
31                return 1;
32            }
33        std::cout<<"Correct results\n";
34    }
35}
```

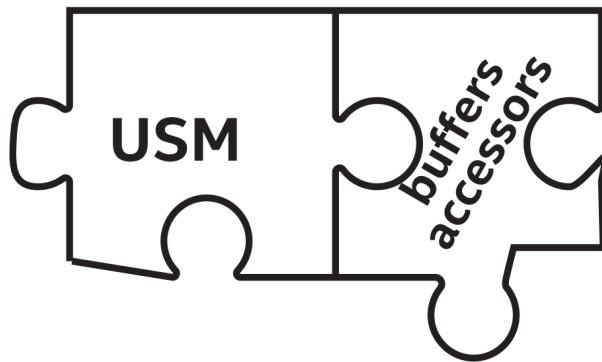
由于工作组请求的大小，图 2-23 所示的代码会在一些 GPU 上执行失败。可以指定一个后备队列作为 submit 函数的参数，如果命令组无法进入主队列，则使用后备队列（本例中是主机设备）。

通过向提交调用传递后备队列来启用后备队列。不过，建议对捕获的初始错误进行处理，而不是使用提供较少可控性的后备队列机制。

总结

本章中，概述了队列、选择与队列相关联的设备，以及如何创建自定义设备选择器。还了解了，如何在设备上异步执行的代码，以及设备代码在主机端如何执行。第 3 章中来了解一下如何控制数据移动。

3 数据管理



超级计算机的架构师们经常哀叹，我们需要“喂养野兽”。“喂养野兽”指的是，向大量并行的计算机的“野兽”喂数据。

在异构机器上进行数据并行，需要确保在需要时数据出现在正确的地方。大程序中，会有很多工作，整理如何管理所有需要的数据移动可能是一场噩梦。

我们将解释管理数据的两种方法：统一共享内存（Unified Shared Memory, USM）和缓冲区。USM 基于指针，C++ 开发者对它很熟悉。缓冲区提供了更高级别的抽象。

我们需要控制数据的移动，本章将介绍实现该目标的方式。

第 2 章中，我们研究了如何控制代码的执行位置。代码需要数据作为输入，生成数据作为输出。因为代码可能运行在多个设备上，而这些设备不一定共享内存，所以需要管理数据移动。即使共享数据，比如 USM，同步和一致性也需要管理。

一个问题是“为什么编译器不自动地做所有的事情？”，编译器可以处理很多事情，但如果把自己定位为开发者，性能通常是次优的，所以编译器没必要做所有的事情。在实践中，为了获得最佳性能，编写异构程序时，需要关注代码执行（第 2 章）和数据移动（本章）。

本章提供了管理数据的介绍，包括数据使用的顺序。前一章展示了如何控制代码的运行位置，本章将让数据在我们要求代码执行的地方出现，这不仅对正确执行应用程序很重要，而且也可以最小化执行时间和功耗。

介绍

没数据，不计算。

加速计算的意义在于更快地得出答案，数据并行计算最重要的是如何访问数据，而在机器中引入加速器设备将使情况复杂化。传统的单 CPU 系统中，只有一个内存。加速器设备通常有自己的内存，这些内存主机不能直接访问。因此，支持独立设备的并行编程模型必须管理这些多内存，以及移动数据的机制。

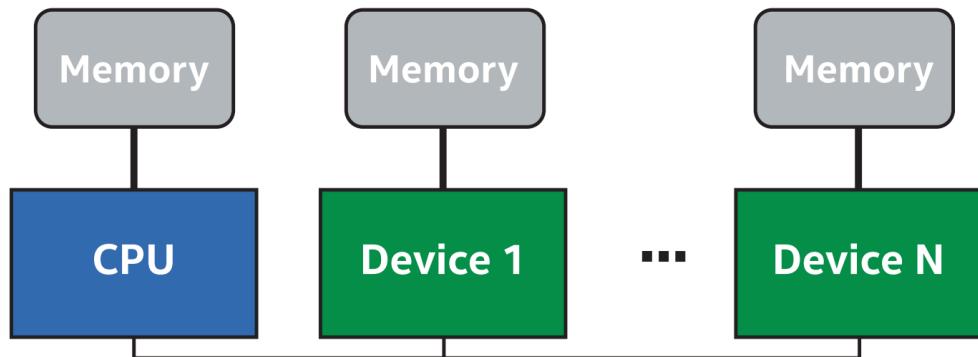
本章中，我们将概述数据管理的各种机制。引入统一共享内存和用于数据管理的缓冲区，并描述了内核执行和数据移动之间的关系。

数据管理的问题

用于并行编程的共享内存模型的优点是提供单一、共享的内存。我们不需要显式的从并行任务中访问内存（除了适当的同步以避免数据竞争）。当某些类型的加速器设备（例如集成 GPU）与主机

CPU 共享内存时，许多独立加速器有自己的内存，不与 CPU 的内存在一起，如图 3-1 所示。

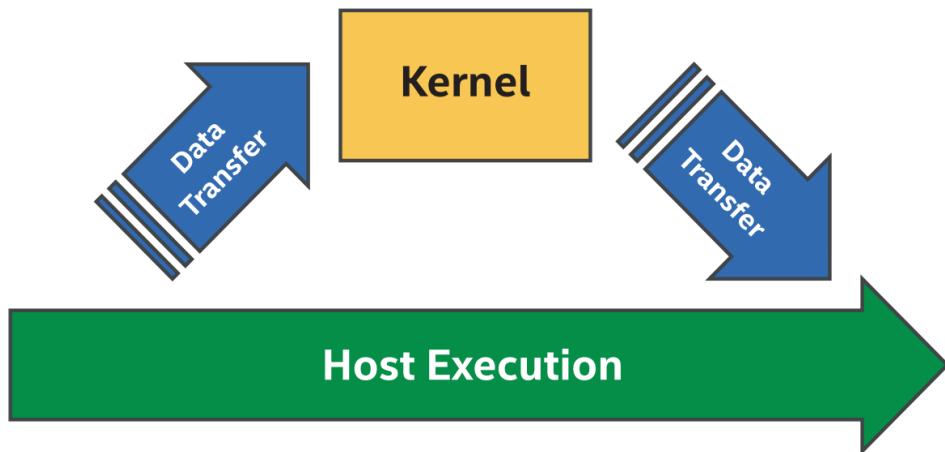
图 3-1 多个独立的内存



本地设备和远程设备

运行在设备上的程序直接使用到设备上的内存，要比远程内存读写数据时性能会更好。我们将直接访问独立内存称为本地访存，访问其他设备的内存是远程访存。远程访存往往比本地访存慢，因为需要以更低的带宽和/或更高的延迟在数据链路上进行传输，所以将计算和数据放在一起有利与计算。为了实现这一点，必须以某种方式确保数据在不同的内存间可以复制或迁移，以便将其移动到更接近计算发生的位置。

图 3-2 数据移动和内核执行



管理多种内存

管理多种内存可以通过两种方式来实现：显式地通过程序实现，或隐式地通过运行时实现。每种方法都有其优点和缺点，可以根据情况或个人喜好进行选择。

显式数据移动

显式地在不同内存之间复制数据。图 3-2 展示了一个有独立加速器的系统，必须先将内核需要的任何数据从主机内存复制到 GPU 内存。在内核计算结果之后，将这些结果复制回 CPU，然后主机才能使用这些数据。

显式数据移动的优点是，可以完全控制数据在不同内存之间的传输时间。因为要在某些硬件上获得最佳性能，将计算与数据传输重叠必不可少。

显式数据移动的缺点是，指定所有数据移动会很繁琐，而且容易出错。传输不正确的数据量，或者没有确保在内核开始计算之前已经传输了所有数据，都可能导致不正确的结果。从一开始就正确地移动所有数据是非常耗时的任务。

隐式数据移动

程序控制的显式数据移动的替代方案，由运行时或驱动程序控制的隐式数据移动。这种情况下，运行时不需要进行显式复制，而是负责确保数据在使用之前就传输到适当的位置。

隐式数据移动的优点是，应用程序直接连接到设备内存，这样会更快，所有工作都由运行时完成。这也减少了引入错误的机会，因为运行时将自动识别何时执行数据传输，以及传输多少数据。

隐式数据移动的缺点是，对运行时的隐式行为控制较少或没有控制。运行时将提供功能的正确性，但可能不会以最佳的方式移动数据，以确保计算与数据传输重叠，这可能会对程序性能产生负面影响。

选择正确的策略

选择最佳策略取决于许多的因素，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，最好的解决方案可以为程序的不同部分混合和适配显式和隐式方法。可以选择使用隐式数据移动，来简化移植到新设备的过程。当开始调优程序性能时，可能会用代码中对性能至关重要的显式部分替换隐式数据移动。未来的章节将涵盖数据传输如何与计算重叠，从而达到优化性能的目的。

统一共享内存、内存和图像

有三种管理内存的方法：统一共享内存 (USM)、缓冲区和图像。

USM 基于指针，C/C++ 开发者应该很熟悉。USM 的优点是更容易与现有 C++ 代码集成。

缓冲区（由缓冲区模板类表示）描述 1、2 或 3 维数组，提供了可以在主机或设备上访问的内存。缓冲区不直接由程序访问，而是通过访问器使用。

图像作为一种特殊的缓冲区，提供特定于图像处理的功能。这个功能包括支持特殊的图像格式，使用采样器读取图像等。缓冲区和图像是许多问题的解决方法，但现有代码中重写所有接口来使用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上是相同的，本章剩下的部分只关注 USM 和缓冲区。

统一共享内存

USM 是可供使用的数据管理工具。当移植大量使用指针的工程时，USM 可以简化工作。支持 USM 的设备支持统一虚拟地址空间，拥有统一虚拟地址空间意味着主机上的 USM 返回的指针都是设备上的有效指针。不需要手动转换主机指针来获得“设备指针”——可以在主机和设备上看到相同的指针。

关于 USM 更详细的解读会在第 6 章继续。

通过指针访问内存

当系统包含主机内存和设备内存时，不是所有的内存都相同，所以 USM 定义了三种不同的分配类型：设备、主机和共享。所有类型的分配都在主机上执行。图 3-3 总结了各分配类型的特点。

图 3-3 USM 分配类型

分配类型	描述	可访问主机？	可访问设备？	位于
设备	设备内存的分配	✗	✓	设备
主机	主机内存的分配	✓	✓	主机
共享	主机和设备共享	✓	✓	随意迁移

设备分配发生在设备内存中，分配的内存可以从设备上读取和写入，但不能直接从主机上访问。必须使用显式的复制操作，在主机内存和设备内存之间移动数据。

主机和设备上都可以访问主机内存，这意味着相同的指针在主机代码和设备内核中都有效。然而，访问这样的指针时，数据总是来自主机内存。当访问设备时，数据不会从主机迁移到设备内存。相反，数据通常通过总线发送，例如 PCI-Express (PCI-E) 将设备连接到主机。

共享分配的内存存在主机和设备上都可以访问，非常类似于主机分配，不同之处在于数据可以在主机内存和设备本地内存之间迁移。迁移之后，对设备的访问将在设备内存中进行，而不是远程访问主机内存。通常，这是通过运行时内部的机制和底层驱动实现。

USM 和数据移动

USM 支持显式和隐式的数据移动策略，不同的分配类型对应不同的策略。设备内存要求显式地在主机和设备之间移动数据，而主机和共享内存提供隐式的数据移动。

USM 显式数据移动

使用 USM 的显式数据移动，通过设备内存和在队列和处理程序中使用特殊的 `memcpy()` 完成的。将 `memcpy()` 操作（动作）放入队列，将数据从主机传输到设备，或从设备传输到主机。

图 3-4 包含操作设备分配的内核。内核执行前后使用 `memcpy()` 操作，`hostArray` 和 `deviceArray` 之间复制数据。队列上调用 `wait()`，确保在内核执行之前复制到设备的操作已经完成，并确保数据复制回主机之前内核已经完成。我们将在本章后面学习如何消除这些调用。

图 3-4 显式地数据移动

```
1 #include <CL/sycl.hpp>
2 #include<array>
3 using namespace sycl;
4 constexpr int N = 42;
5
6 int main() {
7     queue Q;
```

```

8
9 std::array<int, N> host_array;
10 int *device_array = malloc_device<int>(N, Q);
11
12 for (int i = 0; i < N; i++)
13     host_array[i] = N;
14
15 // We will learn how to simplify this example later
16 Q.submit([&](handler &h) {
17     // copy hostArray to deviceArray
18     h.memcpy(device_array, &host_array[0], N * sizeof(int));
19 });
20 Q.wait();
21
22
23 Q.submit([&](handler &h) {
24     h.parallel_for(N, [=](id<1> i) { device_array[i]++; });
25 });
26 Q.wait();
27
28 Q.submit([&](handler &h) {
29     // copy deviceArray back to hostArray
30     h.memcpy(&host_array[0], device_array, N * sizeof(int));
31 });
32 Q.wait();
33
34 free(device_array, Q);
35 return 0;
36 }

```

USM 隐式数据移动

使用 USM 的隐式数据移动是通过主机和共享内存完成的。使用这类型的内存，不需要显式地插入复制操作。相反，**只需访问内核中的指针，任何数据移动都可以自动执行**，无需手动干预（只要设备支持这些内存分配）。这提高了代码的移植性：只需用适当的 USM 分配函数，替换 malloc 或 new(以及 free)，一切就好了。

图 3-5 USM 隐式地数据移动

```

1 #include <CL/sycl.hpp>
2 using namespace sycl;
3 constexpr int N = 42;
4
5 int main() {
6     queue Q;
7     int *host_array = malloc_host<int>(N, Q);
8     int *shared_array = malloc_shared<int>(N, Q);
9

```

```

10  for (int i = 0; i < N; i++) {
11      // Initialize hostArray on host
12      host_array[i] = i;
13  }
14
15  // We will learn how to simplify this example later
16 Q.submit([&](handler &h) {
17     h.parallel_for(N, [=](id<1> i) {
18         // access sharedArray and hostArray on device
19         shared_array[i] = host_array[i] + 1;
20     });
21 });
22 Q.wait();
23
24 for (int i = 0; i < N; i++) {
25     // access sharedArray on host
26     host_array[i] = shared_array[i];
27 }
28
29 free(shared_array, Q);
30 free(host_array, Q);
31 return 0;
32 }

```

图 3-5 中，创建了两个数组，hostArray 和 sharedArray，分别是主机内存和共享内存。虽然主机和共享内存都可以在主机代码中访问，但这里只初始化了 hostArray。类似地，可以在内核直接访问，执行远程读取数据。运行时确保 sharedArray 在内核访问之前，数据在设备上是可用的，并且在之后主机代码读取时会回移数据，这些都不需要开发者干预。

内存

为数据管理提供的另一个方法的是缓冲区对象。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。缓冲区对象可以是标量数据类型（如 int、float 或 double）、向量数据类型（第 11 章）或用户定义的类或结构。缓冲区中的数据结构必须可复制，这意味着可以安全地逐个字节地复制对象，而不需要调用复制构造函数。

虽然缓冲区本身是单个实例，但缓冲区封装的 C++ 类型可以是包含多个对象的数组。缓冲区代表的是数据对象，而不是特定的内存地址，所以不能像常规 C++ 数组那样直接访问。实际上，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，甚至是同一个设备上的多个内存位置。**而我们只能使用访问器，来读取和写入缓冲区。**

第 7 章有更多关于缓冲区的描述。

创建缓冲区

可以通过多种方式创建缓冲区。最简单的方法是构造新的缓冲区，指定缓冲区的大小。然而，以这种方式创建的缓冲区并不会初始化数据，试图从缓冲区中读取数据之前，必须通过其他方式初始化缓冲区。

还可以以主机上的现有数据创建缓冲区。通过调用构造函数来实现，构造函数接受指向主机的内存指针、一组输入迭代器或具有某些属性的容器。构造缓冲区的过程中，数据从现有的主机内存复制到缓冲区对象的主机内存中。如果在 OpenCL 中使用 SYCL 互操作特性，也可以使用现有的 cl_mem 实例创建缓冲区。

访问缓存

主机和设备不能直接访问缓冲区 (除非通过这里没有描述的高级和不经常使用的机制)，而必须创建访问器来读取和写入缓冲区。访问器向运行时提供如何使用缓冲区中的数据信息，从而允许运行时正确地进行数据移动。

图 3-6 缓冲区和访问器

```
1 #include <CL/sycl.hpp>
2 #include <array>
3 using namespace sycl;
4 constexpr int N = 42;
5
6 int main() {
7     std::array<int, N> my_data;
8     for (int i = 0; i < N; i++)
9         my_data[i] = 0;
10
11    {
12        queue q;
13        buffer my_buffer(my_data);
14
15        q.submit([&](handler &h) {
16            // create an accessor to update
17            // the buffer on the device
18            accessor my_accessor(my_buffer, h);
19
20            h.parallel_for(N, [=](id<1> i) {
21                my_accessor[i]++;
22            });
23        });
24
25        // create host accessor
26        host_accessor host_accessor(my_buffer);
27        for (int i = 0; i < N; i++) {
28            // access myBuffer on host
29            std::cout << host_accessor[i] << " ";
30        }
31        std::cout << "\n";
32    }
33
34    // myData is updated when myBuffer is
35    // destroyed upon exiting scope
```

```

36 for (int i = 0; i < N; i++) {
37     std::cout << my_data[i] << " ";
38 }
39 std::cout << "\n";
40 }
```

图 3-7 缓冲区的访问模式

访问模式	描述
read	只能读
write	只写访问 (以前的内容不丢弃)
read_write	可读写

访问模式

创建访问器时，可以通知运行时来提供更多的优化信息，可以通过指定访问模式来实现这一点。图 3-7 中的 Access::mode enum 中定义了访问模式。图 3-6 所示的代码中，访问器 myAccessor 以默认的访问模式创建，access::mode::read_write，这让运行时知道我们打算通过 myAccessor 对缓冲区进行读写操作。访问模式是运行时优化隐式数据移动的方式，例如：使用 access::mode::read 会在内核开始执行之前，需要数据在设备上可用。如果内核只通过访问器读取数据，在内核完成后就不需要将数据复制回主机。同样，access::mode::write 让运行时知道我们将修改缓冲区的内容，并且需要在计算结束后将结果复制回来。

创建适当模式的访问器，可以向运行时提供更多关于如何在程序中使用数据的信息。运行时使用访问器对数据的使用进行排序，也可以使用这些数据来优化内核调度和数据移动。第 7 章中，将继续讨论访问模式和优化标记。

对数据进行排序

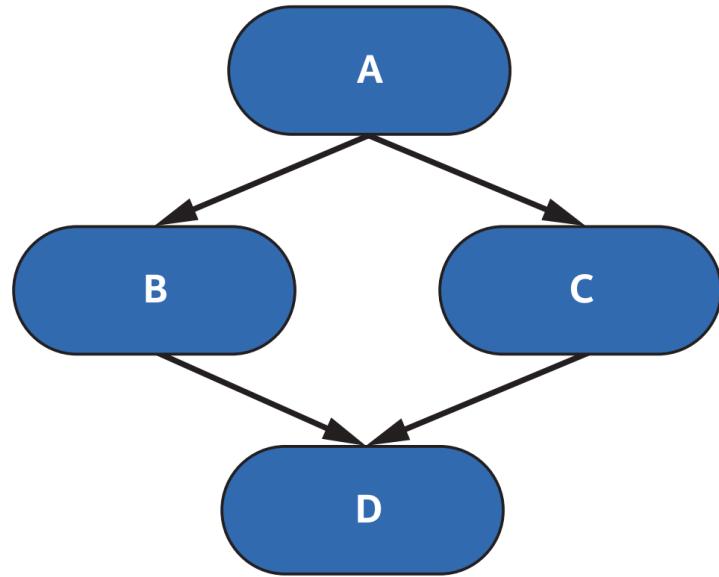
内核可以视为提交执行的异步任务。这些任务必须提交到一个队列中，并安排在设备上执行。许多情况下，内核必须按照特定的顺序执行，才能计算出正确的结果。如果获得正确的结果需要任务 A 在任务 B 之前执行，那么任务 A 和 B 之间存在依赖关系。

然而，内核并不是调度的唯一形式。内核开始执行前，内核访问的任何数据都需要在设备上可用。这些数据依赖可以以数据任务的方式，从一个设备传输到另一个设备。数据传输任务可以是显式编码的拷贝操作，也可以是运行时执行的隐式数据移动。

如果把程序中的所有任务，以及存在的依赖关系画出来，就可以形成一个图。这个任务图是个有向无环图 (DAG)，其中节点是任务，边是依赖项。图是有向的，因为依赖是单向的：任务 A 必须发生在任务 B 之前。因为不包含从节点返回自身的循环或路径，所以无环。

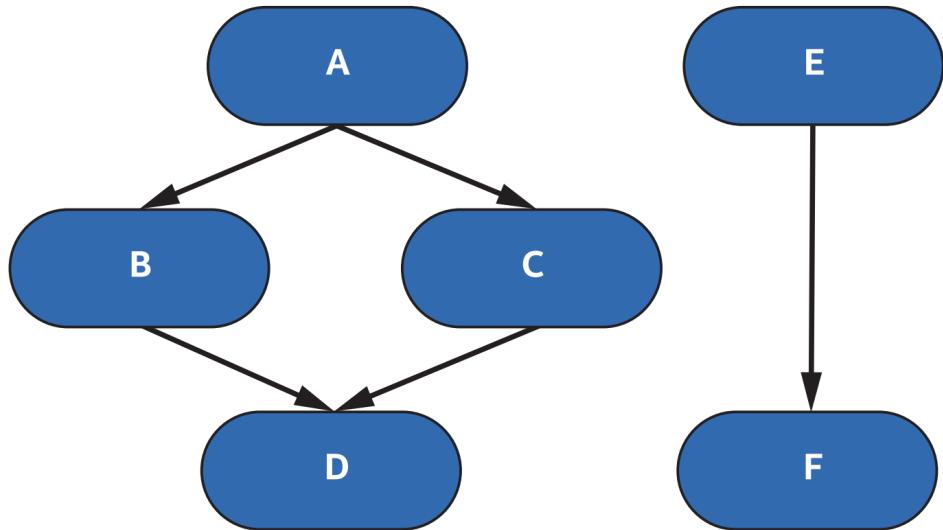
图 3-8 中，A 任务前必须执行任务 B 和 C。同样地，B 和 C 之前必须在 D 之前执行。而 B 和 C 没有依赖，运行时是可以以任何顺序执行（甚至并行）任务。因此，如果 B 和 C 能够同时执行，则这个图可能的法律次序是：A \Rightarrow B \Rightarrow C \Rightarrow D, A \Rightarrow C \Rightarrow B \Rightarrow D，甚至时 A \Rightarrow {B,C} \Rightarrow D。

图 3-8 简单的任务图



任务可能与所有任务的子集有依赖性，我们只指定与正确性有关的依赖项。这种灵活性为优化任务图的执行顺序提供了自由度。图 3-9 中，我们扩展了任务图图 3-8，添加了 E 和 F，并且 E 必须在 F 前执行。然而，任务 E 和 F 与节点 A,B,C,D 没有依赖性。这允许运行时可以选择序执行任务的顺序。

图 3-9 不相交依赖关系的任务图



有两种不同的方法来为任务的执行（例如内核的启动）建模：队列可以按照提交的顺序执行任务，也可以按照自定义的依赖项的任意顺序执行任务。我们有几种机制来定义正确排序所需的依赖项。

有序队列

对任务进行排序的最简单方式是将它们提交给一个有序的队列对象。有序队列按照任务提交的顺序执行任务，如图 3-10 所示。尽管有序队列的任务排序非常简单，它的缺点是即使独立任务之间不存在依赖关系，任务的执行也将串行化。有序队列在启动应用程序时很有用，因为简单、直观、执行顺序确定，并且适用于许多代码。

图 3-10 有序队列的使用方式

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N = 4;

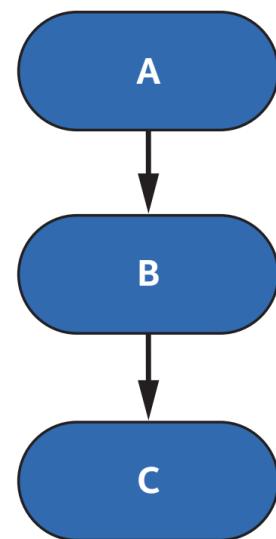
int main() {
    queue Q{property::queue::in_order()};

    // Task A
    Q.submit([&] (handler& h) {
        h.parallel_for(N, [=] (id<1> i) { /*...*/ });
    });

    // Task B
    Q.submit([&] (handler& h) {
        h.parallel_for(N, [=] (id<1> i) { /*...*/ });
    });

    // Task C
    Q.submit([&] (handler& h) {
        h.parallel_for(N, [=] (id<1> i) { /*...*/ });
    });

    return 0;
}
```



无序队列

由于队列是无序队列 (除非使用有序队列属性创建)，必须提供提交任务进行的排序方法。队列允许通知运行时任务间的依赖关系，从而对任务进行排序。并且，可以使用命令组显式或隐式地指定这些依赖项。

命令组是指定任务及其依赖关系的对象。命令组通常以 C++ Lambda 的形式，作为参数传递给队列对象的 submit()。这里 Lambda 的唯一参数是对 handler 对象的引用。handler 对象在命令组中用于指定操作、创建访问器和指定依赖项。

事件的显式依赖

任务之间的显式依赖关系就像我们已经看到的例子 (图 3-8)，其中任务 A 必须在任务 B 之前执行，通过这种方式表达的依赖关系是显式的，并且基于计算，而不是数据。注意，表达计算之间的依赖关系，主要与使用 USM 有关，使用缓冲区的代码通过访问器表达大多数依赖关系。图 3-4

和图 3-5 中，只是告诉队列等待之前提交的所有任务完成后再继续，而我们可以通过事件对象表达任务的依赖关系。当向队列提交命令组时，submit() 方法会返回一个事件对象，事件可以以两种方式使用。

首先，可以通过在事件上显式地调用 wait() 来进行同步。这迫使运行时等待生成事件的任务完成，才继续执行主机程序。显式地等待事件对于调试应用程序非常有用，但 wait() 会过度地限制任务的异步执行，因为阻塞主机线程上的所有执行，也可以在队列对象上调用 wait()，这将阻塞主机上的执行，直到所有进入队列的任务都完成。

这就引出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组，并通知运行时所提交的命令组需要在执行命令组内的操作之前完成指定的事件。图 3-11 展示了如何使用 depends_on() 来排序任务。

图 3-11 使用事件和 depends_on

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N = 4;

int main() {
    queue Q;

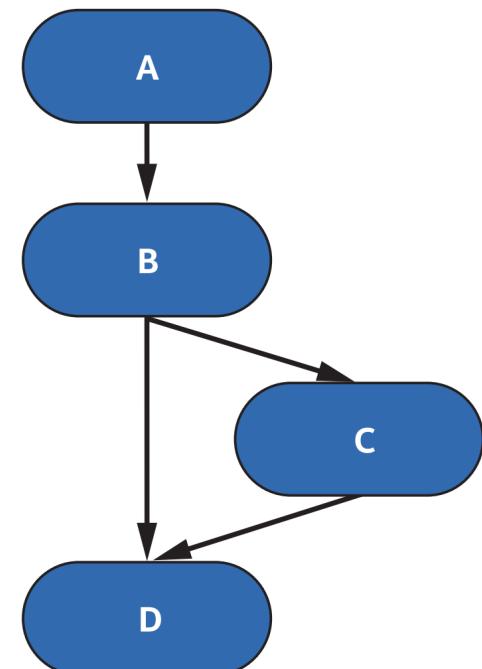
    // Task A
    auto eA = Q.submit([&] (handler &h) {
        h.parallel_for(N, [=] (id<1> i) { /*...*/ });
    });
    eA.wait();

    // Task B
    auto eB = Q.submit([&] (handler &h) {
        h.parallel_for(N, [=] (id<1> i) { /*...*/ });
    });

    // Task C
    auto eC = Q.submit([&] (handler &h) {
        h.depends_on(eB);
        h.parallel_for(N, [=] (id<1> i) { /*...*/ });
    });

    // Task D
    auto eD = Q.submit([&] (handler &h) {
        h.depends_on({eB, eC});
        h.parallel_for(N, [=] (id<1> i) { /*...*/ });
    });

    return 0;
}
```



访问器的隐式依赖

任务之间的隐式依赖关系由数据依赖关系创建，任务之间的数据依赖有三种形式，如图 3-12 所示。

图 3-12 三种的数据依赖关系

依赖类型	描述
Read-after-Write (RAW)	任务 B 需要读取任务 A 计算的数据
Write-after-Read (WAR)	任务 A 读取数据之后，任务 B 写入数据
Write-afterWrite(WAW)	任务 A 写入的数据后，任务 B 写入数据

数据依赖关系以两种方式表示：访问器和执行顺序。两者都必须用于运行时，以正确计算数据依赖关系。如图 3-13 和 3-14 所示。

图 3-13 读后写

```

1 #include <CL/sycl.hpp>
2 #include <array>
3 using namespace sycl;
4 constexpr int N = 42;
5
6 int main() {
7     std::array<int,N> a, b, c;
8     for (int i = 0; i < N; i++) {
9         a[i] = b[i] = c[i] = 0;
10    }
11
12    queue Q;
13
14    // We will learn how to simplify this example later
15    buffer A{a};
16    buffer B{b};
17    buffer C{c};
18
19    Q.submit([&](handler &h) {
20        accessor accA(A, h, read_only);
21        accessor accB(B, h, write_only);
22        h.parallel_for( // computeB
23            N,
24            [=](id<1> i) { accB[i] = accA[i] + 1; });
25    });
26
27    Q.submit([&](handler &h) {
28        accessor accA(A, h, read_only);
29        h.parallel_for( // readA
30            N,
31            [=](id<1> i) {
32                // Useful only as an example
33                int data = accA[i];
34            });
35    });

```

```

36
37 Q.submit([&]( handler &h) {
38     // RAW of buffer B
39     accessor accB(B, h, read_only);
40     accessor accC(C, h, write_only);
41     h.parallel_for( // computeC
42         N,
43         [=](id<1> i) { accC[i] = accB[i] + 2; });
44 });
45
46 // read C on host
47 host_accessor host_accC(C, read_only);
48 for (int i = 0; i < N; i++) {
49     std::cout << host_accC[i] << " ";
50 }
51 std::cout << "\n";
52 return 0;
53 }

```

图 3-14 原始任务图

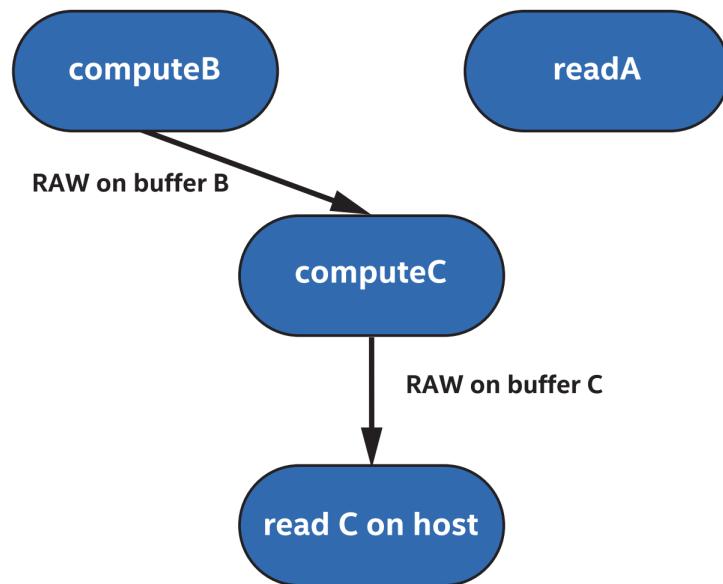


图 3-13 和图 3-14 中，我们执行三个内核——computeB、readA 和 computeC——然后在主机上读取最终结果。内核 computeB 的命令组创建两个访问器，accA 和 accB。这些访问器使用访问标记 read_only 和 write_only 进行优化，以指定不使用默认的访问模式 access::mode::read_write。内核 computeB 读取缓冲区 A 并写入缓冲区 B，缓冲区 A 必须在内核开始执行之前从主机复制到设备上。

内核 readA 也为缓冲区 A 创建一个只读访问器。因为内核 readA 是在内核 computeB 后提交的，所以这会创建一个读后读的场景。然而，读后读并没有对运行时进行限制，内核可以自由地以任何顺序执行。事实上，运行时更喜欢在内核 computeB 之前执行内核 readA，甚至同时执行。两者都需要将缓冲区 A 复制到设备上，但是内核 computeB 也需要复制缓冲区 B，以确保 computeB

的数据覆盖相应的缓冲区。当缓冲区 B 的数据传输时，运行时可以执行内核读取，即使内核只会写入缓冲区，缓冲区的原始内容仍然可移动到设备上，因为不能保证缓冲区中的所有值都由内核修改（参见第 7 章，关于优化标记）。

内核 computeC 读取缓冲区 B，这是内核 computeB 计算的结果。在提交内核 computeB 之后，提交了内核 computeC，这样内核 computeC 对缓冲区 B 有数据依赖。数据依赖也称为真依赖或流依赖，因为数据需要从计算流到另一个计算，从而计算出正确的结果。因为主机希望在内核完成后读取 C，所以还在内核 computeC 和主机之间创建了对缓冲区 C 的依赖，这迫使运行时将缓冲区 C 复制回主机。由于设备上没有对缓冲区 A 的写操作，因为主机已经有了最新的数据副本，所以运行时不需要将缓冲区复制回主机。

图 3-15 写后读和写后写

```
1 #include <CL/sycl.hpp>
2 #include <array>
3 using namespace sycl;
4 constexpr int N = 42;
5
6 int main() {
7     std::array<int, N> a, b;
8     for (int i = 0; i < N; i++) {
9         a[i] = b[i] = 0;
10    }
11
12    queue Q;
13    buffer A{a};
14    buffer B{b};
15
16    Q.submit([&](handler &h) {
17        accessor accA(A, h, read_only);
18        accessor accB(B, h, write_only);
19        h.parallel_for( // computeB
20            N, [=](id<1> i) {
21                accB[i] = accA[i] + 1;
22            });
23        });
24
25    Q.submit([&](handler &h) {
26        // WAR of buffer A
27        accessor accA(A, h, write_only);
28        h.parallel_for( // rewriteA
29            N, [=](id<1> i) {
30                accA[i] = 21 + 21;
31            });
32        });
33
34    Q.submit([&](handler &h) {
35        // WAV of buffer B
```

```

36     accessor accB(B, h, write_only);
37     h.parallel_for( // rewriteB
38     N, [=](id<1> i) {
39         accB[i] = 30 + 12;
40     });
41 });
42
43 host_accessor host_accA(A, read_only);
44 host_accessor host_accB(B, read_only);
45 for (int i = 0; i < N; i++) {
46     std::cout << host_accA[i] << " " << host_accB[i] << "\n";
47 }
48 std::cout << "\n";
49 return 0;
50 }

```

图 3-16 写后读和写后写的任务图

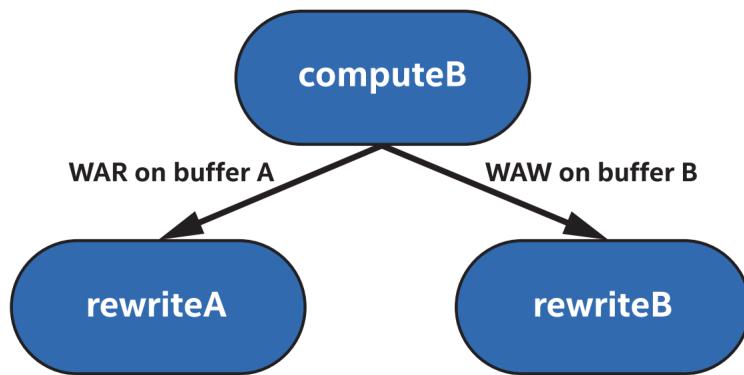


图 3-15 和 3-16 中，再次执行三个内核:computeB、rewriteA 和 rewriteB。内核 rewriteA 写缓冲区 A，内核 rewriteB 写缓冲区 B。内核 rewriteA 理论上可以比内核 rewriteB 更早执行，在内核准备好之前需要传输的数据更少。但必须等到内核 computeB 完成后，因为有一个写后读依赖于缓冲区 A。

这个例子中，内核 computeB 需要从主机获取 A 的原始值，如果内核 rewriteA 在内核 computeB 之前执行，那将读取错误的值。写后读依赖也称为反依赖，原始依赖关系确保数据正确地流向正确的方向，而写后读依赖关系确保在读取现有值之前不会覆盖。内核重写函数中，写后写对缓冲区 B 的依赖与此类似。如果在内核 computeB 和 rewriteB 之间提交了任何对缓冲区 B 的读取，将形成读后写和写后读的依赖关系，从而正确地排序任务。然而，内核 rewriteB 和主机之间存在隐式的依赖关系，最终的数据必须写回主机。写后写依赖关系，也称为输出依赖关系，确保最终的输出数据在主机上的正确性。

选择管理策略

为程序选择正确的数据管理策略很大程度上是偏好的问题。事实上，可以从一种策略开始，然后随着项目的成熟而转向另一种策略。这里有一些指导方针，可以帮助我们选择符合需要的策略。

首先是使用显式数据移动还是隐式数据移动，这极大地影响了对程序进行的操作。隐式数据移动通常更容易，因为所有数据移动都是隐式处理的，从而让我们专注于计算。

如果从一开始就完全控制所有的数据移动，使用 USM 设备分配的显式数据移动就是不错的选择。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。

选择隐式数据移动策略时，仍然可以选择是否使用缓冲区或 USM 主机或共享指针。如果正在移植一个使用指针的 C/C++ 程序，USM 是更简单的方式，无需修改大多数的代码。如果数据表示没有引导我们选择一个策略，则可以问的另一个问题，希望如何表达内核之间的依赖关系。如果更愿意考虑内核之间的数据依赖关系，那么选择缓冲区。如果倾向于把依赖关系看作是在另一个计算之前执行一个计算，并且想使用一个有序队列或显式事件或内核之间的等待来表示依赖关系，那么选择 USM。

使用 USM 指针（显式或隐式数据移动）时，可以选择想要使用类型的队列。有序队列简单直观，但限制了运行时，并可能限制性能。无序队列更复杂，但给了运行时更多的自由来重新排序和重叠执行。如果程序在内核之间有复杂的依赖关系，那么无序队列类是正确的选择。如果程序只是一个接一个地运行多个内核，那么有序队列将是更好的选择。

句柄类：关键成员

我们已经展示了许多使用 handler 类的方法。图 3-17 和图 3-18 更详细地解释了这个非常重要的类的关键成员。我们目前还没有使用所有的成员，后续会对它们进行使用。

另一个 queue 类在第 2 章的末尾也有类似的解释，在线 oneAPI DPC++ 语言手册提供了对这两个类更详细的解释。

图 3-17 简化定义 handler 类的非访问器成员

```
1 class handler {
2 ...
3 // Specifies event(s) that must be complete before the action
4 // defined in this command group executes.
5 void depends_on({event / std::vector<event> & });
6
7 // Enqueues a memset operation on the specified pointer.
8 // Writes the first byte of Value into Count bytes.
9 // Returns an event representing this operation.
10 event memset(void *Ptr, int Value, size_t Count);
11
12 // Enqueues a memcpy from Src to Dest.
13 // Count bytes are copied.
14 // Returns an event representing this operation.
15 event memcpy(void *Dest, const void *Src, size_t Count);
16
17 // Submits a kernel of one work-item for execution.
18 // Returns an event representing this operation.
19 template <typename KernelName, typename KernelType>
20 event single_task(KernelType KernelFunc);
```

```

22 // Submits a kernel with NumWork-items work-items for execution.
23 // Returns an event representing this operation.
24 template <typename KernelName, typename KernelType, int Dims>
25 event parallel_for(range<Dims> NumWork-items, KernelType KernelFunc);
26
27 // Submits a kernel for execution over the supplied nd_range.
28 // Returns an event representing this operation.
29 template <typename KernelName, typename KernelType, int Dims>
30 event parallel_for(nd_range<Dims> ExecutionRange, KernelType KernelFunc);
31 ...
32 };

```

图 3-18 简化定义 handler 类的访问器成员

```

1 class handler {
2 ...
3 // Specifies event(s) that must be complete before the action
4 // Copy to/from an accessor.
5 // Valid combinations:
6 // Src: accessor, Dest: shared_ptr
7 // Src: accessor, Dest: pointer
8 // Src: shared_ptr Dest: accessor
9 // Src: pointer Dest: accessor
10 // Src: accesssor Dest: accessor
11 template <typename T_Src, typename T_Dst,
12     int Dims, access::mode AccessMode,
13     access::target AccessTarget,
14     access::placeholder IsPlaceholder =
15     access::placeholder::false_t>
16 void copy(accessor<T_Src, Dims, AccessMode,
17           AccessTarget, IsPlaceholder> Src,
18           shared_ptr_class<T_Dst> Dst);
19 void copy(shared_ptr_class<T_Src> Src,
20           accessor<T_Dst, Dims, AccessMode,
21           AccessTarget, IsPlaceholder> Dst);
22 void copy(accessor<T_Src, Dims, AccessMode,
23           AccessTarget, IsPlaceholder> Src,
24           T_Dst *Dst);
25 void copy(const T_Src *Src,
26           accessor<T_Dst, Dims, AccessMode,
27           AccessTarget, IsPlaceholder> Dst);
28 template <
29     typename T_Src, int Dims_Src,
30     access::mode AccessMode_Src,
31     access::target AccessTarget_Src,
32     typename T_Dst, int Dims_Dst,
33     access::mode AccessMode_Dst,
34     access::target AccessTarget_Dst,

```

```

35     access::placeholder IsPlaceholder_Src =
36         access::placeholder::false_t ,
37     access::placeholder IsPlaceholder_Dst =
38         access::placeholder::false_t>
39 void copy(accessor<T_Src, Dims_Src, AccessMode_Src,
40             AccessTarget_Src, IsPlaceholder_Src>
41             Src,
42             accessor<T_Dst, Dims_Dst, AccessMode_Dst,
43             AccessTarget_Dst, IsPlaceholder_Dst>
44             Dst);
45
46 // Provides a guarantee that the memory object accessed by the accessor
47 // is updated on the host after this action executes.
48 template <typename T, int Dims,
49     access::mode AccessMode,
50     access::target AccessTarget,
51     access::placeholder IsPlaceholder =
52         access::placeholder::false_t>
53 void update_host(accessor<T, Dims, AccessMode,
54                  AccessTarget, IsPlaceholder> Acc);
55 ...
56 };

```

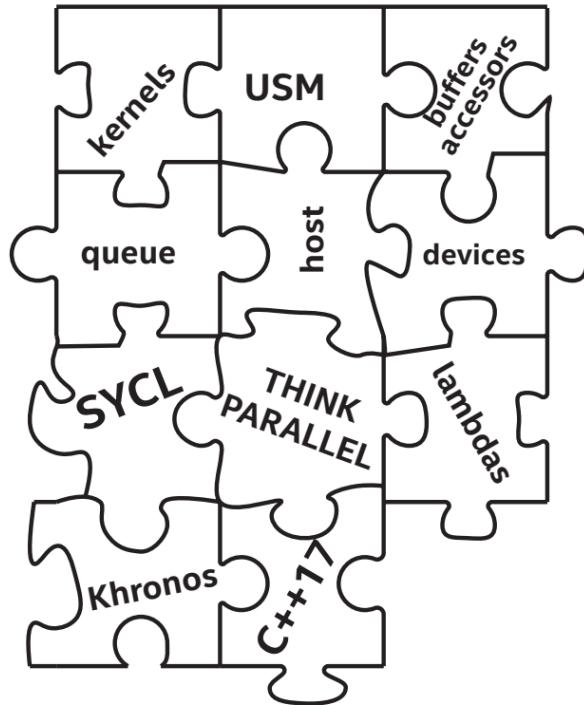
总结

本章中，介绍了解决数据管理的机制，以及如何对数据的使用进行排序。当使用加速器时，管理不同内存的访问是一个挑战，有不同的选择来满足我们的需求。

我们概述了数据使用之间可能存在的不同类型依赖关系，并描述了如何向队列提供关于这些依赖关系的信息，以便正确地对任务进行排序。

本章提供了统一共享内存和缓冲区的概述。我们将在第 6 章更详细地探讨 USM 的所有模式和行为。第 7 章将更深入地探讨缓冲区，包括创建缓冲区和控制其行为的所有不同方法。第 8 章将回顾控制内核执行顺序和数据移动的队列的调度机制。

4 并发表示



现在可以把第一批拼图拼在一起了。了解了如何在设备上执行代码（第 2 章）和数据（第 3 章）——现在必须做的就是如何处理它们。为了达到这个目的，现在补充一些容易忽略的东西。本章标志着从教学示例向实际并行代码的转变，会扩展前面章节中的代码示例的细节。

用一种新的并行语言编写第一个程序似乎很困难，特别是对并行编程的新手。语言规范不是为应用程序开发人员编写的，通常需要对术语有所了解：

- 为什么会有不止一种表达并行性的方法？
- 应该用哪种方法来表达并行性？
- 关于执行模型，需要了解多少？

本章会解决这些问题和其他问题。我们会了解数据并行内核的概念，使用工作代码示例讨论不同内核形式的优缺点，并重点介绍内核执行模型。

内核间的并行

关于执行模型，需要了解多少？近年来，**并行内核**作为一种表达数据并行性的方式出现。基于内核的方法主要为了跨各种设备的可移植性，以便提高开发者工作效率。因此，内核通常不是硬编码处理特定数量或配置硬件资源（例如：内核、硬件线程、SIMD[单指令多数据] 指令）。相反，内核用抽象的概念来描述并行性，实现（即编译器和运行时的组合）可以映射到特定目标设备上的可用硬件并行。尽管这种映射是由具体实现定义，但我们相信其会选择一种合理，且能够有效利用硬件并行性的映射。

以一种硬件无关的方式运行大量的并行计算，确保了程序可以扩展（或缩小）以适应不同平台的能力，但是……

保证功能的可移植性，但不保证高性能！

支持的设备很多，不同的架构为不同的用例设计和优化。想要在特定的设备上达到最高的性能水平，总是需要一些手动优化工作——不管使用什么编程语言！这种特定于设备的优化的例子包括针对缓存的阻塞、选择平摊调度开销的粒度大小、使用专门的指令或硬件单元，以及选择适当的算法。其中一些例子将在第 15、16 和 17 章中继续讨论。

应用程序开发过程中，在性能、可移植性和生产率之间的平衡是必须面对的挑战，也是本书不能完全解决的挑战。然而，我们希望证明 DPC++ 提供了一种高级编程语言，还提供了可维护通用可移植代码和优化目标特定代码所需的所有工具。

多维内核

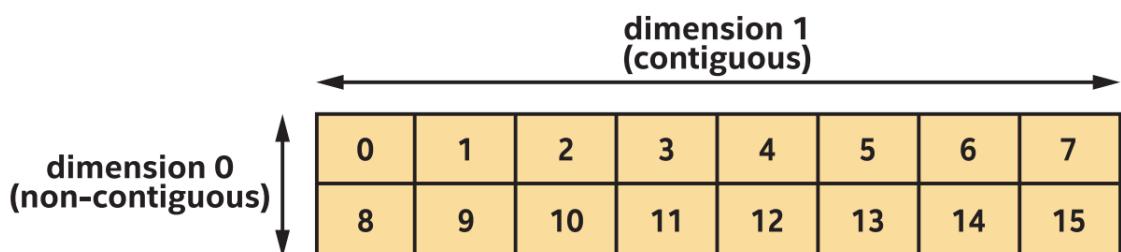
许多语言的并行构造是 1 维的，直接将工作映射到 1 维硬件资源（例如：硬件线程数）。并行内核是一个更高层次的概念，维度更多地反映了代码要解决的问题（在 1、2 或 3 维空间中）。

为了在 1 维空间开发方便，并行内核提供的多维索引。理解这个映射的行为方式可能是某些优化（例如：调优内存访问模式）的重要部分。

需要思考的是，哪个维度是连续的或“单位跨距”（即，多维空间中的数据落在一维空间中的位置）。SYCL 中与并行性相关的多维变量都遵循相同的约定：维度从 0 到 N-1 进行编号。这种约定与标准 C++ 中多维数组的行为一致。

SYCL 将二维空间映射为线性索引的示例如图 4-1 所示。也可以打破这种方式，采用自己的方法来线性化索引，这样做必须小心——打破惯例可能会对让步长为 1 访问方式有性能收益的设备产生负面的性能影响。

图 4-1 映射二维范围 (2,8) 到线性索引上



如果程序数据大于三个维度，则必须使用模算法手动负责多维索引和线性索引之间的映射。

循环与内核

迭代循环是一种串行结构：循环的每次迭代都按顺序执行。优化的编译器可以确定迭代循环的部分或全部是否可以并行执行，但必须是保证，当编译器无法证明并行执行是安全的，则保持循环顺序语义的正确性。

图 4-2 用串行循环表示向量加法

```
1 for (int i = 0; i < N; ++i) {
```

```
2     c[i] = a[i] + b[i];  
3 }
```

考虑图 4-2 中的循环，描述了简单的向量加法。即使在这样的情况下，证明循环可以并行执行也不是一件简单的事情：只有当 c 不重叠 a 或 b 时，并行执行才安全。而在一般情况下，没有运行时检查是无法证明这一点的！为了解决这样的情况，语言添加了一些特性，使我们能够向编译器提供信息，从而简化分析（例如，断言指针不与 `restrict` 重叠）或完全覆盖所有情况进行分析（例如，声明循环的所有迭代都是独立的，或者确切地定义应该如何将循环调度为并行资源）。

并行循环的含义有些含糊不清——因为不同的并行编程语言会重载这个术语——但是许多常见的并行循环构造表示，编译器转换顺序循环。这样的编程模型使我们能够编写顺序循环，并且只需要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，与其他编译器优化集成得很好，并且极大地简化了并行编程，但这不代表鼓励开发者在开发的早期阶段考虑并行性。

并行内核不是循环，也没有迭代。相反，内核表示一个操作，可以多次实例化，并应用于不同的输入数据；当内核并行启动时，该操作的多个实例将同时执行。

图 4-3 将循环重写（伪代码）为并行内核代码

```
1 launch N kernel instances {  
2     int id = get_instance_id(); // unique identifier in [0, N)  
3     c[id] = a[id] + b[id];  
4 }
```

图 4-3 展示了使用伪代码重写为内核的简单循环示例。这个内核中实现并行性是明确的：内核可以由任意数量的实例并行执行，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，可以确定并行运行是安全的（理想情况下应该如此）。

简而言之，基于内核的编程不是一种使用并行改进顺序代码的方法，而是一种编写显式并行程序的方法。

我们越早将思路从并行循环转向内核，就越容易使用 Data Parallel C++ 编写高效的并行程序。

语言的特性

决定编写并行内核时，就必须要启动的内核类型，以及了解如何在程序中表示。并行内核很多种，如果想掌握，就要熟悉每一种方法。

将内核与主机代码分离

有几种分离主机和设备代码的替代方法：C++ Lambda 表达式或函数对象（functors）、OpenCL C 源字符串或二进制文件。其中一些在第 2 章中已经介绍了，这些方式将在第 10 章中进行更详细地介绍。

这些选项都展示并行性的基本概念。为了一致性和简洁性，本章中的所有代码示例都使用 C++ Lambda 来表示内核。

Lambda 无害于性能

为了使用 DPC++, 不需要完全理解 C++ 规范中关于 Lambda 的所有内容——只需要知道 Lambda 可以表示内核，并且可以捕获的变量（按值）作为参数传递给内核。

使用 Lambda 来定义内核不会产生性能影响。DPC++ 编译器能够理解 Lambda 表示并行内核的主体，并能够进行优化执行。

关于 C++ Lambda 函数的复习，以及它们在 SYCL 中的使用说明，请参见第 1 章。有关使用 Lambda 定义内核的详细信息，请参见第 10 章。

不同形式的并行内核

内核有三种不同的形式，支持不同的执行模型和语法。可以使用任何形式编写可移植的内核，并且可以对任何形式的内核进行调优，以在各种设备类型上实现高性能。然而，有时可能希望使用特定的形式使并行算法更容易表达，或者使用某些语言特性。

第一种形式用于基本的数据并行内核，为编写内核提供了最优雅的介绍。牺牲对底层特性的控制，使内核的表达式尽可能简单。单个内核实例如何映射到硬件资源完全由实现控制，因此随着内核的复杂性的增长，对其性能的控制会变得越来越困难。

第二种形式扩展了内核，以提供对底层性能调优的访问。由于历史原因，第二种形式称为 ND-Range(N 维范围) 数据并行方式，其将某些内核实例分组在一起，允许对数据局部性的使用，以及内核实例和硬件资源之间的映射进行控制。

第三种形式提供了另一种语法，可以使用嵌套的内核构造来 ND-Range 内核。第三种形式称为分层数据并行，指的是嵌套内核的层次结构。

我们将在本章的最后再次讨论如何在不同的内核形式之间进行选择，届时将详细讨论了它们的特性。

内核间的数据并行

并行内核最基本的形式适用于基本的并行操作（例如：可以完全独立地、以任何顺序应用于每一块数据的操作）。通过使用这个形式，可以实现对调度的控制。因此，其是一个描述性编程结构的例子——调度决策由实现做出。

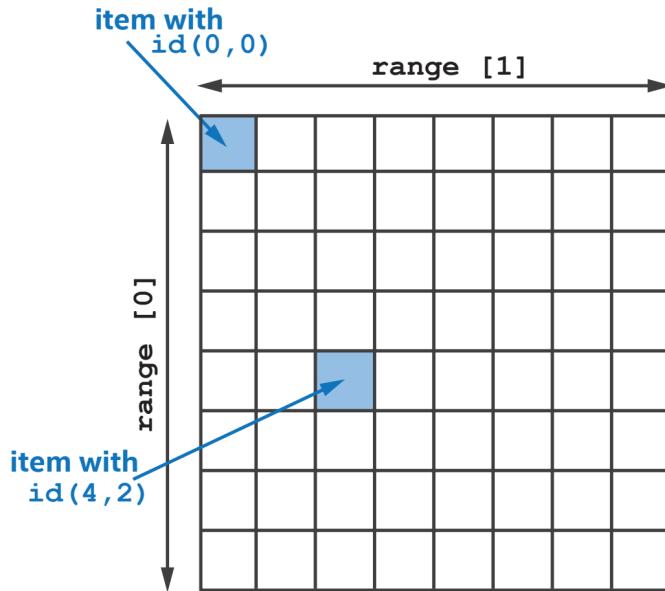
基本的数据并行内核是用单程序多数据 (Single Program, Multiple Data, SPMD) 风格编写的——“程序”（内核）应用于多段数据。注意，由于依赖于数据的分支，这个编程模型仍然允许内核在代码中采取不同的方式书写。

SPMD 编程模型的最大优点，允许同一个“程序”映射到多个级别和类型的并行性，而无需提供任何指示。同一个程序的实例可以流水线化、打包在一起并使用 SIMD 指令执行、跨多个线程分布，或者混合使用这三种方法。

理解数据并行内核

并行内核的执行空间称为执行范围，内核的每个实例称为工作项。如图 4-4 所示。

图 4-4 并行内核的执行空间，显示为 2D 范围内的 64 项



数据并行内核的执行模型非常简单：允许完全并行执行。工作项可以以任何顺序执行，包括在单个硬件线程上顺序执行（即，没有任何并行性）！假设所有工作项都并行执行的内核（例如，尝试同步工作项），则很容易导致程序挂起。

为了保证正确性，必须假定内核可以并行执行。例如，确保对内存的并发访问可被原子操作保护（见第 19 章），以避免条件竞争。

编写数据并行内核

数据并行内核使用 `parallel_for` 函数表示。图 4-5 展示了如何使用这个函数来表示向量加法。

图 4-5 使用 `parallel_for` 表示向量加法的内核代码

```

1 h.parallel_for(range{N}, [=](id<1> idx) {
2     c[idx] = a[idx] + b[idx];
3 });

```

该函数只接受两个参数：第一个参数是一个范围，指定在每个维度中启动工作项的数量，第二个参数是执行的内核函数。可以接受几个不同的类作为内核函数的参数，应该使用哪个类取决于该类公开所需的功能——稍后我们将继续讨论这个问题。

图 4-6 展示了一个非常类似的使用该函数来表示矩阵加法，（数学上）与向量加法相同，只是这次用于二维数据。这反映在内核中——两个代码片段之间的唯一区别是使用的范围和 `id` 类的维度！可以这样编码，因为 SYCL 访问器可以通过多维 `id` 进行索引。虽然看起来很奇怪，但非常强大，使我们能够编写基于数据维度的模板内核。

图 4-6 用 `parallel_for` 表示矩阵加法的内核代码

```

1 h.parallel_for(range{N, M}, [=](id<2> idx) {
2     c[idx] = a[idx] + b[idx];
3 });

```

C/C++ 中，使用多个索引和多个下标操作符为多维数据结构索引更为常见，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 id 描述的更复杂时，这种方式可以提高代码的可读性。

例如，图 4-7 中的矩阵乘法核必须提取索引的两个单独的分量，以便能够描述两个矩阵的行和列之间的点积。使用多个下标操作符（例如： $[j][k]$ ）比混合多个索引模式和构造二维 id 对象（例如， $\text{id}(j,k)$ ）更具可读性。

本章剩下的示例都使用了多个下标操作符，以确保访问的维度没有歧义。

图 4-7 用 parallel_for 表示方阵矩阵乘法的内核代码

```
1 h.parallel_for(range{N, N}, [=](id<2> idx) {
2     int j = idx[0];
3     int i = idx[1];
4     for (int k = 0; k < N; ++k) {
5         c[j][i] += a[j][k] * b[k][i];
6         // c[idx] += a[id(j,k)] * b[id(k,i)]; <<< equivalent
7     }
8});
```

图 4-8 将矩阵乘法映射到执行范围中的工作项上

work-item

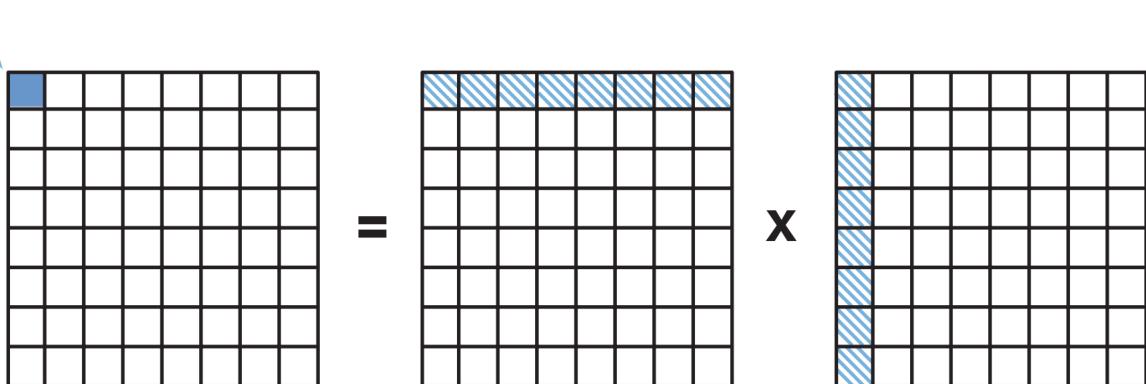


图 4-8 中展示了如何将矩阵乘法内核中的工作映射到各个项。注意工作项的数量是来自输出范围的大小，多个工作项可能使用同样的输入值：每一个工作项计算 C 矩阵的一个值，通过顺序迭代 A 矩阵的行（连续的）和 B（不连续）矩阵的列完成。

数据并行内核的细节

数据并行内核是通过三个 C++ 类表示：range、id 和 item。前面的章节中，已经多次看到了 range 和 id 类，这里将以不同的重点继续讨论。

range 类

range 表示 1、2 或 3 维范围。range 的维度是一个模板参数，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执行范围和缓冲区的大小。

range 类的简化定义显示了查询长度，构造函数和其他各种方法，如图 4-9 所示。

图 4-9 简化的 range 类定义

```
1 template <int Dimensions = 1>
2 class range {
3 public:
4     // Construct a range with one, two or three dimensions
5     range(size_t dim0);
6     range(size_t dim0, size_t dim1);
7     range(size_t dim0, size_t dim1, size_t dim2);
8
9     // Return the size of the range in a specific dimension
10    size_t get(int dimension) const;
11    size_t &operator[](int dimension);
12    size_t operator[](int dimension) const;
13
14    // Return the product of the size of each dimension
15    size_t size() const;
16
17    // Arithmetic operations on ranges are also supported
18};
```

id 类

id 表示 1、2 或 3 维范围的索引。id 的定义在许多方面与 range 相似：维数也必须在编译时已知，并且可以用于在索引内核的单个实例，或在缓冲区中创建偏移。

如图 4-10 中 id 类的简化定义所示，id 在概念上只是包含 1、2 或 3 个整数的容器。可用的操作也非常简单：可以查询每个维度中索引，计算新的索引。

尽管可以构造 id 来表示任意索引，但要获得与特定内核实例关联的 id，必须将它（或包含它的项）作为内核函数的参数。这个 id（或它的成员函数返回的值）必须转发到任何想要查询索引的函数中——目前没有任何可以在程序中任意点查询索引的函数，但是这个问题 DPC++ 会在未来来解决。

每个接受 id 的内核实例只知道分配给它计算的范围内的索引，而对 range 一无所知。如果想让内核实例知道自己的索引和范围，需要使用 item 类。

图 4-10 简化的 id 类定义

```
1 template <int Dimensions = 1>
2 class id {
3 public:
4     // Construct an id with one, two or three dimensions
5     id(size_t dim0);
6     id(size_t dim0, size_t dim1);
7     id(size_t dim0, size_t dim1, size_t dim2);
8
9     // Return the component of the id in a specific dimension
```

```

10    size_t get(int dimension) const;
11    size_t &operator[](int dimension);
12    size_t operator[](int dimension) const;
13
14    // Arithmetic operations on ids are also supported
15 };

```

item 类

item 表示内核函数的单个实例，封装了内核的执行 range 和实例在该 range 内的索引（分别使用一个 range 和一个 id）。与 range 和 id 一样，维度必须在编译时就已知。

图 4-11 给出了工作项类的简化定义。item 和 id 之间的主要区别是，item 公开了额外的函数来查询执行范围的属性（例如：大小、偏移量）和一个计算线性化索引的函数。与 id 一样，获得与特定内核实例相关联的项的唯一方法是将它作为内核函数的参数。

图 4-11 简化的 item 类定义

```

1 template <int Dimensions = 1, bool WithOffset = true>
2 class item {
3 public:
4     // Return the index of this item in the kernel's execution range
5     id<Dimensions> get_id() const;
6     size_t get_id(int dimension) const;
7     size_t operator[](int dimension) const;
8
9     // Return the execution range of the kernel executed by this item
10    range<Dimensions> get_range() const;
11    size_t get_range(int dimension) const;
12
13    // Return the offset of this item (if with_offset == true)
14    id<Dimensions> get_offset() const;
15
16    // Return the linear index of this item
17    // e.g. id(0) * range(1) * range(2) + id(1) * range(2) + id(2)
18    size_t get_linear_id() const;
19 };

```

显式 ND-Range 内核

并行内核的第二种形式是在执行范围内执行，其中工作项属于工作组，符合内核中局域性的概念。不同类型的工作组定义和行为不同，并且可以了解和/或控制将工作映射到特定的硬件平台。

显式的 ND-Range 内核是并行的——对每种工作组的工作进行映射，并且必须遵守这种映射。但这并不是完全定死的，因为工作组本身可以按任何顺序执行，并且实现将每种类型工作组映射到硬件资源上，并保留一定的自由度。这种说明性和描述性编程的结合能进行局部性设计和调优内核，且不会影响可移植性。

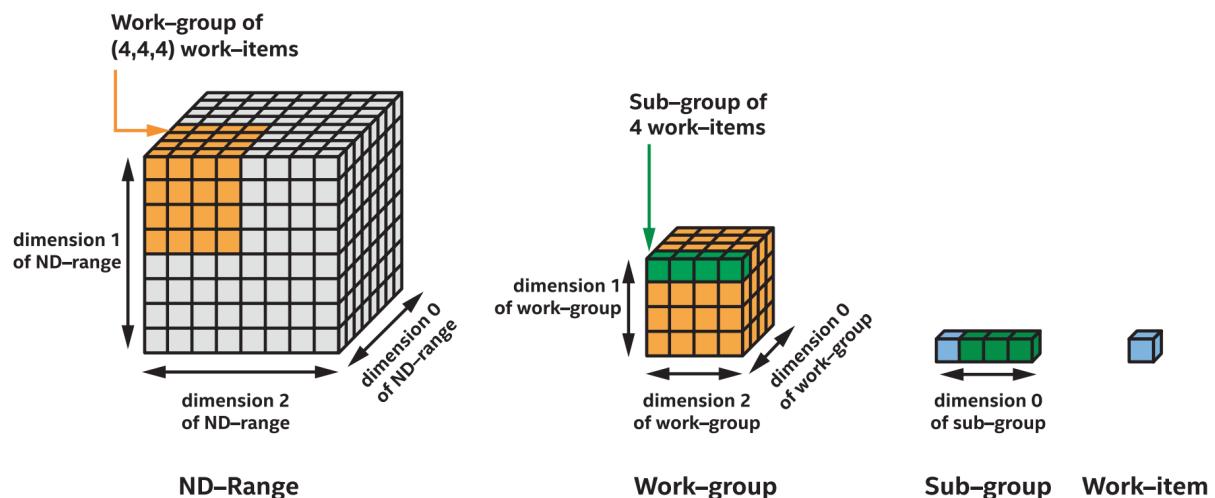
与数据并行内核一样，ND-Range 内核以 SPMD 风格编写，所有工作项执行应用于多个数据块的相同内核。区别是，每个程序实例可以查询在工作组中的位置，并可以访问特定于每种类型组的其他功能。

理解 ND-Range 并行内核

ND-Range 内核的执行范围可以划分为工作组、子工作组和工作项。ND-Range 表示总的执行范围，将其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划分 ND-Range 大小）。每个工作组可以通过实现进一步划分为子工作组。工作项和每种组的执行模型是编写正确和可移植程序的重要部分。

图 4-12 展示了将 $(8,8,8)$ 的 ND-Range 划分为 8 个大小 $(4,4,4)$ 的工作组。每个工作组包含由 4 个工作项组成的 16 个一维子工作组。注意维度的编号：子工作组是一维的，因此 ND-Range 和工作组的维度 2，变成子组的维度 0。

图 4-12 3 维 ND-Range 划分为工作组、子工作组和工作项



每种组到硬件资源的映射由实现定义，这种灵活性使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，可以由硬件线程和/或 SIMD 指令并行执行，甚至可以由特定的硬件执行。

本章中，我们只关注在通用平台上 ND-Range 的执行模型，并且不讨论对任何平台的映射。关于 GPU、CPU 和 FPGA 的硬件映射和性能建议，分别在第 15、16 和 17 章进行介绍。

工作项

工作项表示内核函数的各个实例。没有其他分组的情况下，工作项可以以任何顺序执行，除非通过对全局内存的原子操作（参见第 19 章），否则不能相互通信或同步。

工作组

ND-Range 中的工作项可以组成工作组。工作组可以以任何顺序执行，并且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，使用某些构造时，工作组中的工作项可并发调度，并且这种局部性提供的能力有：

1. 工作组中的工作项可以访问工作组本地内存，这些内存可以映射到某些设备上的专用快速内存（参见第 9 章）。
2. 工作组中的工作项可以使用工作组栅栏进行同步，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函数，提供可通信例程的实现（见第 9 章）和常规并行模式（归约和扫描）（见第 14 章）。

工作组的工作项数量通常在运行时为内核配置，最佳分组将取决于可用的并行度（即 ND-Range 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，来确定特定设备支持的每个工作组的最大工作项数量（见第 12 章），我们的责任是确保每个内核请求的工作组大小有效。

首先，工作组中的工作项可以调度单个计算单元，但工作组的数量和计算单元的数量之间不需要有任何关系。ND-Range 内的工作组数量可能比给定设备可并发执行的工作组数量大很多倍！我们依靠特定于设备的调度，尝试和编写内核同步工作组，但不建议这样做，因为不能保证与实现可能会在与之前不同的设备上运行。

其次，工作组中的工作项可以同时进行工作，但不能保证工作的独立性时——在工作组中使用栅栏和集合，可以对组内工作项进行同步。同一个工作组中工作项之间的通信和同步，只有在使用栅栏和集合操作时才能保证安全，手工编码的同步可能会造成死锁。

对于工作组的思考

工作组在许多方面与其他编程模型中的任务概念相似（例如：线程块）：任务可以按任何顺序执行（由调度程序控制）；开辟大量的任务是可能的（甚至是可取的）；在一组任务之间使用栅栏，通常不是个好主意（因为它可能非常昂贵或与调度器不兼容）。如果已经熟悉了基于任务的编程模型，会发现将工作组看作是数据并行的任务就会更好理解。

子工作组

许多硬件平台上，工作组中的子工作组在执行时具有调度上的保证。例如，子工作组中的工作项可以同时执行，因为可以映射到独立的硬件线程，子工作组本身可以在保证内核进度的情况下执行。

使用单一平台时，很容易在代码中加入执行模型的假设，但这使得内核不安全、不可移植——当在不同的编译器之间使用时，有时来自同一厂商的不同一代硬件之间迁移时，都可能会崩溃！

将子工作组为语言的核心部分，利用子工作组我们可以在底层硬件上执行工作项，并且还可以为跨平台实现高性能级别的应用。

与工作组一样，子工作组中的工作项可以同步、保证内存一致性，或通过工作组功能执行常见的并行模式。但对于子工作组没有等价的工作组本地内存（没有子组本地内存）。相反，子工作组中的工作项可以直接交换数据——无需显式的内存操作——使用 shuffle 操作（第 9 章）。

子工作组的某些功能由实现定义，不在我们的控制范围内。对于给定的设备、内核和 ND-Range 组合，子工作组具有固定的（一维的）大小，可以使用内核类的查询函数来查询（见第 10 章）。默认情况下，每个子工作组的工作项数量也由实现选择——可以通过在编译时请求特定的子组大小来覆盖这个行为，但是必须确保子工作组大小与设备兼容。

与工作组类似，子工作组中的工作项只保证并行执行——实现可以自由地顺序执行每个工作

项，并且只有在遇到集合函数时才会在工作项之间切换。子工作组的特殊之处在于，某些设备保证独立地执行——工作组中的所有子工作组都保证可以执行（取得进展），这是若干生产者-消费者模式的基础。这个独立的执行是否能保持，可以通过查询设备来确定。

对于子工作组的思考

如果考虑显式向量化的编程模型，那么将每个子工作组看作打包到 SIMD 寄存器中的一组工作项可能会更容易理解，其中子工作组中的每个工作项对应于一个 SIMD 通道。当多个子工作组同时执行时，硬件设备可以保证任务的执行，这种模型扩展会把每个子工作组当作并行执行的独立向量指令流。

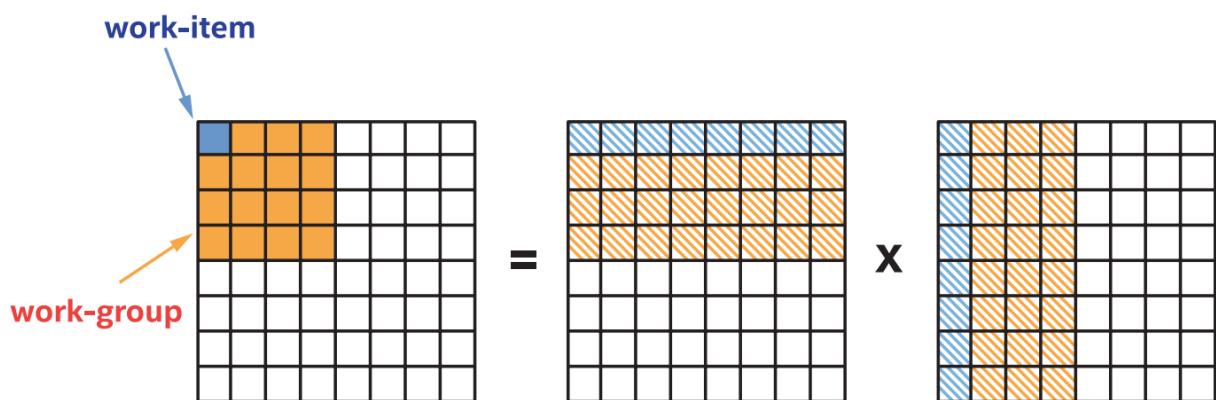
图 4-13 用 ND-Range parallel_for 表示矩阵乘法

```
1 range global{N, N};  
2 range local{B, B};  
3 h.parallel_for(nd_range{global, local}, [=](nd_item<2> it) {  
4     int j = it.get_global_id(0);  
5     int i = it.get_global_id(1);  
6  
7     for (int k = 0; k < N; ++k)  
8         c[j][i] += a[j][k] * b[k][i];  
9 });
```

编写 ND-Range 数据并行内核

图 4-13 重新实现了使用 ND-Range parallel_for 内核编写的矩阵乘法内核，图 4-14 中展示了这个内核是如何映射到每个工作项中的。以这种方式对工作项进行分组，从而保证本地的访问高效性，提高缓存命中率：例如，工作组在图 4-14 的大小是 (4, 4)，包含 16 个工作项，为了每个工作项都能执行正常，相应的数据需要加载 4 次。

图 4-14 将矩阵乘法映射到工作组和工作项



目前为止，矩阵乘法示例依赖于硬件缓存对同一个工作组中工作项 A 和 B 矩阵的重复访问进行优化。这样的硬件缓存在传统的 CPU 架构中很常见，而且在 GPU 架构中也越来越常见，但是

也有其他架构 (如上一代 GPU、FPGA) 带有“暂存”内存。ND-Range 内核可以使用本地访问器来说明工作组的本地内存分配的位置，然后实现就可以自由地将这些分配映射到特定内存 (它存在的地方)。工作组本地内存的使用将在第 9 章中介绍。

ND-Range 数据并行内核的细节

与数据并行内核相比，ND-Range 内核可以使用不同的类:nd_range 替换 range, nd_item 替换 item。还有两个新类，表示工作项属于的不同类型的组: 绑定工作组的功能封装在 group 类中，绑定到子工作组的功能封装在 sub_group 类中。

nd_range 类

nd_range 表示使用 range 类的两个实例组成的执行范围: 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 4-15 给出了 nd_range 类的简化定义。

nd_range 类根本没有提到子工作组: 子工作组范围在构造期间没有指定，不能查询。这有两个原因，1. 子工作组是可以忽略的底层实现细节。2. 有设备只支持固定的的子工作组大小，从而指定大小没有必要。所有与子工作组相关的功能都封装在特定的类中，稍后将对此再进行讨论。

图 4-15 简化定义的 nd_range 类

```
1 template <int Dimensions = 1>
2 class nd_range {
3 public:
4     // Construct an nd_range from global and work-group local ranges
5     nd_range(range<Dimensions> global, range<Dimensions> local);
6
7     // Return the global and work-group local ranges
8     range<Dimensions> get_global_range() const;
9     range<Dimensions> get_local_range() const;
10
11    // Return the number of work-groups in the global range
12    range<Dimensions> get_group_range() const;
13 }
```

nd_item 类

nd_item 是工作项的 ND-Range 形式，封装了内核的执行范围和工作项的索引。nd_item 与 item 的区别在于范围中的位置查询和表示，如图 4-16 中简化的类定义所示。

例如，可以使用 get_global_id() 函数在 (全局)ND-Range 中查询工作项的索引，或者使用 get_local_id() 函数在 (本地) 父工作组中查询工作项的索引。

nd_item 类还提供了获取描述项所属的工作组和子工作组的类句柄的函数。这些类为查询 DN-Range 的工作项索引提供了另一种方式。我们强烈推荐使用这些类来编写内核，而不是依赖于 nd_item，使用 group 和 sub_group 类通常更简洁，更清晰，也更符合 DPC++ 的方向。

图 4-16 简化定义的 nd_item 类

```

1 template <int Dimensions = 1>
2 class nd_item {
3 public:
4     // Return the index of this item in the kernel's execution range
5     id<Dimensions> get_global_id() const;
6     size_t get_global_id(int dimension) const;
7     size_t get_global_linear_id() const;
8
9     // Return the execution range of the kernel executed by this item
10    range<Dimensions> get_global_range() const;
11    size_t get_global_range(int dimension) const;
12
13    // Return the index of this item within its parent work-group
14    id<Dimensions> get_local_id() const;
15    size_t get_local_id(int dimension) const;
16    size_t get_local_linear_id() const;
17
18    // Return the execution range of this item's parent work-group
19    range<Dimensions> get_local_range() const;
20    size_t get_local_range(int dimension) const;
21
22    // Return a handle to the work-group
23    // or sub-group containing this item
24    group<Dimensions> get_group() const;
25    sub_group get_sub_group() const;
26};

```

group 类

group 类封装了与工作组相关的所有功能，简化的定义如图 4-17 所示。

图 4-17 简化定义的 group 类

```

1 template <int Dimensions = 1>
2 class group {
3 public:
4     // Return the index of this group in the kernel's execution range
5     id<Dimensions> get_id() const;
6     size_t get_id(int dimension) const;
7     size_t get_linear_id() const;
8
9     // Return the number of groups in the kernel's execution range
10    range<Dimensions> get_group_range() const;
11    size_t get_group_range(int dimension) const;
12
13    // Return the number of work-items in this group
14    range<Dimensions> get_local_range() const;
15    size_t get_local_range(int dimension) const;

```

```
16 };
```

group 类提供的许多函数在 nd_item 类中都有对等的函数: 例如, group.get_id() 等价于 item.get_group_id(), group.Get_local_range() 等价于 item.get_local_range()。如果不使用该类的任何工作组函数, 还应该使用它吗? 直接使用 nd_item 中的函数, 不是更简单吗? 这里有一个折衷的方式: 使用 group 要求编写更多的代码, 但这些代码可能更容易阅读。例如, 图 4-18 中的代码片段: body 由工作组中的所有工作项调用, parallel_for 的中的 get_local_range() 返回的范围就是工作组的范围。只用 nd_item 就可以很容易地编写相同的代码, 但代码可能很难看懂。

图 4-18 使用 group 类来提高可读性

```
1 void body(group& g);
2 h.parallel_for(nd_range{global, local}, [=](nd_item<1> it) {
3     group<1> g = it.get_group();
4     range<1> r = g.get_local_range();
5     ...
6     body(g);
7 }) ;
```

sub_group 类

sub_group 类封装了与子工作组相关的功能, 简化的定义如图 4-19 所示。与工作组不同, sub_group 类是访问子工作组功能的唯一方法, 其函数在 nd_item 中没有重复。sub_group 类中的查询都是相对于工作项进行解释的: 例如, get_local_id() 返回子工作组中调用工作项的本地索引。

图 4-19 简化定义的 sub_group 类

```
1 class sub_group {
2     public:
3         // Return the index of the sub-group
4         id<1> get_group_id() const;
5
6         // Return the number of sub-groups in this item's parent work-group
7         range<1> get_group_range() const;
8
9         // Return the index of the work-item in this sub-group
10        id<1> get_local_id() const;
11
12        // Return the number of work-items in this sub-group
13        range<1> get_local_range() const;
14
15        // Return the maximum number of work-items in any
16        // sub-group in this item's parent work-group
17        range<1> get_max_local_range() const;
18    };
```

有一些功能用于查询当前子工作组中的工作项数量，以及工作组中任何子工作组中的最大工作项数量。这些方式取决于设备对于子工作组的实现的方式，但其目的是反映编译器目标子工作组大小和运行时子工作组大小之间的差异。例如，非常小的工作组可能包含比编译时子工作组更少的工作项，或者不同大小的子工作组可能用于处理不能被子组大小整除的工作组。

内核间的分层并行

分层数据并行内核提供了实验性的替代语法，可以用工作组和工作项来表示内核，其中层次结构的每一层都使用 `parallel_for` 嵌套调用。这种自顶向下的编程风格类似于编写并行循环，可能比其他两种内核形式使用的自底向上编程风格更为开发者和读者熟悉。

分层内核的复杂性在于，`parallel_for` 的嵌套调用会创建单独的 SPMD 环境，每个范围定义了新的“程序”，其应由与该范围相关的并行工作项执行。这种复杂性要求编译器进行分析，并可能使生成某些设备的代码复杂化。一些平台上用于分层并行内核的编译器技术仍然不成熟，性能与编译器实现的质量密切相关。

由于分层数据并行内核和为特定设备生成的代码之间的关系依赖于编译器，因此分层内核应该比显式的 ND-Range 内核更具描述性的结构。由于分层内核保留了控制任务映射工作项和工作组的能力，因此比基本内核更具有特定性。

理解分层的数据并行内核

分层数据并行内核的底层执行模型，与显式 ND-Range 数据并行内核的执行模型相同。工作项、子工作组和工作组具有相同的语义和执行保证。

然而，分层内核的不同作用域由编译器映射到不同的执行资源：外部作用域对每个工作组执行一次（就像由单个工作项执行一样），而内部作用域由工作组中的工作项并行执行。不同的作用域还控制内存中分配变量的位置，并且作用域的打开和关闭意味着启用工作组栅栏（以强制执行内存一致性）。

尽管工作组中的工作项仍然划分为子工作组，但不能在分层并行内核访问 `sub_group` 类，将子工作组的概念合并到 SYCL 分层并行中，是比引入新类更重要的修改，这个工作正在进行中。

编写分层的数据并行内核

分层内核中，`parallel_for_work_group` 和 `parallel_for_work_item` 可以取代 `parallel_for`，分别对应于工作组和工作项的并行性。`parallel_for_work_group` 作用域中的任何代码在工作组中只执行一次，并且在 `parallel_for_work_group` 作用域中分配的变量对所有工作项都可见（在工作组本地内存中分配）。在 `parallel_for_work_item` 范围内的任何代码，都是由工作组的工作项目并行执行，并且在 `parallel_for_work_item` 范围内分配的变量对单个工作项都可见（可以分配在工作项的私有内存中）。

如图 4-20 所示，分层并行表示的内核与 ND-Range 内核非常相似。因此，应该把分层并行看作是一种生产力象征，不公开任何未通过 ND-Range 内核公开的功能，但可以提高代码的可读性和/或减少代码量。

图 4-20 表示具有层次并行性的矩阵乘法

```

1 range num_groups{N / B, N / B}; // N is a multiple of B
2 range group_size{B, B};
3 h.parallel_for_work_group(num_groups, group_size, [=](group<2> grp) {
4     int jb = grp.get_id(0);
5     int ib = grp.get_id(1);
6     grp.parallel_for_work_item([&](h_item<2> it) {
7         int j = jb * B + it.get_local_id(0);
8         int i = ib * B + it.get_local_id(1);
9         for (int k = 0; k < N; ++k)
10            c[j][i] += a[j][k] * b[k][i];
11    });
12 });

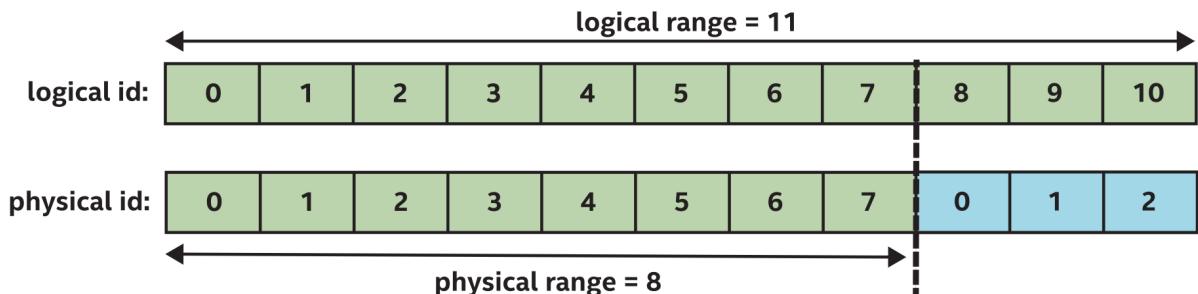
```

需要注意的是，传递给 parallel_for_work_group 的范围指定了组的数量和可选组的大小，而不是像 ND-Range parallel_for 那样指定了工作项的总数和工作组大小。内核函数可传入组类的实例，反映出范围与工作组的关系，而不与单个工作项相关系。

parallel_for_work_item 是 group 类的一个成员函数，只能在 parallel_for_work_group 的作用域内调用。在其最简形式中，唯一的参数是接受 h_item 类实例的函数，该函数可以执行的次数等于每个工作组请求的工作项的数量。parallel_for_work_item 的另一个特性是支持逻辑范围的能力，作为附加参数传递给函数。当指定一个逻辑范围时，每个物理工作项执行零个或多个函数实例，并且逻辑范围的逻辑项会轮询，并分配给物理工作项。

图 4-21 显示了由 11 个逻辑工作项组成的逻辑范围，和由 8 个物理工作项组成的底层物理范围间的映射示例。前三个工作项分配了功能的两个实例，而其他工作项只分配了一个。

图 4-21 将大小为 11 的逻辑范围映射为大小为 8 的物理范围



如图 4-22 所示，将 parallel_for_work_group 的可选组与 parallel_for_work_item 的逻辑范围相结合，可以自由选择工作组大小，方便描述执行范围的能力。请注意，每个组执行的工作量与图 4-20 中相同，但是工作量已经与物理工作组大小不同了。

图 4-22 用层次并行性和逻辑范围表示矩阵乘法

```

1 range num_groups{N / B, N / B}; // N is a multiple of B
2 range group_size{B, B};
3 h.parallel_for_work_group(num_groups, [=](group<2> grp) {
4     int jb = grp.get_id(0);

```

```

5   int ib = grp.get_id(1);
6   grp.parallel_for_work_item(group_size, [&](h_item<2> it) {
7     int j = jb * B + it.get_logical_local_id(0);
8     int i = ib * B + it.get_logical_local_id(1);
9     for (int k = 0; k < N; ++k)
10      c[j][i] += a[j][k] * b[k][i];
11  });
12 });

```

分层数据并行内核的细节

分层数据并行内核重用来自 ND-Range 数据并行内核的 group 类，但需要将 nd_item 替换为 h_item。引入新的私有内存类，以便在 parallel_for_work_group 范围内对分配进行更严格的控制。

h_item 类

h_item 是 item 的变体，只能在 parallel_for_work_item 作用域内使用。如图 4-23，提供了一个类似的接口 nd_item：可以查询工作项的工作组 (get_physical_local_id()) 或逻辑执行范围 parallel_for_work_item(使用 get_logical_local_id())。

图 4-23 简化定义的 h_item 类

```

1 template <int Dimensions>
2 class h_item {
3 public:
4   // Return item's index in the kernel's execution range
5   id<Dimensions> get_global_id() const;
6   range<Dimensions> get_global_range() const;
7
8   // Return the index in the work-group's execution range
9   id<Dimensions> get_logical_local_id() const;
10  range<Dimensions> get_logical_local_range() const;
11
12 // Return the index in the logical execution range of the parallel_for
13 id<Dimensions> get_physical_local_id() const;
14 range<Dimensions> get_physical_local_range() const;
15 };

```

private_memory 类

private_memory 类提供了一种机制来声明每个工作项的私有变量，嵌套在同一个 parallel_for_work_group 作用域内的多个 parallel_for_work_item 都可以访问这些变量。

这个类非常必要，因为不同的分层并行性范围中的变量声明不同：如果编译器能够保证安全，则变量声明是私有的。我们不可能单独使用作用域，来表达变量是工作项私有的。

要了解为什么这是一个问题，回顾一下图 4-22 中的矩阵乘法内核。ib 和 jb 变量是在 parallel_for_work_group 作用域声明的，默认情况下应该在工作组的本地内存中分配！编译器很有可能

不会犯这个错误，因为变量是只读的，可以在每个工作项上进行冗余计算，但语言并没有这样的保证。如果想确定一个变量是否在工作项私有内存中，需要将变量声明包装在 private_memory 类的实例中，如图 4-24 所示。

图 4-24 简化定义的 private_memory 类

```
1 template <typename T, int Dimensions = 1>
2 class private_memory {
3     public:
4         // Construct a private variable for each work-item in the group
5         private_memory(const group<Dimensions>&);
6
7         // Return the private variable associated with this work-item
8         T& operator(const h_item<Dimensions>&);
9 }
```

例如，如果使用 private_memory 类重写矩阵乘法内核，把变量定义为 private_memory<int> ib(grp)，并且对这些变量的访问变成 ib[item]。这样，使用 private_memory 类的代码非常难读，而在 parallel_for_work_item 范围中声明则会更简单。

如果工作项私有变量在多个 parallel_for_work_item 范围内，并且在同一 parallel_for_work_group 上使用，建议只使用 private_memory 类，可以避免多余地计算。要不就依赖现代优化编译器的能力，并且只有在变量分析失败时才在 parallel_for_work_item 范围声明变量（记住，也要向编译器供应商报告问题）。

将计算映射到工作项中

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单个数据块上的单个操作。这是编写内核的一个简单方法，但这种数据与工作项的一一映射不是由 DPC++ 全控制的，所以使得任务参数化是提高应用性能和可移植性的好方法。

一对一映射

当编写与工作项的一一映射的内核时，这些内核需要完成的工作量必须适配范围或 nd_range 的大小。这是编写内核的最简单的方式，这种方式工作得非常好——可以信底层任实现工作项映射到硬件。

然而，对系统和实现的特定组合进行性能调优时，需要更加注意底层调度。工作组对计算资源的调度由实现定义，并且可能是动态的（例如，当一个计算资源完成一个工作组时，执行的下一个工作组可能来自于共享队列）。动态调度对性能的影响不确定，其重要性取决于内核函数每个实例的执行时间，以及调度软件（如 CPU）或硬件（如 GPU）上的实现。

多对一映射

另一种方法是编写多对一映射的内核，范围的含义略有变化：范围不再描述要完成的工作量，而是要使用的工作项的数量。通过改变工作项的数量和分配给每个工作项的工作量，可以调整工作分配，以最大限度地提高效率。

编写这种形式的内核需要做两个更改:

- 内核必须接受描述工作总量的参数。
- 内核必须包含将工作分配给工作项的循环。

图 4-25 给出了这样的内核示例，内部的循环稍微不同寻常——起始索引是全局范围内工作项的索引，而跨距是工作项的总数。这种数据到工作项的循环调度确保了循环的所有 N 次迭代将由一个工作项执行，而且线性工作项可以访问连续的内存位置（以改进缓存局部性和向量化行为）。工作可以分布在多个工作组或单个工作组中的工作项中，以进一步利用局部性。

图 4-25 具有独立数据和执行范围的内核

```
1 size_t N = ...; // amount of work
2 size_t W = ...; // number of workers
3 h.parallel_for(range{W}, [=](item<1> it) {
4     for (int i = it.get_id()[0]; i < N; i += it.get_range()[0]) {
5         output[i] = function(input[i]);
6     }
7});
```

这些分发模式很常见，使用具有逻辑范围的分层并行时，可以使用。我们期望未来的 DPC++ 引入语法糖，来简化 ND-Range 内核中分发模式的表示。

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偏好，并且很受到以前使用其他并行编程模型和语言经验的影响。

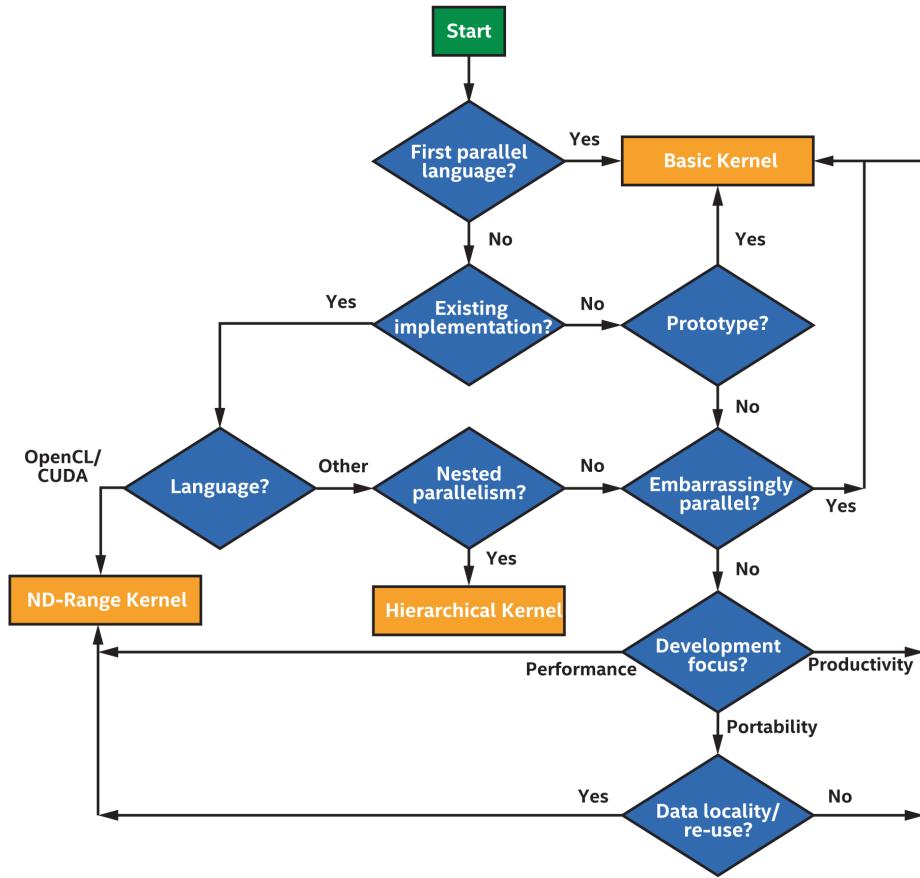
选择内核形式的另一个原因是，公开内核所需的功能。不幸的是，在开发开始之前很难确定需要哪些功能——特别是当我们不熟悉不同的内核形式，以及它们与各种类是如何交互时。

为了帮助读者们进行选择，我们根据自己的经验构建了两个指南。读者可以参考这些经验法则，但最好的方法是选择不同的内核形式进行编写，对不同的方式进行测试和了解，从而在开发应用程序时选择最合适的形式和开发方式。

第一个指南是如图 4-26 所示的流程图：

- 是否有并行编程的经验
- 是从头编写新代码，还是移植用现有的并行程序
- 是包含嵌套的并行，还是在内核函数的不同实例之间重用数据
- 用 SYCL 编写新内核是为了最大化性能，还是为了提高代码的可移植性，使用比底层的语言更高效的方式表示并行性

图 4-26 选择正确的内核形式



第二个指南是图 4-27 中的表格，总结了每种内核形式公开的功能。值得注意的是，这个表反映了 DPC++ 在本书出版时的状态，随着语言的发展，每个内核形式可用的特性会发生变化。预计基本趋势将保持不变：基本的数据并行内核不会公开位置感知特性，显式的 ND-Range 内核会公开所有支持性能的特性，分层内核在公开特性方面会落后于显式的 ND-Range 内核，但是对这些特性的表达将使用更高层的抽象。

图 4-27 每种内核形式的特性

特性	基本内核	ND-Range 内核	层次内核
工作组本地内存	No	Yes	Yes
工作组栅栏	No	Yes	Yes
子工作组	No	Yes	No
工作组函数（比如： scan,reduce）	No	Yes	No

总结

本章介绍了在 DPC++ 中表达并行性的基础知识，并讨论了编写数据并行内核的每种方法的优缺点。

DPC++ 和 SYCL 为许多形式的并行提供了支持，希望已经提供了足够的信息，以便读者准备开始编写代码！

我们只讨论了表面的内容，接下来将深入讨论本章中介绍的许多概念和类：本地内存、栅栏和通信的使用将在第 9 章中讨论；除了使用 Lambda 表达式外，定义内核的不同方法将在第 10 章继续；ND-Range 执行模型到特定硬件的详细映射将在第 15、16 和 17 章中讨论；第 14 章将介绍使用 DPC++ 并行模式的最佳实践。

5 错误处理



阿加莎·克里斯蒂 (Agatha Christie) 在 1969 年写道: “如果计算机相比, 人为错误根本算不了什么。” 错误处理机制可以捕获错误, 处理发生的错误。

开发应用期间, 检测和处理错误很有帮助 (其他开发者也会犯错误), 其在稳定和安全的程序和库中扮演着重要角色。本章将一起来了解 SYCL 中可用的错误处理机制, 以便了解我们有哪些选择, 以及如何构建应用程序。

本章概述了 SYCL 中的同步和异步错误, 描述了不处理错误时, 应用程序的行为, 并深入研究了 SYCL 处理异步错误的机制。

安全第一

不处理检测到 (抛出) 的错误, 那么应用程序将终止, 并显示发生的错误。这种行为在编写应用程序时不关注错误处理, 并且相信错误会以某种方式告知开发人员或用户。当然, 我们并不是建议忽略错误处理! 开发应用程序应该将错误处理作为体系结构的核心, 但是应用程序在开发时通常不觉得。C++ 的目标是即使没有显式处理错误, 那些无法处理错误的代码依旧能够观察到错误。

SYCL 属于数据并行, 原理相同: 如果代码中不做任何管理错误的工作, 并且检测到错误, 程序将会出现异常终止。开发应用程序应该将错误处理作为体系结构的核心, 不仅要报告错误, 还要从错误状态中进行恢复。

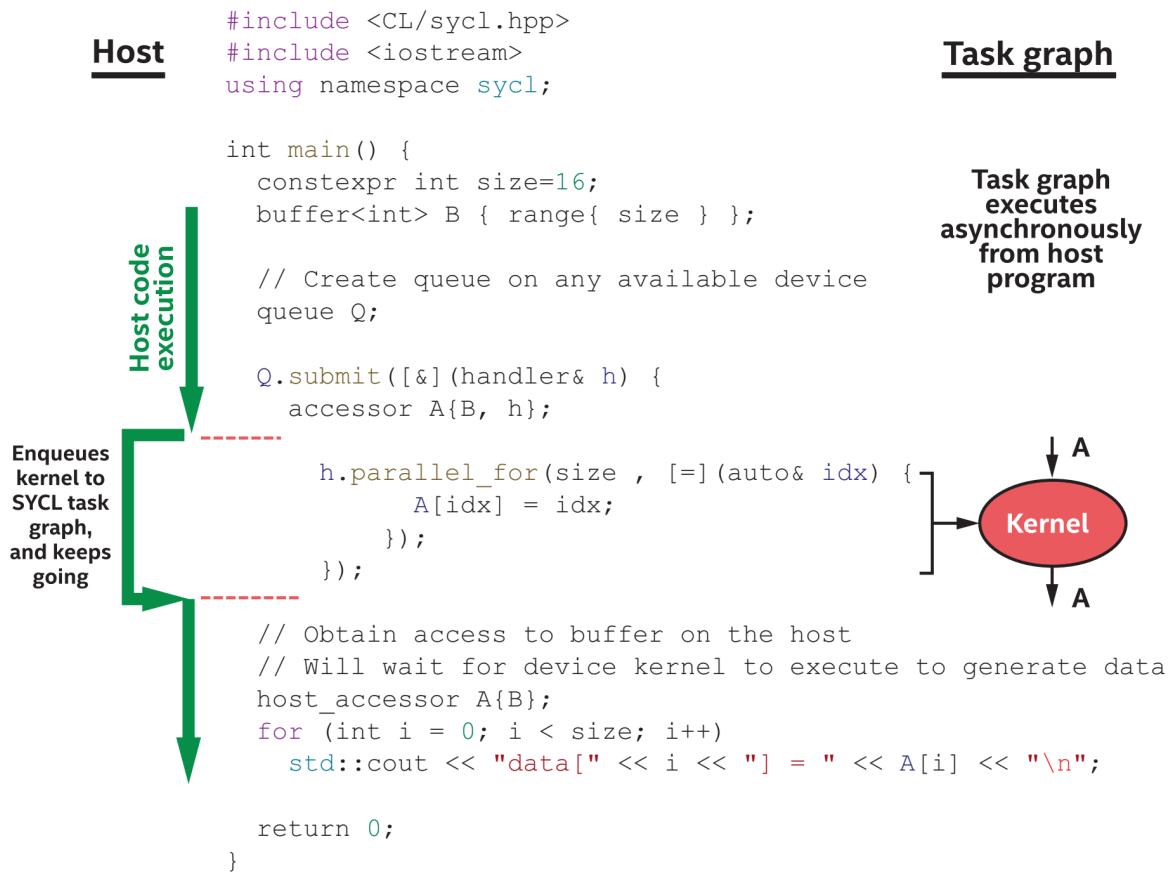
因为不添加任何错误处理代码, 在出现错误时程序会异常终止, 所以需要对错误进行处理。

错误类型

C++ 通过异常机制提供了通知和处理错误的框架。因为有些错误发生在设备上, 或者在设备上启动工作时, 所以异构编程还需要对错误级别进行管理。这些错误通常与主程序不相关, 因此不能与 C++ 异常处理机制集成。为了解决这个问题, 需要有其他机制使异步错误易于管理和控制。

图 5-1 展示了程序的两个部分:(1) 主机代码运行的顺序, 以及执行和提交工作任务图 (2) 的异步执行和函数或对其他设备操作, 对主程序的运行的依赖性。例子使用 `parallel_for` 执行内核函数, 并作为任务图的一部分异步执行, 其他的操作会在第 3、4 和 8 章中再进行讨论。

图 5-1 分离主程序和内核任务



理解图 5-1 中左边和右边 (主机和任务图) 之间的区别，是理解同步和异步错误区别的关键。

主程序执行某个操作 (如 API 调用或对象构造函数)，在检测到错误条件时，会发生同步错误。可以在图左侧的指令完成前检测到，并且可以立即将错误抛出。可以在图的左侧使用 try-catch 来包装指令，在 try 块结束 (从而捕获) 前检测到 try 中操作的错误。C++ 异常机制就是为了处理这些错误而设计的。

当图 5-1 右侧出现异步错误时，只有在执行中的操作才会检测到错误。当检测到异步错误时，主程序通常会继续执行，所以无法使用 try-catch 来捕获这些错误。不过，有异步异常处理框架可以来处理这些错误。

创建一些错误

我们将在下面几节中创建同步和异步错误。

同步错误

图 5-2 创建同步错误

```

1 #include <CL/sycl.hpp>
2 using namespace sycl;
3
4 int main() {

```

```

5   buffer<int> B{ range{16} };

6
7 // ERROR: Create sub-buffer larger than size of parent buffer
8 // An exception is thrown from within the buffer constructor
9 buffer<int> B2(B, id{8}, range{16});

10
11 return 0;
12 }
13 /*
14 Example output:
15 terminate called after throwing an instance of
16 'cl::sycl::invalid_object_error'
17 what(): Requested sub-buffer size exceeds the size of the parent buffer
18 -30 (CL_INVALID_VALUE)
19 */

```

图 5-2 中，缓冲区创建了子缓冲区，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错误，并在构造函数完成之前抛出异常。这是一个同步错误，因为是主机程序的一部分（与主机程序同步）。构造函数返回前错误可检测到，因此错误可以在起始点或主程序中的检测点处进行处理。

代码示例没有执行任何捕获和处理 C++ 异常的操作，因此程序会调用 std::terminate。

异步错误

生成异步错误比较麻烦，因为实现会同步地检测和报告错误。同步错误发生在主程序中特定的起始点，所以更容易调试。不过，为演示目的生成异步错误的一种方法是，向命令组提交添加一个应急/备用队列，并丢弃会抛出的同步异常。图 5-3 就是这样的代码，调用了 handle_async_error 函数。异步错误可以在没有应急/备用队列的情况下发生和报告，因此备用队列只是示例的一部分，异步错误实际不需要这个队列。

图 5-3 创建异步错误

```

1 #include <CL/sycl.hpp>
2 using namespace sycl;
3
4 // Our simple asynchronous handler function
5 auto handle_async_error = [] (exception_list elist) {
6     for (auto &e : elist) {
7         try { std::rethrow_exception(e); }
8         catch (sycl::exception& e) {
9             std::cout << "ASYNC EXCEPTION!!\n";
10            std::cout << e.what() << "\n";
11        }
12    }
13 };
14
15 void say_device (const queue& Q) {
16     std::cout << "Device : "

```

```

17    << Q.get_device().get_info<info::device::name>() << "\n";
18 }
19
20 int main() {
21     queue Q1{ gpu_selector{}, handle_async_error };
22     queue Q2{ cpu_selector{}, handle_async_error };
23     say_device(Q1);
24     say_device(Q2);
25
26     try {
27         Q1.submit([&] (handler &h){
28             // Empty command group is illegal and generates an error
29         },
30         Q2); // Secondary/backup queue!
31     } catch (...) {} // Discard regular C++ exceptions for this example
32     return 0;
33 }
34 /*
35 Example output:
36 Device : Intel(R) Gen9 HD Graphics NEO
37 Device : Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz
38 ASYNC EXCEPTION!!
39 Command group submitted without a kernel or a explicit memory operation. -59 (
40     CL_INVALID_OPERATION)
41 */

```

错误的处理策略

C++ 异常设计目的是将程序中检测到错误与处理的错误区分开，这个概念非常适合 SYCL 中的同步和异步错误。通过抛出和捕获机制，可以定义处理程序的层次。

构建能以可靠的方式处理错误的应用程序，需要预先制定策略，并为错误处理构建相应的体系。C++ 提供了工具来实现策略，但是这样的架构超出了本章讨论的范围。有许多书籍和参考资料专门对这个主题进行讨论，有兴趣可以去全面的了解一下 C++ 错误处理的策略。

错误检测和报告与程序实际功能并不相关。如果目标仅仅是在执行过程中检测并报告错误（但不一定是从错误中恢复），可以通过可靠地检测和报告程序中的错误。下面几节先来介绍忽略错误处理时会发生什么（默认行为并不是那么糟糕！），然后介绍在应用程序中容易实现的错误报告方式。

忽略错误

在 C++ 和 SYCL 中即使没有显式地处理错误，也会出现错误。未处理的同步或异步错误的默认结果是程序异常终止。下面两个示例分别模拟了不处理同步错误和异步错误的情况。

图 5-4 展示了不处理的 C++ 异常的情况，例如：该异常可能是未处理的 SYCL 同步错误。可以使用此代码来测试特定操作系统在这种情况下报告什么错误。

图 5-5 展示了 std::terminate 的输出，这是程序中未处理的 SYCL 异步错误的情况。可以使用此代码来测试特定操作系统在这种情况下将报告什么错误。

因为未捕获的错误将使程序终止，所以需要对错误进行处理！

图 5-4 不处理异常

```
1 #include <iostream>
2
3 class something_went_wrong {};
4
5 int main() {
6     std::cout << "Hello\n";
7
8     throw(something_went_wrong{});
9 }
10 /*
11 Example output in Linux:
12 Hello
13 terminate called after throwing an instance of 'something_went_wrong'
14
15 Aborted (core dumped)
16 */
```

图 5-5 不处理 SYCL 异步异常时，将调用 std::terminate

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello\n";
5
6     std::terminate();
7 }
8 /*
9 Example output in Linux:
10 Hello
11 terminate called without an active exception
12 Aborted (core dumped)
13 */
```

同步的处理错误

SYCL 同步错误与 C++ 异常一样，SYCL 中添加的大多错误机制都与异步错误有关，我们将在下一节中介绍异步错误，而同步错误非常重要，因为实现会以同步的方式检测和报告尽可能多的错误，使其更容易检测和处理。

SYCL 定义的同步错误是 SYCL::exception 类型，是 std::exception 的派生类，其允许通过如图 5-6 所示的 try-catch 来捕获异常。

图 5-6 捕获 sycl::exception 的模式

```

1 try{
2     // Do some SYCL work
3 } catch (sycl::exception &e) {
4     // Do something to output or handle the exception
5     std::cout << "Caught sync SYCL exception: " << e.what() << "\n";
6     return 1;
7 }

```

在 C++ 处理错误机制的基础上，SYCL 添加了 SYCL::exception 异常类型。其他与标准的 C++ 异常处理方式相同，也符合大多数开发者的习惯。

图 5-7 提供了一个处理异常的示例，并通过从 main() 来结束程序。

图 5-7 捕获异常

```

1 try{
2     buffer<int> B{ range{16} };
3     // ERROR: Create sub-buffer larger than size of parent buffer
4     // An exception is thrown from within the buffer constructor
5     buffer<int> B2(B, id{8}, range{16});
6
7 } catch (sycl::exception &e) {
8     // Do something to output or handle the exception
9     std::cout << "Caught sync SYCL exception: " << e.what() << "\n";
10    return 1;
11 } catch (std::exception &e) {
12     std::cout << "Caught std exception: " << e.what() << "\n";
13     return 2;
14 } catch (...) {
15     std::cout << "Caught unknown exception\n";
16     return 3;
17 }
18 return 0;
19
20 /*
21 Example output:
22 Caught sync SYCL exception: Requested sub-buffer size exceeds the size of
23 the parent buffer -30 (CL_INVALID_VALUE)
24 */

```

异步的错误处理

异步错误由 SYCL 运行时 (或底层后端) 检测，错误的发生与主机程序无关。错误存储在 SYCL 运行时的内部，仅在开发者可以控制的点进行处理。我们需要对异步错误的处理进行讨论：

1. 异步处理程序，当有未处理的异步错误需要处理时调用
2. 异步处理程序何时调用

异步处理程序

异步处理程序是个函数定义，使用 SYCL 上下文和/或队列注册。下一节中，有未处理的异步异常需要处理，SYCL 运行时将调用异步处理程序，并将其传递到异常列表中。

异步处理程序以 std::function 的形式传递给上下文或队列构造函数，可以根据常规函数、Lambda 或函数操作符等方式定义。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。

图 5-8 定义为 lambda 的异步处理程序示例

```
1 // Our simple asynchronous handler function
2 auto handle_async_error = []( exception_list elist ) {
3     for ( auto &e : elist ) {
4         try{ std::rethrow_exception(e); }
5         catch ( sycl::exception& e ) {
6             std::cout << "ASYNC EXCEPTION!!\n";
7             std::cout << e.what() << "\n";
8         }
9     }
10 };
```

图 5-8 中，std::rethrow_exception 可以抛出特定的异常类型，并可以使用 catch 对异常进行捕捉，本例中捕捉的是 sycl::exception。还可以在 C++ 中使用其他方法，或者选择处理所有类型的异常。

处理程序在构造时与队列或上下文（第 6 章将详细介绍底层细节）相关联。例如，将图 5-8 中定义的处理程序注册到正在创建的队列中，可以写成：

```
queue my_queue gpu_selector, handle_async_error;
```

同样，要将图 5-8 中定义的处理程序注册到正在创建的上下文中，可以写成：

```
context my_context handle_async_error;
```

大多数应用程序不需要显式地创建或管理上下文（程序会自动创建），大多数开发者应该为特定设备构造处理程序与队列（而不是显式创建上下文）。

应该在队列上定义异步处理程序时（除非已经显式的管理了上下文）。

如果没有为队列或队列的父上下文定义异步处理程序，在处理队列（或上下文中）上发生的异步错误时，会使用默认的异步处理程序，就如图 5-9 所示。

图 5-9 默认异步处理程序的示例

```
1 // Our simple asynchronous handler function
2 auto handle_async_error = []( exception_list elist ) {
3     for ( auto &e : elist ) {
4         try{ std::rethrow_exception(e); }
5         catch ( sycl::exception& e ) {
6             // Print information about the asynchronous exception
7         }
8     }
9 }
```

```
8     }
9
10    // Terminate abnormally to make clear to user
11    // that something unhandled happened
12    std::terminate();
13 }
```

默认处理程序应该向用户显示异常列表中的错误信息，然后以非正常的方式终止应用程序，同时也需要让操作系统记录下这次非正常终止。

异步处理程序中的执行由我们来定，可以记录错误、终止程序、恢复错误，以便程序可以继续正常执行。常见的情况是通过 `sycl::exception::what()` 来展示错误的细节，然后终止程序。

虽然是我们决定异步处理程序做什么，但常见的错误是打印错误消息（会受程序中的其他消息的干扰），然后完成处理函数。除非有适当的错误处理策略，允许恢复已知的程序状态，并确信继续执行的安全，否则应该考虑在异步处理程序函数中终止应用程序，以便减少了在检测到错误的程序中出现结果错误的可能性。许多程序中，当出现异步异常时，终止是首选。

如果没有完善的错误恢复和处理机制，请考虑在输出有关错误的信息之后，终止程序。

处理程序的调用

运行时在特定时间调用异步处理程序。错误发生时不会立即报告，所以管理错误和编程（特别是多线程）将变得更加困难。异步处理程序会在以下特定时间调用：

1. 主程序在特定队列上调用 `queue::throw_asynchronous()` 时
2. 主程序在特定队列上调用 `queue::wait_and_throw()` 时
3. 主程序在特定事件上调用 `event::wait_and_throw()` 时
4. 一个队列销毁时
5. 一个上下文销毁时

方法 1-3 提供了一种控制点机制，让主机程序控制何时处理异步异常，这样就可以管理线程安全和其他特定程序的细节。在控制点上，异步异常可以进入主程序控制流，并且可以像处理同步错误一样处理它们。

如果用户没有显式地调用方法 1-3 的其中一个，那么当队列和上下文销毁时，通常会在程序关闭时报告异步错误。这足以向用户发出错误的信号，并表示不要信任本次程序的最终结果。

不过，程序的正确性依赖错误检测，并不是在所有情况下都有效。例如，如果程序只在某些算法收敛条件达到时才会终止，而这些条件只有通过成功执行设备内核才能实现，那么某个异步异常可能会让算法永远不收敛。这种情况下，以及在有更完整的错误处理策略的生产应用程序中，程序中使用常规调用和控制点调用 `throw_asynchronous()` 或 `wait_and_throw()` 都是有意义的（例如，在检查算法是否收敛之前调用）。

设备上出现错误

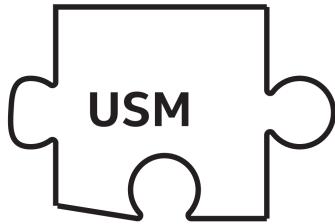
本章讨论基于主机的错误检测和处理机制，主机程序可以检测和处理可能在主程序中或在设备上执行内核期间出现的错误。本章没讨论如何在编写的设备代码中表明出了问题，这是我们的有意为之。

SYCL 显式地禁止在设备代码中使用 C++ 异常处理机制 (比如 throw)，对于某些设备，通常不想牺牲性能。如果检测到设备代码中出现了错误，应该使用现有的处理异常的技术发出错误信号，例如：可以写入缓冲区，该缓冲区记录错误，或返回一些无效结果以表明发生了错误的数值计算。在这些情况下，正确的策略有非常具体的应用。

总结

本章中，我们介绍了同步和异步错误，如果不采取任何措施来处理可能会发生的错误，以及期望的默认行为，还介绍了用于在程序控制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重要主题，在许多程序中代码量占了很大比例。SYCL 沿用了 C++ 处理错误的方法，并提供了灵活的机制来使用已知的错误管理策略。

6 统一共享内存



接下来的两章将深入探讨如何管理数据。有两种方法：统一共享内存（USM）和缓冲区。USM 的接口级别与缓冲区不同——USM 是指针，而缓冲区使用更高级别的接口。本章主要介绍 USM。

USM 是一种基于指针的内存模型，可以通过指针读写内存。

为什么要使用统一共享内存

USM 基于指针，对于基于指针的 C++ 代码来说使用很自然，使用指针作为参数的已有函数不需要进行修改。大多数情况下，唯一需要更改的是使用 USM 的分配方式替换现有的 malloc 或 new，我们将在本章中讨论这些分配方式。

分配类型

虽然 USM 基于指针，但不是所有指针都相同。USM 定义了三种不同的分配类型，每种都有单独的方式。设备可能不支持所有类型的 USM(甚至不支持)，后面我们会去了解到如何查询设备支持的 USM 类型。先来了解一下这三种类型的分配和特征，如图 6-1 所示。

图 6-1 USM 分配方式

类型	描述	主机端可访问？	设备端可访问？	位于
device	在设备内存上进行分配	✗	✓	设备
host	在主机内存上进行分配	✓	✓	主机端
shared	可在主机端和设备端共享	✓	✓	可以在主机和设备之间迁移

设备端内存

这种类型的分配可以拥有指向设备内存（如 (G)DDR 或 HBM）的指针。设备内存可以由运行在设备上的内核函数读取或写入，但是不能从主机直接访问。尝试直接访问设备端内存，可能导致数据不正确或程序崩溃。必须显式使用 USM 的 memcpy 机制在主机和设备之间复制数据，对两个位置上的数据进行复制，本章后面会对此继续讨论。

主机端内存

第二种类型更容易使用，不需要在主机和设备之间复制数据。在主机和设备上都可以访问主机内存，虽然在设备上可以访问，但不能迁移到设备端内存。对该内存进行读写的内核通常通过较慢的总线（如 PCI-Express）进行远程操作，所以必须对编程复杂性和性能进行权衡。尽管主机端的内

存可能导致很高的访问成本，但也有使用的场景，比如：很少访问的数据或存在设备内存中无法容纳的大型数据集。

共享内存

最后一种类型的内存分配结合了设备和主机内存的属性，结合了主机内存对编程复杂性的便利和设备分配提供的更好的性能，共享内存可以在主机和设备上访问。区别是共享内存可以自动地在主机内存和设备内存之间迁移，无需显式干预。如果数据已经迁移到该设备，那么在该设备上执行的内核访问该段数据，会比从主机远程访问该数据有更好的性能。不过，也有缺点。

自动迁移可以通过多种方式实现。不管运行时选择哪种方式来实现共享内存，通常都要付出延迟增加的代价。通过设备内存，可以确切地知道需要复制多少内存，并可以尽可能快地安排数据的复制。自动迁移机制无法对未来进行预见，某些情况下，直到内核尝试访问数据时才开始移动数据，导致内核必须等待或阻塞，直到数据移动完成。其他情况下，运行时很可能不知道内核函数将访问多少数据，可能移动更多的数据，这也会增加内核的延迟。

虽然共享分配可以迁移，但并不一定 DPC++ 的所有实现都会迁移。我们希望大多数实现都通过迁移实现共享，但有些设备可能希望实现与主机内存相同。这样的实现中，分配的内存存在主机和设备上仍然可见，但是可能无法感知迁移带来的性能提升。

分配内存

USM 可以以各种不同的方式分配内存，以满足不同的需求。在更详细地讨论所有方法之前，应该了解一下 USM 分配与 C++ 分配的区别。

需要知道什么？

普通的 C++ 程序可以以多种方式分配内存: new、malloc 或分配器。不管使用哪种语法，内存分配最终都由主机操作系统中的系统分配器执行。当在 C++ 中分配内存时，唯一需要考虑的是“需要多少内存？”和“可以分配多少内存？”但是，USM 需要更多的信息才能进行分配。

首先，USM 分配需要指定分配类型: device、host 或 shared。为了获得该分配所需的行为，使用正确的分配类型非常重要。每个 USM 必须指定一个上下文对象来进行分配，上下文表示可以在对应的设备上执行内核。可以把上下文看作是运行时存储设备状态的容器。大多数 DPC++ 程序中，开发者可能不直接与上下文交互，而只是传递上下文。

不能保证 USM 分配可以跨不同的上下文使用——所有 USM 的分配、队列和内核共享同一个上下文对象。通常，可以从队列中获得上下文。设备分配还要求指定在哪个设备上分配内存，因为我们不想过度分配设备的内存（除非设备能够支持这一点——本章后续会对数据迁移时进行更多的讨论）。USM 分配例程可以通过添加参数来区别于 C++ 原生的方式。

多重样式

想用单一的选项来取悦每个人是不可能的，就像有些人喜欢咖啡而不是茶，或者喜欢 emacs 而不是 vi 一样。USM 支持选择的多样性，并提供了几种不同类型的分配接口：C 风格、C++ 风格和 C++ 分配器风格。将逐个讨论，并指出相同点和不同点。

C 风格的分配方式

第一种类型的分配函数(图 6-2 中列出,在随后的图 6-6 和 6-7 中使用): malloc 函数中的内存分配,函数需要返回一个 void * 指针(模仿 C 语言)。同时,必须指定要分配的总字节数,当要分配 N 个类型为 X 的对象,就必须要求总字节数为 N * sizeof(X)。返回的指针为 void * 类型,必须将其转换为指向 X 类型的指针。这种方式非常简单,但由于需要进行大小计算和类型转换,因此会让代码看起来很冗长。

可以进一步将这种分配方式分为两类:命名函数和单函数。这两种方式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 分配的类型编码在函数名中,单函数 malloc 要求将 USM 分配的类型指定为一个参数。具体要使用哪种方式,还是要取决于我们的偏好。

这里先要提及的概念是内存对齐。malloc 的每个版本都有 aligned_alloc 对应项。malloc 函数返回与设备默认对齐的内存,它将返回一个合法的指针和一个有效的对齐方式。但在某些情况下,我们可能更喜欢手动指定对齐的方式,使用 aligned_alloc 变量指定所需的对齐方式。如果指定的方式非法,不要期望程序能正常工作!原则上的是 2 的幂次方。值得注意的是,在许多设备上,分配是最大对齐的,以对应硬件的特性。因此,我们可能要求分配为 4 字节、8 字节、16 字节或 32 字节对齐,但在实际可能会看到更大字节的对齐。

图 6-2 C 风格的 USM 分配函数

```
1 // Named Functions
2 void *malloc_device(size_t size, const device &dev, const context &ctxt);
3 void *malloc_device(size_t size, const queue &q);
4 void *aligned_alloc_device(size_t alignment, size_t size,
5     const device &dev, const context &ctxt);
6
7 void *aligned_alloc_device(size_t alignment, size_t size, const queue &q);
8
9 void *malloc_host(size_t size, const context &ctxt);
10 void *malloc_host(size_t size, const queue &q);
11 void *aligned_alloc_host(size_t alignment, size_t size, const context
&ctxt);
12 void *aligned_alloc_host(size_t alignment, size_t size, const queue &q);
13
14 void *malloc_shared(size_t size, const device &dev, const context &ctxt);
15 void *malloc_shared(size_t size, const queue &q);
16 void *aligned_alloc_shared(size_t alignment, size_t size,
17     const device &dev, const context &ctxt);
18 void *aligned_alloc_shared(size_t alignment, size_t size, const queue &q);
19
20 // Single Function
21 void *malloc(size_t size, const device &dev, const context &ctxt,
22     usm::alloc kind);
23 void *malloc(size_t size, const queue &q, usm::alloc kind);
24 void *aligned_alloc(size_t alignment, size_t size,
25     const device &dev, const context &ctxt,
```

```

27     usm::alloc kind);
28 void *aligned_alloc(size_t alignment, size_t size, const queue &q,
29     usm::alloc kind);

```

C++ 风格的分配方式

下一个 USM 分配函数 (在图 6-3 中列出) 与第一个非常相似。我们再次拥有分配例程的命名和单函数版本, 以及默认和用户指定的对齐版本。不同之处在于, 现在的函数是 C++ 模板化函数, 它分配类型为 T 的 Count 对象并返回类型为 T * 的指针。可以利用现代 C++ 进行简化, 不再需要以字节为单位手动计算分配的总大小, 或者将返回的指针强制转换为适当的类型。这也会让代码更紧凑、不容易出错。然而, 与 C++ 中的 new 不同, malloc 风格的接口并不为分配的对象调用构造函数——只是分配了足够的字节来适配该类型。

对于使用 USM 编写的代码来说, 这种分配方式是很好的起点。对于大量使用 C 或 C++ malloc 的现有 C++ 代码来说, C 的方式是很好的起点, 我们将在此基础上增加 USM 的使用。

图 6-3 C++ 风格的 USM 分配函数

```

1 // Named Functions
2 template <typename T>
3 T *malloc_device(size_t Count, const device &Dev, const context &Ctxt);
4 template <typename T>
5 T *malloc_device(size_t Count, const queue &Q);
6 template <typename T>
7 T *aligned_alloc_device(size_t Alignment, size_t Count, const device &Dev,
8     const context &Ctxt);
9
10 template <typename T>
11 T *aligned_alloc_device(size_t Alignment, size_t Count, const queue &Q);
12
13 template <typename T> T *malloc_host(size_t Count, const context &Ctxt);
14 template <typename T> T *malloc_host(size_t Count, const queue &Q);
15 template <typename T>
16 T *aligned_alloc_host(size_t Alignment, size_t Count, const context &Ctxt);
17 template <typename T>
18 T *aligned_alloc_host(size_t Alignment, size_t Count, const queue &Q);
19
20 template <typename T>
21 T *malloc_shared(size_t Count, const device &Dev, const context &Ctxt);
22 template <typename T> T *malloc_shared(size_t Count, const queue &Q);
23 template <typename T>
24 T *aligned_alloc_shared(size_t Alignment, size_t Count, const device &Dev,
25     const context &Ctxt);
26 template <typename T>
27 T *aligned_alloc_shared(size_t Alignment, size_t Count, const queue &Q);
28
29 // Single Function
30 template <typename T>

```

```

31 T *malloc(size_t Count, const device &Dev, const context &Ctxt,
32     usm::alloc Kind);
33 template <typename T> T *malloc(size_t Count, const queue &Q, usm::alloc
34 Kind);
35 template <typename T>
36 T *aligned_alloc(size_t Alignment, size_t Count, const device &Dev,
37     const context &Ctxt, usm::alloc Kind);
38 template <typename T>
39 T *aligned_alloc(size_t Alignment, size_t Count, const queue &Q,
40     usm::alloc Kind);

```

C++ 分配器

USM 分配的最后一种风格 (图 6-4) 使用了现代 C++, 这种风格基于 C++ 的 allocator 接口, 该接口定义了直接或间接地在容器 (如 std::vector) 中执行内存分配的对象。如果代码大量使用容器对象, 可以向用户隐藏内存分配和回收的细节, 简化代码减少出现 bug 的机会, 所以这种分配器风格最为实用。

图 6-4 C++ 分配器风格的 USM 分配函数

```

1 template <class T, usm::alloc AllocKind, size_t Alignment = 0>
2 class usm_allocator {
3 public:
4     using value_type = T;
5     template <typename U> struct rebind {
6         typedef usm_allocator<U, AllocKind, Alignment> other;
7     };
8
9     usm_allocator() noexcept = delete;
10    usm_allocator(const context &Ctxt, const device &Dev) noexcept;
11    usm_allocator(const queue &Q) noexcept;
12    usm_allocator(const usm_allocator &Other) noexcept;
13    template <class U>
14        usm_allocator(usm_allocator<U, AllocKind, Alignment> const &) noexcept;
15
16    T *allocate(size_t NumberOfElements);
17    void deallocate(T *Ptr, size_t Size);
18
19    template <
20        usm::alloc AllocT = AllocKind,
21        typename std::enable_if<AllocT != usm::alloc::device, int>::type = 0,
22        class U, class ... ArgTs>
23    void construct(U *Ptr, ArgTs &&... Args);
24
25    template <
26        usm::alloc AllocT = AllocKind,
27        typename std::enable_if<AllocT == usm::alloc::device, int>::type = 0,
28        class U, class ... ArgTs>

```

```

29 void construct(U *Ptr, ArgTs &&... Args);
30
31 template <
32     usm::alloc AllocT = AllocKind,
33     typename std::enable_if<AllocT != usm::alloc::device, int>::type = 0>
34 void destroy(T *Ptr);
35
36 template <
37     usm::alloc AllocT = AllocKind,
38     typename std::enable_if<AllocT == usm::alloc::device, int>::type = 0>
39 void destroy(T *Ptr);
40 };

```

释放内存

无论分配什么，最终都必须释放。USM 定义了一个方法来释放 malloc 或 aligned_malloc 函数分配的内存。这个方法还将分配内存的上下文作为一个参数（可以用队列替换上下文）。如果内存是用 C++ 的 allocator 对象分配，也应该使用该对象来释放内存。

图 6-5 三种配置方式

```

1 constexpr int N = 42;
2
3 queue Q;
4
5 // Allocate N floats
6
7 // C-style
8 float *f1 = static_cast<float*>(malloc_shared(N * sizeof(float) ,Q));
9
10 // C++-style
11 float *f2 = malloc_shared<float>(N, Q);
12
13 // C++-allocator-style
14 usm_allocator<float, usm::alloc::shared> alloc(Q);
15 float *f3 = alloc.allocate(N);
16
17 // Free our allocations
18 free(f1, Q.get_context());
19 free(f2, Q);
20 alloc.deallocate(f3, N);

```

内存分配示例

图 6-5 中，展示了如何使用三种方式进行分配，我们分配 N 个单精度浮点数作为共享分配内存。第一个分配 f1 使用 C 风格的 void * 返回 malloc 例程。对于这种分配，我们显式地传递从队列中获得的设备和上下文。

必须将结果强制转换为 `float*` 类型。第二个分配 `f2` 做了同样的事情，但是使用了 C++ 风格的 `malloc` 模板。因为我们将元素的类型 `float` 传递给分配示例，所以只需要指定分配多少个 `float` 即可，而不需要对结果进行强制转换。还可以使用队列，而不是设备和上下文的形式，完成一段非常简单和紧凑的代码。第三个分配 `f3` 使用了 USM C++ 的 `allocator` 类，实例化了 `allocator` 对象，然后使用该对象执行分配。最后，展示了如何正确地释放分配的内存。

数据管理

了解了如何使用 USM 内存后，来讨论下如何管理数据。可以将其分为两部分：数据初始化和数据移动。

初始化

数据初始化关注的是对内存执行计算前内存的填充值，常见初始化是使用零填充。要对使用 USM 内存进行初始化，可以通过多种方式来实现。最直接的就是写一个内核来做初始化，如果数据集特别大，或者需要复杂的计算，这种方法没问题。也可以实现为遍历所有元素的循环，将每个元素设置为 0，这种方法存在一个问题。循环可以很好地用于主机和共享分配的内存，因为可以在主机上访问。但在主机上不能访问设备分配，主机代码中的循环将不能对设备内存进行写入。这就有了第三种选择。

`memset` 函数可以用来实现这个初始化模式。USM 提供了一个 `memset` 函数，有三个参数：要设置的内存指针，要设置的模式，以及要设置为该模式的字节数。与主机的循环不同，`memset` 是并行发生的。

虽然 `memset` 是一个有用的操作，但只允许指定一个模式来填充。USM 还提供了 `fill` 方法，允许用任意模式填充内存。给它创建一个 `int` 型模板，然后用数字“42”填充内存。与 `memset` 类似，`fill` 接受三个参数：要填充的内存的指针，要填充的值，以及希望将该值写入内存的数量。

数据移动

数据移动可能是 USM 的重点。如果正确的数据没有在正确的时间出现在正确的地点，程序将产生不正确的结果。USM 定义了两种用来管理数据的策略：显式和隐式。使用策略的选择与硬件支持，或使用的 USM 类型有关。

显式

USM 提供的第一个策略是显式数据移动（图 6-6），必须显式地在主机和设备之间复制数据。可以通过调用 `memcpy` 完成，该方法可以在处理程序和队列类上找到。`memcpy` 方法有三个参数：指向目标内存的指针，指向源内存的指针，以及主机和设备之间复制的字节数。不需要指定复制发生的方向——这在源指针和目标指针中是隐式确定的。

显式数据移动最常见的用法是使用 USM 对设备内存中的数据进行复制，因为设备端内存存在主机上不可访问。此外，这可能会造成错误：可能会忽略复制，不正确的数据量可能被复制，或者源或目标指针可能不正确。

然而，显式数据移动也有优点：完全控制数据移动。某些应用程序中，控制复制数据的数量和复制数据的时间，对于获得最佳性能非常重要。理想情况下，可以将计算与数据移动重叠，确保硬

件高效率运行。

其他的 USM 类型，不论是 host，还是 shared，都可以在主机和设备端访问，不需要显式地复制到设备。这就引出了 USM 中数据移动的另一种策略。

图 6-6 USM 显式移动数据

```
1 constexpr int N = 42;
2
3 queue Q;
4
5 std::array<int, N> host_array;
6 int *device_array = malloc_device<int>(N, Q);
7 for (int i = 0; i < N; i++)
8     host_array[i] = N;
9
10 Q.submit([&](handler& h) {
11     // copy hostArray to deviceArray
12     h.memcpy(device_array, &host_array[0], N * sizeof(int));
13 });
14
15 Q.wait(); // needed for now (we learn a better way later)
16
17 Q.submit([&](handler& h) {
18     h.parallel_for(N, [=](id<1> i) {
19         device_array[i]++;
20     });
21 });
22
23 Q.wait(); // needed for now (we learn a better way later)
24
25 Q.submit([&](handler& h) {
26     // copy deviceArray back to hostArray
27     h.memcpy(&host_array[0], device_array, N * sizeof(int));
28 });
29
30 Q.wait(); // needed for now (we learn a better way later)
31
32 free(device_array, Q);
```

隐式

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)。这个策略中，数据移动是隐式发生的，不需要 memcpy，因为可以通过 USM 指针直接访问数据。而系统的任务是确保数据在使用时，在正确的位置上可用。

对于主机内存，可能会争论是否真的进行了数据移动。根据定义，分配的内存始终指向主机内存，因此给定的主机指针表示的内存不能存储在设备上。但当在设备上访问主机内存时，数据移动就会发生。不是将内存迁移到设备，而是通过适当的接口将读或写的值传输到内核中。这对于数据

不需要驻留在设备上的流内核很有用。

隐式数据移动主要与 USM 共享内存有关。这种类型的内存可以在主机和设备上访问，并且可以在主机和设备之间迁移。这种迁移是自动进行的，或者是隐式地进行的，只需访问不同位置的数据即可。接下来，讨论为共享内存进行数据迁移时需要考虑的几个问题。

图 6-7 USM 隐式数据移动

```
1 constexpr int N = 42;
2
3 queue Q;
4
5 int* host_array = malloc_host<int>(N, Q);
6 int* shared_array = malloc_shared<int>(N, Q);
7 for (int i = 0; i < N; i++)
8     host_array[i] = i;
9
10 Q.submit([&](handler& h) {
11     h.parallel_for(N, [=](id<1> i) {
12         // access sharedArray and hostArray on device
13         shared_array[i] = host_array[i] + 1;
14     });
15 });
16
17 Q.wait();
18
19 free(shared_array, Q);
20 free(host_array, Q);
```

迁移

通过显式数据移动，可以控制发生多少数据移动。使用隐式数据移动，系统可处理这一问题，但可能没有那么高效。DPC++ 运行时不是 oracle——不能预测应用将访问什么数据。此外，指针分析对于编译器来说非常困难，可能无法准确地分析和识别内核中可能使用的每个内存分配。因此，隐式数据移动机制的实现，可能会根据支持 USM 设备的功能做出不同的决策，这既影响共享内存的使用方式，也影响了它们的执行方式。

如果设备能力非常强，能够根据需要迁移内存。这种情况下，数据移动将在主机或设备试图访问数据不存在的内存位置时发生。按需获取数据极大地简化了编程，可以在任何地方访问 USM 共享内存，并且可以正常工作。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），仍然能够保证相同的语义，并对共享指针的使用方式进行限制。

限制形式的 USM 共享内存会确定何时何地可以访问共享 neicu8n，以及共享分配的大小。如果设备不能按需迁移内存，则运行时必须保守，并假定内核可以访问其设备附加内存中的任何分配。这会带来两种后果。

首先，主机和设备不该同时访问共享内存，程序应该以阶段替代访问。主机可以访问内存数据，然后内核可以使用该数据进行计算，最后主机读取结果。

如果没有这个限制，主机可以访问内核的不同分配。这种并发访问通常发生在设备内存页上。主机可以访问一个内存页，而设备可以访问另一个内存页。第 19 章将介绍原子访问相同的数据块。

这种受限的共享内存形式的第二个后果是，受到设备内存总量的限制。如果设备不能按需迁移内存，则无法将数据迁移到主机，为不同的数据腾出空间。如果设备支持按需迁移，则可能会超量使用内存，从而允许内核计算超过设备内存通常包含的数据，而这种灵活性可能会因为数据移动，产生性能损失。

细粒度控制

当设备支持按需迁移共享内存时，访问内存位置上没有相应数据时，需要进行数据移动。这时，内核在等待数据移动完成时可能会停止。接下来执行的语句可能会产生更多的数据移动，并给内核执行带来更多的延迟。

DPC++ 提供了一种修改自动迁移机制性能的方法。通过定义两个函数来做到这一点:prefetch 和 mem_advise。图 6-8 展示了每种方法，这些函数向运行时提供了内核如何访问数据的方式，以便运行时选择在内核访问数据之前开始移动数据。请注意，这个例子直接在队列对象上调用 parallel_for 的队列快捷方法，而不是在传递给 submit(命令组) 一个 Lambda 调用。

图 6-8 通过 prefetch 和 mem_advise 进行细粒度控制

```
1 // Appropriate values depend on your HW
2 constexpr int BLOCK_SIZE = 42;
3 constexpr int NUM_BLOCKS = 2500;
4 constexpr int N = NUM_BLOCKS * BLOCK_SIZE;
5
6 queue Q;
7 int *data = malloc_shared<int>(N, Q);
8 int *read_only_data = malloc_shared<int>(BLOCK_SIZE, Q);
9
10 // Never updated after initialization
11 for (int i = 0; i < BLOCK_SIZE; i++)
12     read_only_data[i] = i;
13
14 // Mark this data as "read only" so the runtime can copy it
15 // to the device instead of migrating it from the host.
16 // Real values will be documented by your DPC++ backend.
17 int HW_SPECIFIC_ADVICE_RO = 0;
18
19 Q.mem_advise(read_only_data, BLOCK_SIZE, HW_SPECIFIC_ADVICE_RO);
20
21 event e = Q.prefetch(data, BLOCK_SIZE);
22
23 for (int b = 0; b < NUM_BLOCKS; b++) {
24     Q.parallel_for(range{BLOCK_SIZE}, e, [=](id<1> i) {
25         data[b * BLOCK_SIZE + i] += data[i];
26     });
27     if ((b + 1) < NUM_BLOCKS) {
28         // Prefetch next block
29     }
30 }
```

```

29     e = Q.prefetch(data + (b + 1) * BLOCK_SIZE, BLOCK_SIZE);
30 }
31 }
32
33 Q.wait();
34
35 free(data, Q);
36 free(read_only_data, Q);

```

最简单的方法是调用预取 (prefetch)。此函数作为处理程序或队列类的成员函数，接受指针和字节数。这可以通知运行时某些数据将在设备上使用，以便能够及时地迁移。理想情况下，应该尽早了解这些信息，这样当内核访问数据时，数据就已经驻留在设备上了，从而消除之前所说的延迟。

DPC++ 提供的另一个函数是 mem_advise，这个函数可以确定内核中可以使用哪些特定于设备的内存。这样的话，当数据只在内核中读取，而不是写入，系统就可以意识到这个操作可以复制设备上的数据，这样在内核完成后就不需要更新主机的数据。但是，传递给 mem_advise 的参数是特定于设备的，所以使用此函数之前，请查阅硬件供应商的文档。

查询

最后，并不是所有设备都支持 USM 的特性。如果希望程序可以在不同的设备上移植，不应该假设所有的 USM 功能都可用。USM 的特性可以查询，这些查询可以分为两类：指针查询和设备功能查询。图 6-9 显示了每种方法的简单使用方法。

USM 中的指针查询需要回答两个问题。第一个问题是“这个指针指向哪种 USM 分配类型？”get_pointer_type 函数接受一个指针和 DPC++ 上下文，并返回一个 usm::alloc 类型的结果，可以有四种可能的值：host、device、shared 或 unknown。第二个问题是“这个 USM 指针分配给了什么设备？”可以向函数 get_pointer_device 传递一个指针和一个上下文，并获得一个设备对象。这主要用于设备或共享 USM 内存，对主机内存没什么意义。

USM 提供的第二种查询，关系到设备的性能。USM 扩展了可以通过在设备对象上调用 get_info 来查询的设备信息描述符列表。这些查询可用于测试设备支持哪种 USM 分配类型。此外，可以查询设备上的共享内存是否按照本章前面描述的方式进行了限制。完整的查询列表如图 6-10 所示。第 12 章中，会更详细地了解查询机制。

图 6-9 查询 USM 指针和设备

```

1 constexpr int N = 42;
2
3 template <typename T> void foo(T data, id<1> i) { data[i] = N; }
4
5 queue Q;
6 auto dev = Q.get_device();
7 auto ctxt = Q.get_context();
8 bool usm_shared = dev.get_info<dinfo::usm_shared_allocations>();
9 bool usm_device = dev.get_info<dinfo::usm_device_allocations>();
10 bool use_USM = usm_shared || usm_device;
11

```

```

12 if (use_USM) {
13     int *data;
14     if (usm_shared)
15         data = malloc_shared<int>(N, Q);
16     else /* use device allocations */
17         data = malloc_device<int>(N, Q);
18
19     std::cout << "Using USM with "
20     << ((get_pointer_type(data, ctxt) == usm::alloc::shared)
21     ? "shared"
22     : "device")
23     << " allocations on "
24     << get_pointer_device(data, ctxt).get_info<dinfo::name>()
25     << "\n";
26
27 Q.parallel_for(N, [=](id<1> i) { foo(data, i); });
28 Q.wait();
29 free(data, Q);
30 } else /* use buffers */ {
31     buffer<int, 1> data{range{N}};
32     Q.submit([&](handler &h) {
33         accessor a(data, h);
34         h.parallel_for(N, [=](id<1> i) {
35             foo(a, i);
36         });
37     });
38 }

```

图 6-10 USM 设备信息描述符

设备描述符	类型	描述
info::device::usm_device_allocations	bool	如果该设备支持设备分配，则返回 true
info::device::usm_host_allocations	bool	如果设备可以访问主机分配，则返回 true
info::device::usm_shared_allocations	bool	如果该设备支持共享分配，则返回 true
info::device::usm_restricted_shared_allocations	bool	如果共享分配受本章描述的限制约束，则返回 true

总结

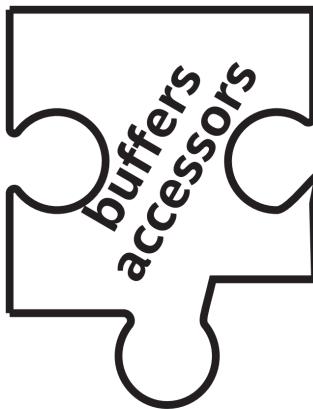
本章中，描述了统一共享内存，一种基于指针的数据管理策略。介绍了 USM 的三种类型的内存分配。讨论了使用 USM 分配和释放内存的所有不同方式，以及数据移动可以由开发者显式控制进行设备分配，也可以由系统隐式控制共享内存。最后，讨论了如何查询设备支持的不同 USM 功

能，以及如何在程序中查询关于 USM 指针的信息。

因为还没有在本书中详细讨论同步，所以在后面的章节中会讨论调度、通信和同步的时候会有更多关于 USM 的内容。将在第 8、9 和 19 章中继续讨论 USM。

下一章中，我们将介绍数据管理的第二种策略：缓冲区。

7 内存



本章中，我们将了解缓冲区。前一章中了解了基于指针的 USM (Unified Shared Memory, USM)。USM 需要我们思考内存存在于哪里，怎么访问。缓冲区是一个高级模型，向开发者隐藏了底层细节。缓冲区只是表示数据，而管理数据在内存中存储和移动的方式就成了运行时的工作。

本章介绍了一种管理数据的替代方法。缓冲区和 USM 之间的选择通常取决于个人偏好和现有代码的风格，应用程序可以自由地混合两种风格来表示应用程序中的不同数据。

USM 只是内存的不同抽象。USM 有指针，而缓冲区是更高级的抽象。缓冲区的抽象级别允许在应用程序的任何设备上可用，包含在运行时数据管理。

我们将详细地了解缓冲区的创建和使用。如果不讨论访问器，对缓冲区的讨论就不完整。虽然缓冲区抽象了程序中表示和存储数据的方式，但不能使用缓冲区直接访问数据。需要使用访问器对象告知运行时如何访问数据，并且访问器会与任务图中的数据依赖机制紧密耦合。介绍了使用缓冲区可以做的所有事之后，还将探讨如何在程序中创建和使用访问器。

介绍

缓冲区是数据的高级抽象。缓冲区不必绑定到单个位置或虚拟内存地址上。运行时可以使用内存中的许多位置（甚至跨不同设备）来表示缓冲区，但运行时必须提供一致的数据视图。程序在主机和任何设备上都可以访问缓冲区。

图 7-1 缓冲区类型的定义

```
1 template <typename T, int Dimensions, AllocatorT allocator>
2 class buffer;
```

buffer 类是一个模板类，有三个模板参数，如图 7-1 所示。第一个模板参数是缓冲区包含的对象的类型，这个类型必须可复制，从而可以安全地逐字节地复制，而不需要使用特殊的 copy 或 move 函数。下一个模板参数是描述缓冲区维度的数量。模板的最后一个参数是可选的，通常使用默认值。此参数指定一个分配器类，用于在主机上分配缓冲区所需的内存。首先，我们来看下究竟创建缓冲区对象的方式。

创建

下图中，展示了创建缓冲区对象的几种方式。如何在程序代码中创建缓冲区决定于使用缓冲区的方式和个人偏好。浏览一下这个示例的每个实例。

图 7-2 创建缓冲区，第 1 部分

```
1 // Create a buffer of 2x5 ints using the default allocator
2 buffer<int, 2, buffer_allocator> b1{range<2>{2, 5}};
3
4 // Create a buffer of 2x5 ints using the default allocator
5 // and CTAD for range
6 buffer<int, 2> b2{range{2, 5}};
7
8 // Create a buffer of 20 floats using a
9 // default-constructed std::allocator
10 buffer<float, 1, std::allocator<float>> b3{range{20}};
11
12 // Create a buffer of 20 floats using a passed-in allocator
13 std::allocator<float> myFloatAlloc;
14 buffer<float, 1, std::allocator<float>> b4{range(20), myFloatAlloc};
```

图 7-2 中创建的第一个缓冲区 b1，是一个包含 10 个整数的二维缓冲区。显式传递所有模板实参，显式传递 buffer_allocator 的默认值作为分配器类型，使用现代 C++ 可以更简单地表达，缓冲区 b2 也是包含 10 个整数的二维缓冲区。这里，使用 C++17 的类模板实参推断 (CTAD) 来自动推断模板实参。

CTAD 需要推断一个类的每个模板参数，或者一个也不推断。本例中，使用带两个参数的 range 初始化 b2，来确定它是一个二维的 range。allocator 模板参数有一个默认值，所以在创建缓冲区时不需要显式地列出。

对于缓冲区 b3，创建了一个包含 20 个浮点数的缓冲区，并使用默认构造的 std::allocator<float> 来分配主机上的内存。当自定义分配器类型与缓冲区一起使用时，通常希望将实际的分配器对象传递给缓冲区使用，而不是默认构造的分配器对象。b4 展示了如何做到这一点。

对于示例中的前四个缓冲区，让缓冲区分配需要的内存，并且在创建数据时不使用任何值初始化数据。使用缓冲区封装内存分配是一种常见的模式，这些内存可能已经初始化了。可以通过向缓冲区构造函数传递初始值进行初始化。

图 7-3 创建缓冲区，第 2 部分

```
1 // Create a buffer of 4 doubles and initialize it from a host pointer
2 double myDoubles[4] = {1.1, 2.2, 3.3, 4.4};
3 buffer b5{myDoubles, range{4}};
4
5 // Create a buffer of 5 doubles and initialize it from a host pointer
6 // to const double
7 const double myConstDbls[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
8 buffer b6{myConstDbls, range{5}};
9
10 // Create a buffer from a shared pointer to int
```

```
11 auto sharedPtr = std::make_shared<int>(42);
12 buffer b7{sharedPtr, range{1}};
```

图 7-3 中, b5 创建了一个具有 4 个 double 的一维缓冲区。除了指定缓冲区大小的范围外, 还将指向 C 数组 mydouble 的主机指针传递给缓冲区构造函数。这里, 可以充分利用 CTAD 来推断缓冲区的所有模板参数。缓冲区的数据, 是通过传递 double 的主机指针获取。维数是自动从一维 range 中推断出来的, 而这个 range 是由一个数字创建的。最后, 使用了默认分配器, 因此不必进行指定。

传递主机指针有几个要点。通过传递指向主机内存的指针, 在缓冲区的生命周期内, 不会尝试访问主机内存。这不是(也不能)由 SYCL 实现强制执行的——确保不违反此契约是我们的责任。当缓冲区存在时, 不应该尝试访问该内存的原因是, 缓冲区可能会选择使用主机上的不同内存来表示缓冲区内容, 这通常出于优化的考虑。如果这样做, 值将从主机指针复制到新内存中。如果后续的内核修改缓冲区, 原始的主机指针将不会更新其内容, 直到特定的同步点。本章的后面, 我们将更多地讨论数据何时会写回主机指针。

b6 与 b5 非常相似, 但有一个区别, 使用指向 const double 的指针初始化缓冲区。我们只能通过宿主指针读取值, 而不能写。但是, 本例中缓冲区的类型是 double, 而不是 const double, 因为推导时没有考虑到常量。这意味着内核可能会修改缓冲区内的数据, 所以需要使用不同的机制来更新主机缓冲区(在本章后面讨论)。

可以使用 C++ 共享内存对象初始化缓冲区。如果程序已经使用了共享内存, 这种初始化方法将正确地计算引用, 并确保内存不会释放。b7 使用一个整数初始化缓冲区维度, 并使用共享内存初始化缓冲区数据。

图 7-4 创建缓冲区, 第 3 部分

```
1 // Create a buffer of ints from an input iterator
2 std::vector<int> myVec;
3 buffer b8{myVec.begin(), myVec.end()};
4 buffer b9{myVec};
5
6 // Create a buffer of 2x5 ints and 2 non-overlapping
7 // sub-buffers of 5 ints.
8 buffer<int, 2> b10{range{2, 5}};
9 buffer b11{b10, id{0, 0}, range{1, 5}};
10 buffer b12{b10, id{1, 0}, range{1, 5}};
```

容器通常在现代 C++ 应用程序中使用, 例如: std::array、std::vector、std::list 或 std::map。可以用两种不同的方式使用容器初始化一维缓冲区。第一种方法, 如图 7-4 所示, 使用 b8 和使用输入迭代器。将两个迭代器传递给缓冲区构造函数, 一个表示数据的开始, 另一个表示结束。通过递增 start 迭代器, 直到等于 end 迭代器所返回的元素个数来计算缓冲区的大小。这对于任何实现 C++ 输入迭代器接口的数据类型都很有用。如果为缓冲区提供初始值的容器对象是连续的, 可以使用更简单的方式来创建缓冲区。b9 通过将 vector 传递给构造函数来 vector 创建缓冲区。缓冲区的大小由初始化容器的大小决定, 缓冲区数据的类型源于容器内的数据类型, 建议在使用 std::vector 和 std::array 时, 这样创建缓冲区。

缓冲区创建的最后一个示例说明了缓冲区类的另一个特性。可以从另一个缓冲区或子缓冲区创建缓冲区。子缓冲区需要三样东西：对父缓冲区的引用、基索引和子缓冲区的范围。不能从子缓冲区创建子缓冲区，同一个缓冲区可以创建多个子缓冲区，可以自由重叠。b10 的创建方式与 b2 完全相同，是一个每行 5 个整数的二维缓冲区。我们从 b10、子缓冲区 b11 和 b12 创建两个子缓冲区。子缓冲区 b11 从 index(0,0) 开始，包含第一行中的每个元素。类似地，子缓冲区 b12 从 index(1,0) 开始，包含第二行中的每个元素。这将产生两个不相交的子缓冲区。由于子缓冲区不重叠，不同的内核可以同时对不同的子缓冲区进行操作，我们会在下一章更多地讨论调度执行图和依赖性相关的话题。

图 7-5 缓冲区属性

```
1 queue Q;
2 int my_ints[42];
3
4 // create a buffer of 42 ints
5 buffer<int> b{range(42)};
6
7 // create a buffer of 42 ints, initialize
8 // with a host pointer, and add the
9 // use_host_pointer property
10 buffer b1{my_ints, range(42),
11     {property::buffer::use_host_ptr{}}};
12
13 // create a buffer of 42 ints, initialize pointer,
14 // with a host and add the use_mutex property
15 std::mutex myMutex;
16 buffer b2{my_ints, range(42),
17     {property::buffer::use_mutex{myMutex}}};
18
19 // Retrive a pointer to the mutex used by this buffer
20 auto mutexPtr =
21     b2.get_property<property::buffer::use_mutex>().
22     get_mutex_ptr();
23
24 // lock the mutex until we exit scope
25 std::lock_guard<std::mutex> guard{*mutexPtr};
26
27 // create a context-bound buffer of 42 ints,
28 // initialized from a host pointer
29 buffer b3{my_ints, range(42),
30     {property::buffer::context_bound{Q.get_context()}}};
```

缓冲区属性

还可以用特殊属性创建缓冲区，以改变缓冲区行为。图 7-5 中，演示三个不同的可选缓冲区属性的示例，并讨论如何使用它们。注意，这些属性在大多数代码中相对不常见。

use_host_ptr

在缓冲区创建期间可以选择性的指定第一个属性是 `use_host_ptr`, 此属性要求缓冲区不在主机上分配任何内存, 并且在缓冲区构造上传递或指定的任何分配器都会忽略。缓冲区必须使用传递给构造函数的主机指针所指向的内存。注意, 这并不需要设备使用相同的内存来保存缓冲区的数据。设备可以自由地将缓冲区内容缓存到存储器中。还有, 此属性只能在主机指针传递给构造函数时使用。当程序希望完全控制所有主机内存分配时, 这个选项很有用。

图 7-5 中, 我们创建了一个缓冲区 `b`。接下来, 我们创建缓冲区 `b1`, 并用指向 `myint` 的指针初始化它。还传递了属性 `use_host_ptr`, 这意味着缓冲区 `b1` 将只使用 `myint` 所指向的内存, 而不会分配任何额外的存储空间。

use_mutex

`use_mutex` 关注缓冲区和主机代码之间的细粒度内存共享。缓冲区 `b2` 使用此属性创建, 该属性采用对互斥对象的引用, 该对象可以从缓冲区中查询。此属性还要求将主机指针传递给构造函数, 并让运行时确定何时可以安全地通过提供的主机指针, 访问主机代码中的数据。运行时保证主机指针能看到缓冲区的值之前, 不能锁定互斥锁。虽然可以与 `use_host_ptr` 属性结合使用, 但这不是必需的。`use_mutex` 是一种机制, 允许主机代码在缓冲区活跃的情况下访问缓冲区中的数据, 而不使用主机访问器机制 (稍后介绍)。除非有特定的理由使用互斥锁, 否则应该优先使用主机访问器机制, 特别是在互斥锁成功锁定和主机代码使用数据之前。

context_bound

最后一个属性展示在示例中创建缓冲区 `b3` 的过程中。42 个整数的缓冲区用 `context_bound` 属性创建, 该属性接受对上下文对象的引用。缓冲区可以在任何设备或上下文上使用。如果使用此属性, 则将缓冲区绑定到指定的上下文, 试图在另一个上下文上使用缓冲区将导致运行时错误。这对于调试程序很有帮助, 例如: 通过识别内核可能提交到错误队列的情况, 可以确定错误产生的位置。实际上, 我们不想在许多程序中看到这个属性, 并且上下文在任何设备上访问缓冲区的能力是缓冲区最强大的属性之一 (这个属性可以撤消)。

可以用缓冲区做什么?

可以用缓冲区做很多事情, 查询缓冲区的特征, 确定缓冲区销毁后是否, 以及在哪里有数据写回主机内存, 或者将缓冲区重新解释为具有不同特征的缓冲区。然而, 不能直接访问缓冲区的数据, 必须创建访问器对象来访问数据。

可以查询缓冲区的示例包括范围、数据元素的总数, 以及存储元素所需的字节数。还可以查询缓冲区正在使用哪个分配器对象, 以及该缓冲区是否为子缓冲区。

缓冲区销毁时更新主机内存, 根据缓冲区创建的方式, 在缓冲区销毁后, 主机内存可能会更新。如果缓冲区是从指向非 `const` 数据的主机指针创建并初始化的, 那么当缓冲区销毁时, 该指针将使用已更新的数据。然而, 还有一种方法可以更新主机内存, 而不管缓冲区如何创建。`set_final_data` 方法是缓冲区的模板方法, 可以接受指针、C++ 输出迭代器或 `std::weak_ptr`。缓冲区销毁时, 数据将写入主机。注意, 如果缓冲区是从指向非 `const` 数据的主机指针创建并初始化, 就好像是用该指针调用了 `set_final_data`。从技术上讲, 指针是输出迭代器的特例。如果传递给 `set_final_data`

的参数是一个 `std::weak_ptr`, 当指针已经过期或已经删除, 则数据不会写入主机。是否发生回写也可以由 `set_write_back` 控制。

访存器

由缓冲区表示的数据不能直接访问, 必须创建访问器对象进行访问。访问器告知运行时希望在何处以及如何访问数据, 从而允许运行时确保正确的数据在正确的时间出现在正确的位置。这是一个非常强大的概念, 特别是与任务图结合使用时, 任务图基于数据依赖来调度内核的执行。

访问器类有 5 个模板参数。第一个参数是访问数据的类型, 这应该与对应缓冲区中存储的数据类型相同。第二个参数描述了数据和缓冲区的维度, 默认值为 1。

图 7-6 访问模式

模式	描述
read	只能读取
write	只能写, 保留以前的内容
read_write	读写都可以

接下来的三个模板参数对访问器的设置。第一个是访问模式, 描述了如何在程序中使用访问器。支持的模式如图 7-6 所示。我们将在第 8 章了解如何使用这些模式来安排内核的执行和数据移动。如果没有指定或自动推断出访问模式参数, 则访问模式参数具有默认值。如果不指定, 对于非 `const` 数据类型, 访问器默认为 `read_write` 访问模式, 对于 `const` 数据类型, 默认为 `read`。这些默认值没有问题, 但提供更准确的信息可以提高运行时执行优化的能力。开始应用程序开发时, 不指定访问模式是安全而简洁的, 然后可以根据对应用程序性能关键区域的分析来细化访问模式。

图 7-7 访问目标

目标	描述
global_buffer	通过全局内存访问一个缓冲区
constant_buffer	通过常量内存访问缓冲区
local	访问工作组本地内存
unsampled_image	访问 <code>unsampled_image</code>
sampled_image	访问 <code>sampled_image</code>
host_buffer	访问主机上的缓冲区
host_unsampled_image	访问 <code>host_unsampled_image</code>
host_sampled_image	访问 <code>host_sampled_image</code>

下一个模板参数是访问目标。缓冲区是数据的抽象, 所以隐藏了数据存储的位置和方式。访问目标既描述了正在访问的数据类型, 也描述了哪些内存将包含该数据, 可访问目标如图 7-7 所示。数据类型是两种类型中的一种: 缓冲区或图像。本书中讨论了图像, 可以把它们看作是为图像处理提供特定领域的缓冲区。

另一方面，设备可能有不同类型的内存，这些内存由不同的地址空间表示，常用的内存类型是设备的全局内存。内核中的大多数访问器都会使用这个目标，所以 global 是默认的目标（如果没有指定）。常量和本地缓冲区使用特殊用途的内存，常量内存用于存储内核内的常量值，本地内存是工作组可用的特殊内存，其他工作组不能访问。我们将在第 9 章了解如何使用本地内存。另一个需要注意的是主机缓冲区，访问主机上的缓冲区时使用的目标。这个模板形参的默认值是 global_buffer，所以在大多数情况下，不需要在代码中指定目标。

最终模板形参决定访问器是否为占位符访问器，这不是开发者直接设置的参数。占位符访问器在命令组之外声明，但用于访问内核内设备上的数据。通过创建访问器的例子，将了解占位符访问器与非占位符访问器的区别。

虽然使用缓冲区对象的 get_access 方法从缓冲区对象中提取访问器，但直接创建（构造）更简单。接下来的例子会非常简单，并且容易理解。

创建访问器

图 7-8 显示了一个示例程序，其中包含了使用访问器所需的所有内容。本例中，有三个缓冲区：A、B 和 C。我们提交给队列的第一个任务是为每个缓冲区创建访问器和定义内核，内核使用这些访问器初始化缓冲区。每个访问器都用的是缓冲区的引用，以及由开发者提交给队列的命令组定义的处理程序对象构造的。这有效地将访问器绑定为命令组的一部分进行提交。常规访问器是设备访问器，默认情况下，它们的目标是存储在设备内存中的全局缓冲区。这是最常见的情况。

图 7-8 简单的访问器创建

```
1 constexpr int N = 42;
2
3 queue Q;
4
5 // create 3 buffers of 42 ints
6 buffer<int> A{range{N}};
7 buffer<int> B{range{N}};
8 buffer<int> C{range{N}};
9 accessor pC{C};
10
11 Q.submit([&](handler &h) {
12     accessor aA{A, h};
13     accessor aB{B, h};
14     accessor aC{C, h};
15     h.parallel_for(N, [=](id<1> i) {
16         aA[i] = 1;
17         aB[i] = 40;
18         aC[i] = 0;
19     });
20 });
21
22 Q.submit([&](handler &h) {
23     accessor aA{A, h};
24     accessor aB{B, h};
```

```

25 accessor aC{C, h};
26 h.parallel_for(N, [=](id<1> i) {
27     aC[i] += aA[i] + aB[i]; });
28 });
29
30 Q.submit([&](handler &h) {
31     h.require(pC);
32     h.parallel_for(N, [=](id<1> i) {
33         pC[i]++;
34     });
35
36 host_accessor result{C};
37 for (int i = 0; i < N; i++)
38     assert(result[i] == N);

```

提交的第二个任务还定义了三个缓冲区访问器。然后，第二个内核中使用这些访问器将缓冲区 A 和 B 的元素添加到缓冲区 C 中。因为第二个任务与第一个任务操作相同的数据，所以运行时将在第一个任务完成后执行。

第三个任务展示了如何使用占位符访问器。创建缓冲区之后，图 7-8 中的示例的开头声明访问器 pC。请注意，构造函数没有传递处理程序对象，因为没有要传递的处理程序对象，可以提前创建一个可重用的访问器对象。然而，为了在内核中使用访问器，需要在提交时将它绑定到一个命令组，使用处理程序对象的 require 方法可以实现。将占位符访问器绑定到命令组时，就可以像使用其他访问器一样，在内核中使用它了。

最后，创建一个 host_accessor 对象，以便在主机上读取计算结果。注意，这与内核中使用的类型不同。主机访问器使用单独的 host_accessor 类来推断模板参数，并提供简单的对外接口。注意，本例中的主机访问器结果不接受处理程序对象，因为我们没有要传递的处理程序对象。主机访问器的特殊类型，允许将它们与占位符区分。主机访问器的一个要点是，只有当数据在主机上可用时，构造函数才会完成，主机访问器的构造可能需要很长时间。构造函数必须等待所需数据的内核上执行完毕，并通过复制数据完成构造。当主机访问器构造完成时，就可以使用它直接访问主机上的数据，并且可以保证在主机上使用的是最新的数据。

虽然示例完全正确，但在创建访问器时，并没有说明如何使用。对于缓冲区中的非 const int 数据，使用默认的访问模式（即读写模式），可能过于保守，可能在操作之间产生不必要的依赖关系或多余的数据移动。如果运行时能够提供更多关于如何使用访问器的信息，可能会更好。但在介绍这样做的示例之前，应该首先介绍另一个工具——访问标识。

访问标识是表达访问器所需的访问模式和目标组合的一种方式。使用访问标记时，将作为参数传递给访问器的构造函数。可能的标识如图 7-9 所示。当使用标识参数构造访问器时，CTAD 可以正确地推导出所需的访问模式和目标，从而提供简单的方法来覆盖这些模板参数的默认值。我们也可以手动指定所需的模板参数，标识提供了一种更简单、更紧凑的方式来获得相同的结果，而无需拼写出完全模板化的访问器。

图 7-9 访问标识

标识类型	标识值	访问模式	访问目标
mode_tag_t	read_write	read_write	default
mode_tag_t	read_only	read	default
mode_tag_t	write_only	write	default
mode_target_tag_t	read_constant	read	constant_buffer

让我们以前面的示例为例，重写以添加访问标识。这个改进示例如图 7-10 所示。

图 7-10 使用指定使用方式创建访问器

```

1 constexpr int N = 42;
2
3 queue Q;
4
5 // Create 3 buffers of 42 ints
6 buffer<int> A{range{N}};
7 buffer<int> B{range{N}};
8 buffer<int> C{range{N}};
9
10 accessor pC{C};
11
12 Q.submit([&](handler &h) {
13     accessor aA{A, h, write_only, noinit};
14     accessor aB{B, h, write_only, noinit};
15     accessor aC{C, h, write_only, noinit};
16     h.parallel_for(N, [=](id<1> i) {
17         aA[i] = 1;
18         aB[i] = 40;
19         aC[i] = 0;
20     });
21 });
22
23 Q.submit([&](handler &h) {
24     accessor aA{A, h, read_only};
25     accessor aB{B, h, read_only};
26     accessor aC{C, h, read_write};
27     h.parallel_for(N, [=](id<1> i) {
28         aC[i] += aA[i] + aB[i];
29     });
30 });
31 Q.submit([&](handler &h) {
32     h.require(pC);
33     h.parallel_for(N, [=](id<1> i) {
34         pC[i]++;
35     });
36 });
37 host_accessor result{C, read_only};

```

```
38  
39 for (int i = 0; i < N; i++)  
40     assert(result[i] == N);
```

首先声明缓冲区，如图 7-8 所示。我们还创建了占位符访问器，现在看看提交给队列的第一个任务。之前，通过传递对缓冲区的引用和命令组的处理程序对象来创建访问器。现在，向构造函数调用添加两个参数。第一个参数是一个访问标识，因为这个内核正在为缓冲区写入初始值，所以使用 write_only 访问标记。这让运行时知道这个内核正在生成新数据，从而不会从缓冲区中读取数据。

第二个参数是一个可选的访问器属性，类似于本章前面看到的缓冲区的可选属性。我们传递的属性 noinit，让运行时知道缓冲区前面的内容可以丢弃，可以让运行时消除不必要的数据移动。本例中，由于第一个任务是为缓冲区写入初始值，所以运行时没有必要在内核执行之前将未初始化的主机内存复制到设备中。noinit 属性对于这个示例很有用，但是不应该用于读-改-写，或者内核函数只更新缓冲区中的一些值的情况。

我们提交给队列的第二个任务与之前的任务相同，但是现在向访问器添加了访问标识。向访问器 A 和 B 添加 read_only 标识，以让运行时知道我们只会通过这些访问器读取缓冲区 A 和 B 的值。第三个访问器 aC 为 read_write 访问标识，因为我们将 A 和 B 的元素之和累积到 c 中。我们在示例中显式地使用标记以保持一致，但这没必要，因为默认的访问模式就是 read_write。

使用占位符访问器的第三个任务中保留了默认用法，这与图 7-8 中看到的简化示例一样。最终访问器，即主机访问器结果，在创建时接收到一个访问标识。因为我们只读取主机上的最终值，所以将 read_only 标记传递给构造函数。如果改写程序，这样主机访问器销毁，启动另一个内核缓冲区 C 操作时，当使用 read_only 标识时，就不需要写回到设备端了，并且运行时知道主机不会修改这个缓冲区。

可以用访问器做什么？

使用访问器对象可以完成很多事情，最重要的是通过访问器的 [] 操作符可以完成对数据的访问。在图 7-8 和 7-10 的例子中使用 [] 操作符，其可以接受索引多维数据的 id 对象或单个 size_t，当访问器有多个维度时使用第二种方式。在查询到达标量值之前，它会返回一个对象，该对象可以再次使用 [] 进行索引。二维情况下，可以表示为 a[i][j]，访问器维度的排序遵循 C++ 的约定。

访问器还可以返回指向底层数据的指针，这个指针可以按照 C++ 规则直接访问。对于这个指针的地址空间，可能会比较复杂。地址空间及其特殊性将在后面的章节中讨论。

还可以通过访问器对象查询许多内容。示例包括通过访问器可访问的元素数量、所覆盖的缓冲区区域的字节大小或可访问的数据范围。

访问器提供了与 C++ 容器类似的接口，可以在许多可能传递容器的情况下使用。访问器支持的容器接口包括 data 方法 (相当于 get_pointer)，以及几种不同类型的前向迭代器和后向迭代器。

总结

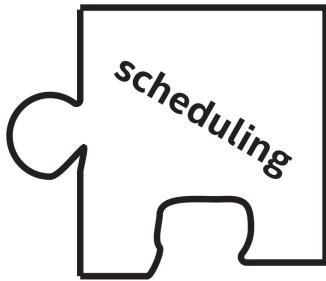
本章中，我们了解了缓冲区和访问器。缓冲区是数据的抽象，向开发者隐藏了内存管理的底层细节。这样做是为了提供更简单、更高层次的抽象。我们介绍了几个示例，这些示例展示了构建缓冲区的不同方法，以及可以指定用来改变其行为的不同可选属性。我们还了解了如何用来自主机内

存的数据初始化缓冲区，以及如何在使用完缓冲区后将数据写回主机内存。

不应该直接访问缓冲区，所以了解了如何使用访问器对象访问缓冲区中的数据，了解了设备访问器和主机访问器之间的区别。并讨论了不同的访问模式和目标，以及它们如何通知运行时程序在何处使用访问器。还展示了使用默认访问模式和目标来使用访问器的最简单方法，并学习了如何区分占位符访问器和非占位符访问器。然后，了解了如何通过向访问器声明添加访问标记，向运行时提供有关访问器使用的更多信息，从而进一步优化示例程序。最后，介绍了程序中使用访问器的不同方式。

下一章中，我们将更详细地了解运行时如何使用访问器提供的信息，来调度不同内核的执行。我们还将看到这些信息，如何通知运行时在主机和设备之间复制缓冲区中数据的时间和方式。我们将学习如何显式地控制涉及缓冲区和 USM 分配的数据移动。

8 调度内核和数据移动



我们需要对并行大师进行讨论。编排并行程序是一件美妙的事情——因为把所有数据安排的明明白白，所以代码无需等待就能全速运行。并且代码进行了分解，以保持硬件最大程度的使用。

生活在快车道上——而不仅仅是一条车道!——我们要认真地对待调度工作。为了做到这一点，可以用任务图来规划工作。

本章中，我们将讨论任务图，用于正确有效地运行复杂内核序列的机制。在应用程序中，有两件事情需要排序：内核和数据移动。任务图是我们用来实现正确排序的机制。

首先，快速回顾如何使用依赖项来编排第 3 章中的任务。接下来，将介绍 DPC++ 运行时如何构建图。我们将讨论 DPC++ 图的基本构造块，即命令组。然后，说明构建图的不同方法。还会讨论数据移动(包括显式和隐式)如何用图表示。最后，将讨论与主机同步图的各种方法。

什么是图调度？

第 3 章中，我们讨论了数据管理和数据使用的排序。描述了 DPC++ 中图的关键：依赖。内核间的依赖基本上是基于内核所访问的数据。计算输出之前，内核需要确定是否读取了正确的数据。

我们描述了三种类型的数据依赖，它们对于正确执行非常重要。第一种是读后写(RAW)，当一个任务需要读取由另一个任务产生的数据时发生。这种类型的依赖描述了两个内核之间的数据流。第二种依赖发生在一个任务在另一个任务读取数据后需要更新数据时，称之为写后读(WAR)依赖。当两个任务试图写入相同的数据时，就会出现最后一种数据依赖，这就是写后写(WAW)依赖关系。

数据依赖关系是用于构建图的构建块。这组依赖关系是我们所需要的，既可以表示简单的线性链，也可以表示具有复杂依赖关系的大型复杂图。无论计算需要哪种类型的图，DPC++ 图都能确保程序基于依赖正确执行。然而，开发者必须确保图能正确地表达程序中的所有依赖关系。

DPC++ 中如何操作图

命令组可以包含三种东西：操作、依赖关系和主机代码。这三样中，必须的是操作。大多数命令组也会有依赖关系，但有些情况下可能不会(比如：程序中提交的第一个操作)，不依赖任何东西来执行，因此不会指定任何依赖关系。在命令组中可能出现的是在主机上执行的 C++ 代码，这可以帮助指定操作或其依赖项，并且在创建命令组时执行此代码。

命令组通常表示为传递给 submit 的 C++ Lambda 表达式。命令组还可以通过队列对象上的快捷方法表示，这些队列对象采用了内核和基于事件的依赖项。

命令组行为

命令组可以执行两种类型的操作：内核操作和显式内存操作。一个命令组只能执行一个操作。正如前面的章节所示，内核通过调用 parallel_for 或 single_task 进行定义，并表示在设备上执行的计算。用于显式数据移动的操作是第二种类型的操作。USM 包括 memcpy、memset 和 fill 操作。缓冲区中包括 copy、fill 和 update_host。

命令组如何声明依赖关系

命令组的另一个主要组成是在执行组定义的操作之前满足的依赖。DPC++ 允许以多种方式指定这些依赖项。

如果程序使用有序的 DPC++ 队列，队列的有序语义指定了进入队列的命令组之间的隐式依赖关系。在之前的任务完成之前，其他任务不能执行。

基于事件的依赖关系，是指定在执行命令组之前必须完成任务的另一种方法。这些基于事件的依赖项可以通过两种方式指定。将命令组指定为传递给队列的 submit 的 Lambda 时，使用第一种方法。这种情况下，开发者调用命令组处理程序对象的 depends_on 方法，传递一个事件或事件组作为参数。当从队列对象上定义的快捷方法创建命令组时，使用另一种方法。当开发者直接调用队列上的 parallel_for 或 single_task 时，一个事件或事件组可以作为参数。

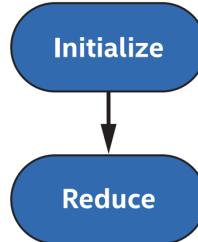
指定依赖项的最后一种方法是通过创建访问器对象。访问器指定如何来读取或写入缓冲区对象中的数据，让运行时使用该信息确定不同内核之间存在的数据依赖关系。如在本章开头的，数据依赖的例子包括：一个内核读取另一个内核生成的数据，两个内核写入相同的数据，或者一个内核在另一个内核读取数据后修改数据。

示例

现在用几个例子来说明刚刚学过的东西。我们将介绍如何用几种方式表达两种不同的依赖模式，说明两种模式是线性依赖的，其中一个任务在另一个任务之后执行，以及“Y”模式，其中两个独立的任务必须在后续任务之前执行。.

这些依赖模式的见图 8-1 和图 8-2。图 8-1 描述了一个线性相关链。第一个节点表示数据的初始化，而第二个节点表示将数据累加为单个结果的归约操作。图 8-2 描述了一个“Y”模式，其中分别初始化两个不同的数据。数据初始化后，加法内核将两个向量相加。最后，图中的最后一个节点将结果累加为一个值。

图 8-1 线性相关



对于每个模式，我们将展示三种不同的实现。第一个实现将使用有序队列，第二个将使用基于事件的依赖项，最后一个实现将使用缓冲区和访问器来表示命令组之间的数据依赖关系。

图 8-2 “Y” 模式依赖

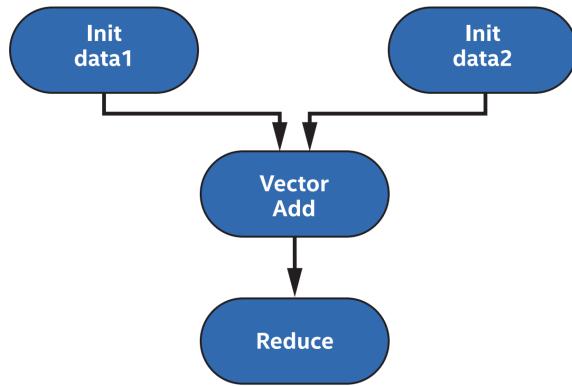


图 8-3 具有序队列的线性依赖链

```
1 constexpr int N = 42;
2
3 queue Q{property::queue::in_order()};
4
5 int *data = malloc_shared<int>(N, Q);
6
7 Q.parallel_for(N, [=](id<1> i) { data[i] = 1; });
8
9 Q.single_task( [=]() {
10     for (int i = 1; i < N; i++)
11         data[0] += data[i];
12 });
13
14 Q.wait();
15
16 assert(data[0] == N);
```

如图 8-3 所示，使用有序队列表示线性依赖。这个示例非常简单，因为有序队列的语义可以保证命令组之间的连续执行顺序。提交的第一个内核将数组的元素初始化为 1。下一个内核取这些元素，并将它们加起来成为第一个元素。由于队列是有序的，不需要做任何其他事情来表示第二个内核在第一个内核完成之前不应该执行。最后，等待队列完成所有任务，并检查是否获得了预期的结果。

图 8-4 与事件线性相关

```
1 constexpr int N = 42;
2
3 queue Q;
4
5 int *data = malloc_shared<int>(N, Q);
6
7 auto e = Q.parallel_for(N, [=](id<1> i) { data[i] = 1; });
```

```

8
9 Q.submit([&](handler &h) {
10    h.depends_on(e);
11    h.single_task(==)() {
12        for (int i = 1; i < N; i++)
13            data[0] += data[i];
14    });
15 });
16
17 Q.wait();
18 assert(data[0] == N);

```

图 8-4 展示了使用无序队列和基于事件依赖的示例。我们捕获第一次调用 parallel_for 所返回的事件。然后，第二个内核能够指定对该事件的依赖，并通过将其作为参数传递给 depends_on 来表示内核执行。我们将在图 8-6 中看到如何使用定义内核的快捷方法，来缩短第二个内核的表达式。

图 8-5 带有缓冲区和访问器的线性依赖

```

1 constexpr int N = 42;
2 queue Q;
3
4 buffer<int> data{range{N}};
5
6 Q.submit([&](handler &h) {
7     accessor a{data, h};
8     h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
9 });
10
11 Q.submit([&](handler &h) {
12     accessor a{data, h};
13     h.single_task(==)() {
14         for (int i = 1; i < N; i++)
15             a[0] += a[i];
16     });
17 });
18
19 host_accessor h_a{data};
20 assert(h_a[0] == N);

```

图 8-5 使用缓冲区和访问器，重写了线性依赖链示例。这里再次使用无序队列，但使用通过访问器指定的数据依赖项，对命令组的执行进行排序。第二个内核读取第一个内核产生的数据，因为基于相同的底层缓冲区对象声明的访问器，所以运行时可以预期相应的操作。这里与前面的示例不同，不需要等待队列完成所有任务。这里使用主机访问器定义了数据之间的依赖关系，并在主机上计算出第二个内核的正确输出，并使用断言来判断内核给出的答案是否正确。请注意，虽然主机访问器提供了主机上数据，但如果在创建缓冲区时指定了原始主机内存，就不能保证已经更新了原始主机内存。除非先销毁缓冲区，或者使用更高级的机制（如第 7 章中描述的互斥机制），否则无法安全地访问原始主机内存。

图 8-6 有序队列的“Y”模式

```
1 constexpr int N = 42;
2
3 queue Q{property::queue::in_order()};
4
5 int *data1 = malloc_shared<int>(N, Q);
6 int *data2 = malloc_shared<int>(N, Q);
7
8 Q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });
9
10 Q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });
11
12 Q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });
13
14 Q.single_task([=]()
15   for (int i = 1; i < N; i++)
16     data1[0] += data1[i];
17   data1[0] /= 3;
18));
19
20 Q.wait();
21 assert(data1[0] == N);
```

如图 8-6 所示，使用有序队列表示“Y”模式。本例中，声明了两个数组 data1 和 data2。然后定义两个内核，每个内核将初始化其中一个数组。这些内核并不相互依赖，但是由于队列是有序的，所以内核必须逐个地执行。而且，交换这两个内核的顺序是完全合法的。第二个内核执行之后，第三个内核将第二个数组的元素添加到第一个数组的元素中。最后一个内核函数将第一个数组的元素求和，从而得到与我们在线性相关链的例子中相同的结果。这个求和内核依赖于前面的内核，但是这个线性链也可以让有序队列捕获。最后，等待所有的内核都完成并验证我们是否成功地计算出了“魔数”。

图 8-7 事件方式的“Y”模式

```
1 constexpr int N = 42;
2 queue Q;
3
4 int *data1 = malloc_shared<int>(N, Q);
5 int *data2 = malloc_shared<int>(N, Q);
6
7 auto e1 = Q.parallel_for(N,
8   [=](id<1> i) { data1[i] = 1; });
9
10 auto e2 = Q.parallel_for(N,
11   [=](id<1> i) { data2[i] = 2; });
12
13 auto e3 = Q.parallel_for(range{N}, {e1, e2},
```

```

14     [=](id<1> i) { data1[i] += data2[i]; });
15
16 Q.single_task(e3, [=]()
17   for (int i = 1; i < N; i++)
18     data1[0] += data1[i];
19   data1[0] /= 3;
20 });
21
22 Q.wait();
23 assert(data1[0] == N);

```

图 8-7 展示了另一种“Y”模式示例，使用了无序队列。由于队列的顺序，依赖关系不再是隐式的，因此必须使用事件显式地指定命令组之间的依赖关系。如图 8-6 所示，首先定义两个没有初始依赖关系的独立内核。用两个事件来表示这些内核，e1 和 e2。们定义第三个内核时，必须指定它依赖于前两个内核。这样做是因为依赖于事件 e1 和 e2 在执行之前完成。本例中使用快捷形式来指定这些依赖项，而不是处理程序的 depends_on 方法。将事件作为参数传递给 parallel_for。想要一次传递多个事件，所以使用接受事件组的形式。幸运的是，现代 C++ 可以通过表达式 {e1, e2}，将其转换为适当的向量，简化了步操作。

图 8-8 访问器的“Y”模式

```

1 constexpr int N = 42;
2 queue Q;
3
4 buffer<int> data1{range{N}};
5 buffer<int> data2{range{N}};
6
7 Q.submit([&](handler &h) {
8   accessor a{data1, h};
9   h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
10 });
11
12 Q.submit([&](handler &h) {
13   accessor b{data2, h};
14   h.parallel_for(N, [=](id<1> i) { b[i] = 2; });
15 });
16
17 Q.submit([&](handler &h) {
18   accessor a{data1, h};
19   accessor b{data2, h, read_only};
20   h.parallel_for(N, [=](id<1> i) { a[i] += b[i]; });
21 });
22
23 Q.submit([&](handler &h) {
24   accessor a{data1, h};
25   h.single_task(==() {
26     for (int i = 1; i < N; i++)
27       a[0] += a[i];

```

```
28     a[0] /= 3;
29 }
30 }
31
32 host_accessor h_a{data1};
33 assert(h_a[0] == N);
```

如图 8-8 所示，最后一个例子中，再次用缓冲区和访问器替换 USM 指针和事件。这个例子将两个数组 data1 和 data2 表示为缓冲区对象。内核不再使用快捷方法，因为必须将访问器与命令组处理程序关联起来。同样，第三个内核必须了解对前两个内核的依赖关系，这里通过为缓冲区声明访问器来完成。因为之前已经为这些缓冲区声明了访问器，所以运行时能够正确地排列这些内核的执行顺序。此外，当声明访问器 b 时，还向运行时提供了信息。添加了访问标记 `read_only`，让运行时知道只读取该数据。正如线性依赖的缓冲区和访问器示例中所示，最终的内核通过更新第三个内核产生的值来对自身排序。通过声明主机访问器来获取计算的最终值，该访问器将等待最终内核完成执行，然后将数据移回主机，在那里可以读取数据，并断言内核是否计算出了正确的结果。

CG 的各个部分是什么时候执行的？

因为任务图是异步的，所以知道命令组究竟是什么时候执行是有意义的。目前为止，只要满足了内核的依赖关系，就可以执行内核，但是命令组的主机部分会发生什么呢？

当命令组提交到队列时，会立即在主机上执行（在 `submit` 返回之前）。命令组的主机部分只执行一次，在命令组中定义的任何内核或显式数据操作，都将排队在设备上执行。

数据传送

数据移动是 DPC++ 图的另一个重点，它对于理解应用程序性能至关重要。数据移动在程序中隐式地发生（使用缓冲区和访问器或使用 USM 共享分配），常常会忽略。接下来，我们将研究数据移动在 DPC++ 中影响图执行的不同方式。

显式

显式数据移动的优点是会显式地出现在图中，使开发者可以清楚地看到图中发生了什么。我们把显式数据操作分为用于 USM 的和用于缓冲区的。

如在第 6 章，当需要在设备内存和主机内存间复制数据时，USM 中的显式数据移动就会发生，可以通过 `memcpy` 完成。提交操作或命令组将返回一个事件，该事件可用于与其他命令组对操作进行排序。

通过调用命令组处理程序对象的 `copy` 或 `update_host`，将发生缓冲区的显式数据移动。复制方法可用于在主机内存和设备上的访问器对象之间交换数据，一个简单的例子是检查一个长时间运行的计算序列。使用 `copy` 的话，数据可以以单向方式从设备写入到主机内存中。如果是使用缓冲区完成的，大多数情况下（例如，不是以 `use_host_ptr` 创建的）将要求数据首先复制到主机，然后从缓冲区的内存到所需的主机内存。

`update_host` 是一种非常特殊的复制形式。如果基于主机内存创建了缓冲区，则此方法将把访问器表示的数据复制回原始主机内存。如果程序手动将主机数据与 `use_mutex` 属性创建的缓冲区同步，那么这个操作将非常有用。但是，这种用例不太可能出现在大多数的程序中。

隐式

隐式数据移动可能对 DPC++ 中的命令组和任务图产生隐藏的效果。通过隐式数据移动，数据可以通过 DPC++ 运行时或某种硬件和软件的组合在主机和设备之间复制。这两种情况下，复制不需要显式进行。让我们再次分别看看 USM 和缓冲区的例子。

对于 USM，隐式数据移动发生在主机和共享内存中。主机内存并没有真正移动数据，而是远程访问数据，共享内存可能在主机和设备之间迁移。由于这种迁移是自动进行的，所以对于 USM 隐式数据移动和命令组实际上没有什么需要考虑的。然而，共享内存有一些微妙之处值得了解。

预取操作以类似于 `memcpy` 的方式工作，以便让运行时在内核尝试使用共享分配之前开始迁移。然而，与 `memcpy` 不同，数据必须复制才能确保结果正确，预取通常视为运行时提高性能的提示，预取不会使内存中的指针值失效。如果预取在内核开始执行之前没有完成，程序仍会正确执行，因为预取并不是一个功能需求，所以很多代码可能会选择使图中的命令组不依赖于预取操作。

缓冲区也有一些细微差别。使用缓冲区时，命令组必须为指定如何使用数据的缓冲区构造访问器。这些数据依赖关系表示了不同命令组之间的顺序，并允许构建任务图。然而，带有缓冲区的命令组有时会进行另外的操作：指定数据移动。

访问器指定内核将读取或写入缓冲区。由此得出的结论是，设备上的数据必须可用，如果不可用，运行时必须在内核开始执行之前将其移到设备上。因此，DPC++ 运行时必须跟踪当前缓冲区位置，以便安排数据移动操作。访问器在图中创建了一个隐藏节点。如果数据移动是必需的，运行时必须先执行。这样，提交的内核才能正常执行。

再看看图 8-8。这个例子中，前两个内核将需要将缓冲区 `data1` 和 `data2` 复制到设备中，运行时隐式地创建图节点来执行数据移动。当提交第三个内核的命令组时，这些缓冲区可能仍然在设备上，因此运行时将不需要执行数据移动。第四个内核的数据也可能不需要数据移动，但是主机访问器的创建需要运行时在访问器可用之前将缓冲区 `data1` 移动回主机。

与主机同步

我们将讨论的最后一个主题是，如何同步图和主机执行。本章中已经涉及到这一点，但现在将研究程序实现这一点的方法。

主机同步的第一个方法是：等待队列。队列对象有两个方法，`wait` 和 `wait_and_throw`，阻塞执行，直到提交到队列的每个命令组都完成。非常简单的方法，可以处理许多常见的情况。然而，这种方法是粒度非常粗。如果需要更细粒度的同步，需要使用另一种方法。

主机同步的另一种方法是对事件进行同步。这比在队列上同步更灵活，因为只允许程序在特定的操作或命令组上同步。这可以通过在事件上调用 `wait` 方法或在事件类上调用 `wait` 来完成，`wait` 可以接受一个事件组。

图 8-5 和图 8-8 中使用了不同的方法：主机访问器。主机访问器执行两个功能。首先，使主机上的数据可用。其次，通过在当前访问的图和主机之间定义依赖关系来与主机同步。这可以确保复制回主机的数据是图计算的正确结果。但是，如果缓冲区是由主机内存构造，那么这个原始内存不能保证数值的一致性。

注意，主机访问器是阻塞的。在数据可用之前，主机上的执行可能不会在创建主机访问器之后继续。同样，当主机访问器存在并保持其数据可用时，不能在设备上使用缓冲区。一种常见的模式

是在 C++ 作用域中创建主机访问器，以便在不再需要主机访问器时释放数据。这是另一种主机同步的方法。

DPC++ 中的某些对象在销毁和调用其析构函数时具有特殊行为。当缓冲区和图像销毁或离开生命周期时，也有特殊的行为。当缓冲区销毁时，将等待所有使用该缓冲区的命令组完成执行。当任何内核或内存操作不再使用缓冲区，运行时必须将数据复制回主机。如果缓冲区使用主机指针初始化，或者主机指针传递给 `set_final_data`，则会发生复制。然后，运行时库将复制该缓冲区的数据，并在销毁对象之前更新主机内存。

与主机同步的最后一个选项涉及在第 7 章中描述的一个不常见的特性。回想一下，缓冲区对象的构造函数可选地接受属性列表。创建缓冲区时，传递的属性是 `use_mutex`。当以这种方式创建缓冲区时，增加了一个要求，即该缓冲区的内存可以与主机程序共享。对该内存的访问由互斥锁控制，当可以安全地访问与缓冲区共享内存时，主机能够获得锁。如果无法获得锁，用户可能需要将内存移动操作排队，以便与主机同步数据。这种用法非常小众，在大多数 DPC++ 应用程序中不太能看到。

总结

本章中，我们学习了图，以及如何在 DPC++ 中构建、调度和执行图。详细介绍了什么是命令组，它的作用是什么。讨论了命令组中可以包含的三种内容：依赖关系、操作和主机代码。回顾了如何使用事件，以及通过访问器描述的数据依赖来指定任务之间的依赖关系。了解到命令组中的单个操作可以是内核操作，也可以是显式内存操作，然后看了几个示例，展示了构建执行图的不同方法。接下来，回顾了数据移动是如何成为 DPC++ 图的重要部分，并了解了数据移动如何显式或隐式地出现在图中。最后，了解了与主机同步执行图的所有方法。

理解程序流程可使我们理解在运行失败时，打印的调试信息。第 13 章“调试运行时失败”一节中有一个表，根据我们在本书中所获得的知识，这个表会更有意义一些。然而，本书并不打算详细讨论这些编译器的高级信息。

希望了解完这章，可以让您觉得自己是一个执行图专家，可以构建复杂的图，从线性链到具有数百个节点、复杂数据和任务依赖的巨大执行图！下一章中，我们将开始深入研究在特定设备上改进应用程序性能的底层细节。

9 通信和同步



第 4 章中，讨论了表达并行性的方法，包括使用数据并行内核、ND-Range 内核或分层并行内核。讨论了数据并行内核如何将相同的操作应用到每一块数据上。还讨论了 ND-Range 内核和分层并行内核如何划分工作组中的工作项。

本章提供了关于 ND-Range 内核和分层并行内核的更多细节，并描述了如何使用工作项分组来提高算法的性能。描述工作项/组如何并行工作，以及如何执行提供保证，并且引入支持工作项/组的特性。在第 15、16、17 章中针对特定设备进行优化时，以及在第 14 章中介绍常见的并行模式，这些思想和概念都非常重要。

工作组和工作项

ND-Range 和分层并行内核将工作项组织到工作组中，并且工作组中的工作项可以并发执行。工作项可以保证并发时，工作组中的协作问题才能解决。

图 9-1 二维 ND-Range 中 (8,8) 个工作项分为四个 (4,4) 工作组

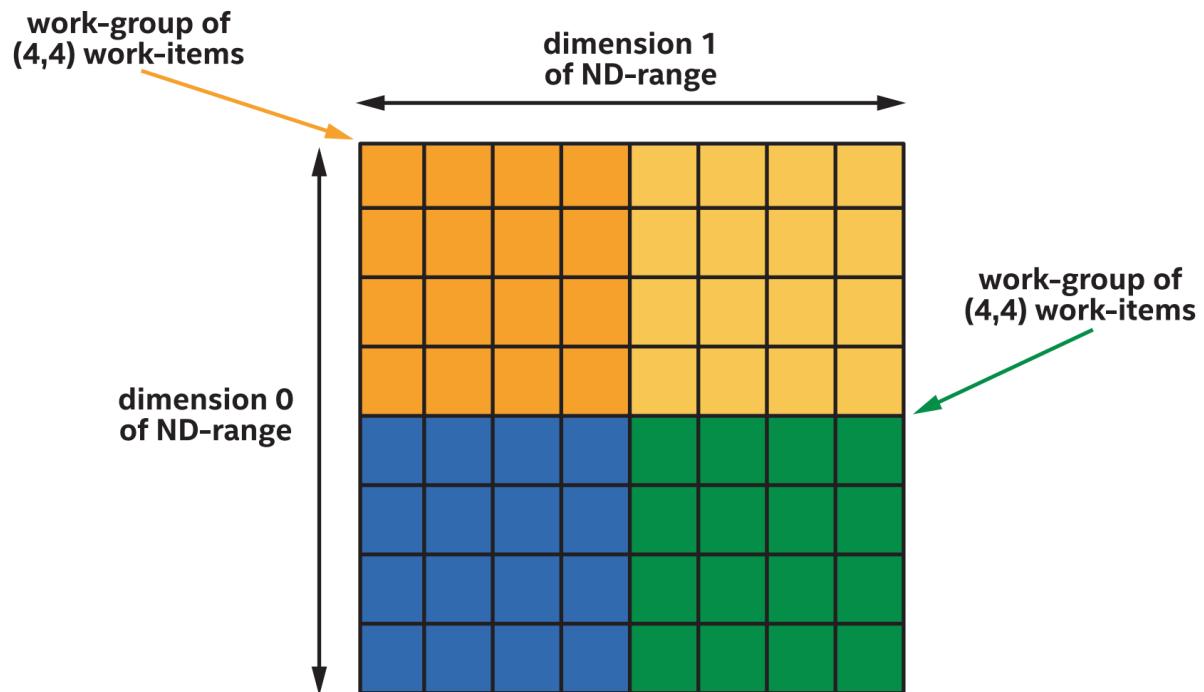


图 9-1 将 ND-Range 划分为不同的工作组，每个工作组用不同的颜色表示。工作组中的工作项可以同时执行，因此工作项可以与相同颜色的工作项通信。

因为不同工作组不能保证并发执行，所以同一种颜色的工作项不能与不同颜色的工作项通信，并且如果一个工作项试图与当前未执行的另一个工作项通信，可能会导致死锁。当我们希望内核完成执行，必须确保当工作项之间通信时，必须在同一个工作组中。

有效通讯的基础

本节介绍了提供组中工作项间通信的构建块。一些基本的构建块，可以构建自定义算法，而另一些高级别的，则描述了许多内核的常规操作。

通过栅栏进行同步

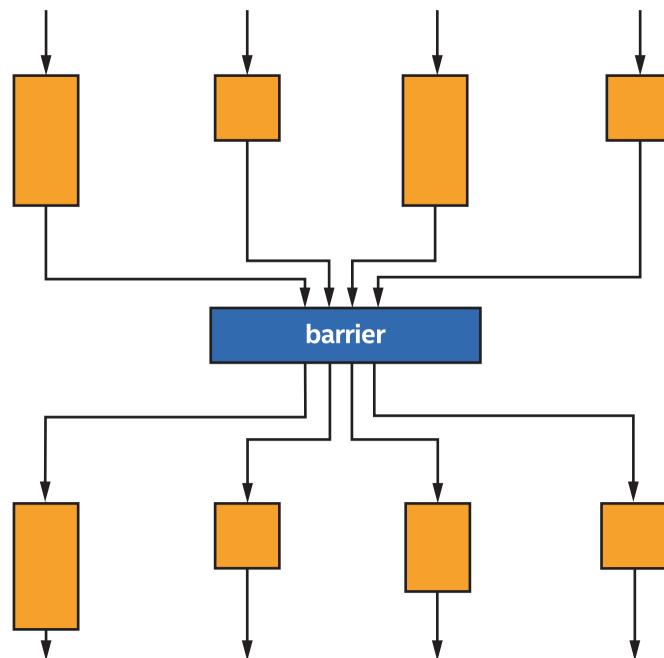
通讯最基本的部分是栅栏功能。栅栏功能有两个目的：

首先，栅栏功能可以同步组中工作项的执行。通过同步，工作项可以确保另一个工作项在使用该操作的结果之前完成操作。在另一个工作项使用操作的结果之前，让工作项完成操作。

其次，栅栏函数同步每个工作项，并检查内存状态。这种类型的同步操作称为强制内存一致或隔离内存（详见第 19 章）。内存一致与同步执行一样重要，因为在栅栏前执行的内存操作对栅栏之后执行的其他工作项可见。如果没有内存一致性，工作项中的操作就像森林中倒下的一棵树，其他工作项不一定听得到声音！

图 9-2 显示了在栅栏功能上同步组中的四个工作项。尽管每个工作项的执行时间可能不同，但没有任何工作项可以跨过栅栏执行，直到所有工作项都遇到了栅栏。执行栅栏功能之后，所有工作项就有了一致的内存。

图 9-2 组中的四个工作项在栅栏功能上的同步



为什么内存默认情况下不一致?

对于许多开发者来说，内存一致性的概念——以及不同的工作项可以看到不同的内存数据——可能感觉非常奇怪。如果默认情况下所有工作项的所有内存都是一致的，不是更容易吗？是的，但实现的代价非常昂贵。允许工作项看到不一致的内存数据，并且只要求程序执行期间定义的点上保证内存一致性，这样加速器硬件价格会更便宜，性能更好。

因为栅栏功能同步执行，所以要么组中的所有工作项执行栅栏操作，要么组中没有工作项执行栅栏操作。如果组分支中的某些工作项绕过栅栏功能，则组中的其他工作项可能永远在栅栏处等待！

集合功能

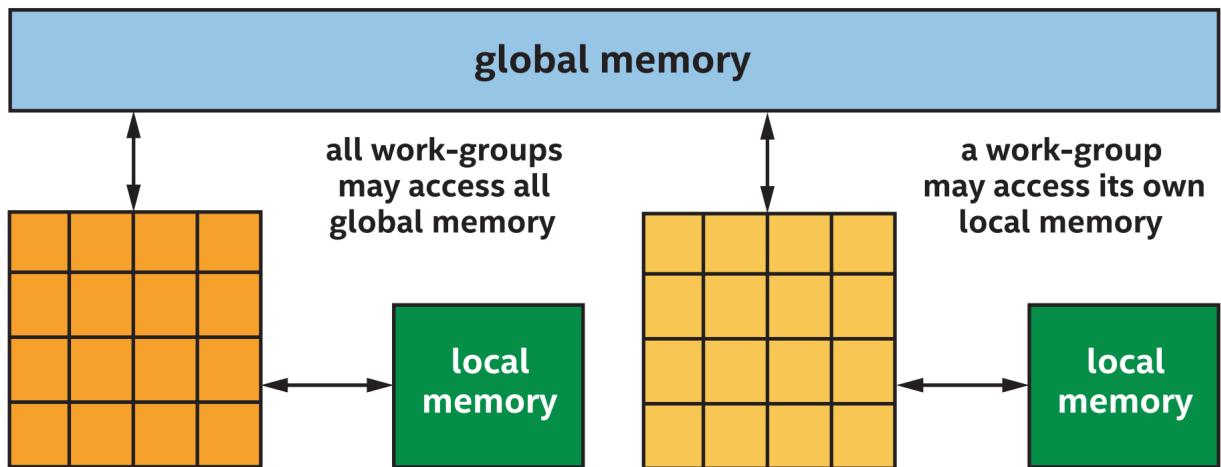
当一个功能需要由组中的所有工作项执行时，可以将其称为集合功能，因为操作由工作组执行，不由组中的单个工作项执行。栅栏函数并不是 SYCL 中唯一可用的集合函数。本章稍后将介绍其他集合功能。

工作组的本地内存

工作组栅栏功能可以协调工作组中工作项的通信，但是通信必须通过内存进行。通信可以通过 USM 或缓冲区进行，但这可能效率低下：需要为通讯开辟专用内存，并需要在工作组之间进行分配。

为了简化内核开发，并加速工作组中工作项的通信，SYCL 定义了专用于工作组中工作项之间通信的本地内存。

图 9-3 每个工作组可以访问所有全局内存，但只能访问自己的本地内存



如图 9-3 所示，两个工作组可以访问全局内存空间中的 USM 和缓冲区。每个工作组可以访问自己的本地内存中的变量，但不能访问另一个工作组的本地内存中的变量。

当一个工作组开始时，其本地内存的内容未初始化，并且本地内存存在工作组执行完毕后不会持续存在。由于这些属性，本地内存只能在工作组执行时作为临时存储使用。

对于某些设备，本地内存是一种抽象概念，使用与全局内存相同的内存子系统来实现。设备上使用本地内存，主要是以便通信机制的使用。一些编译器可能会使用内存空间信息进行编译器优化；否则，设备上使用本地内存进行通信，并不会比通过全局内存进行通信的性能更好。

对于其他设备，比如：多 GPU 设备，有专用的本地内存资源。这些设备上，通过本地内存通信比通过全局内存通信性能更好。

使用本地内存时，工作组内部的通信可以更方便、更快捷！

可以使用设备查询 `info::device::local_mem_type` 来确定加速器是否为本地内存提供了专用资源，或者本地内存是否实现为全局内存。有关查询设备属性的更多信息请参见第 12 章，有关本地内存如何为 CPU、GPU 和 FPGA 实现的更多信息详见第 15、16 和 17 章。

使用工作组栅栏和本地内存

现在已经确定了工作项之间通信的基本构建块，可以描述如何在内核中表示工作组栅栏和本地内存。记住，工作项之间的通信需要工作组的概念，因此这些概念只能在 ND-Range 内核和分层内核中表示。

本章将以第 4 章中介绍的矩阵乘法内核示例为基础，通过介绍执行矩阵乘法的工作组中工作项之间的通信。在许多设备上——不一定是所有设备！——通过本地内存通信提高矩阵乘法内核的性能。

关于矩阵乘法的注意事项

本书中，矩阵乘法内核用于演示内核的变化如何影响性能。尽管使用本章的技术可以在某些设备上提高矩阵乘法的性能，但鉴于矩阵乘法是非常常见的操作，以至于许多供应商已经实现了矩阵乘法的高度优化版本。厂商投入大量的时间和精力来实现和验证特定设备的功能，并且在某些情况下会使用在标准并行内核中难以或不可能使用的功能或技术。

使用供应商提供的库！

当供应商提供函数的库实现时，尽可能使用它，而不是将函数重新实现！对于矩阵乘法，可以将 oneMKL 作为 DPC++ 英特尔 oneAPI 解决方案工具包的一部分。

图 9-4 展示了第 4 章的矩阵乘法内核代码。

图 9-4 第 4 章中的矩阵乘法内核

```
1 h.parallel_for(range{M, N}, [=](id<2> id) {
2     int m = id[0];
3     int n = id[1];
4
5     T sum = 0;
6     for (int k = 0; k < K; k++)
7         sum += matrixA[m][k] * matrixB[k][n];
8 }
```

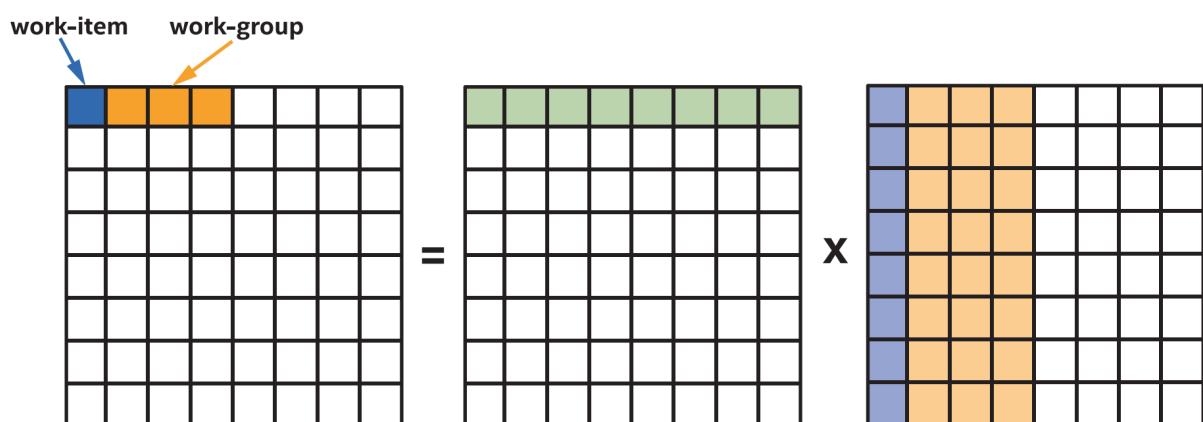
```
9     matrixC[m][n] = sum;  
10 } );
```

第 4 章中，我们注意到矩阵乘法算法具有高度的重用性，并且对工作项进行分组可以改善访问的局部性，从而提高缓存命中率。本章中，修改后的矩阵乘法内核将使用本地内存作为缓存，以保证访问的局部性，而不是依赖缓存来提高性能。

对于许多算法来说，可以将本地内存看作缓存。

图 9-5 是第 4 章修改后的代码，展示了由单行组成的工作组，这使得使用本地内存的算法更容易理解。注意，对于结果矩阵一行中的元素，每个结果元素都是使用来自一个输入矩阵的数据列计算，用蓝色和橙色显示。因为这个输入矩阵没有数据共享，所以不是理想的本地内存候选。但是，该行中的每个结果元素都访问另一个输入矩阵（绿色部分）中的完全相同的数据。由于此数据是重用的，是工作组本地内存中最佳的方式。

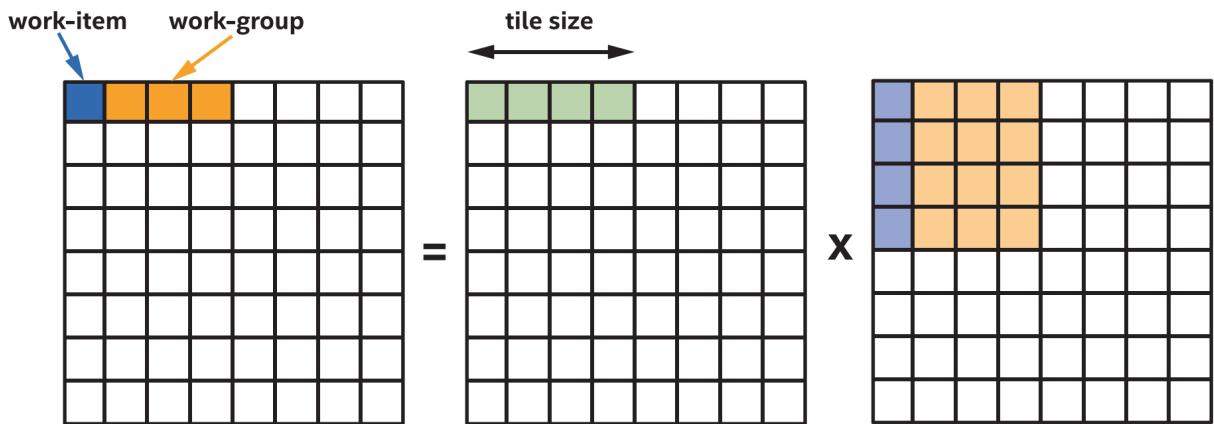
图 9-5 矩阵乘法中工作组和工作项的映射



因为要将非常大的矩阵相乘，而且工作组本地内存可能是有限的资源，所以修改后的内核将处理每个矩阵的子部分，将其称为矩阵块。对于每个块，修改后的内核将把矩阵块的数据加载到本地内存中，同步组中的工作项，然后从本地内存而不是全局内存加载数据。第一个块所访问的数据如图 9-6 所示。

内核中选择了与工作组大小相等的块大小。这不是必需的，但简化了本地内存的数据传输，所以块的大小通常可以选择工作组大小的倍数。

图 9-6 处理第一个块：绿色的输入数据（X 的左边）重用并从本地内存读取，蓝色和橙色的输入数据（X 的右边）从全局内存读取



ND-Range 内核中的工作组栅栏和本地内存

本节描述如何在 ND-Range 内核中表示工作组栅栏和本地内存。对于 ND-Range 内核：内核声明并操作本地地址空间分配的本地访问器，调用栅栏函数来同步工作组中的工作项。

本地内存访问器

要声明在 ND-Range 内核中使用的本地内存，请使用本地访问器。与其他访问器一样，本地访问器在命令组处理程序中构造，但与第 3 章和第 7 章讨论的访问器对象不同，本地访问器不是从缓冲区对象创建的。可以通过指定类型和描述该类型元素数量的范围来创建本地访问器。与其他访问器一样，本地访问器可以是 1 维、2 维或 3 维的。图 9-7 演示了如何声明本地访问器并在内核中使用。

本地内存 在每个工作组开始时未初始化，每个工作组完成后不会持续保存数据。这意味着本地访问器必须是 `read_write`，否则内核将无法分配本地内存的内容或查看分配的结果。但本地访问器可以选择为原子的，可以通过访问器对本地内存进行原子访存。原子访存将在第 19 章进行更详细的讨论。

图 9-7 声明和使用本地访问器

```

1 // This is a typical global accessor.
2 accessor dataAcc {dataBuf, h};
3
4 // This is a 1D local accessor consisting of 16 ints:
5 local_accessor<int> localIntAcc{16, h};
6
7 // This is a 2D local accessor consisting of 4 x 4 floats:
8 local_accessor<float> localFloatAcc{{4,4}, h};
9
10 h.parallel_for(nd_range<1>{{size}, {16}}, [=](nd_item<1> item) {
11     auto index = item.get_global_id();
12     auto local_index = item.get_local_id();
13
14     // Within a kernel, a local accessor may be read from
15     // and written to like any other accessor.

```

```
16 localIntAcc[local_index] = dataAcc[index] + 1;
17 dataAcc[index] = localIntAcc[local_index];
18});
```

同步功能

同步 ND-Range 内核工作组中的工作项，可以使用 `nd_item` 类中的 `barrier` 函数。因为 `barrier` 函数是 `nd_item` 类的成员，所以只对 ND-Range 的内核可用，而对简单的数据并行内核或分层内核不可用。

`barrier` 函数目前接受一个参数来描述要同步的内存空间，但是 `barrier` 函数的参数将来可能会随着 SYCL 和 DPC++ 中内存模型的发展而改变。但是，`barrier` 函数的参数提供了关于同步内存空间或内存同步域的控制。

当没有参数传递给 `barrier` 函数时，`barrier` 函数将使用功能上正确和默认值。本章中的代码示例使用了这种方式，以获得最大的可移植性和可读性。对于高度优化的内核，建议精确地描述哪些内存空间或哪些工作项必须同步，这可能会提高性能。

完整的 ND-Range 内核示例

现在知道了如何声明本地内存访问器并使用 `barrier` 函数同步对它的访问，可以实现矩阵乘法的 ND-Range 内核版本，协调工作组中工作项之间的通信，以减少对全局内存的通信。完整示例如图 9-8 所示。

图 9-8 用 ND-range parallel_for 和工作组本地内存表示块矩阵乘法内核

```
1 // Traditional accessors, representing matrices in global memory:
2 accessor matrixA{bufA, h};
3 accessor matrixB{bufB, h};
4 accessor matrixC{bufC, h};
5
6 // Local accessor, for one matrix tile:
7 constexpr int tile_size = 16;
8 local_accessor<int> tileA{tile_size, h};
9
10 h.parallel_for(
11 nd_range<2>{{M, N}, {1, tile_size}}, [=](nd_item<2> item) {
12     // Indices in the global index space:
13     int m = item.get_global_id() [0];
14     int n = item.get_global_id() [1];
15
16     // Index in the local index space:
17     int i = item.get_local_id() [1];
18
19     T sum = 0;
20     for (int kk = 0; kk < K; kk += tile_size) {
21         // Load the matrix tile from matrix A, and synchronize
22         // to ensure all work-items have a consistent view
23         // of the matrix tile in local memory.
```

```

24    tileA [ i ] = matrixA [ m ] [ kk + i ];
25    item . barrier ( );
26
27    // Perform computation using the local memory tile , and
28    // matrix B in global memory .
29    for ( int k = 0; k < tile_size ; k ++ )
30        sum += tileA [ k ] * matrixB [ kk + k ] [ n ];
31
32    // After computation , synchronize again , to ensure all
33    // reads from the local memory tile are complete .
34    item . barrier ( );
35 }
36
37 // Write the final result to global memory .
38 matrixC [ m ] [ n ] = sum ;
39 });

```

这个内核中的主循环可以看作两个不同的阶段：第一个阶段，组中工作项将共享数据从 A 矩阵加载到本地内存中；第二种情况下，工作项使用共享的数据执行自己的计算。为了确保所有工作项在进入第二个阶段之前已经完成了第一阶段，需要使用 barrier 来同步所有工作项，并提供内存栅栏来分隔。这种模式很常见，内核中使用工作组本地内存总是需要使用工作组栅栏。

注意，还必须调用 barrier 来同步当前块的计算阶段和下一个矩阵块的加载阶段之间的操作。如果没有此同步操作，则在用另一个工作项完成计算之前，工作组中的工作项结果可能会覆盖当前结果矩阵块的结果。当工作项在本地内存中读写由另一个工作项读写的数据时，就需要同步。图 9-8 中，同步是在循环结束时完成的，在每个循环开始时也需要同步。

分层内核中的工作组栅栏和本地内存

本节描述如何在分层内核中表示工作组栅栏和本地内存。与 ND-Range 内核不同，分层内核中的本地内存和栅栏是隐式的，不需要特殊的语法或函数调用。一些开发者会发现分层内核表示更直观、更容易使用，而其他开发者会喜欢 ND-Range 内核提供的直接控制。大多数情况下，相同的算法可能使用两种表示来描述，因此可以选择最容易开发和维护的方式。

本地内存和栅栏的作用域

回顾第 4 章，分层内核通过使用 parallel_for_work_group 和 parallel_for_work_item 来表示两级的并行执行。并行执行的这两个级别（或作用域）用于表示变量是否位于工作组本地内存中，并在工作组中的所有工作项之间共享，或者变量是否位于每个工作项的私有内存中（不在工作项之间共享）。这两个作用域还用于同步工作组中的工作项，并强制内存一致性。

图 9-9 展示了一个层次结构内核，在本地内存中声明一个在工作组作用域的变量，然后在工作项作用域中使用该变量。在工作组作用域对本地内存的写入和工作项作用域对本地内存的读取之间存在隐式的栅栏。

图 9-9 具有局部内存变量的分层内核

```

1 range group_size { 16 };

```

```

2 range num_groups = size / group_size;
3
4 h.parallel_for_work_group(num_groups, group_size, [=](group<1> group) {
5     // This variable is declared at work-group scope, so
6     // it is allocated in local memory and accessible to
7     // all work-items.
8     int localIntArr[16];
9
10    // There is an implicit barrier between code and variables
11    // declared at work-group scope and the code and variables
12    // at work-item scope.
13
14    group.parallel_for_work_item([&](h_item<1> item) {
15        auto index = item.get_global_id();
16        auto local_index = item.get_local_id();
17
18        // The code at work-item scope can read and write the
19        // variables declared at work-group scope.
20        localIntArr[local_index] = index + 1;
21        data_acc[index] = localIntArr[local_index];
22    });
23 });

```

分层内核的主要优点是，看起来非常类似于标准 C++ 代码，标准 C++ 代码中，一些变量可以在作用域中赋值，并在嵌套作用域中使用。当然，这也可以说是一种缺点，因为哪些变量在本地内存中，以及什么时候由分层内核编译器插入栅栏，这些不是很明显。对于实现栅栏操作非常昂贵的设备来说尤其如此！

完整的分层内核示例

既然知道了如何在分层内核中表示本地内存和栅栏，就可以编写一个分层内核，可以实现与图 9-7 中的 ND-Range 内核相同的算法，如图 9-10 所示。

尽管分层内核与 ND-Range 内核非常相似，但有一个关键的区别：ND-Range 内核中，矩阵乘法的结果在写入到内存中的输出矩阵之前累积到每个工作项变量和中，而分层内核则累积到内存中。也可以在分层内核中积累为每个工作项变量，但这需要特殊的 private_memory 在工作组作用域声明每个工作项数据，而选择使用分层内核语法的原因之一是避免使用特殊语法！

分层内核不需要特殊的语法来声明工作组本地内存中的变量，但是需要特殊的语法来声明工作项私有内存中的一些变量！

为了避免每个工作项的特殊数据语法，分层内核中工作项循环的常见模式是将中间结果写入工作组本地内存或全局内存。

图 9-10 块矩阵乘法内核作为分层内核实现

```

1 const int tileSize = 16;

```

```

2 range group_size{1, tileSize};
3 range num_groups{M, N / tileSize};
4
5 h.parallel_for_work_group(num_groups, group_size, [=](group<2> group) {
6     // Because this array is declared at work-group scope
7     // it is in local memory
8     T tileA[16];
9
10    for (int kk = 0; kk < K; kk += tileSize) {
11        // A barrier may be inserted between scopes here
12        // automatically, unless the compiler can prove it is
13        // not required
14
15        // Load the matrix tile from matrix A
16        group.parallel_for_work_item([&](h_item<2> item) {
17            int m = item.get_global_id()[0];
18            int i = item.get_local_id()[1];
19            tileA[i] = matrixA[m][kk + i];
20        });
21
22        // A barrier gets inserted here automatically, so all
23        // work items have a consistent view of memory
24
25        group.parallel_for_work_item([&](h_item<2> item) {
26            int m = item.get_global_id()[0];
27            int n = item.get_global_id()[1];
28            for (int k = 0; k < tileSize; k++)
29                matrixC[m][n] += tileA[k] * matrixB[kk + k][n];
30        });
31
32        // A barrier gets inserted here automatically, too
33    }
34});
```

图 9-10 中内核有一个有趣的属性与循环变量 `kk` 有关: 由于循环处于工作组作用域, 循环迭代变量 `kk` 可以从工作组本地内存中分配, 就像 `tileA` 数组一样。由于 `kk` 的值对于工作组中的所有工作项都是相同的, 所以编译器可能会选择在每个工作项内存中分配 `kk`, 特别是对于本地内存稀缺的设备。

子工作组

目前为止, 工作项已经通过工作组本地内存交换数据, 并通过隐式或显式 `barrier` 函数进行同步(这取决于内核是如何编写的), 与工作组中的其他工作项进行了通信。

第 4 章中, 讨论了另一种工作项分组。子工作组是工作组中定义子工作集, 在相同的硬件资源或调度上一起执行。因为实现决定如何将工作项分组到子工作组中, 子工作组中的工作项可能比任意工作组中的工作项更有效地通信或同步。

本节描述子工作组中工作项之间通信的构建块。注意, 子工作组目前仅为 ND-Range 内核实

现，并且子工作组不能通过分层内核表示。

同步操作与子工作组的栅栏

就像 ND-Range 内核中工作组中的工作项，可以使用工作组 barrier 功能进行同步，子工作组中的工作项可以使用子工作组栅栏功能进行同步。工作组同步可以通过调用工作项 group_barrier 或 nd_item 类的 barrier 功能完成。子工作组中的工作项的同步，是通过调用 group_barrier 函数或 sub_group 类上的 barrier 函数（可使用 nd_item 类查询），如图 9-11。

图 9-11 查询和使用 sub_group 类

```

1 h.parallel_for(nd_range{{size}, {16}}, [=](nd_item<1> item) {
2     auto sg = item.get_sub_group();
3     ...
4     sg.barrier();
5     ...
6 });

```

像工作组栅栏一样，子工作组的栅栏可以接受参数，来更精确地控制栅栏操作。无论子工作组栅栏功能是同步全局内存还是本地内存，只同步子工作组中的工作项，都可能比同步工作组中的所有工作项要划算。

子工作组内的数据交换

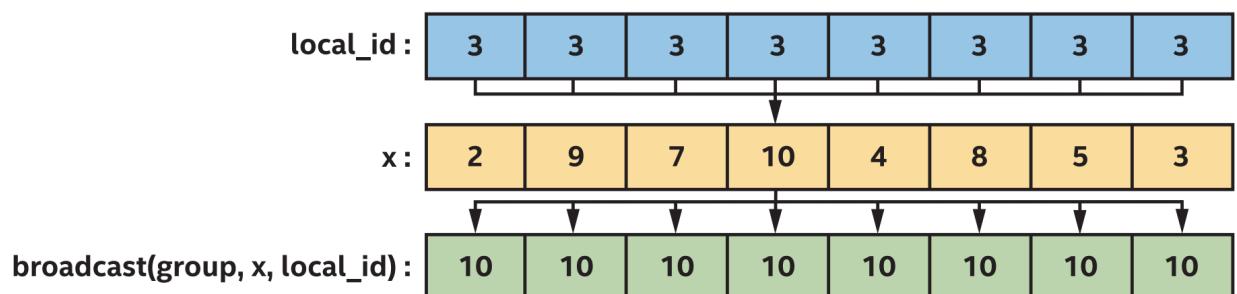
与工作组不同，子工作组没有用于交换数据的专用内存空间。子工作组中的工作项可以通过工作组的本地内存、全局内存或通过使用子工作组集合函数来交换数据。

如前所述，集合功能是描述由一组工作项执行的操作的功能，而不是单个工作项，并且因为栅栏同步功能是由一组工作项执行的操作，所以是集合功能。

其他集合功能表示公共通信模式。我们将在本章后面详细描述许多集合函数，现在简要描述广播集合函数，将使用子工作组来实现矩阵乘法。

广播集合函数从组中的一个工作项中获取值，并将其传递给组中的所有其他工作项，配置示例如图 9-12 所示。注意，广播函数的语义要标识组中哪个值要通信的 local_id 对于组中的所有工作项必须相同，以确保广播函数的结果对于组中的所有工作项是相同的。

图 9-12 广播功能



如果查看本地内存矩阵乘法内核的最内层循环，如图 9-13 所示，可以看到对矩阵块的访问是一个广播操作，因为组中的每个工作项从矩阵块中读取相同的值。

图 9-13 包含广播操作的矩阵乘法内核

```
1 h.parallel_for<class MatrixMultiplication>(
2     nd_range<2>{ {M, N}, {1, tileSize} }, [=](nd_item<2> item) {
3         ...
4
5         // Perform computation using the local memory tile, and
6         // matrix B in global memory.
7         for( size_t k = 0; k < tileSize; k++ ) {
8             // Because the value of k is the same for all work-items
9             // in the group, these reads from tileA are broadcast
10            // operations.
11            sum += tileA[k] * matrixB[kk + k][n];
12        }
13        ...
14    });
}
```

使用子组广播功能来实现不需要工作组本地内存或栅栏的矩阵乘法内核。许多设备上，子工作组广播比使用工作组本地内存和栅栏的广播更快。

完整的子组 ND-Range 内核示例

图 9-14 使用子工作组实现矩阵乘法的完整示例。注意，此内核不需要工作组本地内存或显式同步，而是使用子工作组广播集合函数在工作项之间通信矩阵块的内容。

图 9-14 用 NDrange parallel_for 和子组集合函数表示块矩阵乘法内核

```
1 // Note: This example assumes that the sub-group size is
2 // greater than or equal to the tile size!
3 static const int tileSize = 4;
4
5 h.parallel_for(
6     nd_range<2>{{M, N}, {1, tileSize}}, [=](nd_item<2> item) {
7         auto sg = item.get_sub_group();
8
9         // Indices in the global index space:
10        int m = item.get_global_id()[0];
11        int n = item.get_global_id()[1];
12
13        // Index in the local index space:
14        int i = item.get_local_id()[1];
15
16        T sum = 0;
17        for (int_fast64_t kk = 0; kk < K; kk += tileSize) {
18            // Load the matrix tile from matrix A.
19            T tileA = matrixA[m][kk + i];
```

```

20
21 // Perform computation by broadcasting from the matrix
22 // tile and loading from matrix B in global memory. The loop
23 // variable k describes which work-item in the sub-group to
24 // broadcast data from.
25 for (int k = 0; k < tileSize; k++)
26     sum += intel::broadcast(sg, tileA, k) * matrixB[kk + k][n];
27 }
28
29 // Write the final result to global memory.
30 matrixC[m][n] = sum;
31 );
32 );

```

通用功能

本章的“子工作组”部分，我们描述了集合功能以及集合功能如何表达公共通信模式。我们特别讨论了广播集合功能，它用于将值从组中的一个工作项传递到组中的其他工作项。本节介绍其他集合函数。

虽然本节中描述的集合函数，可以在程序中使用原子、工作组本地内存和栅栏等特性直接实现，但许多设备包括专用硬件对集合函数进行了加速。即使设备不包括专门的硬件，供应商提供的集合函数的实现可能针对它们的设备进行调优，因此调用内置集合函数通常会比编写的通用实现执行得更好。

为公共通信模式使用集合函数来简化代码并提高性能！

工作组和子工作组都支持许多集合功能。其他集合功能仅支持子工作组

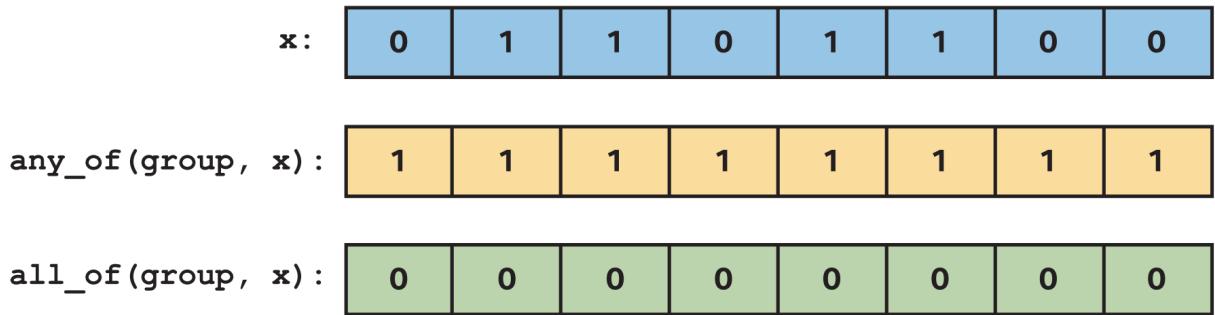
广播

广播功能允许组中的一个工作项与组中其他工作项共享变量的值。广播功能的工作原理如图 9-12 所示。工作组和子工作组都支持广播功能。

投票

`any_of` 和 `all_of` 功能（今后将共同称为“投票”功能）使组内工作项比较的结果为一个布尔值：至少一个工作项的条件为真时，`any_of` 返回 `true`；只有在所有工作项的条件为真时，`all_of` 返回 `true`。对于一个输入示例，这两个函数的比较如图 9-15 所示。

图 9-15 比较 `any_of` 函数和 `all_of` 函数



工作组和子工作组都支持 any_of 和 all_of 投票函数。

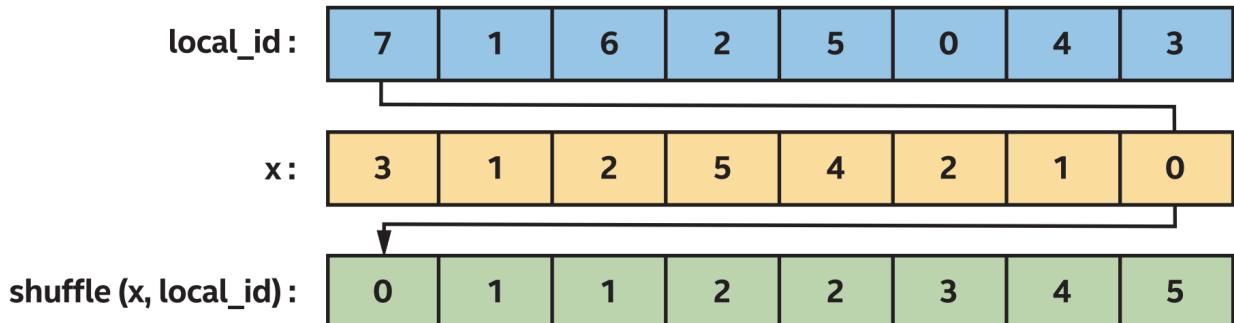
打乱

子工作组最有用的特性之一是能够在单个工作项之间直接通信，而不需要显式的内存操作。许多情况下，例如：子工作组矩阵乘法内核，这些打乱操作使我们能够从内核中删除工作组本地内存的使用和/或避免对全局内存不必要的重复访问。这些打乱函数有几种类型。

最通用的打乱函数，称为 shuffle，如图 9-16 所示，它允许子工作组中的任何工作项之间进行通信。然而，这种通用性可能会以性能为代价，我们强烈鼓励尽可能使用更特化的 shuffle 函数。

图 9-16 中，使用 shuffle 对预先计算的置换索引对子组的 x 值进行排序。展示了子工作组中一个工作项的箭头，其中 shuffle 的结果是 local_id 等于 7 的工作项的 x 值。

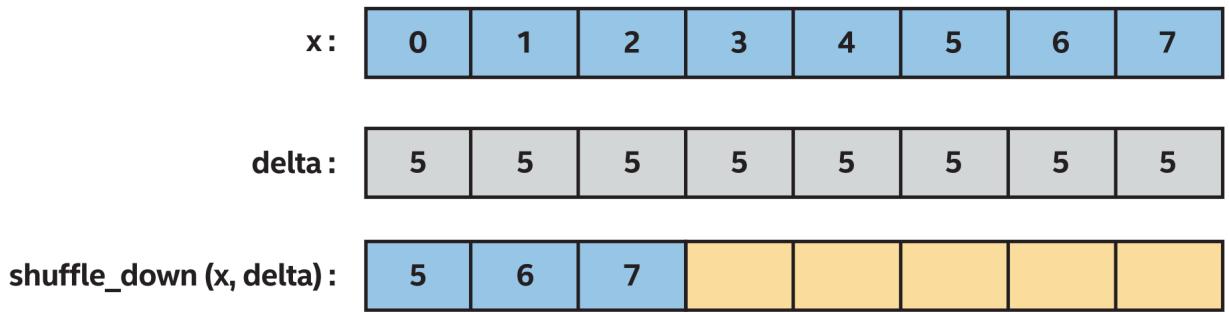
图 9-16 使用 shuffle 对预先计算的排列索引对 x 值排序



注意，可以将子工作组广播函数视为 shuffle 的特化版，其中 shuffle 索引对于子工作组中的所有工作项是相同的，使用广播而不是 shuffle 可为编译器提供信息，并可能提高某些实现的性能。

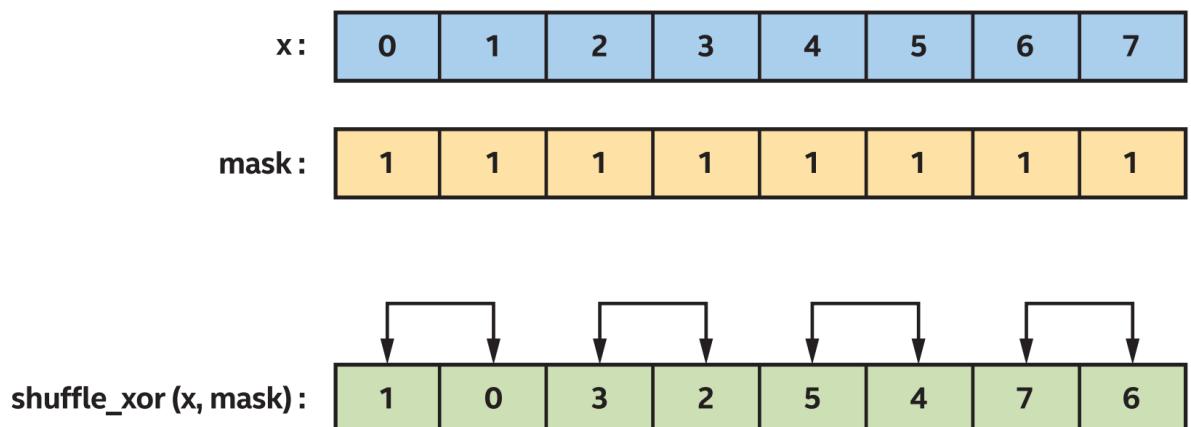
shuffle_up 和 shuffle_down 函数有效地将子工作组的内容向给定方向移动一定数量元素的长度，如图 9-17 所示。注意，返回到子工作组中最后五个工作项的值是未定义的，并且在图 9-17 中显示为空白。对于并行化带有循环依赖的循环，或者在实现扫描等常见算法时，移位非常有用。

图 9-17 使用 shuffle_down 将子组的 x 值移动 5 个元素的长度



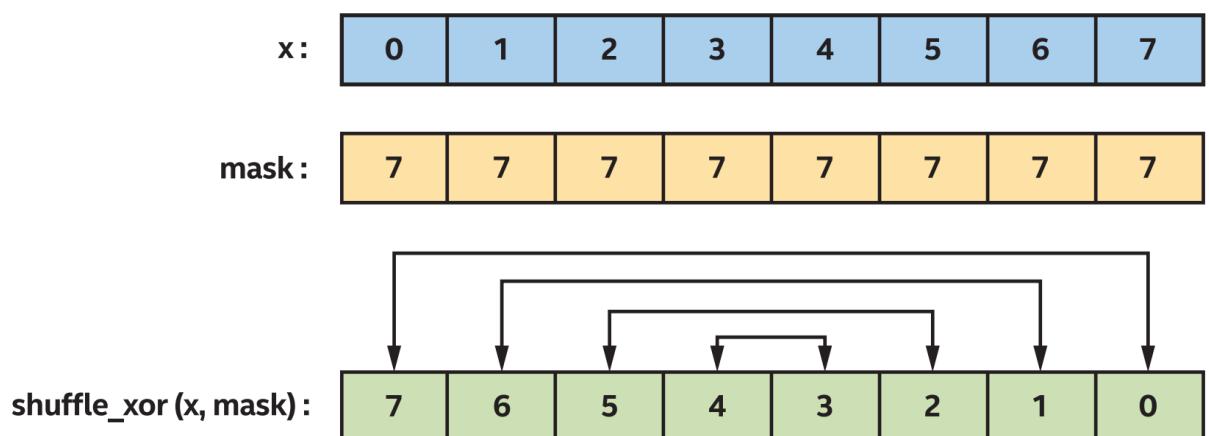
shuffle_xor 函数交换两个工作项的值，这些值由应用于工作项的子工作组本地 id 和常量的 XOR 操作结果的指定。如图 9-18 和 9-19 所示，几种常见的通信模式可以用异或表示，例如：交换相邻值对。

图 9-18 使用 shuffle_xor 交换相邻的 x 对



或者反转子工作组的结果。

图 9-19 使用 shuffle_xor 反转 x 的值



使用优化的广播，投票和集合函数

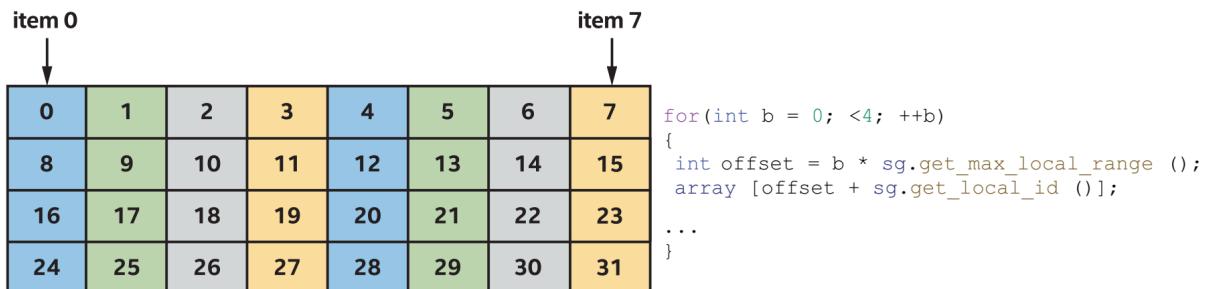
应用于子工作组的广播、投票和其他集合函数的行为与应用于工作组的行为是相同的，但是需要注意的是，它们会在某些编译器中启用主动优化，例如：编译器可能能够减少广播到子工作组中所有工作项的变量的寄存器使用，或者能够基于 any_of 和 all_of 函数的使用推断出控制流的方向。

加载和存储

子工作组的加载和存储函数有两个目的：第一，通知编译器子工作组中的所有工作项都从内存中的相同（统一）位置加载连续数据；第二，能够请求优化的加载/存储大量连续数据。

对于 ND-Range parallel_for，编译器可能不清楚由不同工作项计算的地址如何关联。例如，如图 9-20 所示，从每个工作项的角度来看，从索引 [0,32) 访问一个连续的内存块似乎是跨越的访问模式。

图 9-20 一个子工作组访问四个连续块的内存



有些体系结构包括专用硬件，用于检测子工作组中的工作项，何时访问连续数据并组合内存请求，而其他体系结构要求提前知道这一点并将其编码到加载/存储指令中。子工作组的加载和存储在任何平台上，都不是为了正确性存在的，但在某些平台上可能会提高性能，应视为一种优化提示。

总结

本章讨论了一个组中的工作项如何交流和合作以提高某些类型内核的性能。

首先讨论了 ND-Range 内核和分层内核如何支持将工作项分组到工作组中。讨论了如何将工作项分组到工作组中，从而更改并行执行模型，以确保工作组中的工作项并发执行，并支持通信和同步。

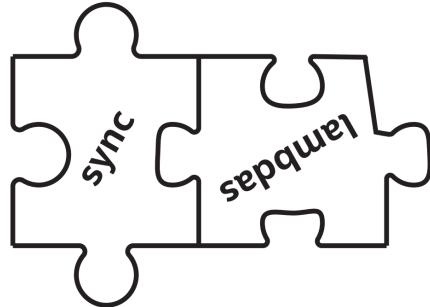
接下来，讨论了工作组中的工作项如何使用 barrier 进行同步，以及如何在 ND-Range 内核中显式地表示 barrier，或者在分层内核的工作组和工作项范围内隐式地表示 barrier。还讨论了如何通过工作组本地内存执行工作组中工作项之间的通信，以简化内核并提高性能，以及如何使用用于 ND-Range 内核的本地访问器来表示工作组本地内存，以及如何使用分层内核工作组范围内的本地内存。

讨论了如何将 ND-Range 内核中的工作组进一步划分为子工作组，其中子工作组可能支持其他的通信模式或调度。

对于工作组和子工作组，我们讨论了如何通过使用集合功能来表达和加速公共通信模式。

本章中的概念是理解第 14 章中描述的常见并行模式，以及理解第 15、16 和 17 章中如何针对特定设备进行优化的基础。

10 定义内核函数



目前为止，代码示例都是使用 C++ Lambda 表达式表示内核。Lambda 表达式是表示内核的一种方法，但不是 SYCL 中表示内核的唯一方法。本章中，我们将探索各种详细定义内核的方法，从而选择最适合 C++ 编码需求的内核形式。

本章解释和比较了三种表示内核的方法：

- Lambda 表达式
- 命名函数对象 (functor)
- 与其他语言或 API 创建的内核

本章讨论了如何显式地操作程序对象中的内核，控制何时以及如何编译内核。

为什么用三种方法来表示内核函数？

深入讨论细节之前，首先了解一下为什么有三种定义内核的方式，以及每种方法的优缺点。图 10-1 给出了一个表。

内核是用来表示计算单元的，内核的许多实例通常会在加速器上执行。SYCL 支持多种方式来表示内核，以便无缝地集成到各种代码库中，同时在各种加速器类型上高效地执行。

图 10-1 三种表示内核的方法

内核表示	描述
Lambda 表达式	<p>优点:</p> <ul style="list-style-type: none"> • Lambda 表达式是一种简洁的方式，可以在使用的地方表示内核。 • Lambda 表达式是现代 C++ 代码库中表示类内核操作的一种方式。 • Lambda 捕获规则自动将数据传递给内核。 <p>缺点:</p> <ul style="list-style-type: none"> • 以 Lambda 表达式表示的内核不能模板化，不能重用，也不能作为库提供。 • 一些代码库可能不支持 Lambda 语法。
命名函数对象 (Functor)	<p>优点:</p> <ul style="list-style-type: none"> • 函数可以模板化、重用，并作为库的一部分提供。 • 函数提供对传递给内核数据的更多控制。 <p>缺点:</p> <ul style="list-style-type: none"> • 用函数对象表示内核比用 Lambda 表达式表示的内核使用更多的代码。 • 内核参数必须显式地传递给函数对象，不能自动捕获。
与其他语言或 API	<p>优点:</p> <ul style="list-style-type: none"> • 允许重用以前编写的内核或库。 • 允许大型应用程序代码库增量地添加对 SYCL 的支持。 • 来自其他 API 的内核语言可能支持尚未添加或难以用 SYCL 表达的特性。 <p>缺点:</p> <ul style="list-style-type: none"> • 互操作性是一个可选特性，不是所有 SYCL 实现或设备都支持该特性。 • 用其他 API 编写的内核不是由 SYCL 设备编译器编译的，这可能会限制编译时对语法、内核参数的类型检查和优化。 • 用其他 API 编写的内核可能不支持最新的 C++ 特性。

使用 Lambda 表示内核函数

C++ Lambda 表达式，也称为匿名函数、未命名函数、闭包。本节介绍了如何将内核表示为 C++ Lambda。本节扩展了第 1 章中关于 C++ Lambda 函数的介绍。

C++ Lambda 表达式非常强大，表示内核时，只需要（并支持）完整 C++ Lambda 语法的子集即可。

图 10-2 使用 Lambda 表达式定义的内核

```
1 h.parallel_for(size,
```

```

2 // This is the start of a kernel lambda expression:
3 [=](id<1> i) {
4     data_acc[i] = data_acc[i] + 1;
5 }
6 // This is the end of the kernel lambda expression.
7 );

```

内核 Lambda 的构成

图 10-2 展示了用 Lambda 编写的内核——书中目前为止的代码示例都使用这种语法。

图 10-3 中的图显示了可以与内核一起使用的 Lambda 的更多组件，其中许多都不是必须的。大多数情况下，默认值就足够了，所以内核 Lambda 看起来更像图 10-2 中的表达式，而不是图 10-3 中那样。

图 10-3 内核 Lambda 的更多要素，包括可选组件

```

accessor data_acc {data_buf, h};
h.parallel_for(size,
    1 2 4 5 6 3 7
    [=] (id<1> i) noexcept [[cl::reqd_work_group_size(8,1,1)]] -> void {
        data_acc[i] = data_acc[i] + 1;
    });

```

1. Lambda 的第一部分描述捕获。从周围捕获变量使其可以在 Lambda 表达式中使用，而不用显式地将其作为参数传递给表达式。

C++ Lambda 表达式支持通过复制或创建对变量的引用的方式来捕获变量，但对于内核 Lambda，变量只能通过复制来捕获。一般的做法是简单地使用默认捕获模式 [=]，隐式地按值捕获所有变量，不过也可以显式地命名每个捕获的变量。在内核中使用的任何未捕获的变量都会将导致编译时错误。

2. Lambda 的第二部分是传递给表达式的参数，就像传递参数给命名函数一样。

对于内核 Lambda，参数取决于调用内核的方式，并且通常标识为并行执行空间中工作项的索引。关于各种并行执行空间，以及如何识别执行空间中工作项的索引的细节信息，请参阅第 4 章。

3. Lambda 的最后一部分定义了函数体。对于内核 Lambda，函数体描述了在并行执行空间中的每个索引处执行的操作。

内核也支持 Lambda 的其他部分，要么是可选的，要么不常用：

4. 可能会支持一些说明符（比如 mutable），但不建议使用，并且在未来的 SYCL 版本（在 SYCL 2020 中已经没有了）或 DPC++ 中可能会删除这种支持。
5. 如果提供了异常，就必须是 noexcept，因为内核不支持异常。
6. 支持 Lambda 属性，可以用来控制如何编译内核。例如，reqd_work_group_size 属性可用与定义工作组大小。
7. 可以指定返回类型，但必须返回 void，因为内核不支持非 void 返回类型。

Lambda: 隐式捕获还是显式捕获?

一些 C++ 风格指南建议不要使用隐式(或默认)捕获 Lambda, 因为可能存在悬空指针问题, 特别是在表达式跨越作用域边界时。当 Lambda 用于表示内核时, 也会出现同样的问题, 因为内核在设备上与主机代码异步执行。

因为隐式捕获简单好用, 是 SYCL 内核的常见做法, 也是本书中经常使用的一种方法, 但最终需要权衡隐式捕获的简洁和显式捕获的清晰。

命名表示内核的 Lambda

当内核写成 Lambda 时, 某些情况下必须提供另外一个标识: 因为表达式是匿名的, 有时 SYCL 需要显式的内核名称模板参数来标识写成表达式的内核。

图 10-4 命名内核的 Lambda

```
1 // In this example, "class Add" names the kernel lambda:  
2  
3 h.parallel_for<class Add>(size, [=](id<1> i) {  
4     data_acc[i] = data_acc[i] + 1;  
5});
```

命名内核 Lambda 是由主机代码编译器确定, 再由单独的设备代码编译器编译内核时, 确定调用哪个内核的一种方法。命名内核 Lambda 还可以对已编译的内核进行运行时选择, 或通过名称构建内核, 如图 10-9 所示。

为了在不需要内核名的情况下支持更简洁的代码,DPC++ 编译器支持通过-fsyclnamed -lambda 编译器选项来省略内核名。使用该选项时, 不需要显式的内核名模板参数, 如图 10-5 所示。

图 10-5 使用未命名的内核 Lambda

```
1 // In many cases the explicit kernel name template parameter  
2 // is not required.  
3 h.parallel_for(size, [=](id<1> i) {  
4     data_acc[i] = data_acc[i] + 1;  
5});
```

大多数情况下不需要 Lambda 的内核名称模板参数, 所以可以从未命名的 Lambda 开始, 只有在需要内核名称的情况下添加即可。

不需要内核名称时, 首选未命名的内核 Lambda。

使用命名的函数对象表示内核函数

命名函数对象, 也称为函子, 允许在定义接口的同时操作任意数据集合。当用于内核时, 已命名函数对象的成员变量定义了内核可能操作的状态, 重载函数调用 operator() 将为并行执行的每个

工作项调用。

命名函数对象需要比 Lambda 更多的代码来表示内核，但提供了更多的功能。例如，更容易分析和优化以命名函数对象表示的内核，因为内核使用的任何缓冲区和数据都必须显式地传递给内核，而不是自动捕获。

最后，因为命名函数对象就像 C++ 类一样，表示为命名函数对象的内核可以模板化。以命名函数对象表示的内核也更容易重用，可以作为头文件或库的一部分。

内核命名函数对象的组成

图 10-6 中的代码描述了用命名函数对象表示的内核。

图 10-6 内核作为命名函数对象

```
1 class Add {
2 public:
3     Add(accessor<int> acc) : data_acc(acc) {}
4     void operator()(id<1> i) {
5         data_acc[i] = data_acc[i] + 1;
6     }
7
8 private:
9     accessor<int> data_acc;
10 };
11
12 int main() {
13     constexpr size_t size = 16;
14     std::array<int, size> data;
15
16     for (int i = 0; i < size; i++)
17         data[i] = i;
18
19     {
20         buffer data_buf{data};
21
22         queue Q{ host_selector{} };
23         std::cout << "Running on device: "
24             << Q.get_device().get_info<info::device::name>() << "\n";
25
26         Q.submit([&](handler& h) {
27             accessor data_acc {data_buf, h};
28             h.parallel_for(size, Add(data_acc));
29         });
30     }
31 }
```

当内核表示为命名函数时，必须遵循 C++11 规则才能复制。命名函数对象可以安全地逐字节地复制，使命名函数对象的成员变量能够传递给在设备上执行的内核代码。

重载函数调用 operator() 的参数取决于内核，就像用 Lambda 表示的内核一样。

因为是命名的函数对象，所以主机代码编译器可以使用函数对象类型与设备代码编译器生成的内核代码相关联。因此，命名内核函数对象不需要内核名称。

与其他 API 的互动性

当 SYCL 实现构建在另一个 API 上时，该实现可能能够与使用底层 API 机制定义的内核进行互操作。这允许应用程序将 SYCL 集成到现有代码库中。

因为 SYCL 实现可能基于许多 API，所以本节中描述的功能是可选的，并不是所有实现都支持。底层 API 甚至可能根据特定的设备类型，或设备供应商而有所不同！

广义地说，一个实现可能支持两种互操作性机制：从 API 定义的源或中间表示 (IR) 或从特定于 API 的句柄。这两种机制中，从 API 定义的源或中间表示创建内核的能力更具可移植性，因为一些源或 IR 格式由多个 API 支持。例如，OpenCL C 内核可直接使用，或者可以编译成某个 API 可以理解的形式，但是来自某个 API 的特定于 API 的内核句柄，不太可能让另一个 API 理解。

记住，所有形式的操作都是可选的！不同的 SYCL 实现可能支持从不同 API 句柄创建的内核——或者根本不支持。要了解详细信息，请查阅文档！

与 API 定义的互动性

使用这种形式的互动性，内核的内容描述为源代码或使用不是由 SYCL 定义的中间表示，但是内核对象仍然使用 SYCL API 创建。这种形式的互动性可以重用，使用其他源语言编写的内核库，或者使用中间表示形式生成代码的特定语言 (DSI)。

实现必须理解内核源代码或中间表示，才能利用这种形式的互动性。例如，如果内核以源码的形式使用 OpenCL C 编写，那么实现必须支持从 OpenCL C 内核源代码构建 SYCL 程序。

图 10-7 展示了如何将 SYCL 内核编写为 OpenCL C 内核源代码。

图 10-7 使用 OpenCL C 内核源码创建 SYCL 内核

```
1 // Note: This must select a device that supports interop!
2 queue Q{ cpu_selector{} };
3
4 program p{Q.get_context()};
5
6 p.build_with_source(R"CLC(
7     kernel void add(global int* data) {
8         int index = get_global_id(0);
9         data[index] = data[index] + 1;
10    }
11 )CLC",
12     "-cl-fast-relaxed-math");
13
14 std::cout << "Running on device: "
15     << Q.get_device().get_info<info::device::name>() << "\n";
16
```

```

17 Q.submit([&]( handler& h ) {
18     accessor data_acc{data_buf, h};
19
20     h.set_args(data_acc);
21     h.parallel_for(size, p.get_kernel("add"));
22 });

```

例子中，内核以字符串形式表示，在同一个文件中对 SYCL 的主机 API 进行调用（但这并不是必须的），一些程序可以从文件读取内核字符串然后生成内核。

因为 SYCL 编译器无法看到用内核，所以必须使用 set_arg() 或 set_args() 接口显式地传递内核参数。SYCL 运行时和 API 定义语言必须将对象作为内核参数。本例中，访问器 dataAcc 作为内核的全局指针参数进行数据传递。

build_with_source() 接口支持传递 API 定义的构建选项，来控制内核的编译方式。例子中，编译选项为-cl-fast-relaxation -math，用于表示内核编译器可以使用更快、低精度的数学库。编译选项是可选的，如果不需要生成选项，可以省略。。

与 API 定义内核的互动性

内核对象本身在另一个 API 中创建，然后导入 SYCL。这种形式的互动性使，应用程序的一部分可以使用底层 API 直接创建和使用内核对象，而应用程序的另一部分可以使用 SYCL API 重用相同的内核。图 10-8 中的代码展示了如何从 OpenCL 内核对象创建 SYCL 内核。

图 10-8 OpenCL 内核对象创建的内核

```

1 // Note: This must select a device that supports interop
2 // with OpenCL kernel objects!
3 queue Q{cpu_selector{}};
4 context sc = Q.get_context();
5
6 const char* kernelSource =
7 R"CLC(
8     kernel void add(global int* data) {
9         int index = get_global_id(0);
10        data[index] = data[index] + 1;
11    }
12 )CLC";
13 cl_context c = sc.get();
14 cl_program p =
15     clCreateProgramWithSource(c, 1, &kernelSource, nullptr, nullptr);
16 clBuildProgram(p, 0, nullptr, nullptr, nullptr, nullptr);
17 cl_kernel k = clCreateKernel(p, "add", nullptr);
18
19 std::cout << "Running on device: "
20      << Q.get_device().get_info<info::device::name>() << "\n";
21
22 Q.submit([&]( handler& h ) {
23     accessor data_acc{data_buf, h};

```

```

24
25     h.set_args(data_acc);
26     h.parallel_for(size, kernel{k, sc});
27 });
28
29 clReleaseContext(c);
30 clReleaseProgram(p);
31 clReleaseKernel(k);

```

与其他形式的互动性一样，SYCL 编译器对 API 定义的内核对象不可见。因此，必须使用 set_arg() 或 set_args() 接口显式传递参数，并且 SYCL 运行时和底层 API 必须遵循传递参数的约定。

程序对象中的内核函数

前面的部分中，内核是 API 定义的，或者是由特定的句柄创建的，内核通过两个步骤创建：首先创建程序对象，然后从程序对象创建内核。程序对象是内核及其调用的函数集合，这些函数编译为一个单元。

对于以 Lambda 或命名函数对象表示的内核，包含内核的程序对象是隐式的，对应用程序来说不可见。对于需要更多控制的应用程序，可以显式地管理内核和封装程序对象。为了说明为什么这样做，简要地了解下有多少 SYCL 实现管理即时 (JIT) 内核编译就能知道了。

虽然规范并不要求，但许多实现使用“惰性的”方式编译内核。这是一个很好的策略，因为可以确保快速启动程序，并且仅编译执行的内核。这种策略的缺点是，第一次使用内核的时间通常比后续要长，因为包括编译所需的时间，以及提交和执行内核的时间。对于某些复杂的内核，编译内核的时间可能很长，因此需要在应用程序执行期间将编译进行转移，比如：在应用程序加载时，或者在后台线程中。

一些内核还可能受益于实现定义的“构建选项”，以精确地控制内核的编译方式。例如：可以指示内核编译器使用精度较低，但性能更好的数学库。

为了更好地控制编译内核的时间和方式，应用程序可以使用特定的构建选项，在使用内核之前显式地编译内核。然后，将预编译的内核提交到一个队列中执行。原理如图 10-9 所示。

图 10-9 使用构建选项编译内核 Lambda 函数

```

1 // This compiles the kernel named by the specified template
2 // parameter using the "fast relaxed math" build option.
3 program p(Q.get_context());
4
5 p.build_with_kernel_type<class Add>("-cl-fast-relaxed-math");
6
7 Q.submit([&](handler& h) {
8     accessor data_acc {data_buf, h};
9
10    h.parallel_for<class Add>(
11        // This uses the previously compiled kernel.
12        p.get_kernel<class Add>(),

```

```
13     range{size},
14     [=](id<1> i) {
15         data_acc[i] = data_acc[i] + 1;
16     });
17 };
```

本例中，从 SYCL 上下文创建程序对象，使用 `build_with_kernel_type` 函数构建由指定的模板形参定义的内核。对于这个示例，程序构建选项-`cl-fast-relaxed-math` 表示内核编译器可以使用更快的数学库，但程序构建选项是可选的，如果不需要特殊的程序构建选项，可以省略。本例中，需要指定内核 Lambda 的模板参数，以确定要编译的内核。

也可以从特定设备的上下文上创建程序对象，程序对象可以使用不同的构建选项，将内核编译到不同的设备对象上。

除了内核 Lambda 外，之前编译的内核使用 `get_kernel` 将内核传递给 `parallel_for`。这可以确保使用使用高效数学库构建的内核。如果之前编译的内核没有传递给 `parallel_for`，那么内核将再次编译，不需要任何构建选项。这可能在功能上是正确的，但肯定不是预期的行为！

许多情况下，这些步骤不太可能对应用程序的行为产生影响，但是在调优应用程序的性能时，需要考虑这些步骤所带来的性能影响。

改进互动性和程序对象管理

虽然本章介绍了描述的用于互动性和程序对象管理的 SYCL 接口，但它们可能会在 SYCL 和 DPC++ 的未来版本中得到改进和增强。请参考最新的 SYCL 和 DPC++ 文档，以找到更新。

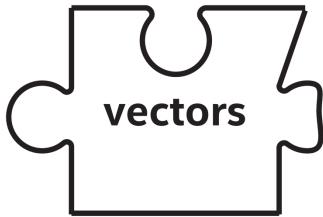
总结

本章中，探索了定义内核的不同方法。描述了如何通过将内核表示为 C++ Lambda 或命名函数对象，来集成到现有的 C++ 库中。对于新的代码，还讨论了不同内核表示的优缺点，以帮助应用程序或库的需要选择定义内核的最佳方式。

还介绍了如何与其他 API 进行互动，通过 API 的表示中创建内核，或者通过句柄到内核的 API 表示创建内核对象。互动性使应用程序可以随着时间的推移从底层 API 迁移到 SYCL，或者与为其他 API 编写接口。

最后，描述了在 SYCL 应用程序中内核是如何编译的，以及如何直接操作程序对象中的内核来控制编译。尽管大多数应用程序不需要这种级别的控制，但在调优时了解这一点这方面的知识是有用的。

11 向量



向量是数据的集合，计算机中的并行性来自于计算硬件的集合，数据通常以相关的分组进行处理（例如，RGB 像素中的颜色通道）。如此重要的特性，值得用一章来讨论向量类型的优点，以及如何使用它们。本章中，不会深入到向量化，因为它会根据设备类型和实现而变化。向量化将在第 15 和 16 章中讨论。

本章旨在解决以下问题：

- 什么是向量类型？
- 关于向量接口，需要知道多少？
- 应该使用向量类型来表示并行性吗？
- 什么时候使用向量类型？

我们将使用代码示例讨论可用向量类型的优缺点，并重点介绍向量类型的使用。

如何使用向量的方式思考

当与并行编程专家交谈时，向量是一个有争议话题。根据作者的经验，这是因为不同的人以不同的方式定义和思考这个术语。

有两种对向量数据类型（数据的集合）的理解：

1. **作为一种方便类型：**例如，将像素的颜色通道（例如 RGB、YUV）分组为单个变量（例如：float3），可以是一个向量。可以定义一个像素类或结构，并在其上定义像 + 这样的数学运算符，但向量类型可以方便地使用。其类型可以在许多着色器语言中找到，所以这种思维方式在许多 GPU 开发者中已经是共识了。
2. **作为一种描述代码机制，是如何映射到硬件适配的 SIMD 指令集上的呢？**例如，一些语言和实现中，float8 上在理论上可以映射到硬件中的 8 通道 SIMD 指令。Vector 类型在多种语言中作为特定指令集的一种高级替代存在。

尽管这两种解释非常不同，但当 SYCL 和其他语言同时适用于 CPU 和 GPU 时，可以将其组合在一起。SYCL 1.2.1 规范中的向量与这两种理解兼容（稍后将再次讨论这一点）。在进一步讨论之前，需要了解 DPC++ 中的理解。

本书中，讨论了如何将工作项组合在一起以便使用强大的通信和同步原语，例如：子工作组栅栏和混洗。为了使这些操作在向量硬件上效率最优，假设子工作组中的不同工作项可以合并，并映射到 SIMD 指令。换句话说，编译器可以将多个工作项组合在一起，可以映射到硬件的 SIMD 指令上。第 4 章中，SPMD 编程模型操作需要在支持向量操作的硬件上进行，一个通道的工作项构成的 SIMD 指令，而不是一个工作项定义了整个操作的 SIMD 指令。当硬件中映射到 SIMD 指令，并使用 DPC++ 编译器以 SPMD 风格编程时，可以认为编译器总是在跨工作项进行向量化。

对于本书中描述的特性和硬件，向量主要用于本节的第一个解释——向量是一种类型，不应该认为是对 SIMD 指令的映射。工作项分组在一起，在指令的 (CPU、GPU) 硬件上形成 SIMD 指令。向量应该认为是提供方便的操作符，如 swizles 和 math 函数，使代码中对数据组的通用操作更加方便 (例如，添加两个 RGB 像素)。

若开发者没有接触过 GPU 渲染语言中的向量，可以将 SYCL 向量作为一个本地工作项，如果有两个具有 4 个元素向量做加法，可能需要四个指令的硬件 (这是从标量的角度)。向量的每个元素可以通过不同的指令/时钟周期相加。解释一下应该很容易懂，可以在源代码的单个操作中直接操作两个向量，而非对四个标量进行操作。

对于有 CPU 背景的开发人员，应该知道对 SIMD 硬件的隐式向量在编译器中以几种独立于向量类型的方式默认发生。编译器在工作项之间执行这种隐式向量化，从循环中提取向量操作，或者在映射到指令操作的向量类型一更多信息请参见第 16 章。

其他可能的实现

SYCL 和 DPC++ 的不同编译器和实现在理论上，可以对代码中的向量数据类型如何映射到向量硬件指令做出不同的决定。应该阅读供应商的文档和优化指南，以理解如何编写将映射到有效 SIMD 指令的代码。本书主要是针对 DPC++ 编译器编写，因此介绍了 DPC++ 的编程思维和模式。

变化即将发生

要将向量类型视为方便类型，并在考虑到设备上的硬件映射，并期待跨工作项的向量化。这将成为 DPC++ 编译器和工具链的默认行为。然而，还有另外两个变化需要注意。

首先，可以期待一些 DPC++ 特性，这些特性将允许编写直接映射到硬件中 SIMD 指令的代码，特别是对于那些希望为特定体系结构进行代码调优，并从编译器向量器获得控制权的专家来说。虽然只会有少数开发人员使用的小众特性，但是可以期待这种编程机制。这些编程机制将确定代码风格 (显式向量化风格)，这样就不会在编写现有代码时，与显式 (且不可移植) 风格之间产生混淆。

其次，本书这一节 (讨论向量的解释) 的需要强调了向量的含义存在混淆，这将在 SYCL 中解决。SYCL 2020 临时规范中描述了一种数学数组类型 (marray)，这是本节的第一个解释——与向量硬件指令无关的类型。应该期望另一种类型最终会覆盖第二种解释，很可能与 C++ 的 std::simd 模板一致。有了这两种类型与向量数据类型的解释相关联，作为开发者将在编写的代码中清楚地传递信息。这将减少错误和混乱，当“什么是向量？”问题出现时，甚至可能减少专家级开发人员之间的激烈讨论，

向量的类型

SYCL 中的向量类型是跨平台的类模板，可以在设备和主机 C++ 代码中工作，并允许在主机和设备之间共享向量。向量类型包括允许从混合组件元素构造新向量，这样新向量的元素可以按照任意顺序从旧向量的元素中挑选。vec 是一种向量类型，可编译为目标设备后端上的内置向量类型，

并在主机上提供兼容支持。

vec 类是根据元素数量和元素类型模板化。参数 numElements 的元素个数可以是 1、2、3、4、8 或 16 中的一个，任何其他值都将产生编译失败。元素类型参数 dataT，必须是设备代码中支持的标量类型。

SYCL ec 类模板提供了与 vector_t 定义的底层 vector 类型的互动，该类型仅在为设备编译时可用。vec 类可以从 vector_t 的实例构造，并可以隐式地转换为 vector_t 实例，以支持与内核函数（例如，OpenCL 后端）的本地 SYCL 后端的互操作。当元素的数量为 1 时，还可以隐式地将 vec 类模板的实例转换为数据类型的实例，以便单元素向量和标量间的互换。

为便于编程，SYCL 提供了许多使用的单类型别名 $\langle type \rangle \langle elems \rangle = \text{vec} < \langle storage-type \rangle, \langle elems \rangle \rangle$ ，这里的 $\langle elems \rangle$ 为 2, 3, 4, 8 和 16 和 $\langle type \rangle$ 配对，并且 $\langle storage-type \rangle$ 为整型 char \Leftrightarrow int8_t, uchar \Leftrightarrow uint8_t, short \Leftrightarrow int16_t, ushort \Leftrightarrow uint16_t, int \Leftrightarrow int32_t, uint \Leftrightarrow uint32_t, long \Leftrightarrow int64_t，以及 ulong \Leftrightarrow uint64_t 对于浮点类型 half, float 和 double。例如：uint4 是 $\text{vec} < \text{uint32_t}, 4 \rangle$ 的别名，float16 是 $\text{vec} < \text{float}, 16 \rangle$ 的别名。

向量的接口

vector 类型的功能是通过 vec 类使用，vec 类表示一组数据元素。vec 类模板的构造函数、成员函数和非成员函数的接口描述在图 11-1、11-4 和 11-5 中。

图 11-2 中的 XYZW 成员只有在 numElements ≤ 4 时才可用。RGBA 成员只有在 numElements == 4 时可用。

图 11-3 中的成员 lo、hi、odd 和 even 只有在 numElements > 1 的情况下才可用。

图 11-1 vec 类声明和成员函数

```
1 vec Class declaration
2 template <typename dataT, int numElements> class vec;
3 vec Class Members
4 using element_type = dataT;
5 vec();
6 explicit vec(const dataT &arg);
7 template <typename ... argTN> vec(const argTN&... args);
8 vec(const vec<dataT, numElements> &rhs);
9
10 #ifdef __SYCL_DEVICE_ONLY__ // available on device only
11 vec(vector_t openclVector);
12 operator vector_t() const;
13 #endif
14
15 operator dataT() const; // Available only if numElements == 1
16 size_t get_count() const;
17 size_t get_size() const;
18
19 template <typename convertT, rounding_mode roundingMode>
20 vec<convertT, numElements> convert() const;
21 template <typename asT> asT as() const;
```

图 11-2 swizzled_vec 的成员函数

```
1 template<int ... swizzleIndexes>
2     __swizzled_vec__ swizzle() const;
3     __swizzled_vec__ XYZW_ACCESS() const;
4     __swizzled_vec__ RGBA_ACCESS() const;
5     __swizzled_vec__ INDEX_ACCESS() const;
6
7 #ifdef SYCL_SIMPLE_SWIZZLES
8 // Available only when numElements <= 4
9 // XYZW_SWIZZLE is all permutations with repetition of:
10 // x, y, z, w, subject to numElements
11     __swizzled_vec__ XYZW_SWIZZLE() const;
12
13 // Available only when numElements == 4
14 // RGBA_SWIZZLE is all permutations with repetition of: r, g, b, a.
15     __swizzled_vec__ RGBA_SWIZZLE() const;
16#endif
```

图 11-3 vec 的函数操作符

```
1     __swizzled_vec__ lo() const;
2     __swizzled_vec__ hi() const;
3     __swizzled_vec__ odd() const;
4     __swizzled_vec__ even() const;
5
6 template <access::address_space addressSpace>
7     void load(size_t offset, mult_ptr<dataT, addressSpace> ptr);
8 template <access::address_space addressSpace>
9     void store(size_t offset, mult_ptr<ptr<dataT, addressSpace> ptr) const;
10
11 vec<dataT, numElements> &operator=(const vec<dataT, numElements> &rhs);
12 vec<dataT, numElements> &operator=(const dataT &rhs);
13 vec<RET, numElements> operator!();
14
15 // Not available for floating point types:
16 vec<dataT, numElements> operator~();
```

图 11-4 vec 的成员函数

成员函数 (OP) 类型	对于所有类型可能支持的操作	对于整型可能支持的操作
vec<dataT, numElements> operatorOP(const vec<dataT, numElements>&rhs) const;	+,-,*,/	%,&, ,^, <<, >>
vec<dataT, numElements> operatorOP(const dataT &rhs) const		
vec<dataT, numElements> &operatorOP(const vec<dataT, numElements>&rhs) const;	+,-,*,/=	%=,&=, =, ^=,<<=, >>=
vec<dataT, numElements> &operatorOP(const dataT &rhs) const;		
vec<RET, numElements> operatorOP(const vec<dataT, numElements>&rhs) const;	&&, ,==, !=,<,>,<=,	
vec<RET, numElements> operatorOP(const dataT &rhs) const;	>=	
vec<dataT, numElements> &operatorOP() const;	++,-	
vec<dataT, numElements> operatorOP(int) const;		

图 11-5 vec 的非成员函数

成员函数 (OP) 类型	对于所有类型可能支持的操作	对于非浮点类型可能支持的操作
template<typename dataT, int numElements> vec<dataT, numElements> operatorOP(const dataT, &lhs, const vec<dataT, numElements>& rhs);	+,-,*,/	%,&, ,^,<<. >>
template<typename dataT, int numElements> vec<RET, numElements> operatorOP(const dataT &lhs, const vec<dataT, numElements>&rhs);	&&, ,==, !=,<,>,<=, >=	

加载和存储成员函数

向量的加载和存储操作，是用于加载和存储向量元素的 vec 类的成员。这些操作可以是指向或来自与向量通道类型相同的元素数组。如图 11-6 所示。

图 11-6 使用加载和存储成员函数。

```
1 buffer fpBuf(fpData);
```

```

2 queue Q;
3 Q.submit([&](handler& h){
4     accessor buf{fpBuf, h};
5
6     h.parallel_for(size, [=](id<1> idx){
7         size_t offset = idx[0]/16;
8         float16 inpf16;
9         inpf16.load(offset, buf.get_pointer());
10        float16 result = inpf16 * 2.0f;
11        result.store(offset, buf.get_pointer());
12    });
13 });

```

vec 类中，`dataT` 和 `numElements` 是反映 vec 的组件类型和维数的模板参数。

`load()` 成员函数模板从 `multi_ptr` 地址的内存中读取 `dataT` 类型的值，`dataT` 元素的 `offset` 乘以 `numElements*offset`，并将这些值写入 vec 的通道中。

`store()` 成员函数模板将读取向量的通道，并将这些值写入 `multi_ptr` 地址的内存中，`dataT` 元素中的 `offset` 乘以 `numElements*offset`。

形参是 `multi_ptr`，而不是访问器，可以使用本地创建的指针，以及主机指针。

`multi_ptr` 的数据类型是 `dataT`，即 vec 类专门化组件的数据类型。这要求传递给 `load()` 或 `store()` 的指针必须与 vec 实例本身的数据类型匹配。

混合 (Swizzle) 操作

图形应用程序中，混合意味着重新安排矢量的数据元素。例如，如果 `a = {1,2,3,4, }`，并且知道一个四元向量的分量可以称为 `{x, y, z, w}`，可以写成 `b = a.wxyz()`，变量 b 的结果是 `{4,1,2,3}`。这种形式的代码在 GPU 程序中很常见，GPU 应用程序中有高效的硬件进行此类操作。混合可以通过两种方式进行：

- 通过调用 vec 的 swizzle 成员函数，该函数接受从 0 到 `numElements-1` 之间的可变数目的整型模板参数，指定 swizzle 索引
- 通过调用简单的 swizzle 成员函数，如 `XYZW_SWIZZLE` 和 `RGBA_SWIZZLE`

简单的 swizzles 函数只对最多 4 个元素的向量可用，并且只有在包含 `SYCL.hpp` 之前定义了宏 `SYCL_SIMPLE_SWIZZLES` 时才可用。这两种情况下，返回类型是一个 `__swizzled_vec__` 实例，实现定义的临时类表示原始 vec 实例的 swizzle。swizzle 成员函数模板和简单的 swizzle 成员函数都允许重复 swizzle 索引。图 11-7 展示了 `__swizzled_vec__` 的简单用法。

图 11-7 使用 `__swizzled_vec__` 类的示例

```

1 constexpr int size = 16;
2
3 std::array<float4, size> input;
4 for (int i = 0; i < size; i++)
5     input[i] = float4(8.0f, 6.0f, 2.0f, i);
6
7 buffer B(input);

```

```

8
9 queue Q;
10 Q.submit([&](handler& h) {
11     accessor A{B, h};
12
13     // We can access the individual elements of a vector by using
14     // the functions x(), y(), z(), w() and so on.
15     //
16     // "Swizzles" can be used by calling a vector member equivalent
17     // to the swizzle order that we need, for example zyx() or any
18     // combination of the elements. The swizzle need not be the same
19     // size as the original vector.
20     h.parallel_for(size, [=](id<1> idx) {
21         auto b = A[idx];
22         float w = b.w();
23         float4 sw = b.xyzw();
24         sw = b.xyzw() * sw.wzyx();;
25         sw = sw + w;
26         A[idx] = sw.xyzw();
27     });
28 });

```

在并行内核中使用向量

如第 4 章和第 9 章所述，工作项是并行层次结构的叶节点，并表示内核函数的单个实例。工作项可以以任何顺序执行，并且不能相互通信或同步，除非通过对局部和全局内存的原子内存操作，或者通过组集合函数（例如 shuffle、barrier）。

正如本章开始所描述的，DPC++ 中的向量是为了方便使用，每个向量对于单个工作项都是本地的（而不是与硬件中的向量化相关），因此可以看作是工作项中的 numElements 的私有数组。例如，“float4 y4” 声明的存储等价于 float y4[4]。如图 11-8 所示。

图 11-8 向量执行示例

```

1 Q.parallel_for(8, [=](id<1> i){
2     ...
3     float x = a[i]; // i = 1,2,3...,7
4     float4 y4 = b[i]; // i = 1,2,3...,7
5     ...
6 });

```

对于标量变量 x，在具有 SIMD 指令（例如，CPU、GPU）的硬件上使用多个工作项的内核执行可能会使用向量寄存器和 SIMD 指令，但向量化是跨工作项的，并且与代码中的任何向量类型无关。每个工作项可以在 vec_x 中的不同位置上操作，如图 11-9 所示。工作项中的标量数据，可以看作在工作项之间的隐式向量化（合并到 SIMD 硬件指令中），但编写的工作项代码没有对此进行编码——这是 SPMD 编程风格的核心。

work-item ID	vec_y0	vec_y1	vec_y2	vec_y3
w0	b0	b0	b0	b0
w1	b1	b1	b1	b1
w2	b2	b2	b2	b2
w3	b3	b3	b3	b3
w4	b4	b4	b4	b4
w5	b5	b5	b5	b5
w6	b6	b6	b6	b6
w7	b7	b7	b7	b7

图 11-9 从标量变量 x 到 vec_x[8] 的向量展开

work-item ID	w0	w1	w2	w3	w4	w5	w6	w7
vec_x	a0	a1	a2	a3	a4	a5	a6	a7

通过编译器标量变量 x 到 vec_x[8] 的隐式向量展开 (如图 11-9 所示), 编译器从出现在多个工作项中的标量操作创建一个 SIMD 操作。

对于向量变量 y4, 内核执行多个工作项 (例如: 8 个工作项) 的结果不会通过在硬件中使用向量操作来处理 vec4。每个工作项独立地使用向量, 并且该向量上元素的操作发生在多个时钟周期/指令上 (编译器扩展了向量), 如图 11-10 所示。

图 11-10 垂直展开到 y4 的 vec_y[8][4] 的等量, 横跨 8 个工作项

每个工作项都看到 y4 的原始数据布局, 这提供了一个直观的模型来进行推理和调优。性能的缺点是编译器必须为 CPU 和 GPU 生成收集/分散内存指令, 如图 11-11 所示 (向量在内存中是连续的, 并且相邻的工作项并行地在不同的向量上操作), 因此, 当编译器将跨工作项 (例如: 跨子组) 向量化时, 标量通常是显式向量的有效方法。详见第 15 章和第 16 章。

图 11-11 带有地址转义的向量代码示例

```

1 Q.parallel_for(8, [=](id<1> i) {
2   ...
3   float x = a[i]; // i = 1,2,3...,7
4
5   // "dowork" expects y4 with vec_y[8][4] data layout
6   float x = dowork(&y4);
7
8 });

```

当编译器能够证明 y4 的地址没有从当前内核工作项中转移, 或者所有调用的函数都将内联, 那么编译器可能会执行优化, 就像使用一组向量寄存器从 y4 水平展开到 vec_y[4][8] 一样, 如图 11-12 所示。这种情况下, 编译器可以在不收集/分布 SIMD 指令的情况下获得最佳性能。编译器优化报告向程序员提供了关于这种类型转换的信息, 可以提供关于如何调整代码以提高性能的提示。

图 11-12 将 y_4 扩展到 $\text{vec_y}[4][8]$

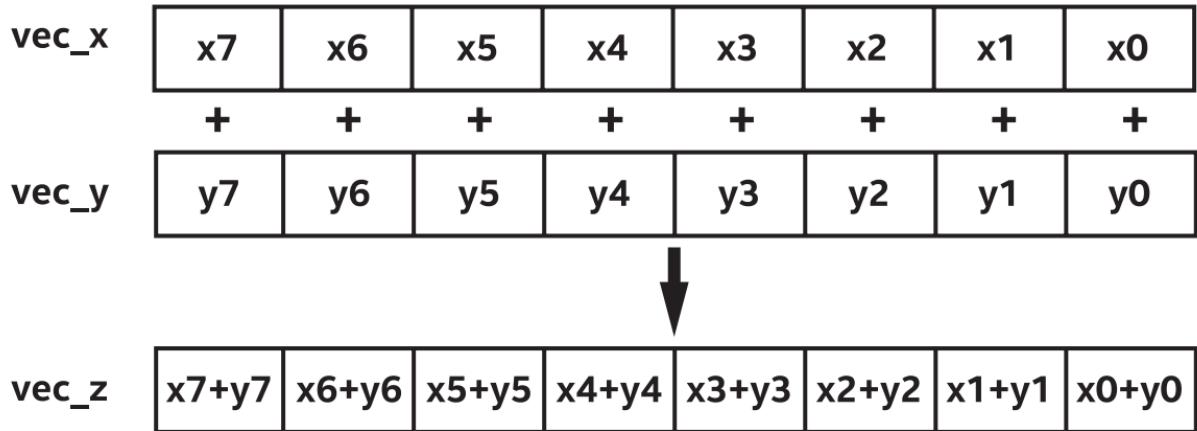
work-item ID	w0	w1	w2	w3	w4	w5	w6	w7
y_0	b0	b1	b2	b3	b4	b5	b6	b7
y_1	b0	b1	b2	b3	b4	b5	b6	b7
y_2	b0	b1	b2	b3	b4	b5	b6	b7
y_3	b0	b1	b2	b3	b4	b5	b6	b7

向量并行

尽管 DPC++ 中的向量应该解释为仅适用于单个工作项的工具，但不提到硬件中的 SIMD 指令是如何操作的话，这一章关于向量的内容就不完整。这个主题并不与向量耦合，而是与向量无关，本书后面描述特定设备类型 (GPU, CPU, FPGA) 的章节时，就会了解讨论这个话题的重要性了。

现代的 CPU 和 GPU 包含 SIMD 指令硬件，对包含在一个向量寄存器或寄存器文件中的多个数据值进行操作。例如，对于 Intel x86 AVX-512 和其他现代 CPU SIMD 硬件，SIMD 指令可以用来利用数据并行性。在提供 SIMD 的 CPU 和 GPU 上，可以考虑向量加法操作，例如：一个 8 元素向量上，如图 11-13 所示。

图 11-13 SIMD 增加了 8 路数据并行性



本例中的向量加法可以在向量硬件上的一条指令中执行，将向量寄存器 vec_x 和 vec_y 与 SIMD 指令并行地相加。

以一种与硬件无关的方式公开并行性，确保应用程序可以扩展 (或缩小) 规模，以适应不同平台的功能，包括那些带有向量指令的平台。在应用程序开发过程中，工作项和其他形式的并行性之间的平衡，是我们必须面对的挑战，这将在第 15、16 和 17 章中详细讨论。

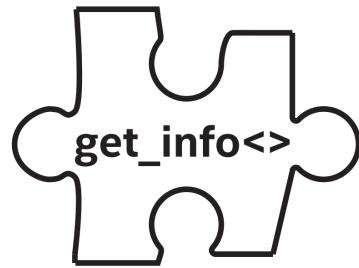
总结

编程语言中，向量一词有多种解释，想要编写高性能和可扩展的代码时，理解语言或编译器的解释很重要。DPC++ 编译器围绕这样的思想构建：源码中的向量是工作项的本地函数，编译器跨

工作项的隐式向量值，可能映射到硬件中的 SIMD 指令。当我们想要编写显式映射到 SIMD 指令集时，应该查看供应商的文档和对 SYCL 和 DPC++ 的扩展。使用多个工作项（例如 ND-Range）编写内核，并依赖编译器跨工作项向量化应该是大多数应用程序的方式，这样做利用了 SPMD 强大的抽象功能，它提供了一个简单的编程模型，提供了跨设备和架构的可扩展性。

本章描述了 `vec` 接口，想要对类似的数据类型进行操作（例如，一个具有多个颜色通道的像素）时，提供了便利。本章还简要介绍了硬件中的 SIMD 指令，以便在第 15 和 16 章中进行更详细的讨论。

12 设备信息



第 2 章介绍了将工作导向特定设备的机制——控制代码执行的位置。本章中，我们将探索如何适应运行时出现的设备。

我们希望程序可以移植，程序需要适应设备的功能。我们可以将程序参数化，只使用现有的特性，并根据设备的具体情况调整代码。如果程序不能适应环境，那么不好的事情就会发生，比如执行缓慢或失败。

幸运的是，SYCL 规范考虑到了这一点，并提供了解决这个问题的接口。SYCL 规范定义了一个设备类，封装了可以执行内核的设备。查询设备类的能力，使程序能够适应设备的特性和能力，这是本章的核心内容。

许多人将从开始思考如何将“是否存在 GPU?”通知正在执行的程序，并让程序自身做出选择。如我们将看到的，有更多的信息可以帮助我们使程序的健壮和性能更好。

对程序进行参数化可以帮助提高程序的正确性、功能可移植性和性能可移植性。

本章将深入探讨查询，以及如何在程序中如何有效地使用。

设备特定的属性可以使用 `get_info` 查询，但是 DPC++ 不同于 SYCL 1.2.1，完全重载了 `get_info`，以减少使用 `get_work_group_info` 获取工作组的信息，而工作组信息实际上是设备特定的信息。DPC++ 不支持 `get_work_group_info`，这意味着特定于设备的内核和工作组属性可以查询特定于设备的属性 (`get_info`)。这纠正了 SYCL 1.2.1 中从 OpenCL 继承而来的历史问题。

优化内核代码

考虑到编码，内核大致可以分为以下三类：

- 泛型内核代码：在任何地方运行，而不是调优到特定的设备类。
- 设备类型特定的内核代码：运行在一种类型的设备（例如，GPU, CPU, FPGA）上，而不是针对某一设备特定的类型。因为许多设备类型具有共同的特性，所以可以编写一些不适用于所有设备的通用代码。
- 调优特定于设备的内核代码：运行在特定的设备上，使用对设备的特定参数作出反应的调优——这涵盖了从少量调优到详细的优化工作的可能性。

作为开发者，我们的工作是确定不同的设备类型，以及何时需要不同的模式（第 14 章）。我们会在第 14、15、16 和 17 章来阐明这一重要的思想。

最常见的做法是先实现通用内核代码以使其工作。第 2 章特别讨论了在开始内核实现时哪些方法最容易调试。一旦有了可以工作的内核，就可以将它发展为针对特定设备类型或设备模型的功能。

深入研究设备问题之前，第 14 章提供了一个思考并行性的框架。对模式（又名算法）的选择决定了代码模式，我们的工作是确定不同的设备何时需要不同的模式。第 15 章（GPU）、第 16 章（CPU）和第 17 章（FPGA）深入地探讨了区分这些设备类型和使用模式选择的特性。当不同设备类型上的方法（模式选择）不同时，这些特性促使我们考虑为不同的设备编写不同版本的内核。

当为特定类型的设备（例如，特定的 CPU、GPU、FPGA 等）编写内核时，将其适应于特定的供应商甚至此类设备的模型是合乎逻辑的。好的编码风格是基于特性（例如，从设备查询中找到的项目大小支持）参数化的代码。

我们应该编写代码来查询描述设备实际性能的参数，而不是从互联网上查询它的市场信息；查询设备的型号并对此作出反应是非常糟糕的编程实践——这种代码的可移植性较差。

为支持的每种设备类型编写不同的内核是很常见的（内核的 GPU 版本和内核的 FPGA 版本，也许还有通用版本）。当支持特定的设备供应商甚至设备模型时，可以参数化内核而不是复制时，工作会轻松很多。只要我们认为合适，可以自由选择其中任何一种。有太多参数调整的代码可能难以阅读或在运行时负担过重。然而，参数可以很好地适用于内核的每个版本。

算法大致相同，当针对特定设备的功能进行了调整时，参数化最有意义。当使用完全不同的方法、模式或算法时，编写不同的内核要干净得多。

如何枚举设备和功能

第 2 章列举并解释了选择要执行的设备的五种方法。本质上，方法 1 是不规定内核在什么地方运行，而方法 5 则相反，会考虑在设备上执行一个精确的模型。介于两者之间的枚举方法提供了灵活性和规定性。图 12-1、12-2 和 12-3 说明了如何选择设备。

图 12-1 展示了实现会为选择一个默认设备（第 2 章中的方法 1），可以查询有关所选设备的信息。

图 12-2 展示了如何使用特定的设备（本例中是 GPU）设置队列。如果没有可用的 GPU，则在主机上显式地返回。这给了我们选择设备的控制权，如果简单地使用默认队列，最终可能会得到意想不到的设备类型（例如，DSP、FPGA）。如果明确地想要在没有 GPU 设备的情况下使用主机设备，代码则可以会做到。回想一下，主机设备总是存在的，所以使用 host_selector 时不用担心。

不建议使用如图 12-2 所示的解决方案。除了看起来有点吓人和容易出错之外，图 12-2 没有给我们选择什么 GPU 的控制权，如果有多个可用的 GPU，其会依赖于实现进行选择。尽这个例子有教育意义和实用价值，但还是有更好的方法可以替代。建议编写自定义设备选择器，如下面的代码示例（图 12-3）所示。

自定义设备选择器

图 12-3 使用自定义设备选择器。自定义设备选择器在第 2 章中作为方法 5 讨论，用来选择的代码运行的位置（图 2-15）。自定义设备选择器会为应用程序可用的每个设备调用 operator()，如图 12-3 所示。在这个例子中，选中的设备为得分最高的设备，我们将使用选择器进行一些有趣的操作：

- 拒绝供应商名称包含“Martian”(返回-1)的 GPU。
- 建议使用供应商名称包含单词“ACME”的 GPU(返回 824)。
- 任何其他 GPU(返回 799)。
- 如果没有 GPU，我们选择主机设备(返回 99)。
- 忽略所有设备(返回-1)。

下一节，“get_info<>”深入研究了 get_devices()、get_platforms() 和 get_info<> offer 的信息。这些接口为选择设备提供了逻辑参考，包括图 2-15 和 12-3 中所示的简单的供应商名称检查。

图 12-1 使用默认分配的设备

```

1 queue Q;
2
3 std::cout << "By default, we are running on "
4     << Q.get_device().get_info<info::device::name>() << "\n";
5
6 // sample output:
7 // By default, we are running on Intel(R) Gen9 HD Graphics NEO.
```

关于设备的查询依赖于已安装的软件(特殊的用户级驱动程序)。SYCL 和 DPC++ 也依赖这些软件，就像操作系统需要驱动程序来访问硬件一样——仅仅将硬件安装在一台机器上是不够的。

图 12-2 使用 try-catch 选择 GPU 设备和主机设备

```

1 auto GPU_is_available = false;
2
3 try {
4     device testForGPU((gpu_selector()));
5     GPU_is_available = true;
6 } catch (exception const& ex) {
7     std::cout << "Caught this SYCL exception: " << ex.what() << std::endl;
8 }
9
10 auto Q = GPU_is_available ? queue(gpu_selector()) : queue(host_selector());
11
12 std::cout << "After checking for a GPU, we are running on:\n"
13     << Q.get_device().get_info<info::device::name>() << "\n";
14
15 // sample output using a system with a GPU:
16 // After checking for a GPU, we are running on:
17 // Intel(R) Gen9 HD Graphics NEO.
18 //
19 // sample output using a system with an FPGA accelerator, but no GPU:
20 // Caught this SYCL exception: No device of requested type available.
21 // ... (CL_DEVICE_NOT_FOUND)
```

```
22 // After checking for a GPU, we are running on:  
23 // SYCL host device.
```

图 12-3 自定义设备选择器——首选的解决方案

```
1 class my_selector : public device_selector {  
2 public:  
3     int operator()(const device &dev) const {  
4         int score = -1;  
5  
6         // We prefer non-Martian GPUs, especially ACME GPUs  
7         if (dev.is_gpu()) {  
8             if (dev.get_info<info::device::vendor>().find("ACME")  
9                 != std::string::npos) score += 25;  
10  
11            if (dev.get_info<info::device::vendor>().find("Martian")  
12                == std::string::npos) score += 800;  
13        }  
14  
15        // Give host device points so it is used if no GPU is available.  
16        // Without these next two lines, systems with no GPU would select  
17        // nothing, since we initialize the score to a negative number above.  
18        if (dev.is_host()) score += 100;  
19        return score;  
20    }  
21};  
22  
23 int main() {  
24     auto Q = queue{ my_selector{} };  
25  
26     std::cout << "After checking for a GPU, we are running on:\n"  
27         << Q.get_device().get_info<info::device::name>() << "\n";  
28  
29     // Sample output using a system with a GPU:  
30     // After checking for a GPU, we are running on:  
31     // Intel(R) Gen9 HD Graphics NEO.  
32     //  
33     // Sample output using a system with an FPGA accelerator, but no GPU:  
34     // After checking for a GPU, we are running on:  
35     // SYCL host device.  
36     return 0;  
37 }
```

get_info<>

为了让程序“知道”在运行时哪些设备可用，可以让程序从设备类中查询可用的设备，然后可以使用 get_info<> 查询特定的设备来了解更多细节。我们提供了一个简单的程序，叫做 curious(参见图 12-4)，它使用这些接口输出信息直接查看。开发或调试使用这些接口的程序时，进行完整性

检查非常有用，接口失败通常是软件驱动程序没有正确安装。图 12-5 显示了该程序的示例输出，其中包含了有关当前设备的高级信息。

图 12-4 设备查询机制的简单使用:curious.cpp

```
1 // Loop through available platforms
2 for (auto const& this_platform : platform::get_platforms() ) {
3     std::cout << "Found platform: "
4         << this_platform.get_info<info::platform::name>() << "\n";
5
6 // Loop through available devices in this platform
7 for (auto const& this_device : this_platform.get_devices() ) {
8     std::cout << " Device: "
9         << this_device.get_info<info::device::name>() << "\n";
10 }
11 std::cout << "\n";
12 }
```

图 12-5 来自 curious.cpp 的示例输出

```
% make curious
dpcpp curious.cpp -o curious

% ./curious
Found platform 1...
Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
Device: Intel(R) FPGA Emulation Device

Found platform 2...
Platform: Intel(R) OpenCL HD Graphics
Device: Intel(R) Gen9 HD Graphics NEO

Found platform 3...
Platform: Intel(R) OpenCL
Device: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz

Found platform 4...
Platform: SYCL host platform
Device: SYCL host device
```

了解更多：详细的枚举

程序命名为 verycurious.cpp(图 12-6), 以说明使用 `get_info<>` 可以获得的详细信息。同样, 发现自己编写这样的代码是为了在开发或调试程序时提供帮助。图 12-5 显示了该程序的示例输出, 其中包含有关于当前设备的底层信息。

现在我们已经展示了如何访问信息, 接下来将讨论在应用程序中最重要的查询和操作的信息字段。

图 12-6 设备查询机制的更详细使用:verycurious.cpp

```
1 template <auto query , typename T>
2 void do_query( const T& obj_to_query , const std::string& name, int indent=4)
3 {
4     std::cout << std::string(indent, ' ') << name << " is "
5     << obj_to_query.template get_info<query>() << "\n";
6 }
7
8 // Loop through the available platforms
9 for (auto const& this_platform : platform::get_platforms() ) {
10     std::cout << "Found Platform:\n";
11     do_query<info::platform::name>(this_platform ,
12         "info :: platform :: name");
13     do_query<info::platform::vendor>(this_platform ,
14         "info :: platform :: vendor");
15     do_query<info::platform::version>(this_platform ,
16         "info :: platform :: version");
17     do_query<info::platform::profile>(this_platform ,
18         "info :: platform :: profile");
19
20 // Loop through the devices available in this platform
21 for (auto &dev : this_platform.get_devices() ) {
22     std::cout << " Device: "
23     << dev.get_info<info::device::name>() << "\n";
24     std::cout << " is_host(): "
25     << (dev.is_host() ? "Yes" : "No") << "\n";
26     std::cout << " is_cpu(): "
27     << (dev.is_cpu() ? "Yes" : "No") << "\n";
28     std::cout << " is_gpu(): "
29     << (dev.is_gpu() ? "Yes" : "No") << "\n";
30     std::cout << " is_accelerator(): "
31     << (dev.is_accelerator() ? "Yes" : "No") << "\n";
32
33     do_query<info::device::vendor>(dev , "info :: device :: vendor");
34     do_query<info::device::driver_version>(dev ,
35         "info :: device :: driver_version");
36     do_query<info::device::max_work_item_dimensions>(dev ,
37         "info :: device :: max_work_item_dimensions");
38     do_query<info::device::max_work_group_size>(dev ,
39         "info :: device :: max_work_group_size");
40     do_query<info::device::mem_base_addr_align>(dev ,
```

```

41     " info :: device :: mem_base_addr_align");
42 do_query<info :: device :: partition_max_sub_devices>(dev,
43     " info :: device :: partition_max_sub_devices");
44
45     std :: cout << " Many more queries are available than shown here! \n";
46 }
47 std :: cout << "\n";
48 }
```

get_info<>

has_extension() 接口允许程序直接测试某个特性，而不是像前面的代码示例所显示的那样，从 get_info <info::platform::extensions> 遍历扩展列表。SYCL 2020 临时规范定义了新的机制来查询扩展和设备的详细信息，但本书中不涉及这些特性（它们刚刚定稿）。更多信息请参考在线 oneAPI DPC++ 语言手册。

设备信息描述符

“curious” 程序示例，使用了最常用的 SYCL 设备类成员函数（例如，is_host, is_cpu, is_gpu, is_accelerator, get_info, has_extension）。这些成员函数记录在 SYCL 规范的“SYCL 设备类的成员函数”的表中（在 SYCL 1.2.1 中，是表 4.18）。

“curious” 程序示例也使用 get_info 成员函数查询信息。所有 SYCL 设备（包括主机设备）都必须支持组询。这些项目的完整列表在 SYCL 规范中一个名为“设备信息描述符”的表中描述（在 SYCL 1.2.1 中，是表 4.20）。

特定于设备的内核信息描述符

像平台和设备一样，可以使用 get_info 函数查询关于内核的信息。这些信息（例如，支持的工作组大小、工作组大小、工作项所需的私有内存量）是特定于设备的，因此内核类的 get_info 成员函数接受一个设备作为参数。

SYCL 1.2.1 中的设备信息

SYCL 继承了 OpenCL 中 kernel::get_info 和 kernel::get_work_group_info 的查询组合，分别返回关于内核对象的信息和关于内核在特定设备上执行的信息。

在 DPC++ 和 SYCL 中使用重载（截至 2020 年临时）允许通过单个 get_info API 支持这两种类型的信息。

正确性

我们将把细节划分为关于必要条件（正确性）的信息，和对调优有用但对正确性不是必需的信息。

第一个正确性类别中，将列举内核正确启动所应满足的条件，不遵守这些设备限制将导致程序失败。图 12-7 显示了如何获取这些参数中的部分，这些值可以在主机代码和内核代码中使用（通过 Lambda 捕获）。可以修改代码来利用这些信息，例如：可以调整缓冲区大小或工作组大小。

提交一个不满足这些条件的内核将生成一个错误。

图 12-7 获取可用于形成内核的参数

```
1 std::cout << "We are running on:\n"
2     << dev.get_info<info::device::name>() << "\n";
3
4 // Query results like the following can be used to calculate how
5 // large our kernel invocations should be.
6 auto maxWG = dev.get_info<info::device::max_work_group_size>();
7 auto maxGmem = dev.get_info<info::device::global_mem_size>();
8 auto maxLmem = dev.get_info<info::device::local_mem_size>();
9
10 std::cout << "Max WG size is " << maxWG
11     << "\nMax Global memory size is " << maxGmem
12     << "\nMax Local memory size is " << maxLmem << "\n";
```

查询设备

device_type: cpu, gpu, accelerator, custom, automatic, host, all。最常使用的是 *is_host()*, *is_cpu()*, *is_gpu()* (见图 12-6):

max_work_item_sizes: *nd_range* 的工作组的每个维度中允许的最大工作项数。对于非定制设备，最小值为 (1,1,1)。

max_work_group_size: 单个计算单元上执行内核的工作组中允许的最大工作项数。最小值为 1。

global_mem_size: 全局内存大小 (以字节为单位)。

local_mem_size: 本地内存大小 (以字节为单位)。除定制设备外，最小为 32K。

extensions: 特定于设备的信息在 SYCL 规范中没有特别详细，通常是特定于供应商的，如 verycurious 程序所示 (图 12-6)。

max_compute_units: 说明设备实现定义的可用并行性的数量，请谨慎使用!

sub_group_sizes: 返回设备支持的子工作组大小。

usm_device_allocations: 如果该设备支持显式 USM 中描述的设备内存，则返回 true。

usm_host_allocations: 如果该设备可以访问主机内存，则返回 true。

usm_shared_allocations: 如果此设备支持共享内存，则返回 true。

usm_restricted_shared_allocations: 如果该设备支持受设备上“restricted USM”限制所控制的共享内存，则返回 true。此属性要求 *usm_shared_allocations* 对此设备返回 true。

usm_system_allocator: 如果系统分配器可能用于此设备上的共享内存，而不是 USM 分配机制，则返回 true.

我们建议避免在程序逻辑中使用最大计算单元！

部分原因是定义不够清晰，在代码调优中没有用处。大多数程序应该表达并行性，并让运行时将其映射为可用的并行性，而不是使用 `max_compute_units`。只有在使用特定于实现和设备的信息时，最大计算单元才有意义。专家可能会这样做，但大多数开发人员不这样做，也不需要这样做！在这种情况下，让运行时来完成这项工作！

查询内核

第 10 章“程序对象中的内核”中讨论的机制需要执行这些内核查询：

`work_group_size`: 返回可用于在特定设备上执行内核的最大工作组大小

`compile_work_group_size`: 返回由内核指定的工作组大小（如果适用），否则返回 (0,0,0)

`compile_sub_group_size`: 返回由内核指定的子工作组大小（如果适用），否则返回 0

`compile_num_sub_groups`: 返回由内核指定的子工作组的数量（如果适用），否则返回 0

`max_sub_group_size`: 返回以指定工作组大小启动的内核的最大子工作组大小

`max_num_sub_groups`: 返回内核的最大子工作组数

调优/优化

还有一些参数可以作为内核的微调参数。这些可以忽略，而不会影响程序的正确性。可以使我们的内核能够真正利用硬件的特性来提高性能。

在调优缓存（如果存在的话）时，这些查询的结果会有所帮助。

设备查询

`global_mem_cache_line_size`: 全局内存缓存行的大小（字节）。

`global_mem_cache_size`: 全局内存缓存大小（以字节为单位）。

`local_mem_type`: 支持的本地内存类型。可以是 `info::local_mem_type::local` 表示专用的本地内存存储，如 SRAM 或 `info::local_mem_type::global`。后一种类型意味着本地内存只是作为全局内存之上的抽象实现的，没有任何性能提高。本地内存类型也可以是 `info::local_mem_type::none`，表示不支持本地内存。

内核查询

`preferred_work_group_size`: 在特定设备上执行内核的首选工作组大小。

`preferred_work_group_size_multiple`: 在特定设备上执行内核的首选工作组大小。

运行时和编译时属性

本章描述的查询是通过运行时 API(`get_info`) 进行的，这意味着直到运行时才知道结果。这涵盖了许多方面，但是 SYCL 规范也正在进行修改，以提供属性的编译时查询（当可以被工具链感知时），以允许更高级的编程方式，如基于设备属性的内核模板。可以在运行时查询，但基于在代码编

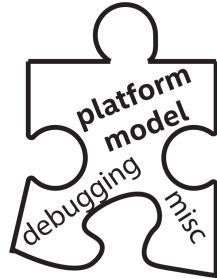
译时的查询是不可能的，这种能力对于高级优化或编写使用某些扩展的内核很重要。撰写本文时，这些接口的定义还不够好，本书不足以在中描述这些接口，但可以期待 SYCL 和 DPC++ 中很快会出现更强大的查询和代码适应机制！查看在线 oneAPI DPC++ 语言手册和 SYCL 规范以获得更新。

总结

可移植的程序将查询系统中可用的设备，并根据运行时信息调整行为。本章为我们打开了一扇通往信息的大门，这些信息允许我们对代码进行修缮，以适应运行时出现的硬件。

通过参数化，应用程序以适应硬件的特性，程序的移植性会更好，性能也更可移植，并且更具有前瞻性。还可以测试现有的硬件是否在程序设计的预想范围内，当发现硬件不在预想范围内时，要么发出警告，要么中止程序。

13 实践技巧



本章包含了许多有用的信息、实用的技巧、建议和技术，这些在编程 SYCL 和使用 DPC++ 时非常有用。这些主题都没挖掘到极致，所以我们的目的是提高人们的意识，鼓励人们根据需要学习更多的知识。

获取 DPC++ 编译器和代码示例

第 1 章介绍了如何获得 DPC++ 编译器 (oneapi.com/implementations 或 github.com/intel/llvm) 以及在哪里获得代码示例 (www.apress.com/9781484255735——查找本书的源代码)。可以尝试这些示例（包括进行修改）可以获得实际经验。

在线论坛和文档

Intel 开发者区举办了一个论坛，讨论 DPC++ 编译器，DPC++ 库（第 18 章），DPC++ 兼容性工具（CUDA 迁移——稍后将在讨论），以及 oneAPI 工具包中包含的 gdb（本章也涉及到调试）。这是一个很好的地方来发表关于编写代码的问题，包括编译器错误。你会在这个论坛上发现一些作者的帖子就是这样做的，尤其是在写这本书的时候。该论坛的网址是 <https://software.intel.com/en-US/forums/oneapidata-parallel-c-compiler>。

在线 oneAPI DPC++ 手册是一个很好的资源，可以找到类和成员定义的完整列表，编译器选项的详细信息等等。

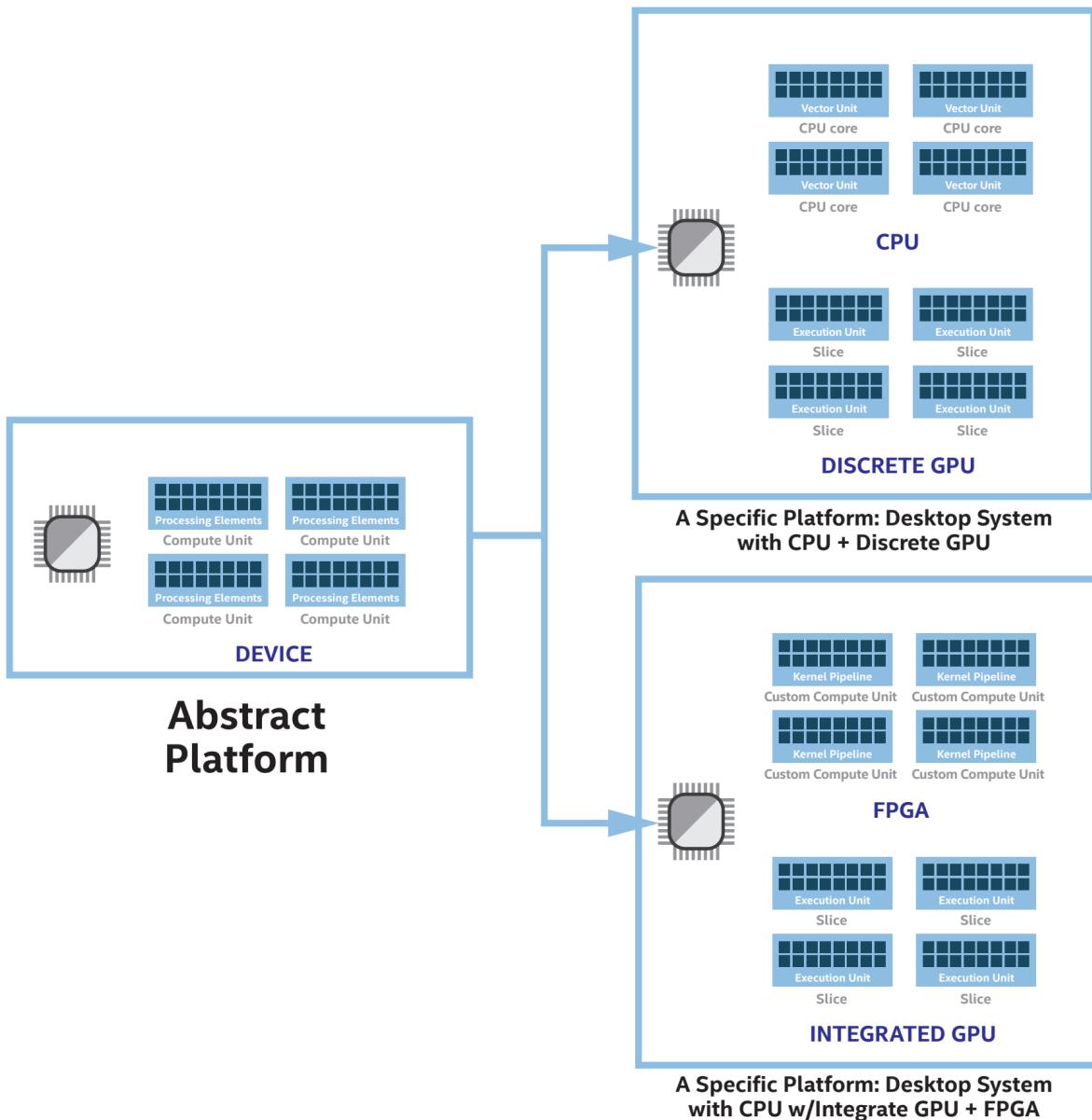
平台模型

SYCL 或 DPC++ 编译器的设计，与之前使用过的任何其他 C++ 编译器一样。不过，普通 C++ 编译器只为 CPU 生成代码。有必要在较高的层次上理解其内部工作原理，它使编译器能够为主机 CPU 和设备生成代码。

SYCL 和 DPC++ 所使用的平台模型（图 13-1）指定了主机来协调和控制在设备上执行的计算工作。第 2 章描述了如何将工作分配给设备，第 4 章深入研究了如何对设备进行编程。第 12 章描述了在不同层次上使用平台模型的特殊性。

正如第 2 章中所述，总有一个与主机相对应的设备，称为主机设备。为设备代码提供保证可用的目标，并允许在编写设备代码时，至少有一个设备可用，即使是主机本身！在哪个设备上运行设备代码是在程序控制下的——作为开发者，如果想在特定的设备上执行代码，以及如何执行代码，完全由自己选择。

图 13-1 平台模型: 可以抽象地使用, 也可以有针对性地使用



多体系结构二进制文件

由于我们的目标是使用单一源代码来支持异构机器, 因此很自然地希望结果是单个可执行文件。

一个多体系结构二进制文件 (又名宽二进制文件) 是一个单独的二进制文件, 已经扩展到包含异构机器所需的所有已编译的和中间代码。多架构二进制文件的概念并不新鲜。例如, 一些操作系统支持 32 位和 64 位的多架构库和可执行文件。一个多架构二进制文件的作用就像我们常用的任何 a.out 或 A.exe 一样, 但它包含异构机器所需的一切。这有助于自动的为特定设备选择正确的代码。正如下面要讨论的, 一种可能的设备代码形式是中间格式, 它将设备指令的最终创建推迟到运行时。

编译模型

SYCL 和 DPC++ 的单源特性允许编译操作像普通的 C++ 编译。不需要为设备使用额外的工具，也不需要处理设备和主机代码的绑定，这些都由编译器自动处理。当然，了解事情的细节很重要，原因有几个。如果希望更有效地针对特定的体系结构，这是非常有用的知识，并且了解是否需要调试编译过程中发生的故障也非常重要。

回顾编译模型，以便了解何时需要这些知识。由于编译模型支持同时在主机或多个设备上执行的代码，因此编译器、链接器和其他支持工具发出的命令比我们习惯的 C++ 编译（只针对一种体系结构）要复杂得多。

DPC++ 编译器向我们隐藏了这种异构的复杂性。DPC++ 编译器可以生成与传统 C++ 编译器类似的可执行代码（提前（AOT）编译，有时称为脱机编译），或者可以生成中间表示，可以在运行时将其即时（JIT）编译为特定的目标。

只有在设备目标提前已知（编译程序时）时，编译器才能提前编译。延迟即时编译提供了更多的灵活性，但需要编译器和运行时在程序运行时执行额外的工作。

DPC++ 编译可以是“提前编译”或“即时编译”。

默认情况下，当为大多数设备编译代码时，设备代码的输出以中间形式存储。在运行时，系统上的设备处理程序将实时地将中间形式编译为在设备上运行的代码，以匹配系统上可用的内容。

我们可以要求编译器为特定的设备或设备类进行提前编译。这样做的优点是节省运行时间，但缺点是增加了编译时间和更宽的二进制文件！提前编译的代码不如即时编译的代码可移植性好，因为不能在运行时进行调整。我们可以将两者包含在二进制文件中，以获得两者的优点的叠加。

提前编译特定设备还有助于在构建时，对设备进行检查。使用即时编译，程序可能会在运行时编译失败（这可以通过第 5 章中的机制捕捉到）。第 5 章详细介绍了如何在运行时捕获这些错误，以避免终止程序。

图 13-2 说明了 DPC++ 从源代码到宽二进制文件（可执行文件）的编译过程。无论选择什么组合，都会组合成一个宽二进制文件。宽二进制文件在应用程序执行时由运行时使用（在主机上执行的就是这个二进制文件！）有时，可能希望在单独的编译中编译特定设备的设备代码。希望这样一个单独编译的结果最终组合到宽二进制文件中。完全编译的耗时很长，这对于 FPGA 开发非常有用，而且实际上 FPGA 开发需要避免在运行时系统上安装合成工具。图 13-3 展示了此类需求所支持的绑定/解绑定的流程。总是可以一次编译所有内容，但是在开发过程中，拆分编译的选项会非常有用。

每个 SYCL 和 DPC++ 编译器都有一个相同目标的编译模型，但是确切的实现细节会有所不同。这里显示的图表是 DPC++ 编译器工具链。

特定于 DPC++ 的组件如图 13-2 所示，作为集成头文件生成器，本书中将不再提及。我们不需要知道它是什么，或者能做什么就可以编程。然而，这里有一些信息需要知道：集成头文件生成器会生成一个头文件，提供关于在转译单元中找到的 SYCL 内核的信息。这包括 SYCL 内核类型名称如何映射到符号名称，以及关于内核参数及其在相应 Lambda 或 functor 对象中的位置的信息，这些信息是由编译器创建。集成头文件是一种机制，用于通过 C++ Lambda/functor 对象在主机代码上创建内核，可以将我们从设置参数、按名称解析内核等耗时的任务中解放出来。

图 13-2 编译过程: 提前和即时

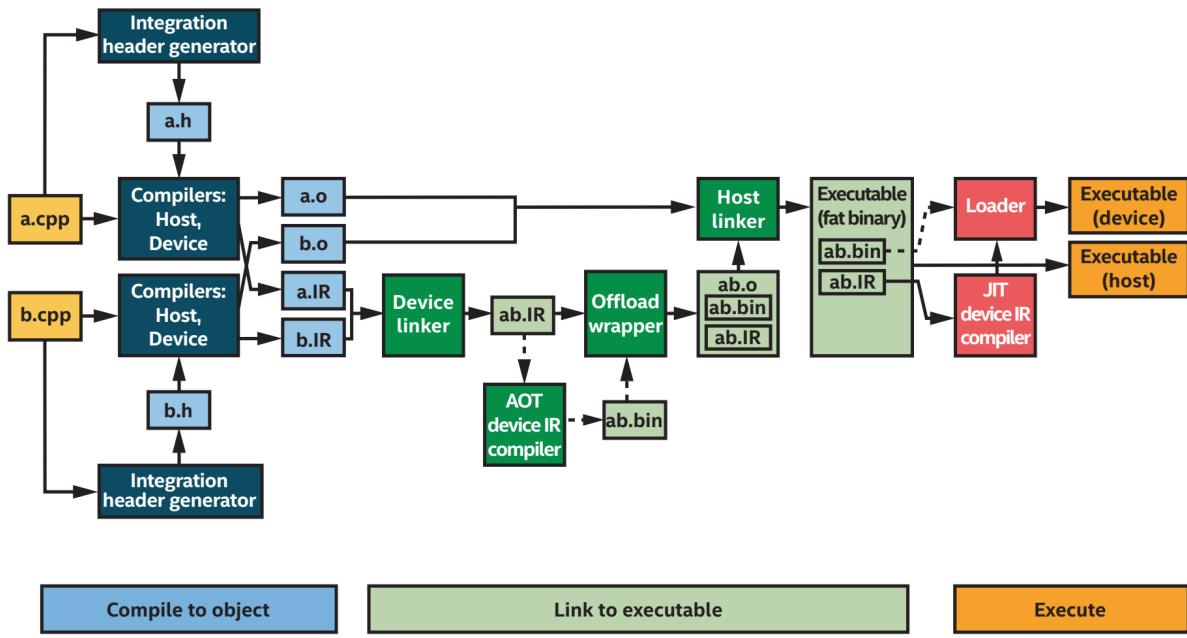
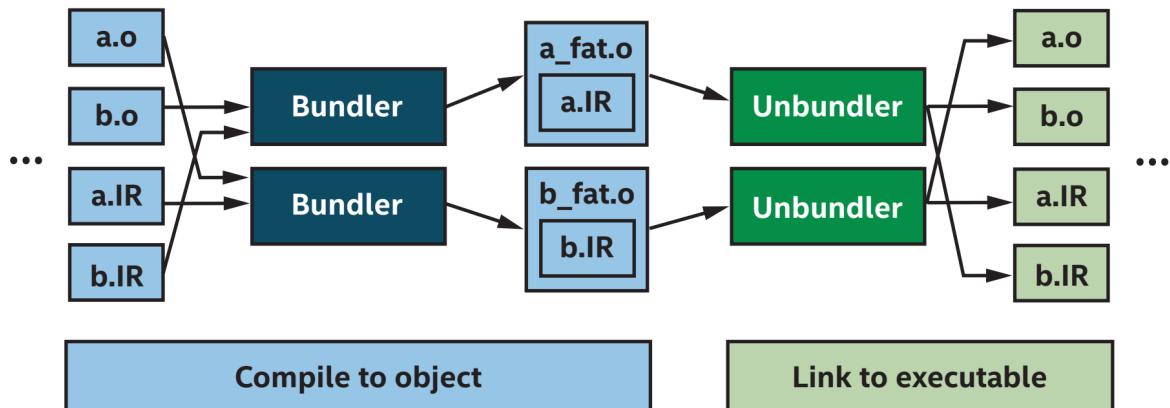


图 13-3 编译过程: 组合打包/分解打包



将 SYCL 添加到现有 C++ 代码中

向 C++ 程序中添加适当的并行性开发是使用 SYCL 的第一步。如果应用程序已经并行执行，在带来性能收益时，也可能带来令人头痛的问题。因为应用程序的工作方式为并行，极大地影响了可以用来做什么。当开发者谈到重构时，指的是重新安排程序内的执行流和数据流，以便为利用并行性做好准备。这是一个复杂的话题，我们只会简单地谈一谈。关于如何将程序并行化没有一个通用的答案，但是有一些技巧值得注意。

向 C++ 应用程序添加并行性时，一种简单的方法是在程序中找到可并行性最大的点。可以从那里开始修改，然后根据需要继续在其他领域添加并行性。复杂的因素是，重构（例如，重新安排程序流和重新设计数据结构）可能提高并行性的机会。

当程序找到并行性机会最大的点，就需要考虑如何在程序中使用 SYCL。

高层次上，引入并行性的关键步骤包括：

1. 并发性的安全性 (传统 CPU 编程中通常称为线程安全性): 将所有共享的可变数据 (可以更改和并发共享的数据) 调整为并发使用
2. 引入并发和/或并行
3. 并行性调优 (最佳扩展、吞吐量或延迟优化)

许多应用程序已经为并发性进行了重构，将 SYCL 作为并行性的来源，关注内核中使用的数据以及可能与主机共享的数据的安全性。如果程序中有其他并行性技术 (OpenMP、MPI、TBB 等) 的使用，这是 SYCL 编程之上的另一个关注点。需要注意的是，单个程序中使用多种技术是可以的——SYCL 不是程序中并行性的唯一来源。本书不涉及与其他并行技术混合的高级主题。

调试

本节提供一些调试建议，以减少调试并行程序 (特别是针对异构机器的并行程序) 的挑战。

当应用程序在主机设备上运行时，可以选择调试，这个调试技巧在第 2 章中描述为方法 2。因为设备通常很少包含调试方式，所以调试可以在主机上的完成。主机上运行的另一个好处是，与同步相关的许多错误将消失，包括在主机和设备之间移动内存导致的问题。虽然最终需要调试所有错误，但这可以允许增量调试，可以先解决一些错误。

调试技巧 功能强大的调试工具可以运行在主机上。

在主机上运行所有代码时，工具通常更容易检测和消除并行编程错误，特别是数据竞争和死锁。当在主机和设备的组合上运行时，经常会看到这种并行编程的错误导致程序运行失败。出现这样的问题时，记住使用 host-only 的方式更有助于调试。SYCL 和 DPC++ 可以让我们使用这个选项，访问相关的内存。

调试技巧 如果程序出现死锁，请检查主机访问器是否正确销毁。

开始调试时，可以使用下面的 DPC++ 编译器选项：

- -g: 在输出中输入调试信息。
- -ferror-limit=1: 当使用带有 SYCL/DPC++ 等 C++ 模板库时，可以保持完整性。
- -Werror -Wall -Wpedantic: 让编译器强制执行编码，以避免生成运行时可调试的错误代码。

不需要仅仅为了使用 DPC++ 而去修复迂腐的警告，所以可以选择不使用-Wpedantic。

在运行时让代码及时编译时，我们可以检查代码。这高度依赖于编译器，因此查看编译器手册以获得建议是一个好主意。

调试内核代码

调试内核代码时，首先要在主机设备上运行 (如第 2 章所述)。第 2 章中的设备选择器，可以很容易地修改运行时选项或编译时选项，以便在调试时将工作重定向到主机。

调试内核代码时，SYCL 定义了一个 C++ 风格的输出流，可以在内核中使用 (图 13-4)。DPC++ 还提供了一个 C 风格 printf 的实验性实现，功能有一些限制。更多的细节可以参考在线 oneAPI

DPC++ 手册。

图 13-4 `sycl::stream`

```
1 Q.submit ([&] ( handler &h) {  
2     stream out(1024, 256, h);  
3     h.parallel_for (range{8}, [=] (id<1> idx) {  
4         out << "Testing my sycl stream (this is work-item ID:" << idx << ")\n";  
5     });  
6 });
```

调试内核代码时，可以在 `parallel_for` 之前或在 `parallel_for` 内部设置断点。即使是在执行下一个操作之后，放置在 `parallel_for` 的断点也可以多次触发。C++ 调试方式适用于许多模板展开，其中模板调用上的断点将在编译器展开时，转换为一组复杂的断点。可能有方法可以缓解这种情况，但这里的关键是，可以通过在 `parallel_for` 上设置断点来避免混淆。

调试运行时的错误

发生运行时错误时，要么是在处理编译器/运行时错误，要么是无意中编写了废代码。深入研究这些 bug 可能有点令人生畏，但可以让更好地了解导致特定问题的原因。可能会产生一些额外的信息，指导我们避免这个问题，或者向编译器团队提交错误报告。不管怎样，了解一些能够提供帮助的工具很重要。

运行时失败的程序输出如下所示：

```
origin>: error: Invalid record (Producer: 'LLVM9.0.0' Reader: 'LLVM 9.0.0')  
terminate called after throwing an instance of 'cl::sycl::compile_program_error'
```

看到这里的 `throw`，可以知道主程序可以捕捉这个错误。虽然这不能解决问题，但确实意味着运行时编译器失败不需要中止应用程序。第 5 章会深入探讨这个主题。

当看到运行时失败并难以快速调试时，不妨尝试使用提前编译进行重新构建。如果所使用的设备具有提前编译选项，这可能是很容易的事情，可能会产生更容易理解的诊断信息。如果错误是在编译时，而不是在 JIT 或运行时，通常会在编译器的错误消息中找到有用的信息，而不是在 JIT 或运行时看到的少量错误信息。对于特定选项，请查看在线 oneAPI DPC++ 手册，了解提前编译。

SYCL 程序运行在 OpenCL 运行时上，并使用 OpenCL 后端时，可以使用 OpenCL 拦截：github.com/intel/opencl-intercept-layer 运行程序。可以检查、记录和修改应用程序（或更高级别运行时）生成 OpenCL 命令的工具。它支持很多控件，但是最好的初始设置是 ErrorLogging、BuildLogging 和 CallLogging（会生成很多输出）。DumpProgramSPIRV 可以提供有用的转储。OpenCL 拦截是一个独立的程序，不是 OpenCL 实现的一部分，所以可以与 SYCL 编译器一起工作。

对于 Intel GPU 在 Linux 系统上的编译器问题，可以从 Intel 图形编译器转储中间编译器输出。通过将环境变量

`IGC_ShaderDumpEnable` 设置为 1（用于某些输出）或将环境变量 `IGC_ShaderDumpEnableAll` 设置为 1（用于大量输出）来做到这一点，转储的输出在 `/tmp/IntelIGC` 目录下。这种技术可能不适用于所有图形驱动程序的构建，但值得一试。

图 13-5 列出了这些变量以及编译器或运行时支持的一些附加环境变量 (在 Windows 和 Linux 上支持), 以帮助进行高级调试。这些是 DPC++ 实现相关的高级调试选项, 用于检查和控制编译模型。

这些选项在本书中没有详细描述, 这里提到是为了在需要时打开高级调试的通道。这些选项可以让我们深入了解如何解决问题。使用这些选项是为了对编译器本身进行调试。因此, 选项更多地与编译器开发人员联系在一起, 而不是编译器用户。一些高级用户认为这些选项很有用, 因此在这里提及。更多信息可以参考 GitHub for DPC++ 在 [llvm/sycl/doc/EnvironmentVariables.md](#) 下关于所有环境变量的文档。

调试技巧当其他选项都用尽, 且无法解决需要调试运行时问题时, 可能寻找为提供相关原因的工具。

图 13-5 DPC++ 的高级调试选项

环境变量	值	描述
SYCL_PI_TRACE	1(basic), 2(advanced), -1(all)	运行时: 值为 1 启用运行时插件接口 (PI) 跟踪插件，并寻找设备； 值为 2 表示跟踪所有 PI 呼叫； 值为-1 释放所有级别的跟踪。
SYCL_PRINT_EXECUTION_GRAPH	always(或者通过指定 只转储选定文件; before_addcg、 after_addgc、 before_adddcopyback、 after_adddcopyback、 before_addhostacc、 或 after_addhostacc)	运行时: 创建跟踪执行图的文本文件 (DOT 扩展名)，这样浏览运行时 发生的跟踪会更容易。
CL_CONFIG_USE_VBVECTORIZER	true 或 false	运行时: 请求 CPU 编译器启用 或禁用向量。
CL_CONFIG_CPU_TARGET_ARCH	skx, core-avx2	运行时: 要求 Intel CPU 编译器支持 Inter Advanced Vector Extensions (AVX512 和 AVX2) 的代码。
CL_CONFIG_DUMP_ASM	true 或 false	运行时: 通过英特尔 CPU 编译器转 储出 CPU 汇编代码。
IGC_ShaderDumpEnable	0 或 1	只限定于 Linux。 运行时: 通过 Intel 图形编译器 (JIT) 转储一些信息。
IGC_ShaderDumpEnableALL	0 或 1	只限定于 Linux。 运行时: 通过 Intel 图形编译器 (JIT) 转储大量信息。
SYCL_DUMP_IMAGES	true 或 false	编译时: 通过 SYCL_DUMP_IMAGES=1 请求编译器转储出 SPV 文件， 其中包含在执行期间传递给 JIT 编译器的中间码。
SYCL_USE_KERNEL_SPV	<device binary>	运行时: 从指定文件加载设备镜像。 如果运行时无法读取该文件，则 抛出 cl::sycl::runtime_error 异常。

初始化数据并访问内核输出

本节中，我们将深入讨论一个主题，其会给 SYCL 的新用户带来困惑，以及新 SYCL 开发人员会遇到（以我们的经验）的第一个 bug。

简单地说，当从主机内存分配（例如，数组或向量）创建缓冲区时，主机不能直接访问。缓冲区在整个生命周期内，缓冲区拥有在构造时传入的主机内存。很少有机制允许在缓冲区还存在时访问主机内存（例如，缓冲区互斥），但是这些高级特性对调试早期 bug 没有帮助。

如果使用主机内存构造缓冲区，不能直接访问主机内存，直到缓冲区销毁！当缓冲区处于活动状态时，其会占有相应内存。

当主机程序访问主机内存，而缓冲区仍然拥有该内存时，会出现一个错误，因为不知道缓冲区的类型。如果数据是错误的，不要感到惊讶。如第 3 章和第 8 章所述，SYCL 是围绕异步任务图构建的。尝试使用任务图操作输出数据之前，需要确保已经到达了同步点，并使数据对主机可用。缓冲区销毁和主机访问器的创建都会触发同步。

图 13-6 展示了代码的模式，通过关闭定义缓冲区的作用域来销毁缓冲区。通过销毁缓冲区，可以给缓冲区构造函数的主机内存，从而可以安全地读取内核结果。

图 13-6 从主机内存创建通用模式的缓冲区

```
1 constexpr size_t N = 1024;
2
3 // Set up queue on any available device
4 queue q;
5
6 // Create host containers to initialize on the host
7 std::vector<int> in_vec(N), out_vec(N);
8
9 // Initialize input and output vectors
10 for (int i=0; i < N; i++) in_vec[i] = i;
11 std::fill(out_vec.begin(), out_vec.end(), 0);
12
13 // Nuance: Create new scope so that we can easily cause
14 // buffers to go out of scope and be destroyed
15 {
16
17 // Create buffers using host allocations (vector in this case)
18 buffer in_buf{in_vec}, out_buf{out_vec};
19
20 // Submit the kernel to the queue
21 q.submit([&](handler& h) {
22     accessor in{in_buf, h};
23     accessor out{out_buf, h};
24
25     h.parallel_for(range{N}, [=](id<1> idx) {
26         out[idx] = in[idx];
27     });
28 });
29
30 // Wait for the queue to finish
31 q.wait();
32
33 // Read back the results
34 std::vector<int> result(N);
35 q.read(out_buf, result);
36
37 // Check the results
38 for (int i=0; i < N; i++) {
39     if (result[i] != i) {
40         std::cout << "Error: result[" << i << "] = " << result[i] << std::endl;
41         return -1;
42     }
43 }
44
45 // Clean up
46 q.destroy();
47
48 // Print the results
49 std::cout << "Result: ";
50 for (int i=0; i < N; i++) {
51     std::cout << result[i] << " ";
52 }
53 std::cout << std::endl;
```

```

27     });
28 }
29
30 // Close the scope that buffer is alive within! Causes
31 // buffer destruction which will wait until the kernels
32 // writing to buffers have completed, and will copy the
33 // data from written buffers back to host allocations (our
34 // std::vectors in this case). After the buffer destructor
35 // runs, caused by this closing of scope, then it is safe
36 // to access the original in_vec and out_vec again!
37 }
38
39 // Check that all outputs match expected value
40 // WARNING: The buffer destructor must have run for us to safely
41 // use in_vec and out_vec again in our host code. While the buffer
42 // is alive it owns those allocations, and they are not safe for us
43 // to use! At the least they will contain values that are not up to
44 // date. This code is safe and correct because the closing of scope
45 // above has caused the buffer to be destroyed before this point
46 // where we use the vectors again.
47 for (int i=0; i<N; i++)
48     std::cout << "out_vec[" << i << "]=" << out_vec[i] << "\n";

```

将缓冲区与现有主机内存关联有两个常见的原因，如图 13-6 所示：

1. 简化缓冲区的初始化。可以从已经初始化的主机内存（或应用程序的其他部分）中构造缓冲区。
2. 减少输入的字符，因为用“}”关闭作用域比创建缓冲区的 host_accessor 更简洁（更容易出错）。

如果使用主机内存转储或验证内核的输出值，需要将缓冲区分配放入作用域（或其他作用域）中，这样就可以控制缓冲区的销毁了。然后，必须确保访问主机内存，在获得内核输出之前销毁缓冲区。图 13-6 展示了这方面的正确实现，而图 13-7 展示了一个错误实现，即仍在缓冲区活动时访问输出。

高级用户可能更喜欢使用缓冲区销毁，将结果数据从内核返回到主机内存分配内存中。对于大多数用户，特别是新手开发者，建议使用作用域内的主机访问器。

图 13-7 常见错误：缓冲区生命周期内直接从主机分配内存中读取数据

```

1 constexpr size_t N = 1024;
2
3 // Set up queue on any available device
4 queue q;
5
6 // Create host containers to initialize on the host
7 std::vector<int> in_vec(N), out_vec(N);

```

```

8
9 // Initialize input and output vectors
10 for (int i=0; i < N; i++) in_vec[i] = i;
11 std::fill(out_vec.begin(), out_vec.end(), 0);
12
13 // Create buffers using host allocations (vector in this case)
14 buffer in_buf{in_vec}, out_buf{out_vec};
15
16 // Submit the kernel to the queue
17 q.submit([&](handler& h) {
18     accessor in{in_buf, h};
19     accessor out{out_buf, h};
20
21     h.parallel_for(range{N}, [=](id<1> idx) {
22         out[idx] = in[idx];
23     });
24 });
25
26 // BUG!!! We're using the host allocation out_vec, but the buffer out_buf
27 // is still alive and owns that allocation! We will probably see the
28 // initialization value (zeros) printed out, since the kernel probably
29 // hasn't even run yet, and the buffer has no reason to have copied
30 // any output back to the host even if the kernel has run.
31 for (int i=0; i < N; i++)
32     std::cout << "out_vec[" << i << "]=" << out_vec[i] << "\n";

```

请使用主机访问器，而不是直接访问内存，特别是在开始时。

为了避免这些错误，建议在开始使用 SYCL 和 DPC++ 时使用主机访问器，而不是直接访问内存。主机访问器提供了对来自主机的缓冲区的访问。当构造完成时，就可以保证之前对缓冲区的任何写入（例如，创建 host_accessor 之前提交的内核）都已经执行并且可见。本书混合使用了这两种风格（即，主机访问器和传递给缓冲区构造函数的主机分配），以便熟悉这两种风格。开始时，使用主机访问器不容易出错。图 13-8 展示了如何使用主机访问器从内核读取输出，而不破坏缓冲区。

图 13-8 建议：使用主机访问器读取内核结果

```

1 constexpr size_t N = 1024;
2
3 // Set up queue on any available device
4 queue q;
5
6 // Create host containers to initialize on the host
7 std::vector<int> in_vec(N), out_vec(N);
8
9 // Initialize input and output vectors
10 for (int i=0; i < N; i++) in_vec[i] = i;
11 std::fill(out_vec.begin(), out_vec.end(), 0)

```

```

12 );
13 // Create buffers using host allocations (vector in this case)
14 buffer<int> in_buf{in_vec}, out_buf{out_vec};
15
16 // Submit the kernel to the queue
17 q.submit([&](handler& h) {
18     accessor in{in_buf, h};
19     accessor out{out_buf, h};
20
21     h.parallel_for(range{N}, [=](id<1> idx) {
22         out[idx] = in[idx];
23     });
24 });
25
26 // Check that all outputs match expected value
27 // Use host accessor! Buffer is still in scope / alive
28 host_accessor A{out_buf};
29
30 for (int i=0; i<N; i++) std::cout << "A[" << i << "]=" << A[i] << "\n";

```

只要缓冲区没销毁，就可以使用主机访问器，例如：在缓冲区生命周期的两端，用于初始化缓冲区内容和从内核读取结果，如图 13-9 所示。

图 13-9 建议：使用主机访问器进行缓冲区初始化和读取结果

```

1 constexpr size_t N = 1024;
2
3 // Set up queue on any available device
4 queue q;
5
6 // Create buffers of size N
7 buffer<int> in_buf{N}, out_buf{N};
8
9 // Use host accessors to initialize the data
10 { // CRITICAL: Begin scope for host_accessor lifetime!
11     host_accessor in_acc{ in_buf }, out_acc{ out_buf };
12     for (int i=0; i < N; i++) {
13         in_acc[i] = i;
14         out_acc[i] = 0;
15     }
16 } // CRITICAL: Close scope to make host accessors go out of scope!
17
18 // Submit the kernel to the queue
19 q.submit([&](handler& h) {
20     accessor in{in_buf, h};
21     accessor out{out_buf, h};
22
23     h.parallel_for(range{N}, [=](id<1> idx) {
24         out[idx] = in[idx];

```

```

25 } );
26 });
27
28 // Check that all outputs match expected value
29 // Use host accessor! Buffer is still in scope / alive
30 host_accessor A{out_buf};
31
32 for (int i=0; i<N; i++) std::cout << "A[ " << i << "]=" << A[i] << "\n";

```

最后要提到的是，主机访问器有时会在应用程序中引发错误，因为它们也有生命周期。当缓冲区的 host_accessor 在生命周期中，运行时将不允许任何设备使用该缓冲区！运行时不会分析主程序来确定什么时候可以访问主机访问器，所以主机程序访问缓冲区的唯一方法是运行 host_accessor 析构函数。如图 13-10 所示，如果主程序正在等待内核运行（例如，queue::wait() 或获取另一个主机访问器），如果 DPC++ 运行时正在等待之前的主机访问器销毁，在销毁之后才能操作使用缓冲区的内核，这可能会导致应用程序挂起等待。

使用主机访问器时，请确保内核或其他主机访问器不再需要解锁缓冲区时销毁它们。

图 13-10 host_accessors 使用不当是会挂起程序

```

1 constexpr size_t N = 1024;
2
3 // Set up queue on any available device
4 queue q;
5
6 // Create buffers using host allocations (vector in this case)
7 buffer<int> in_buf{N}, out_buf{N};
8
9 // Use host accessors to initialize the data
10 host_accessor in_acc{ in_buf }, out_acc{ out_buf };
11 for (int i=0; i < N; i++) {
12     in_acc[i] = i;
13     out_acc[i] = 0;
14 }
15
16 // BUG: Host accessors in_acc and out_acc are still alive!
17 // Later q.submit will never start on a device, because the
18 // runtime doesn't know that we've finished accessing the
19 // buffers via the host accessors. The device kernels
20 // can't launch until the host finishes updating the buffers,
21 // since the host gained access first (before the queue submissions).
22 // This program will appear to hang! Use a debugger in that case.
23
24 // Submit the kernel to the queue
25 q.submit([&](handler& h) {
26     accessor in{in_buf, h};
27     accessor out{out_buf, h};

```

```

28
29     h.parallel_for(range{N}, [=](id<1> idx) {
30         out[idx] = in[idx];
31     });
32 });
33
34 std::cout <<
35     "This program will deadlock here!!! Our host_accessors used\n"
36     "for data initialization are still in scope, so the runtime won't\n"
37     "allow our kernel to start executing on the device (the host could\n"
38     "still be initializing the data that is used by the kernel). "
39     "The next line\n of code is acquiring a host accessor for"
40     "the output, which will wait for\n the kernel to run first. "
41     "Since in_acc and out_acc have not been\n"
42     "destructed, the kernel is not safe for the runtime to run, "
43     "and we deadlock.\n";
44
45 // Check that all outputs match expected value
46 // Use host accessor! Buffer is still in scope / alive
47 host_accessor A{out_buf};
48
49 for (int i=0; i<N; i++) std::cout << "A[" << i << "]=" << A[i] << "\n";

```

多个转译单元

要调用在不同的平移单元中定义的内核内部函数时，这些函数需要用 SYCL_EXTERNAL 标记。如果没有此属性，编译器将只编译用于设备代码外部的函数（从设备代码内部调用外部函数是非法的）。

如果在同一个转换单元中定义 SYCL_EXTERNAL 函数，则有一些限制：

- SYCL_EXTERNAL 只能用于函数。
- SYCL_EXTERNAL 函数不能使用原始指针作为形参或返回类型。必须使用显式指针类。
- SYCL_EXTERNAL 函数不能调用 parallel_for_work_item。
- SYCL_EXTERNAL 函数不能在 parallel_for_work_group 范围内调用。

如果一个内核调用的函数不在同一个转译单元中，并且没有使用 SYCL_EXTERNAL 声明，那么估计会出现如下错误

```
error: SYCL kernel cannot call an undefined function without SYCL_EXTERNAL attribute
```

如果函数本身在编译时没有 SYCL_EXTERNAL 属性，则可以看到链接或运行时失败，例如

```
terminate called after throwing an instance of 'cl::sycl::compile_program_error' ...error: undefined reference to ...
```

DPC++ 支持 SYCL_EXTERNAL。SYCL 不需要编译器支持 SYCL_EXTERNAL，这是一个可选的特性。

多重转译单元的性能影响

编译模型的含义（见本章前面的部分）是，如果将设备代码分布到多个转译单元中，那么与设备代码位于同一位置时相比，可能会触发更多的即时编译。这是高度依赖于实现，并且会随着实现发生变化。

这种方式对性能的影响很小，在大多都可以忽略，但当进行微调以最大化代码性能时，可以考虑两件事来减轻影响：(1) 将设备代码组合在同一个单元中，(2) 使用提前编译来完全避免即时编译效应。由于这两种方法都需要一定的工作量，所以只有在完成开发，并试图从应用程序中榨出每一分性能时才会这样做。当使用这种调优时，有必要进行测试，以观察对 SYCL 实现的影响。

需要为匿名的 Lambda 函数命名

SYCL 提供了为 Lambda 定义的名称赋值，以备工具需要时使用，并用于调试目的（例如，根据用户定义的名称启用显示）。本书的大部分内容中，匿名 Lambda 用作内核函数，因为在使用 DPC++ 时不需要名称（除非传递编译选项，如第 10 章中讨论的 Lambda 命名所述）。从 SYCL 2020 开始，名称是可选的。

当需要在代码库中混合来自多个供应商的 SYCL 工具时，该工具可能要求命名 Lambda 函数。这可以通过在使用 Lambda 的 SYCL 动作构造中添加 <class uniquename> 来实现（例如，parallel_for）。这种命名允许供应商的工具在编译中，以定义的方式进行交互，还可以在调试工具和层中定义中显示内核名称。

将 CUDA 迁移到 SYCL

迁移 CUDA 代码到 SYCL 或 DPC++ 在本书中没有详细介绍。有一些可用的工具和资源可以做到这一点。迁移 CUDA 代码相对简单，因为是一种基于内核的并行化方法。当用 SYCL 或 DPC++ 编写时，目标设备比单独支持 CUDA 增强了很多。针对 NVIDIA GPU，新程序仍然可以使用支持 NVIDIA GPU 的 SYCL 编译器。

迁移到 SYCL 为 SYCL 支持的设备的多样性打开了大门，这就不仅限于 GPU 的范畴了。

当使用 DPC++ 兼容性工具时，-report-type=value 选项提供了关于迁移代码的非常有用的数据信息。本书的一位评论家称它为“Intel dpct 提供的优美标识”。-in-root 选项在迁移 CUDA 代码时非常有用。

要了解更多关于 CUDA 迁移的信息，可以从以下两方面开始：

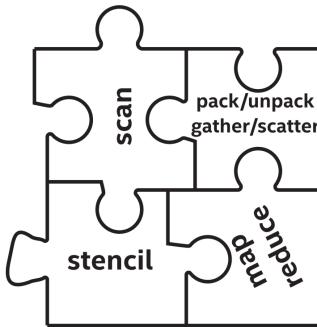
- 英特尔的 DPC++ 兼容工具将 CUDA 应用程序转换为 DPC++ 代码 (tinyurl.com/CUDAToDPCpp)。
- Codeplay 教程“从 CUDA 迁移到 SYCL” (tinyurl.com/codeplayCUDAToSYCL)。

总结

通常把“小窍门”称为“生活技巧”。不幸的是，编程文化经常赋予黑客一种负面含义，所以作者没有将这一章命名为“SYCL Hacks”。毫无疑问，本章只是触及了使用 SYCL 和 DPC++ 的

实用技巧的皮毛。可以在在线论坛上分享更多的技巧，一起学习如何使用 DPC++，从而充分利用 SYCL。

14 常见的并行模式



当我们在工作中认识到模式，并且证明了其是最佳解决方案的技术。并行编程也不例外，需要研究已有的模式。考虑大数据应用中的 MapReduce 框架，他们的成功原因是基于两个简单而有效的并行模式——map 和 reduce。

并行编程中有很多常见的模式，与编程语言无关。这些模式是通用的，可以进行任何级别的并行（例如：子工作组、工作组、设备）和在任何设备（例如：CPU, GPU, FPGA）上使用。但是，模式的某些属性（例如可扩展性）可能会影响其不同的设备。某些情况下，使应用程序适应新设备可能只需要选择适当的参数或微调模式的实现，我们可以通过选择完全不同的模式来提高性能。

理解如何、何时以及在何处使用这些常见的并行模式，是提高对 DPC++（以及一般的并行编程）熟练程度的关键。对于那些具有并行编程经验的人来说，了解这些模式是如何在 DPC++ 中表达的，可以快速了解该语言的功能。

本章旨在解答以下问题：

- 理解哪些模式是最重要的？
- 模式如何与不同设备的功能相关联？
- 哪些模式已经作为 DPC++ 函数和库提供？
- 如何使用直接实现模式？

了解模式

这里讨论的模式是 McCool 等人在《结构化并行编程》一书中描述并行模式的子集。我们不讨论与并行性类型相关的模式（例如 fork-join、分支绑定），而是关注对编写数据并行内核最有用的算法模式。

理解并行模式的这个子集对于成为一个高效的 DPC++ 程序员至关重要。图 14-1 中的表给出了不同模式的概述，包括主要用例、关键属性，以及属性如何影响不同硬件设备的亲和性。

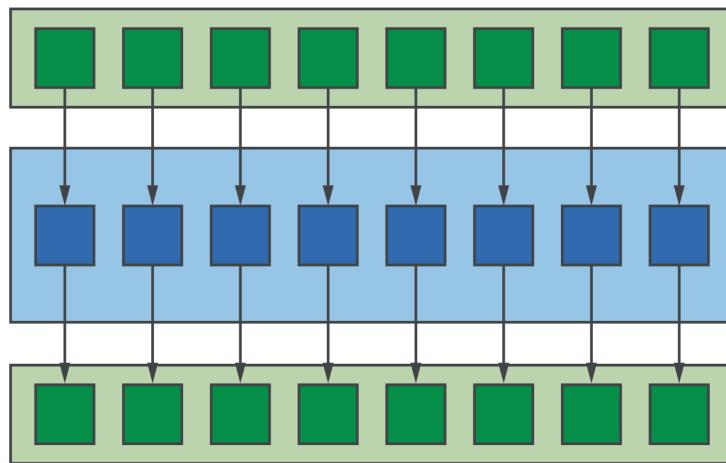
图 14-1 并行模式及其对不同设备类型的亲和性

模式	用于	关键属性	设备亲和度
Map	简单的并行内核	无数据依赖性和高可扩展性	所有设备
Stencil	结构化数据依赖性	数据依赖和数据重用	取决于模具的大小
Reduction	合并部分结果	数据依赖	所有设备
Scan Pack/Unpack	筛选和重组数据	有限的可扩展性	取决于问题的大小

Map——映射

映射模式是所有并行模式中最简单的，具有函数式编程语言经验的读者很快就会熟悉它。如图 14-2 所示，范围内的每个输入元素通过应用某个函数独立映射到一个输出。许多数据并行操作可以表示为映射模式（例如，向量加法）。

图 14-2 映射模式



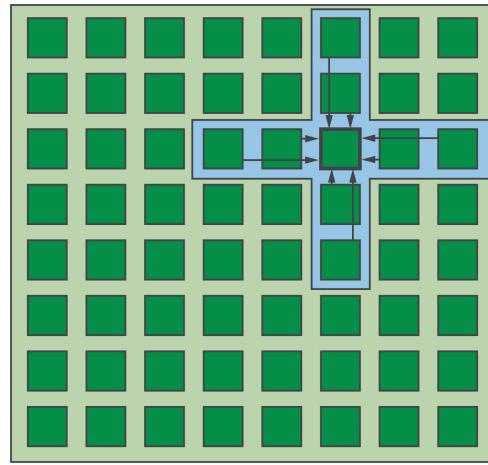
由于每个应用程序的函数都完全独立，所以映射的表达式通常非常简单，依赖于编译器和/或运行时来做大部分工作。写入映射模式的内核应该能够适用于任何设备，并且这些内核的性能能根据可用的硬件并行度进行很好地扩展。

然而，在决定将整个应用程序重写为一系列映射内核之前，应该仔细考虑！这样的开发方法是高效的，保证了应用程序可以移植到各种设备类型，但却忽略那些可能显著提高性能的优化（例如，数据重用，融合内核）。

Stencil——模具

模具模式与映射模式密切相关。如图 14-3 所示，一个函数应用到一个输入和一组由模板描述的邻近输入，以产生单个输出。模板模式经常出现在许多领域，包括科学/工程应用（如有限差分）和计算机视觉/机器学习应用（如图像卷积）。

图 14-3 模具模式



当模具模式在不同的地方执行时 (例如, 将输出写入一个单独的存储位置), 该函数可以独立地应用于每个输入。现实中调度模板通常比这更复杂: 计算相邻的输出需要相同的数据, 并且多次从内存中加载该数据会降低性能; 我们可能希望就地应用模板 (即, 重写原始输入值), 以减少应用程序的内存占用。

因此, 模板内核对不同设备的适用性高度依赖于模具的属性和输入。以下有一些经验法则:

- 小模具可以利用 GPU 的便笺存储器。
- 大型模具可以利用 CPU(相对而言) 的缓存。
- 通过在 FPGA 上实现收缩阵列, 在小输入上操作的小模板可以获得显著的性能增益。

模具很容易描述, 但要有效地实现却很复杂, 所以模具是领域特定语言 (DSL) 开发中最活跃的领域之一。已经有一些嵌入式的 DSL 利用 C++ 的模板元编程能力在编译时生成高性能的模具内核, 我们希望这些移植到 DPC++ 上只是时间问题。

Reduction——归约

归约模式是一种常见的并行模式, 使用典型的关联和交换操作符 (例如加法) 组合内核调用的每个实例的部分结果。例子是计算一个和 (例如, 计算一个点积) 或计算最小/最大值 (例如, 使用最大速度来设置时间步长)。

图 14-4 归约模式

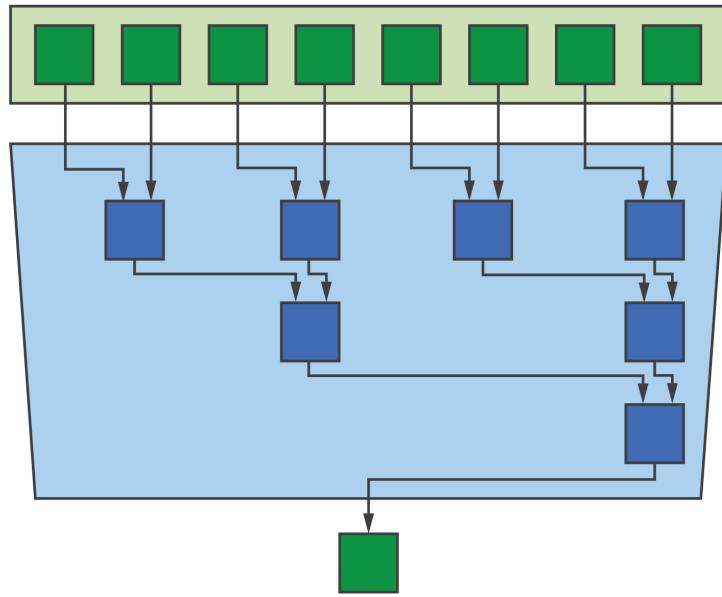


图 14-4 显示了通过树归约实现的归约模式，需要对 N 个输入元素的范围进行 $\log_2(N)$ 组合操作。尽管树型归约很常见，但是其他的实现也是可能的——归约一个以特定的顺序组合值。

内核间很少会地并行，即使是并行的，也经常与归约（如 MapReduce 框架）配对。这使得归约成为需要理解的最重要的并行模式之一，也是必须能够在任何设备上有效执行的模式。

针对不同设备进行扩展需要进行性能权衡，即计算部分所花费的时间和组合它们所花费的时间。使用过少的并行度会增加计算时间，而使用过多的并行度会增加合并时间。

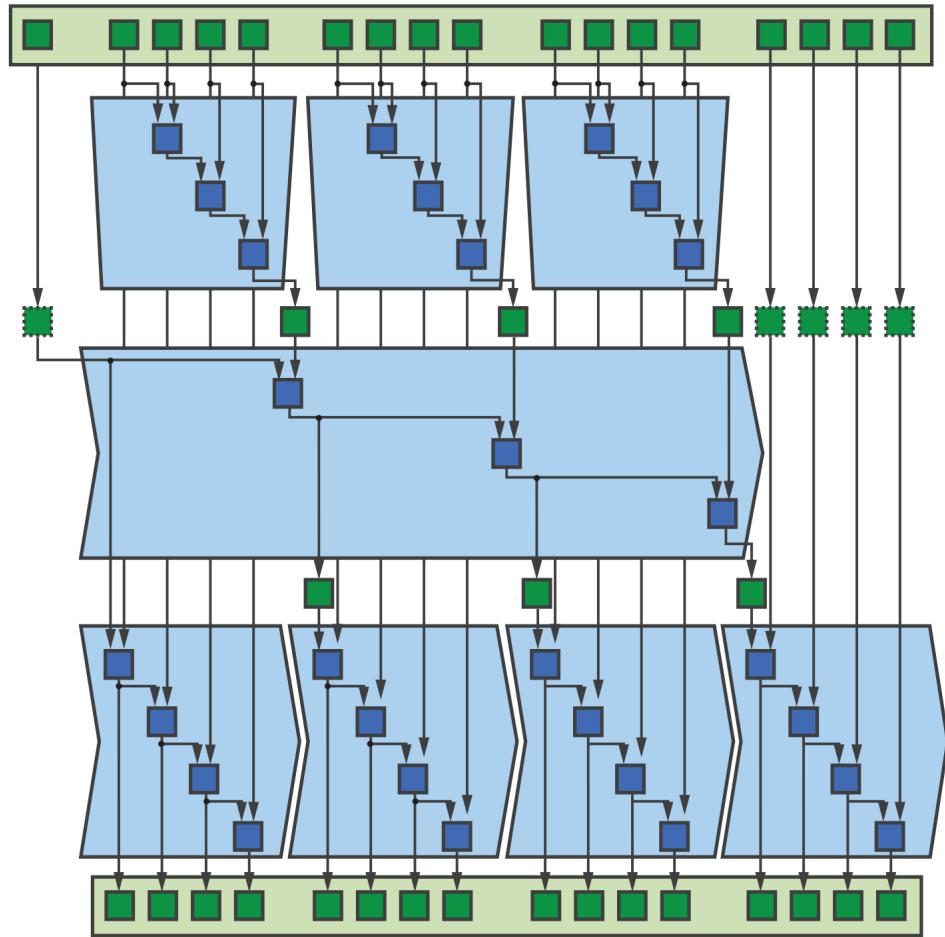
通过使用不同的设备来执行计算和组合步骤，来提高系统的整体利用率可能很不错，但是这种调优工作必须注意在设备之间移动数据的成本。实践中，我们发现在产生数据的同时在同一设备上直接执行数据归约通常是最好的方法。因此，使用多个设备来提高归约模式的性能并不依赖于任务的并行性，而是依赖于另一种级别的数据并行性（即，每个设备对输入数据的一部分执行归约）。

Scan——扫描

扫描模式使用二进制关联运算符计算一个广义前缀和，输出的每个元素表示一个部分结果。扫描包容第一个元素，如果元素的部分和的范围在 $[0,i]$ （即之和（包括 i ）。扫描排除第一个元素，如果元素的部分和的范围在 $[0,i]$ （例如：和不包括 i ）。

乍一看，扫描似乎是一个固有的串行操作，因为每个输出的值取决于前一个输出的值！虽然与其他模式相比，扫描获得并行性的机会确实较少（因此可能伸缩性较差），但图 14-5 显示了使用对相同数据的多次扫描实现并行扫描是可能的。

图 14-5 扫描模式



因为在扫描操作中并行性的机会是有限的，所以执行扫描的最佳设备高度依赖于问题的大小：较小的问题更适合 CPU，因为只有较大的问题才会包含足够的数据并行性使 GPU 饱和。对于 FPGA 和其他架构来说，问题的大小不太重要，因为扫描本身就有助于管道并行性。处理归约的一个经验法则是，在产生数据的设备上执行扫描操作——考虑在优化期间扫描操作适合应用程序的位置和方式——通常会比专注于单独优化扫描操作产生更好的结果。

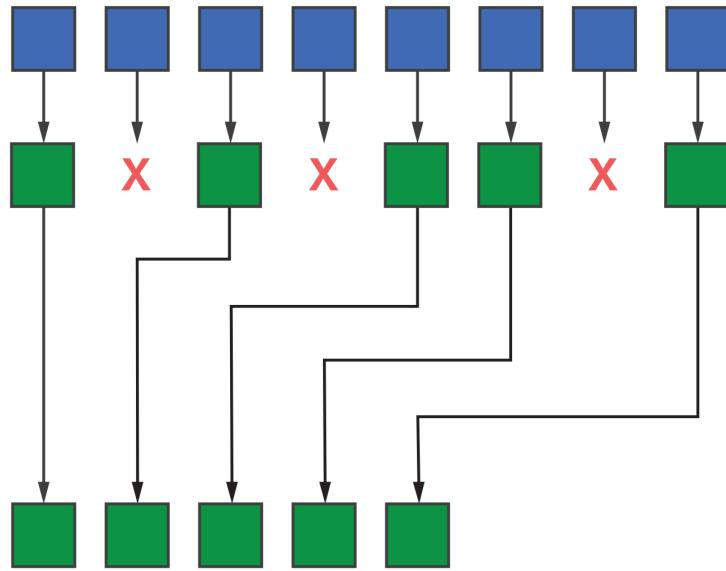
打包和解包

打包和解包模式与扫描密切相关，通常在扫描功能的基础上实现。在这里单独讨论，因为它们是常见操作（例如，添加到列表）的高性能实现，这些操作可能与求前缀和没有明显的联系。

打包

打包模式，如图 14-6 所示，基于布尔条件丢弃输入范围中的元素，将未丢弃的元素打包到输出范围的连续位置。这个布尔条件可以是预先计算的掩码，也可以通过对每个输入元素应用某个函数在线计算。

图 14-6 打包模式

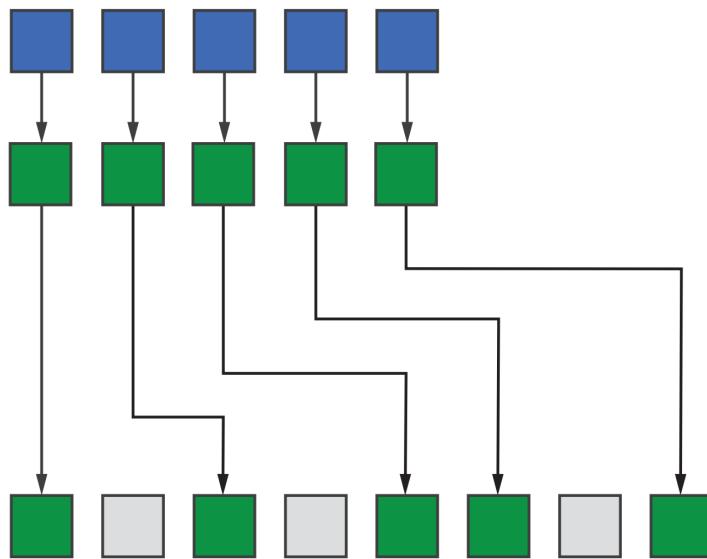


与扫描一样，打包操作具有串行性。给定要打包/复制的输入元素，计算在输出范围中的位置需要关于有多少先前的元素也打包/复制到输出中的信息。这个信息相当于对布尔条件的独占性扫描。

解包

如图 14-7 所示 (顾名思义)，解包模式与打包模式相反。输入范围内的连续元素被解包为输出范围内的非连续元素，而不影响其他元素。这个模式最明显的用例是解包以前打包的数据，也可以用来填充以前计算产生的数据中的“空白”。

图 14-7 解包模式



使用内置函数和库

这些模式中的许多都可以直接使用 DPC++ 的内置函数，或用 DPC++ 使用供应商提供的库来编写。利用这些功能和库是在真正的大型软件工程项目中，是平衡性能、可移植性和生产力的最

佳方法。

DPC++ 中的归约库

DPC++ 不要求每个人都维护自己的归约内核库，而是提供了一个抽象来用归约语义描述变量。这种抽象简化了内核的表达式，并使显式执行成为可能，允许实现为不同的设备、数据类型和归约操作组合选择不同的归约算法。

图 14-8 reduce 表示为使用归约库的 ND-Range 数据并行内核

```
1 h.parallel_for(
2     nd_range<1>{N, B},
3     reduction(sum, plus<>()),
4     [=](nd_item<1> it, auto& sum) {
5         int i = it.get_global_id(0);
6         sum += data[i];
7     });
}
```

图 14-8 中的内核展示了使用归约库的示例。注意，内核体不包含任何对归约的引用——必须指定包含归约的内核，使用加法算子组合了 sum 变量的实例。这为实现自动生成优化的归约序列提供了足够的信息。

编写本文时，归约库只支持具有单个归约变量的内核。未来版本的 DPC++ 预计将支持同时执行多个归约的内核，方法是在传递给 parallel_for 的 nd_range 和函数参数之间指定多个归约，并将多个归约作为内核函数的参数。

归约的结果在内核完成之前不能保证会写回到原来的变量。除了这个限制，访问归约的结果与访问 SYCL 中的任何其他变量的行为相同：访问存储在缓冲区中的归约结果需要创建适当的设备或主机访问器，访问存储在 USM 分配中的归约结果可能需要显式的同步和/或内存移动。

DPC++ 的归约库与其他语言中的归约抽象的区别是，限制了在内核执行期间对归约变量的访问——不能检查归约变量的中间值，禁止使用除了指定的组合函数以外的任何东西来更新归约变量。这些限制可以避免一些难以调试的错误（例如，在计算最大值的同时添加一个归约变量），并确保归约可以在各种不同的设备上高效地实现。

归约类

归约类是用来描述内核中归约的接口。构造归约对象的唯一方法是使用图 14-9 所示的函数。

图 14-9 归约函数的原型

```
1 template <typename T, typename BinaryOperation>
2 unspecified reduction(T* variable, BinaryOperation combiner);
3
4 template <typename T, typename BinaryOperation>
5 unspecified reduction(T* variable, T identity, BinaryOperation combiner);
```

该函数的第一个版本允许指定归约变量和用于组合每个工作项的操作符。第二个版本允许提供与归约操作符相关联的可选标识值——这是对用户定义归约的优化。

请注意，归约函数的返回类型未指定，归约类本身完全由实现定义。虽然对于 C++ 类来说有点不寻常，但它允许实现使用不同的类（或具有任意数量模板参数的单个类）来表示不同的约简算法。未来版本的 DPC++ 可能会重新考虑这个设计，以便在特定的执行上下文中显式地请求特定的归约算法。

reducer 类

reducer 类的实例封装了一个归约变量，暴露了有限的接口，就可以进行安全的归约变量更新。reducer 类的简化定义如图 14-10 所示。像归约类一样，reducer 类是实现定义的——reducer 的类型取决于 reduce 是如何执行的，为了最大化性能，在编译时知道这一点很重要。允许更新归约变量的函数和操作符是定义好的，并且保证 DPC++ 可以支持。

图 14-10 reducer 类的简化定义

```
1 template <typename T,
2         typename BinaryOperation ,
3         /* implementation-defined */>
4 class reducer {
5     // Combine partial result with reducer's value
6     void combine(const T& partial);
7 };
8
9 // Other operators are available for standard binary operations
10 template <typename T>
11 auto& operator +=(reducer<T, plus::<T>>&, const T&);
```

每个 reducer 都提供一个 combine()，它将部分结果（来自单个工作项）与 reducer 变量的值组合在一起。这个组合函数的行为由实现定义，但不是在编写内核时需要考虑的问题。还需要一个 reducer，使其他操作可依赖于 reducer 变量，例如：+= 运算符定义为加归约。这些附加操作符仅作为开发者方便和提高可读性提供，操作符具有与直接调用 combine() 相同的行为。

用户定义的归约

一些常见的归算法（例如，树状归约）不会看到每个工作项直接更新单个共享变量，而是将一些部分结果累加到私有变量中，这些私有变量将在将来的某个时候合并在一起。这样的私有变量引入了一个问题：实现应该如何初始化？从每个工作项初始化到第一个贡献的变量有潜在的性能后果，因为需要额外的逻辑来检测和处理未初始化的变量。将变量初始化为归运算符的标识来避免性能损失，但只有当标识已知时才有可能。

DPC++ 实现只能在还原操作简单的算术类型，且还原操作符是标准函数（例如，plus）时自动确定要使用的正确标识值。对于用户定义的归约（例如，对用户定义的类型进行操作和/或使用用户定义的函数），可以通过指定标识来提高性能。

图 14-11 使用用户定义的归约查找具有 ND-Range 内核的最小值的位置

```
1 template <typename T, typename I>
2 struct pair {
```

```

3  bool operator<(const pair& o) const {
4      return val <= o.val || (val == o.val && idx <= o.idx);
5  }
6  T val;
7  I idx;
8 };
9
10 template <typename T, typename I>
11 using minloc = minimum<pair<T, I>>;
12
13 constexpr size_t N = 16;
14 constexpr size_t L = 4;
15
16 queue Q;
17 float* data = malloc_shared<float>(N, Q);
18 pair<float, int>* res = malloc_shared<pair<float, int>>(1, Q);
19 std::generate(data, data + N, std::mt19937{});
20
21 pair<float, int> identity = {
22     std::numeric_limits<float>::max(), std::numeric_limits<int>::min()
23 };
24 *res = identity;
25
26 auto red = reduction(res, identity, minloc<float, int>());
27
28 Q.submit([&](handler& h) {
29     h.parallel_for(nd_range<1>{N, L}, red, [=](nd_item<1> item, auto& res) {
30         int i = item.get_global_id(0);
31         pair<float, int> partial = {data[i], i};
32         res.combine(partial);
33     });
34 }).wait();
35
36 std::cout << "minimum value = " << res->val << " at " << res->idx << "\n";

```

对用户定义归约的支持，仅限于简单的可复制类型和没有副作用的组合函数，但这足以支持许多实际用例。例如，图 14-11 中的代码演示了如何使用用户定义的归约来计算向量中的最小元素及其位置。

oneAPI DPC++ 库

C++ 标准模板库 (STL) 包含了几种算法，对应于本章讨论的并行模式。STL 中的算法通常适用于由迭代器指定的序列。从 C++17 开始，可以使用执行策略参数，表示算法应该顺序执行还是并行执行。

oneAPI DPC++ 库 (oneDPL) 利用了这个执行策略参数来提供高效的并行编程方法，这种方法在底层利用了用 DPC++ 编写的内核。如果应用程序可以仅使用 STL 算法的功能来表达，那么 oneDPL 就可以在系统中使用加速器，而无需编写任何 DPC++ 内核代码！

图 14-12 中的表格显示了 STL 中可用的算法如何与本章描述的并行模式以及在适当的情况下与串行算法 (在 C++17 之前可用) 相关。关于如何在 DPC++ 应用程序中，使用这些算法的更详细的说明可以在第 18 章找到。

图 14-12 将并行模式与 C++17 算法库相关联

模式	串行算法	并行算法
Map	transform	transform
Stencil	transform	transform
Reduction	accumulate	reduce transform_reduc
Scan	partial_sum	inclusive_scan exclusive_scan transform_inclusive_scan transform_exclusive_scan
Pack	N/A	copy_if
Unpack	N/A	N/A

组函数

DPC++ 设备代码中对并行模式的支持是由单独的组函数库提供的。这些组函数利用特定工作组 (例如，一个工作组或一个子工作组) 的并行性，在有限的范围内实现通用的并行算法，并且可以作为构建块来构建更复杂的算法。

如 oneDPL 一样，DPC++ 中组函数的语法基于 C++。每个函数的第一个参数接受一个 group 或 sub_group 对象来代替执行策略，C++ 算法的任何限制都适用。组功能是由指定组中的所有工作项协作执行的，因此必须以类似于组栅栏的方式对待——组中的所有工作项必须在控制流中遇到相同的算法 (即，组中的所有工作项必须以类似的方式执行或不执行到算法调用)，并且所有工作项必须提供相同的参数，以确保操作的一致性。

reduce、exclusive_scan 和 inclusive_scan 函数仅限于基本数据类型和最常用的归约操作符 (例如，plus、minimum 和 maximum)。这对于许多用例来说已经足够了，但 DPC++ 的未来版本预计将扩展对用户定义类型和操作符的支持。

直接编程

尽管建议尽可能利用库，但是通过观察如何使用“原生”DPC++ 内核实现每个模式，可以学到更多东西。

本章剩余部分的内核不会达到与高度调优库相同的性能水平，但是对于更好地理解 DPC++ 的功能很有用——甚至可以作为新库的功能原型。

使用供应商提供的库!

当供应商提供函数库实现时，使用它而不是将函数重新实现总是有益的!

Map——映射模式

由于其简单性，映射模式可以作为基本的并行内核直接实现。图 14-13 所示的代码显示了这样一个实现，使用映射模式计算范围内每个输入元素的平方根。

图 14-13 数据并行内核中实现映射模式

```
1 Q.parallel_for(N, [=](id<1> i) {  
2     output[i] = sqrt(input[i]);  
3 }).wait();
```

Stencil——模具模式

如图 14-14 所示，将模具直接实现为多维缓冲区的数据并行内核，很简单，也很容易理解。

图 14-14 数据并行内核中实现模具模式

```
1 id<2> offset(1, 1);  
2 h.parallel_for(stencil_range, offset, [=](id<2> idx) {  
3     int i = idx[0];  
4     int j = idx[1];  
5  
6     float self = input[i][j];  
7     float north = input[i - 1][j];  
8     float east = input[i][j + 1];  
9     float south = input[i + 1][j];  
10    float west = input[i][j - 1];  
11    output[i][j] = (self + north + east + south + west) / 5.0f;  
12});
```

不过，这个模式的表达式非常简单，不能期望能执行得很好。正如本章前面提到的，需要使用局部性（通过空间或时间阻塞）来避免从内存中重复读取相同的数据。一个使用工作组本地内存的空间阻塞如图 14-15 所示。

图 14-15 使用工作组本地内存 ND-Range 内核中实现模具模式

```
1 range<2> local_range(B, B);  
2 // Includes boundary cells  
3 range<2> tile_size = local_range + range<2>(2, 2);  
4 auto tile = local_accessor<float, 2>(tile_size, h);  
5  
6 // Compute the average of each cell and its immediate neighbors  
7 id<2> offset(1, 1);  
8
```

```

9 h.parallel_for(
10 nd_range<2>(stencil_range, local_range, offset), [=](nd_item<2> it) {
11     // Load this tile into work-group local memory
12     id<2> lid = it.get_local_id();
13     range<2> lrange = it.get_local_range();
14     for (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
15         int gi = ti + B * it.get_group(0);
16         for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
17             int gj = tj + B * it.get_group(1);
18             tile[ti][tj] = input[gi][gj];
19         }
20     }
21     it.barrier(access::fence_space::local_space);
22
23     // Compute the stencil using values from local memory
24     int gi = it.get_global_id(0);
25     int gj = it.get_global_id(1);
26
27     int ti = it.get_local_id(0) + 1;
28     int tj = it.get_local_id(1) + 1;
29
30     float self = tile[ti][tj];
31     float north = tile[ti - 1][tj];
32     float east = tile[ti][tj + 1];
33     float south = tile[ti + 1][tj];
34     float west = tile[ti][tj - 1];
35     output[gi][gj] = (self + north + east + south + west) / 5.0f;
36 });

```

要对给定的模具进行最佳优化，需要对块大小、邻域和模板函数本身进行压缩时检查，这其实很复杂，需要有更多讨论才能理解。。

Reduction——归约模式

通过利用提供工作项之间同步和通信功能（例如，原子操作、工作组和子工作组功能、子工作组混合）的特性，在 DPC++ 中实现归约内核。图 14-16 和图 14-17 中的内核显示了两种可能的归约实现：使用 parallel_for 和工作项的原子操作的简单归约，以及稍微一些简化，可以使用 ND-Range parallel_for 和工作组 reduce 函数对局部性进行利用。我们将在第 19 章更详细地讨论这些原子操作。

图 14-16 为数据并行内核的简单归约

```

1 Q.parallel_for(N, [=](id<1> i) {
2     atomic_ref<
3         int,
4         memory_order::relaxed,
5         memory_scope::system,
6         access::address_space::global_space>(*sum) += data[i];

```

```
7 }) . wait () ;
```

图 14-17 为 ND-Range 内核的简单归约

```
1 Q.parallel_for(nd_range<1>{N, B}, [=](nd_item<1> it) {
2     int i = it.get_global_id(0);
3     int group_sum = reduce(it.get_group(), data[i], plus<>());
4     if (it.get_local_id(0) == 0) {
5         atomic_ref<
6             int,
7             memory_order::relaxed,
8             memory_scope::system,
9             access::address_space::global_space>(*sum) += group_sum;
10    }
11 }) . wait () ;
```

还有许多其他方法可以编写归约内核，由于对原子操作的硬件支持、工作组本地内存大小、全局内存大小、快速设备范围栅栏的可用性、甚至专用归约指令的可用性也不同，不同的设备可能更喜欢不同的实现。某些体系结构上，使用 $\log_2(N)$ 单独的内核调用来执行树形归约可能会更快（或更必要）。

强烈建议只有在 DPC++ 归约库不支持的情况下，或者在为特定设备的功能对内核进行微调时，才需要考虑手动实现归约——即使这样，也要在 100% 确定归约库的性能不佳之后才考虑这样做！

Scan——扫描模式

实现并行扫描需要对数据进行多次扫描，每次扫描之间都要进行同步。由于 DPC++ 没有提供同步 ND-Range 内所有工作项的机制，所以必须使用多个内核来实现设备范围内扫描的实现，这些内核通过全局内存传递结果。

图 14-18、14-19 和 14-20 所示的代码演示了使用多个内核实现的包含第一个元素扫描。第一个内核将输入值分布到多个工作组，在工作组本地内存中，计算工作组本地扫描（注意，可以使用工作组 inclusive_scan 函数来代替）结果。第二个内核使用单个工作组计算本地扫描，对每个块的最终值进行计算。第三个内核结合这些中间结果来最后确定前缀和。这三个内核对应图 14-5 中的三个层。

图 14-18 ND-Range 内核中实现全局包含扫描的第 1 阶段：跨工作组计算

```
1 // Phase 1: Compute local scans over input blocks
2 q.submit([&](handler& h) {
3     auto local = local_accessor<int32_t, 1>(L, h);
4     h.parallel_for(nd_range<1>(N, L), [=](nd_item<1> it) {
5         int i = it.get_global_id(0);
6         int li = it.get_local_id(0);
7
8         // Copy input to local memory
9         local[li] = input[i];
```

```

10    it.barrier();
11
12    // Perform inclusive scan in local memory
13    for (int32_t d = 0; d <= log2((float)L) - 1; ++d) {
14        uint32_t stride = (1 << d);
15        int32_t update = (li >= stride) ? local[li - stride] : 0;
16        it.barrier();
17        local[li] += update;
18        it.barrier();
19    }
20
21    // Write the result for each item to the output buffer
22    // Write the last result from this block to the temporary buffer
23    output[i] = local[li];
24    if (li == it.get_local_range()[0] - 1)
25        tmp[it.get_group(0)] = local[li];
26    });
27 }).wait();

```

图 14-19 ND-Range 内核中实现全局包含扫描的第 2 阶段: 扫描每个工作组的结果

```

1 // Phase 2: Compute scan over partial results
2 q.submit([&](handler& h) {
3     auto local = local_accessor<int32_t, 1>(G, h);
4     h.parallel_for(nd_range<1>(G, G), [=](nd_item<1> it) {
5         int i = it.get_global_id(0);
6         int li = it.get_local_id(0);
7
8         // Copy input to local memory
9         local[li] = tmp[i];
10        it.barrier();
11
12        // Perform inclusive scan in local memory
13        for (int32_t d = 0; d <= log2((float)G) - 1; ++d) {
14            uint32_t stride = (1 << d);
15            int32_t update = (li >= stride) ? local[li - stride] : 0;
16            it.barrier();
17            local[li] += update;
18            it.barrier();
19        }
20
21        // Overwrite result from each work-item in the temporary buffer
22        tmp[i] = local[li];
23    });
24 }).wait();

```

图 14-20 ND-Range 内核中实现全局包含扫描的第三阶段 (最终阶段)

```

1 // Phase 3: Update local scans using partial results
2 q.parallel_for(nd_range<1>(N, L), [=](nd_item<1> it) {
3     int g = it.get_group(0);
4     if (g > 0) {
5         int i = it.get_global_id(0);
6         output[i] += tmp[g - 1];
7     }
8 }).wait();

```

图 14-18 和 14-19 非常相似，唯一的区别是范围的大小，以及如何处理输入和输出值。该模式的实际实现可以使用带有不同参数的函数来实现这两个阶段，这里将它们作为不同的代码。

打包和解包

打包和解包也称为收集和分散操作。这些操作处理数据在内存中的排列方式，以及呈现给计算资源的方式上。

打包

由于打包依赖于排除第一个元素扫描，实现一个适用于 ND-Range 的所有元素的打包，必须通过全局内存和在几个内核排队的过程中进行。但是，对于包有一个通用的用例，不需要在 ND-Range 的所有元素上应用操作——仅在特定工作组或子工作组中的项上应用打包。

图 14-21 中的代码片段展示了如何在排除第一个元素的扫描中，实现组打包操作。

图 14-21 排除第一个元素的扫描中实现组打包操作

```

1 uint32_t index = exclusive_scan(g, (uint32_t) predicate, plus<>());
2 if (predicate)
3     dst[index] = value;

```

图 14-22 中的代码演示了如何在内核中使用这样的包操作，来构建一个元素列表，这些元素需要一些额外的后处理（在将来的内核中）。所示的示例基于分子动力学模拟中的一个真实内核：分配给粒子 i 的子工作组中的工作项协作识别与 i 固定距离内的所有其他粒子，只有这个“相邻列表”中的粒子将被用于计算作用于每个粒子上的力。。

图 14-22 使用工作组打包操作构建需要额外后处理的元素列表

```

1 range<2> global(N, 8);
2 range<2> local(1, 8);
3 Q.parallel_for(
4     nd_range<2>(global, local),
5     [=](nd_item<2> it) [[ cl::intel_reqd_sub_group_size(8) ]] {
6         int i = it.get_global_id(0);
7         sub_group sg = it.get_sub_group();
8         int sglid = sg.get_local_id()[0];
9         int sgrange = sg.get_max_local_range()[0];
10    });

```

```

11  uint32_t k = 0;
12  for (int j = sg lid; j < N; j += sgrange) {
13
14      // Compute distance between i and neighbor j
15      float r = distance(position[i], position[j]);
16
17      // Pack neighbors that require post-processing into a list
18      uint32_t pack = (i != j) and (r <= CUTOFF);
19      uint32_t offset = exclusive_scan(sg, pack, plus<>());
20      if (pack)
21          neighbors[i * MAX_K + k + offset] = j;
22
23      // Keep track of how many neighbors have been packed so far
24      k += reduce(sg, pack, plus<>());
25  }
26  num_neighbors[i] = reduce(sg, k, maximum<>());
27 }) . wait();

```

注意，打包模式不重新排序元素——打包到输出数组中的元素的顺序与输入数组中的顺序相同。打包的这个属性很重要，使我们能够使用打包功能来实现其他更抽象的并行算法（比如 std::copy_if 和 std::stable_partition）。然而，在打包功能上还可以实现其他并行算法，这些算法不需要维护顺序（比如：std::partition）。

解包

和打包一样，可以使用 scan 实现解包。在排除第一个元素扫描的基础上实现子工作组解包操作，如图 14-23 所示。

图 14-23 排除第一个元素扫描的基础上实现子工作组解包操作

```

1 uint32_t index = exclusive_scan(sg, (uint32_t) predicate, plus<>());
2 return (predicate) ? new_value[index] : original_value;

```

图 14-24 中的代码展示了如何使用这样的子工作组解包，来改平衡有不同控制流的内核（本例中，是计算 Mandelbrot 集）。每个工作项分配了单独的像素来计算和迭代，直到收敛或达到最大迭代次数。然后使用解包操作将已完成的像素替换为新像素。

图 14-24 使用子工作组解包操作来平衡不同控制流的内核

```

1 // Keep iterating as long as one work-item has work to do
2 while (any_of(sg, i < Nx)) {
3     uint32_t converged =
4         next_iteration(params, i, j, count, cr, ci, zr, zi, mandelbrot);
5     if (any_of(sg, converged)) {
6         3
7         // Replace pixels that have converged using an unpack
8         // Pixels that haven't converged are not replaced
9         uint32_t index = exclusive_scan(sg, converged, plus<>());

```

```
10     i = (converged) ? iq + index : i;
11     iq += reduce(sg, converged, plus<>());
12
13 // Reset the iterator variables for the new i
14 if (converged)
15     reset(params, i, j, count, cr, ci, zr, zi);
16 }
17 }
```

这种方法提高效率（并减少执行时间）的程度高度依赖于应用程序和输入，因为检查完成和执行解包操作都会带来开销！因此，在实际应用程序中成功地使用此模式，需要一些基于发散量和正在执行的计算的微调（例如，引入启发式方法，仅在活动工作项的数量低于某个阈值时执行解包操作）。

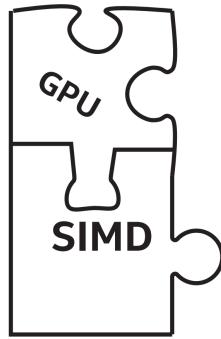
总结

本章演示了如何使用 DPC++ 和 SYCL 特性实现一些最常见的并行模式，包括内置函数和库。SYCL 和 DPC++ 的生态仍在开发中，随着开发人员从该语言以及生产级应用程序和库的开发中获得更多经验，我们希望为这些模式找到最佳实践。

更多信息

- 结构化并行编程: 高效计算模式, Michael McCool, Arch Robison, and James Reinders, 2012, Morgan Kaufmann 出版, ISBN 978-0-124-15993-8
- 英特尔 oneAPI DPC++ 库指南, <https://software.intel.com/en-us/oneapi-dpcpp-library-guide>
- 算法库, C++ 在线手册, <https://en.cppreference.com/w/cpp/algorithms>

15 GPU 编程



过去的几十年里，图形处理单元 (GPU) 已经从在屏幕上绘制图像的硬件设备，发展到能够执行复杂并行内核的通用设备。现在，几乎每台计算机都有一个 GPU 和一个 CPU，许多程序可以通过将部分并行算法从 CPU 转移到 GPU 来加速。

本章中，我们将描述 GPU 是如何工作的，GPU 软件和硬件是如何执行 SYCL 应用程序的，以及当我们为 GPU 编写和优化并行内核时需要的技巧和技术。

性能说明

与处理器类型一样，不同厂商的 GPU，甚至不同产品的 GPU 也有所不同。因此，对于一种设备的最佳实践可能并不适用于不同的设备。本章的建议可能利于多数 GPU，无论是现在还是将来，但是……

为了实现特定 GPU 的最佳性能，请查阅 GPU 供应商的文档！

本章末尾提供了许多 GPU 厂商的文档链接。

GPU 的工作原理

本节描述了常规 GPU 的工作原理，以及 GPU 与其他类型加速器的区别。

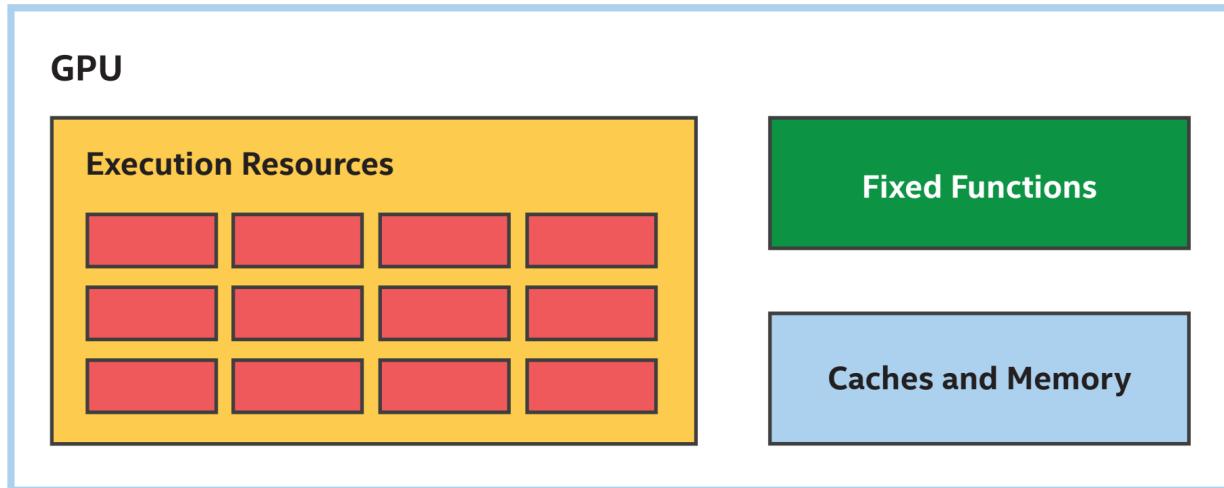
GPU 构建块

图 15-1 展示了一个非常简化的 GPU，由三个构建块组成：

- **执行资源:** GPU 是执行计算工作的处理器。不同的 GPU 供应商对其执行资源使用不同的名称，所有现代 GPU 都由多个可编程处理器组成。处理器可以是异构和专门的，也可以是同构的。大多数现代 GPU 的处理器都是同构的。
- **固定功能:** GPU 是硬件单元，其可编程性低于执行资源，专门用于单个任务。当 GPU 用于图形时，图形管道的许多部分（如栅格化或光线追踪）都使用 GPU 来执行，以提高能效和性能。当 GPU 用于数据并行计算时，可能用于工作负载调度、纹理采样和依赖跟踪等任务。
- **缓存和内存:** 像其他处理器类型一样，GPU 有缓存来存储执行资源访问的数据。GPU 缓存可能是隐式的，不需要开发者做什么，或者显式的暂存存储器，开发者只要在使用数据之前，有

目的地将数据移动到缓存中即可。许多 GPU 也有很大的内存池，来提供对执行资源使用数据的快速访问。

图 15-1 常规 GPU 的构建块

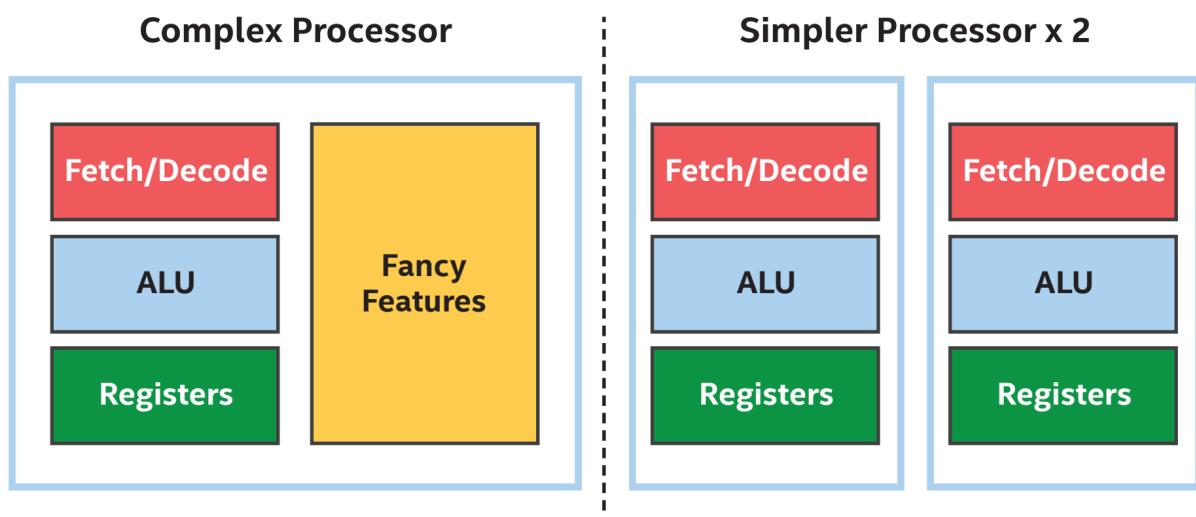


更简单的处理器

以前执行图形操作时，GPU 处理大量数据。例如，游戏帧或渲染工作量涉及数千个顶点，这些顶点每帧产生数百万个像素。为了保持帧率，必须尽快处理这些大量数据。

常规 GPU 设计权衡是消除形成执行资源的处理器特性，从而加速单线程性能，并使用节省来资源进行额外的构建，如图 15-2 所示。例如，GPU 处理器可能不包括复杂的乱序执行能力或其他类型处理器使用的分支预测逻辑。由于这些权衡，单个数据元素在 GPU 上的处理速度可能会比在其他处理器上慢，但更多的处理器数量使 GPU 能够快速有效地处理许多数据元素。

图 15-2 GPU 处理器更简单，但数量多



为了利用这种权衡，给 GPU 足够大的数据范围就很重要。为了说明加载大量数据的重要性，请参考本书中的矩阵乘法内核。

关于矩阵乘法

本书中，矩阵乘法内核用于演示内核中的更改或它的分配方式是如何影响性能的。虽然使用本章描述的技术可以显著提高矩阵乘法的性能，但矩阵乘法是如此重要和常见的操作，许多硬件 (GPU、CPU、FPGA、DSP 等) 供应商已经实现了许多例程的高度调优版本，包括矩阵乘法。供应商投入大量的时间和精力来实现和验证特定设备的功能，在某些情况下可能会使用在标准内核中难以或不可能使用的功能或技术。

使用供应商提供的库！

当供应商提供函数库实现时，使用它而不是重新实现内核！对于矩阵乘法，可以将 oneMKL 作为适合 DPC++ 开发者的英特尔 oneAPI 解决方案工具包的一部分。

矩阵乘法内核可以通过将其作为单个任务，提交到一个队列中，并在 GPU 上执行。这个矩阵乘法内核的主体，看起来就像在主机 CPU 上执行的函数，如图 15-3 所示。

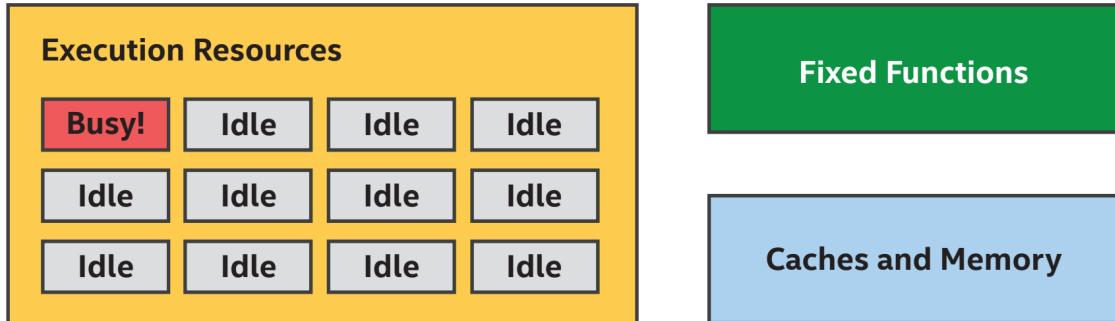
图 15-3 单个任务矩阵乘法看起来很像 CPU 主机代码

```
1 h.single_task( [=]() {
2     for (int m = 0; m < M; m++) {
3         for (int n = 0; n < N; n++) {
4             T sum = 0;
5             for (int k = 0; k < K; k++)
6                 sum += matrixA[m * K + k] * matrixB[k * N + n];
7             matrixC[m * N + n] = sum;
8         }
9     }
10});
```

如果尝试在 CPU 上执行这个内核，可能会执行得很好——如果不是很好，因为没利用 CPU 的并行能力，但对于较小的矩阵大小来说可能就足够了。如图 15-4 所示，如果试图在 GPU 上执行这个内核，可能会执行得非常糟糕，因为单个任务将只使用单个 GPU 处理器。

图 15-4 GPU 上的单个任务内核会使许多执行资源闲置

GPU



表达并行性

提高这个内核在 CPU 和 GPU 上的性能，可以通过将一个循环转换为 parallel_for 来代替提交要并行处理的数据元素。对于矩阵乘法内核，可以选择提交表示两个最外层循环的数据元素。图 15-5 中，选择并行处理结果矩阵的行。

图 15-5 类似矩阵乘法

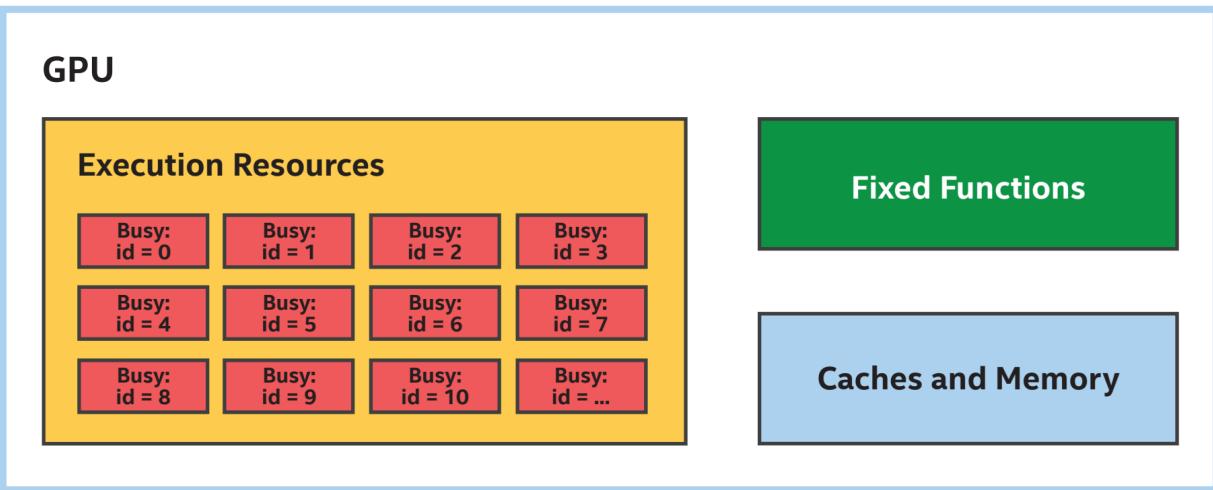
```
1 h.parallel_for(range{M}, [=](id<1> idx) {
2     int m = idx[0];
3     for (int n = 0; n < N; n++) {
4         T sum = 0;
5         for (int k = 0; k < K; k++)
6             sum += matrixA[m * K + k] * matrixB[k * N + n];
7         matrixC[m * N + n] = sum;
8     }
9 })
```

选择如何并行化

选择并行化哪个维度是对 GPU 和其他设备类型调优的重要方法。本章的后续部分将描述，为什么在一个维度上并行比在不同维度上并行执行得更好。

尽管并行内核与单任务内核非常相似，但应该在 CPU 上运行得更好，在 GPU 上运行得更好。如图 15-6 所示，parallel_for 使表示结果矩阵行的工作项能够在多个处理器资源上并行处理，因此所有执行资源都保持忙碌。

图 15-6 某种程度上的并行内核使更多的处理器资源处于繁忙状态



注意，没有指定将行分区并分配给不同处理器资源的确切方式，这为选择在设备上执行内核提供了灵活性。例如，可以选择在同一个处理器上执行连续的行，而不是在处理器上执行单个行，以利用局部性。

表达更多的并行性

通过选择并行处理两个外部循环，可以将矩阵乘法内核更加的并行化。因为 parallel_for 可以表示最多三个维度上的并行循环，如图 15-7 所示。图中，传递给 parallel_for 的范围和表示并行执行空间中索引的项都是二维的。

图 15-7 更多的并行矩阵乘法

```

1 h.parallel_for(range{M, N}, [=](id<2> idx) {
2     int m = idx[0];
3     int n = idx[1];
4     T sum = 0;
5     for (int k = 0; k < K; k++)
6         sum += matrixA[m * K + k] * matrixB[k * N + n];
7     matrixC[m * N + n] = sum;
8 });

```

在 GPU 上运行时，并行性可能会提高矩阵乘法内核的性能。即使当矩阵的行数超过 GPU 处理器的数量时，这也可能是正确的。接下来的几节将描述这种情况的原因。

简化控制逻辑 (SIMD 指令)

许多 GPU 处理器通过大多数数据元素在内核中采用相同的控制流路径来优化控制逻辑。例如，在矩阵乘法内核中，每个数据元素执行最内层循环的次数相同，因为循环边界不变。

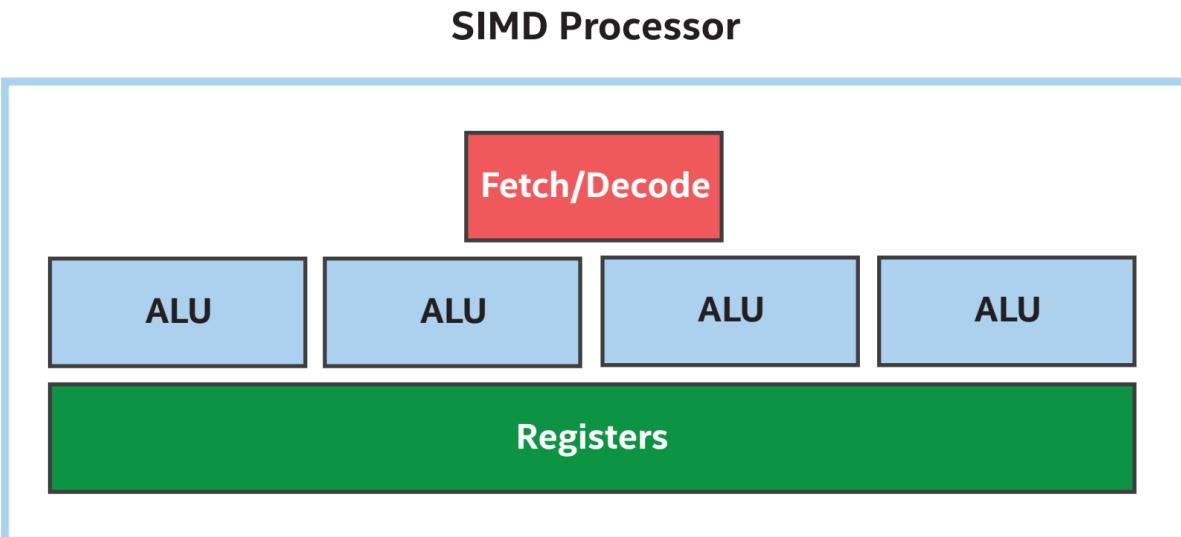
当数据元素以相同的控制流通过内核时，处理器可以通过在多个数据元素之间共享控制逻辑，并将它们作为一个组来执行来降低管理指令流的成本。一种方法是实现单指令、多数据或 SIMD 指令集，其中多条数据元素由一条指令同时处理。

线程与指令流

在许多并行编程上下文中和 GPU 文献中，术语“线程”用来表示“指令流”。上下文中，“线程”与传统的操作系统线程不同，通常更轻量级。但情况并非总是如此，在某些情况下，“线程”用来描述完全不同的东西。

由于术语“线程”是重载的，很容易误解，本章使用术语“指令流”代替。

图 15-8 4 个宽 SIMD 处理器:4 个 ALU 共享取/解码逻辑



单个指令同时处理的数据元素的数量，有时称为指令的 SIMD 宽度或执行指令的处理器。图 15-8 中，4 个 ALU 共享相同的控制逻辑，因此可以将其描述为一个宽度为 4 的 SIMD 处理器。

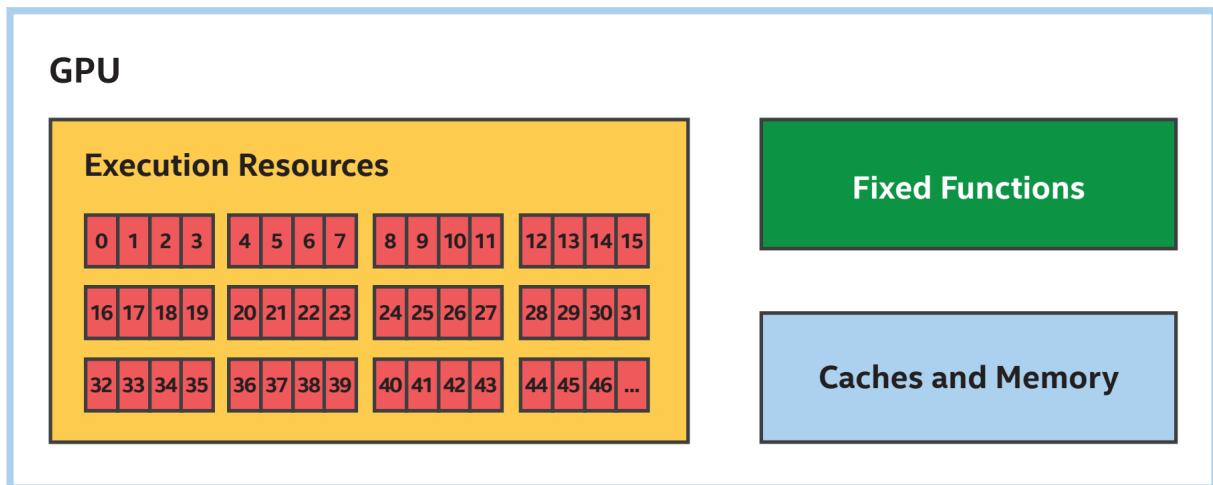
GPU 并不是唯一实现 SIMD 指令集的处理器。其他处理器类型也实现了 SIMD 指令集，以提高处理大型数据集时的效率。GPU 和其他处理器类型之间的主要区别是，GPU 依赖于并行执行多个数据元素来获得良好的性能，而且 GPU 可能比其他处理器类型支持更宽的 SIMD 宽度。例如，GPU 支持 16、32 或更多数据元素的 SIMD 宽度并不少见。

编程模型:SPMD 和 SIMD

虽然 GPU 实现不同宽度的 SIMD 指令集，但这通常是实现细节，并且对在 GPU 上执行数据并行内核的应用程序是透明的。这是因为许多 GPU 编译器和运行时 API 实现了单程序、多数据或 SPMD 编程模型，其中 GPU 编译器和运行时 API 决定用 SIMD 指令流处理最有效的一组数据元素，而不是显式地表达为 SIMD 指令。第 9 章的“子工作组”一节探讨了数据元素分组对应用程序可见的情况。

图 15-9 扩展了执行资源，以支持宽度为 4 的 SIMD，允许并行处理四倍的矩阵行。

图 15-9 SIMD 处理器上执行并行的内核



通过使用并行处理多个数据元素的 SIMD 指令，图 15-5 和 15-7 中的并行矩阵乘法内核的性能可以超越单个处理器的数量。通过在同一个处理器上执行连续的数据元素，SIMD 指令的使用还在许多情况下提供了局域性优势，包括矩阵乘法。

内核受益于处理器之间的并行性和处理器内部的并行性！

预测和屏蔽

只要在内核中所有数据元素通过相同的路径通过条件代码，那么在多个数据元素之间共享指令流就可以很好地工作。当数据元素在条件代码中采取不同的路径时，控制流称为分叉。当控制流在 SIMD 指令流中分叉时，通常执行两个控制流路径，并屏蔽或预测一些通道。这确保了正确的行为，但是这种正确性以性能为代价，因为屏蔽的通道不会执行有用的操作。

为了展示预测和屏蔽是如何工作的，请考虑图 15-10 中的内核，将具有“奇数”索引的每个数据元素乘以 2，并将具有“偶数”索引的每个数据元素加 1。

图 15-10 具有发散控制流的内核

```

1 h.parallel_for(array_size, [=](id<1> i) {
2     auto condition = i[0] & 1;
3     if (condition)
4         dataAcc[i] = dataAcc[i] * 2; // odd
5     else
6         dataAcc[i] = dataAcc[i] + 1; // even
7 });

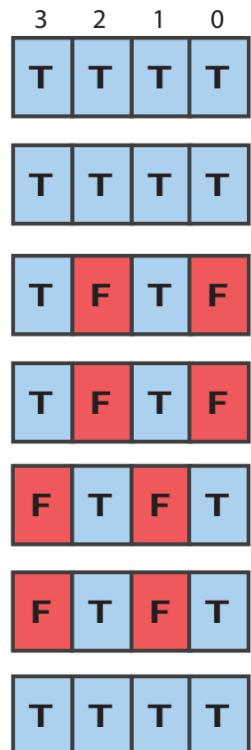
```

假设在图 15-8 所示的宽度为 4 的 SIMD 处理器上执行这个内核，并且在一个 SIMD 指令流中执行前 4 个数据元素，在不同的 SIMD 指令流中执行接下来的 4 个数据元素，以此类推。图 15-11 显示了屏蔽通道和预测执行的一种方式，正确地使用不同的控制流执行了这个内核。

图 15-11 发散型内核可能的通道掩码

Channel Mask:

```
// All Channels Enabled Initially:  
  
// All Channels Enabled for Condition:  
    condition = i.get(0) & 1  
  
// Even Channels Disabled by "if":  
    if condition  
  
// Odd Indices Multiplied by Two:  
    dataAcc[i] = dataAcc[i] * 2  
  
// Enabled Channels Inverted by "else":  
    else  
  
// Even Indices Incremented by One:  
    dataAcc[i] = dataAcc[i] + 1  
  
// Possible Re-Convergence After "if":
```



SIMD 效率

SIMD 效率度量的是与等效标量指令流相比 SIMD 指令流执行得有多好。图 15-11 中，由于控制流将通道划分为两个相等的组，所以发散控制流中的每条指令执行效率只有一半。最坏的情况下，对于高度分散的内核，效率可能会降低。

所有实现 SIMD 指令集的处理器都会受到影响 SIMD 效率的差异惩罚，但因为 GPU 通常比其他处理器类型支持更宽的 SIMD，当优化 GPU 的内核时，重新构造算法来最小化不同的控制流和最大化融合执行可能特别有用。但也并不总是可以这样做，选择以更收敛的执行并行化维度，可能比以高度发散的执行并行化另一个维度执行要好。

SIMD 效率和组中工作项

目前为止，本章中的所有内核都是基本的数据并行内核，没有在执行范围内指定任何项目分组，这给了实现为设备选择最佳分组的机会。例如，具有较宽 SIMD 的设备可能更喜欢较大的分组，具有较窄 SIMD 的设备可能适合较小的分组。

当内核具有明确工作项分组的 ND-Range 内核时，应该注意选择能够最大化 SIMD 效率的 ND-Range 工作组大小。当工作组的大小不能被处理器的 SIMD 宽度整除时，部分工作组可能会在内核的整个过程中禁用通道。可以使用 `preferred_work_group_size_multiple` 查询来选择有效的工作组大小。关于如何查询设备属性的更多信息，请参阅第 12 章。

选择包含单个工作项的工作组大小可能会表现得非常糟糕，因为许多 GPU 将通过屏蔽除其他所有 SIMD 通道来实现单个工作项工作组。例如，图 15-12 中的内核可能比图 15-5 中非常相似的内核性能要差得多，两者之间唯一显著的区别是从一个基本的数据并行内核，改为一个低效的单工

作项 ND-Range 内核 (`nd_range<1>{M, 1}`)。

图 15-12 低效的单项矩阵乘法

```
// A work-group consisting of a single work-item is inefficient!
1 h.parallel_for(nd_range<1>{M, 1}, [=](nd_item<1> idx) {
2     int m = idx.get_global_id(0);
3
4     for (int n = 0; n < N; n++) {
5         T sum = 0;
6         for (int k = 0; k < K; k++)
7             sum += matrixA[m * K + k] * matrixB[k * N + n];
8         matrixC[m * N + n] = sum;
9     }
10 }
11});
```

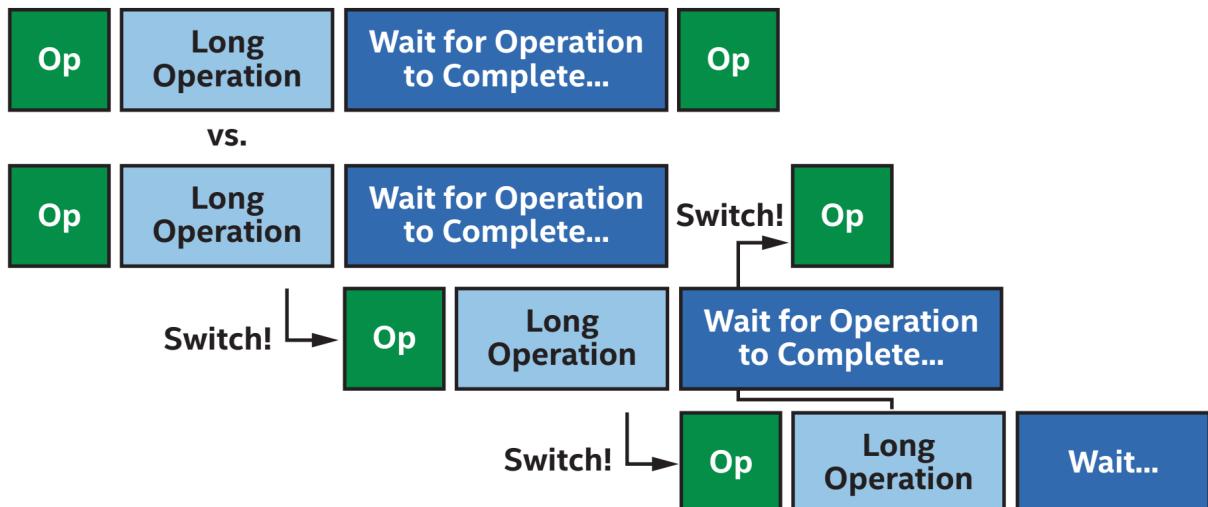
切换工作以隐藏延迟

许多 GPU 实现了另一种技术来简化控制逻辑，最大化执行资源，并提高性能：许多 GPU 允许多个指令流同时驻留在一个处理器上，而不是在处理器上执行单个指令流。

处理器上驻留多个指令流利于性能，可以让处理器选择要执行的工作。如果指令流执行了长延迟的操作，比如：从内存读取，处理器可以切换并运行的不同指令流，而不是阻塞等待操作完成。有了足够的指令流，当处理器切换回原始指令流时，长延迟的操作可能已经完成，而不需要处理器阻塞等待。

图 15-13 展示了处理器如何使用多个并发指令流，来隐藏延迟并提高性能。尽管第一个指令流在多个指令流中执行的时间要长一些，但通过切换到其他指令流，处理器能够找到准备执行的工作，而不需要等待长操作完成。

图 15-13 切换指令流以隐藏延迟



GPU 分析工具可以使用占用率等术语，来描述 GPU 处理器当前执行的指令流数量与理论指令流总数的对比。

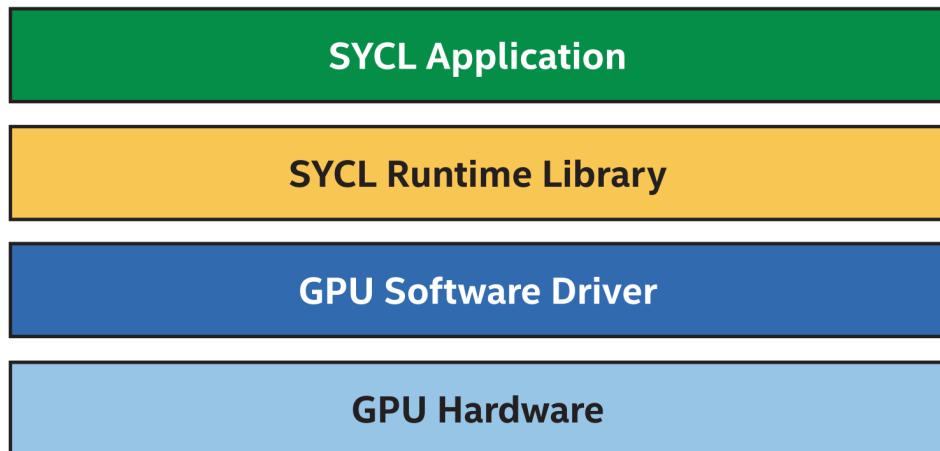
低占用率并不一定意味着低性能，因为少量指令流可能会使处理器繁忙。同样，高占用率并不一定意味着高性能，因为如果所有指令流执行低效的、长延迟的操作，GPU 仍然需要等待。其他条件相同的情况下，增加占用率可以最大化 GPU 隐藏延迟的能力，通常也会提高性能。增加占用率是图 15-7 中并行的内核提高性能的另一个原因。

这种在多个指令流之间切换，以隐藏延迟的技术特别适合于 GPU 和数据并行处理。从图 15-2 中可以看出，GPU 通常比其他处理器更简单，因此缺乏复杂的延迟隐藏特性。这使得 GPU 更容易受到延迟问题的影响，但是由于数据并行编程涉及到处理大量的数据，GPU 通常有大量指令流要执行！

将内核函数加载到 GPU

本节描述一个应用程序、SYCL 运行库和 GPU 软件驱动程序如何一起在 GPU 硬件上加载内核。图 15-14 中展示了这些抽象层的软件堆栈。许多情况下，这些层的存在对程序是透明的，但在调试或分析应用程序时，理解它们就很重要了。

图 15-14 将并行内核加载到 GPU(简化)



SYCL 运行时库

SYCL 运行库是 SYCL 应用程序交互的主要软件库。运行时库负责实现类，如队列、缓冲区和访问器，以及这些类的成员函数。运行时库的部分实现可能位于头文件中，因此可以直接编译为应用程序可执行文件。运行时库的其他部分作为库函数实现，这些库函数作为应用程序构建过程的一部分链接到应用程序可执行文件。运行时库通常不是特定于设备的，同一个运行时库可能会将负载分配到 CPU、GPU、FPGA 或其他设备上。

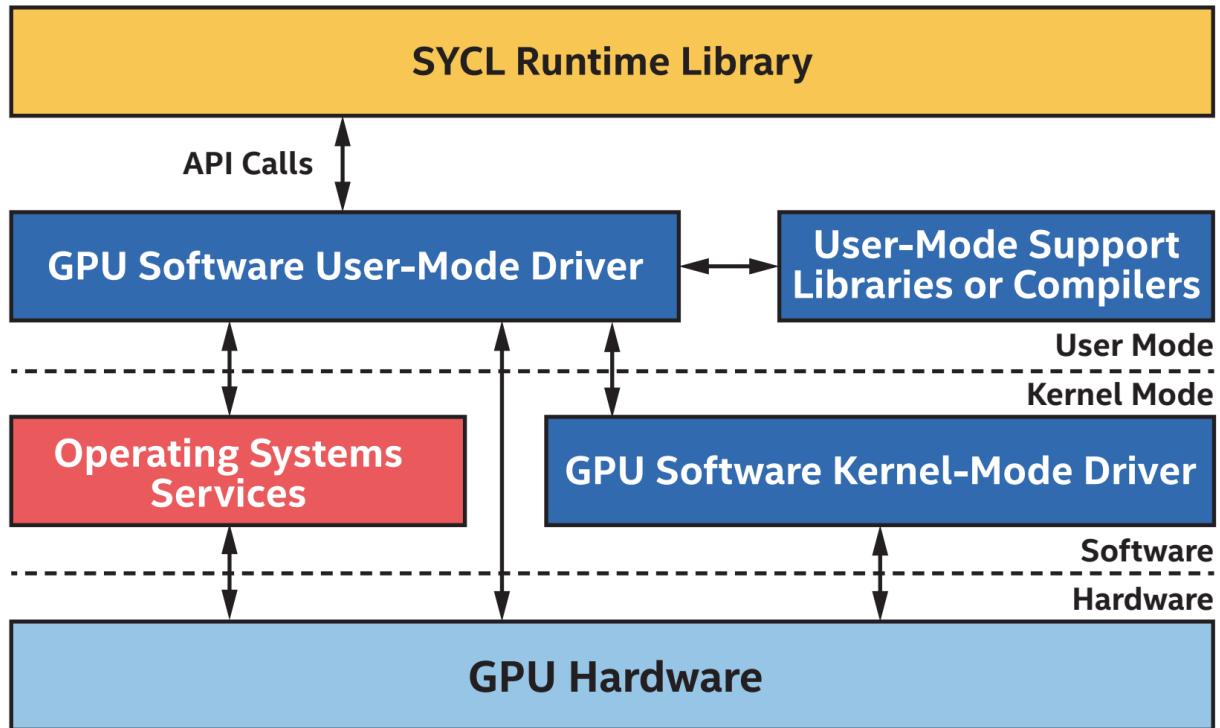
GPU 软件驱动

虽然从理论上讲，SYCL 运行库可以直接加载给 GPU，但大多数 SYCL 运行库都与 GPU 软件驱动程序接口，以便将工作提交给 GPU。

GPU 软件驱动程序通常是 API 的实现，如 OpenCL、Level Zero 或 CUDA。大多数 GPU 软件驱动程序是在 SYCL 运行时调用的用户模式驱动程序库中实现的，用户模式驱动程序可以调用

操作系统或内核模式驱动程序来执行系统级的任务，比如：分配内存或向设备提交工作。用户模式驱动程序也可以调用其他用户模式库，例如：GPU 驱动程序可以调用 GPU 编译器来实时编译一个内核，从中间表示到 GPU ISA(指令集架构)。各个软件模块，及其相互作用如图 15-15 所示。

图 15-15 GPU 软件驱动模块



GPU 硬件

当运行时库或 GPU 软件用户模式驱动程序被显式请求提交工作时，或者 GPU 软件启发式地决定应该开始工作时，通常会通过操作系统或内核模式驱动程序调用来开始在 GPU 上执行工作。某些情况下，GPU 软件用户模式驱动程序可能会直接向 GPU 提交工作，但不是所有设备或操作系统都支持。

当工作在 GPU 上执行的结果需要在主机处理器或另一个加速器，GPU 必须发出工作完成的信号。工作完成所涉及的步骤与工作提交的步骤非常相似，但执行方式相反:GPU 可能会向操作系统或内核驱动程序发出信号，表示已经完成了执行，然后驱动程序会得到通知，最后运行时库会通过 GPU 软件 API 调用观察到相应的工作已经完成。

每一步都引入了延迟，运行库和 GPU 软件需要在较低的延迟和较高吞吐量之间进行权衡。例如，频繁地向 GPU 提交工作可能会减少延迟，但频繁地提交也可能由于提交产生的开销而降低吞吐量。收集大量的工作将增加延迟，但将提交开销分摊到更多的工作上，就会引入更多并行执行的机会。运行时和驱动都作出了权衡，但如果怀疑驱动启发式的提交导致工作效率低下，应该查阅文档确定是否有方法来替代默认驱动程序行为，这里可能会使用特定于 API，甚至是使用特定的机制。

注意加载成本！

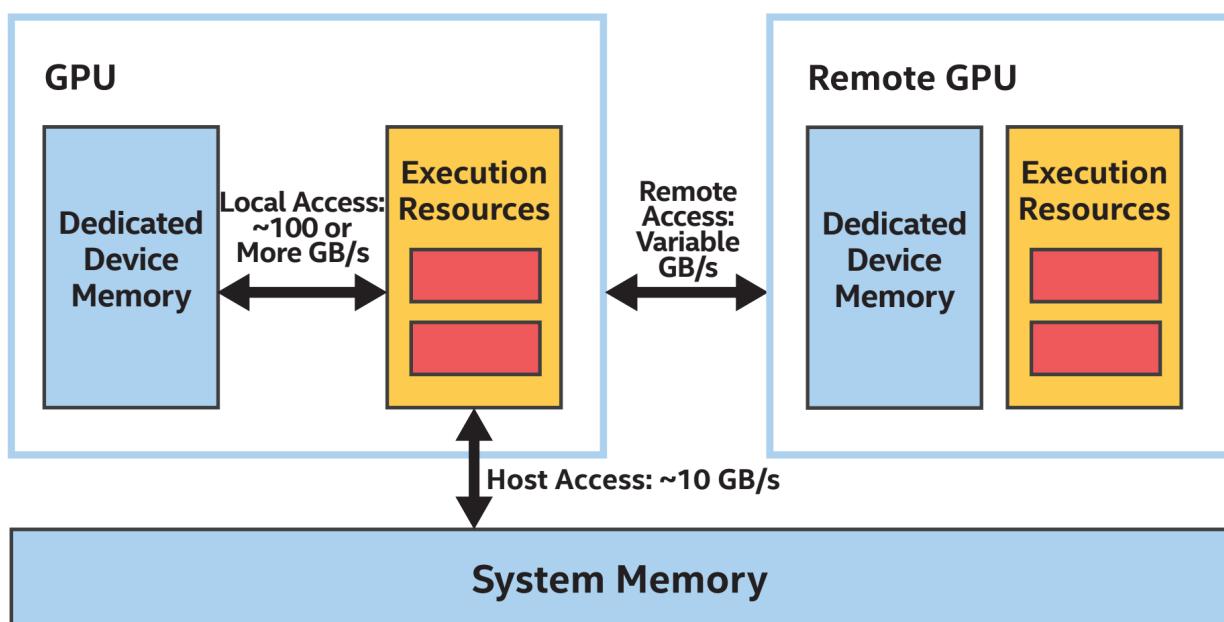
尽管 SYCL 实现和 GPU 供应商正在不断创新和优化，以降低将工作转移到 GPU 的成本，但在启动 GPU 工作和在主机或其他设备上观察结果时，总会有开销。当选择在哪里执行算法时，要考虑在设备上执行算法的好处，以及将算法和需要的数据移动到设备上的成本。某些情况下，最有效的方法可能是使用主机处理器执行并行操作——或者在 GPU 上低效地执行算法的串行部分——以避免将算法从一个处理器移动到另一个处理器的开销。

从整体上考虑算法的性能——在设备上执行算法的部分效率可能最高，而不是将执行转移到另一个设备上！

与设备存储器之间的传输

对于使用专用内存的 GPU，要特别注意内存和主机或其他设备上内存之间的传输成本。系统中不同内存类型的内存带宽差异如图 15-16 所示。

图 15-16 设备内存、远程内存和主机内存之间的差异



回顾一下第 3 章，GPU 更喜欢在专用设备内存上运行，这比在主机内存或其他设备内存上运行要快多个数量级。尽管访问专用设备内存比访问远程内存或系统内存要快得多，但如果数据尚未在专用设备内存中，则必须复制或迁移数据。

频繁访问数据的话，将其移到专用设备内存中是性能有益的，特别是当 GPU 执行资源忙于处理另一个任务时，传输任务可以异步执行。当数据访问频率不高或不可预测时，最好是节省传输成本，并在远程或系统内存中操作数据（即使每次访问成本更高）。第 6 章描述了控制内存分配、复制和预取数据到专用设备内存的不同方法。这些技术在为 GPU 优化程序执行时非常重要。

GPU 内核最佳实践

前几节描述了传递给 `parallel_for` 的参数，如何影响内核分配给 GPU 资源，以及在 GPU 上执行内核所涉及的软件层和开销。本节介绍了内核在 GPU 上执行的最佳实践。

内核要么是内存式的，其性能受到数据读写操作在 GPU 上执行资源的限制，要么是计算式的，其的性能受到 GPU 上执行资源的限制。这是为 GPU 和许多其他处理器优化内核的良好第一步！——用来确定内核是内存式，还是计算式的，因为改进内存式内核的技术通常不会有益于计算式内核，反之亦然。所以供应商通常提供分析工具来帮助确定。

需要不同的优化技术，这取决于内核是内存式还是计算式！

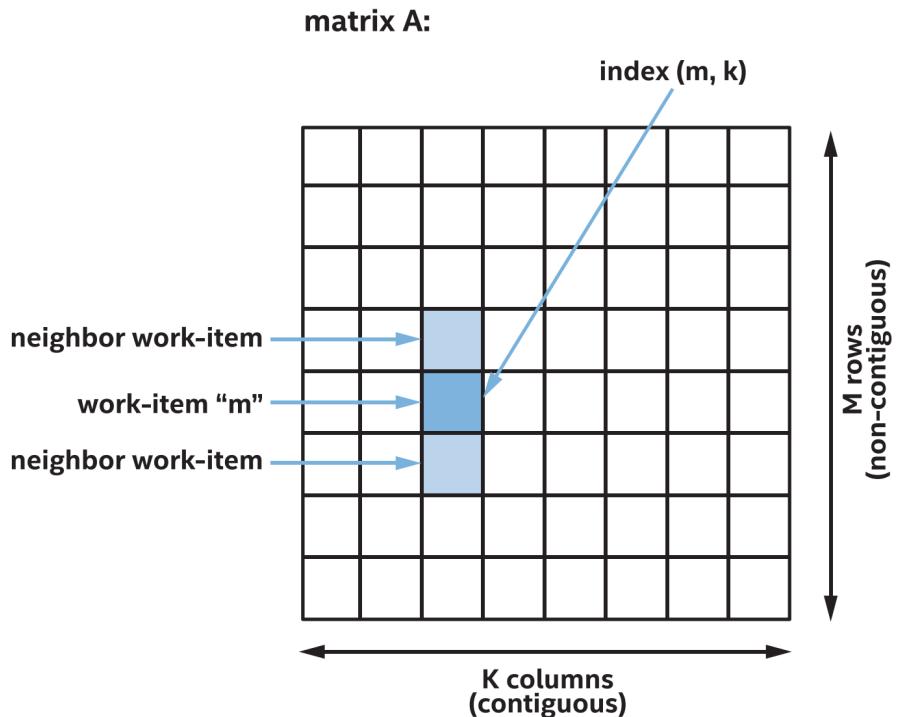
因为 GPU 倾向于拥有许多处理器和较宽的 SIMD，内核倾向于内存式，而不是计算式。如果不确定从哪里开始，检查内核如何访问内存是很好的一步。

访问全局内存

高效地访问全局内存对于优化应用程序性能至关重要，因为工作项或工作组操作的几乎所有数据都源自全局内存。当内核在全局内存上的操作效率很低，那性能会很差。尽管 GPU 通常包含专用硬件集合和计算单元，用于读写内存中的任意位置，但访问全局内存的性能通常是由数据访问的局部性驱动。当工作组中的工作项正在访问内存中的一个元素，该元素与工作组中另一个工作项访问的元素相邻，则全局内存访问性能可能会很好。如果工作组中的工作项访问的内存是跨步的或随机的，则全局内存访问性能会差。一些 GPU 文档将附近内存访问操作描述为**合并访问**。

在图 15-15 中某种程度上并行的矩阵乘法内核，可以选择是并行处理一行还是一列，这里选择并行处理结果矩阵的行。这是一个糟糕的选择：如果一个 id 等于 m 的工作项与一个 id 等于 $m-1$ 或 $m+1$ 的相邻工作项分组，用于访问 `matrixB` 的索引对于每个工作项都是相同的，但是用于访问 `matrixA` 的索引不同于 K ，这意味着访问是高度跨步的。`matrixA` 的访问模式如图 15-17 所示。

图 15-17 对 `matrixA` 的访问是高度跨越和低效的



如果选择并行处理结果矩阵的列，则访问模式具有更好的局部性。图 15-18 中的内核在结构上与图 15-5 中相似，唯一的区别是图 15-18 中的每个工作项操作的是结果矩阵的一列，而不是结果矩阵的一行。

图 15-18 并行计算结果矩阵的列，而不是行

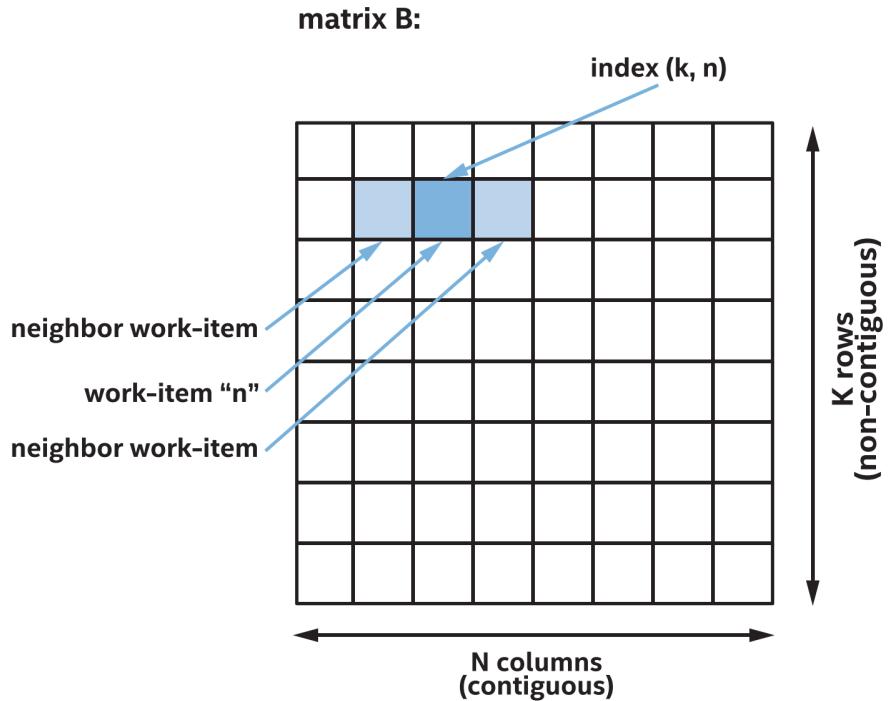
```

1 // This kernel processes columns of the result matrix in parallel.
2 h.parallel_for(N, [=](item<1> idx) {
3     int n = idx[0];
4
5     for (int m = 0; m < M; m++) {
6         T sum = 0;
7         for (int k = 0; k < K; k++)
8             sum += matrixA[m * K + k] * matrixB[k * N + n];
9         matrixC[m * N + n] = sum;
10    }
11 });

```

尽管这两个内核在结构上非常相似，但在许多 GPU 上操作数据列的内核将显著优于操作数据行的内核，纯粹是因为更高效的内存访问：如果一个 id 等于 n 的工作项与一个 id 等于 n-1 或 n+1 的相邻工作项分组，用于访问 matrixA 的索引现在对于每个工作项都是相同的，并且用于访问 matrixB 的索引是连续的。matrixB 的访问模式如图 15-19 所示。

图 15-19 对 matrixB 的访问是连续的和有效的



对连续数据的访问通常非常有效。根据经验，访问一组工作项的全局内存的性能取决于访问的 GPU 缓存行的数量有访问都在一个缓存行内，访问将以最高性能执行。一个访问需要两条缓存线，比如通过访问每个其他元素或者从缓存不对齐的地址开始，访问可能会以一半的性能运行。组中的每个工作项访问唯一的高速缓存行，例如：对于非常频繁的或随机的访问，就可能以最低的性能运行。

设置内核变体

对于矩阵乘法，选择沿着一个维度并行化，显然可以提高内存访问效率，但对于其他内核，选择可能不那么明显。在实现最佳性能非常重要的内核中，如果不清楚并行化的维度，那么有时值得开发和分析沿每个维度并行化的不同内核变体，以了解哪种方法更适合设备和数据集。

访问工作组本地内存

前一节中，我们描述了对全局内存的访问如何从局部性中获益，从而最大化缓存性能。某些情况下，可以设计算法来有效地访问内存，比如选择在一个维度上并行化。然而，并非所有情况下都能使用这种技术。本节描述如何使用工作组本地内存来有效地支持更多的内存访问模式。

从第 9 章开始，工作组中的工作项可以通过工作组本地内存通信和使用工作组栅栏同步来协作解决问题。这种技术对 GPU 有利，因为典型的 GPU 有专属硬件来实现栅栏和工作组本地内存。不同的 GPU 厂商和不同的产品可能会以不同的方式实现工作组本地内存，但是工作组本地内存与全局内存相比通常有两个优势：本地存储器可以支持比对全局存储器的访问更高的带宽和更低的等待时间，即使当全局存储器访问命中高速缓存时，并且本地存储器通常被划分为不同的存储器区域，称为存储库。只要组中的每个工作项访问不同的存储库，本地内存访问就会以完全的性能执行。存储库访问允许本地存储器支持比全局存储器更多的具有峰值性能的访问模式。

许多 GPU 供应商会将连续的本地内存地址分配给不同的存储地址。这确保了连续的内存访问总是完全的性能操作，而不管起始地址是什么。但当内存访问是有跨距访问时，组中的一些工作项可能访问分配给同一存储地址。当发生这种情况时，认为是存储地址冲突，从而导致序列化访问和较低的性能。

为了获得最大的全局内存性能，应尽量减少访问的高速缓存行的数量。

为了获得最大的本地内存性能，尽量减少内存地址冲突！

图 15-20 概述了全局内存和本地内存的访问模式和预期性能。假设当 `ptr` 指向全局内存时，指针与 GPU 缓存行的大小对齐。访问全局内存时，可以通过从缓存对齐的地址连续访问内存来获得最佳性能。访问未对齐的地址可能会降低全局内存性能，因为访问可能需要访问额外的缓存行。因为访问未对齐的本地地址不会导致存储地址冲突，所以本地内存性能没有改变。

有跨距的案例值得更详细地描述。访问全局内存中的每一个元素都需要访问更多的缓存行，这可能会导致较低的性能。访问本地内存中的每一个元素可能会导致存储地址冲突和性能下降，但只有当存储地址的块数量能被 2 整除时。如果存储地址块的数量是奇数，这种情况也将全速运行。

当访问之间的跨越非常大时，每个工作项访问唯一的缓存行，从而导致最差的性能。但是对于本地内存，性能取决于步幅和存储地址块的数量。当步长 N 等于存储地址块数量时，每次访问都会导致内存块冲突，所有访问都会序列化，导致性能最差。但是，如果步数 M 和内存块数量没有关系，则访问将以全速运行。因此，许多优化的 GPU 内核会在本地内存中填充数据结构，以选择减少或消除内存块冲突的方法。

图 15-20 对于不同的访问模式，全局和本地内存可能的性能情况

	Global Memory:	Local Memory:
<code>ptr[id]</code>	Full Performance!	Full Performance!
<code>ptr[id + 1]</code>	Lower Performance	Full Performance!
<code>ptr[id * 2]</code>	Lower Performance	Lower Performance
<code>ptr[id * N]</code>	Worst Performance	Worst Performance
<code>ptr[id * M]</code>	Worst Performance	Full Performance!

子工作组不使用本地内存

正如第 9 章所讨论的，子工作组集合功能是在组中的工作项之间交换数据的一种替代方法。对于许多 GPU，子工作组代表由单个指令流处理的工作项集合。这些情况下，子工作组中的工作项

可以在不使用工作组本地内存的情况下，低成本地交换数据和同步。许多性能好的 GPU 内核都使用子工作组，所以对于内核，我们的算法是否可以重新定义，为使用子工作组集合函数是值得研究的课题。

使用小数据类型优化计算

本节描述消除或减少内存访问瓶颈后优化内核的技术。记住，GPU 以前是用来在屏幕上绘制图片的。尽管 GPU 的纯计算能力随着时间的推移不断发展和改进，但在传统的图形学领域能力优势仍然很明显。

例如，考虑对内核数据类型的支持。许多 GPU 都为 32 位浮点操作进行了高度优化，因为这些操作在图像和游戏中很常见。对于处理较低精度的算法，许多 GPU 还支持较低精度的 16 位浮点型，以精度换取更快的处理速度。相反，尽管许多 GPU 支持 64 位双精度浮点操作，但高精度是有代价的，32 位操作通常比 64 位操作执行得更快。

整数类型也是如此，32 位整数数据类型通常比 64 位整数数据类型执行得更好，而 16 位整数可能执行得更好。如果可以用更小的整数来组织计算，那么内核可能会执行得更快。特别需要注意的是寻址操作，这些操作通常在 64 位的 `size_t` 数据类型上进行，但有时可以重新安排以使用 32 位的数据类型执行大部分计算。某些本地内存情况下，16 位就足够了。

优化数学函数

内核可能为了性能而牺牲精确性的另一个领域涉及了 SYCL 内置函数。SYCL 包含一组丰富的数学函数，在一系列输入中具有定义良好的精度。大多数 GPU 本身并不支持这些函数，而是使用一长串其他指令来实现。虽然数学函数实现通常为 GPU 进行了很好的优化，但如果应用程序可以容忍较低的精度，应该考虑较低精度和更高性能的不同实现。关于 SYCL 内置函数的更多信息，请参阅第 18 章。

对于常用的数学函数，SYCL 库包括快速或本机函数变体，这些变体有较少的或精度要求。对于某些 GPU 来说，这些函数比等价的函数快一个数量级，所以对于算法来说是否有足够的精度值得考虑。例如，许多图像后处理算法具有定义良好的输入，可以容忍较低的精度，因此是使用快速或原生数学函数的好选择。

如果算法可以容忍较低的精度，可以使用较小的数据类型或较低的精度数学函数来提高性能!

专用的功能和扩展

为 GPU 优化内核时的最后一个考虑是在许多 GPU 中常见的专用指令。例如，几乎所有的 GPU 都支持 mad 或 fma 乘加指令，在一个时钟中执行两个操作。GPU 编译器通常非常擅长识别和优化单独的乘法和加法，而不是使用一条指令，SYCL 也包括 mad 和 fma 函数，还可以显式调用。当然，如果希望 GPU 编译器优化乘法和加法，应该确保不会通过禁用浮点截断来阻止优化！

其他特定的 GPU 指令可能只能通过编译器优化或对 SYCL 语言的扩展来使用。例如，一些 GPU 支持一种特殊的“点乘加”指令，编译器将试图识别并优化它，或者可以直接调用。关于如何查询 GPU 实现支持的扩展的更多信息，请参阅第 12 章。

总结

本章中，首先描述了传统的 GPU 是如何工作的，以及 GPU 与传统 CPU 的不同之处。介绍了 GPU 是如何针对大量数据进行优化的，通过交换处理器特性，为其他处理器加速单个指令流。

介绍了 GPU 如何使用宽 SIMD 指令并行处理多个数据元素，以及 GPU 如何使用预测和隐藏延迟来使用 SIMD 指令执行具有复杂流控制的内核。讨论了预测和隐藏延迟如何提高 SIMD 的效率和内核的性能，以及如何选择在一个维度上并行处理，以减少 SIMD 的发散执行。

因为 GPU 有如此多的处理资源，讨论了如何给 GPU 足够的工作来保持高占用率。还介绍了 GPU 如何使用指令流来隐藏延迟，这使得让 GPU 执行大量工作变得更加重要。

接下来，讨论了将内核加载到 GPU 所涉及的软件和硬件层，以及加载的成本。讨论了如何在单个设备上执行算法比在一个设备到另一个设备上执行更有效。

最后，介绍了在 GPU 上执行内核时的最佳实践。介绍了有多少内核从内存式开始，如何有效地访问全局内存和本地内存，如何通过使用子工作组操作来避免使用本地内存。当为计算式内核时，介绍了如何通过用较低的精度换取更高的性能，或使用自定义的 GPU 扩展来访问专门的指令来优化计算。

更多信息

关于 GPU 编程还有很多需要学习的内容，而这一章只是皮毛而已！

GPU 规格和白皮书是学习更多关于特定 GPU 和 GPU 架构的好方法。许多 GPU 供应商提供了关于他们的 GPU 和如何编程的非常详细的信息。

撰写本文时，有关图形处理器的相关信息可以在 software.intel.com、devblogs.nvidia.com 和 amd.com 上找到。

一些 GPU 供应商有开源驱动程序或驱动程序组件。在可用的情况下，检查或遍历驱动程序代码，可以了解哪些操作是昂贵的，或者应用程序中可能存在哪些开销，或是有益的。。

本章主要讨论通过缓存访问器或统一共享内存对全局内存的传统访问，但大多数 GPU 也有纹理采样器，可以加速对图像的操作。有关图像和采样器的更多信息，请参阅 SYCL 规范。

16 CPU 编程



内核编程最初作为 GPU 编程的方式。由于是通用内核编程，理解编程风格如何影响代码向 CPU 的映射就很重要。

CPU 在过去的几年里不断发展。在 2005 年左右发生了一个重大的变化，当时增加时钟速度带来的性能的提升减少了。并行性是最受欢迎的解决方案——CPU 生产商引入了多核芯片，而不是提高时钟频率。计算机可以同时执行多个任务！

虽然多核是提高硬件性能的主流方式，但在软件上释放这种能力需要付出很大的努力。多核处理器要求开发人员对算法进行修改，这样硬件性能增益才会明显，但想要做到却很难。拥有的核芯越多，就越难高效地工作。DPC++ 是解决这些挑战的编程语言，有助于在 CPU(和其他体系结构)上开发各种形式的并行。

本章讨论了 CPU 架构的一些细节，CPU 硬件如何执行 DPC++ 应用程序，并提供了在为 CPU 平台编写 DPC++ 代码的最佳实践。

性能说明

DPC++ 为并行化应用程序铺了一条路。当程序在 CPU 上运行时，其性能在很大程度上取决于以下因素：

- 内核代码的调用方式和执行底层的性能
- 并行内核中运行代码的百分比和可扩展性
- CPU 利用率、数据共享、数据局部性和负载均衡
- 工作项之间同步和通信的数量
- 为创建、恢复、管理、挂起、销毁和同步工作项所执行的线程而引入的开销，串行到并行或并行到串行转换的数量会使性能变得更糟
- 由共享内存引起的冲突
- 共享资源（如内存、写入组合缓冲区和内存带宽）的性能限制

此外，与任何处理器类型一样，CPU 可能因厂商的不同而不同，甚至因产品的不同而不同。对于 CPU 的最佳实践，可能不是针对其他 CPU 或配置的最佳实践。

要在 CPU 上实现最佳性能，请尽可能多地了解相应 CPU 的架构！

通用 CPU 的基础知识

多核 CPU 的出现和发展推动了共享内存并行计算平台的接受度。CPU 提供了笔记本电脑、台式机和服务器级的并行计算平台，使得它们无处不在。CPU 体系结构的最常见形式是缓存

相关的非一致性内存访问 (cc-NUMA)，其特征是访问时间不完全一致。甚至许多小型双插槽 CPU 系统也有这种内存系统。由于处理器中的核心数量和插槽数量不断增加，这种体系结构已占据主导地位。

在 cc-NUMA 的 CPU 系统中，每个套接字连接到系统中内存的一个子集。缓存相关的交互将所有的套接字粘在一起，并为开发者提供一致的内存视图。这样的内存系统是可扩展的，因为聚合内存带宽随系统中插槽的数量变化。交互的好处是应用程序可以透明地访问系统中的内存，而不管数据放在哪里。然而，这也是有代价的：内存访问数据和指令的延迟不再一致（例如，固定的访问延迟）。延迟取决于数据存储在系统中的位置。数据来自直接连接到代码运行的套接字的内存。最坏的情况下，数据源位于系统中很远的内存，由于 cc-NUMA 的 CPU 系统上插槽之间的数量增加，内存访问成本可能会增加。

图 16-1 是一个具有 cc-NUMA 内存的通用 CPU 架构。这种简化的系统架构，包含多插槽系统中的核心和内存组件。本章的其余部分，将用来说明对应代码示例的映射。

为了获得最佳性能，需要确保了解特定系统的 cc-NUMA 配置。例如，Intel 最近的服务器使用了网状互连架构，其中核心、缓存和内存控制器组织成行和列。理解处理器与内存的连接性对于实现系统的最佳性能非常重要。

图 16-1 通用多核 CPU 系统

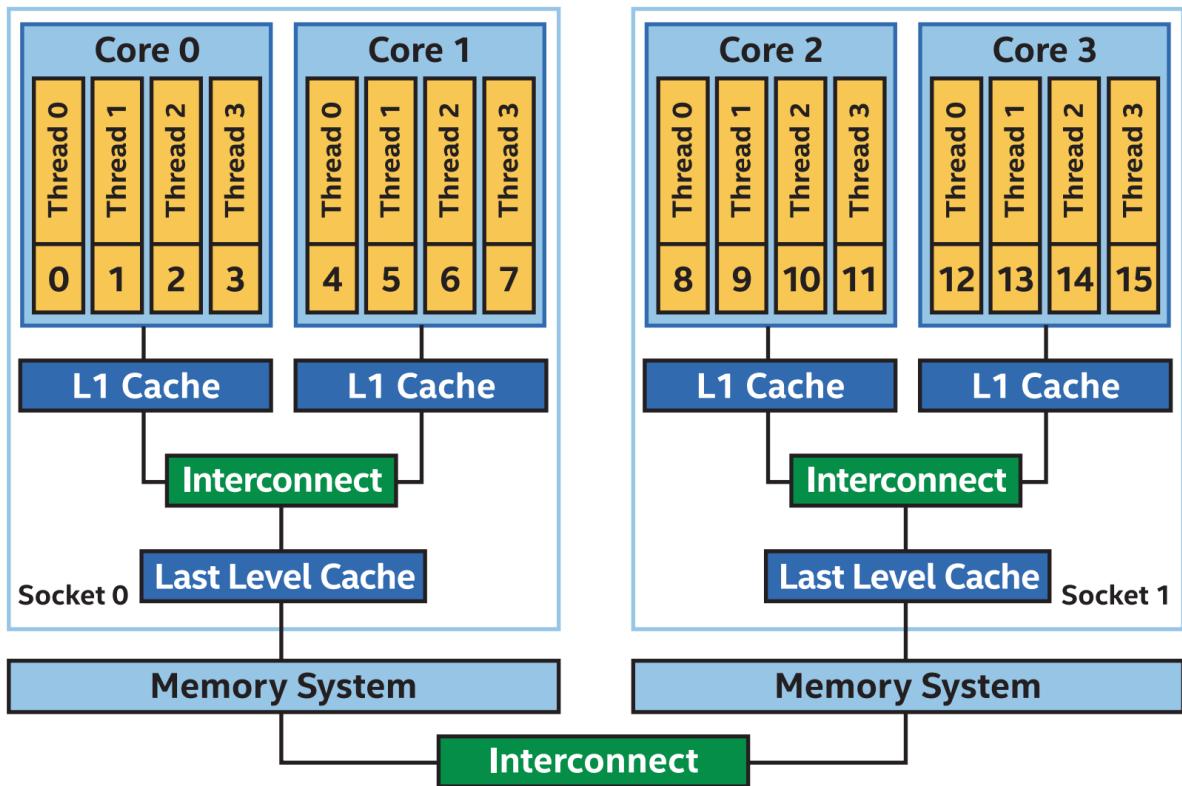


图 16-1 中的系统有两个插槽，每个插槽有两个核，每个核有四个硬件线程。每个核都有自己的 L1 缓存。L1 缓存连接到共享的最后一级缓存，后者连接到套接字上的内存系统，套接字内的内存访问延迟是一致的。

这两个卡槽通过缓存相互连接起来。内存分布在整个系统中，但是所有内存都可以从系统中的任何地方访问。当不在运行访问内存的代码时，内存读写延迟不一致，这意味着当访问远程数据

时，可能会施加不一致的延迟。然而，互连的关键是一致性。不需要担心跨内存系统的数据不一致（这将是一个功能问题），只需要担心访问分布式内存系统的方式对性能的影响。

CPU 中的硬件线程是执行工具。这些是执行指令流的单元（CPU 术语中的线程）。图 16-1 中的硬件线程从 0 到 15 连续编号，这是用于简化本章示例的符号。除特别说明外，本章中有关 CPU 系统的描述均以图 16-1 中的 cc-NUMA 系统为参考。

SIMD 硬件的基础知识

1996 年，第一个 SIMD（根据 Flynn 的分类法，Single Instruction, Multiple Data）指令集是在 x86 架构之上的 MMX。从那以后，许多 SIMD 指令集扩展既遵循 Intel 架构，也在行业中都广泛的应用。CPU 核心通过执行指令来完成它的工作，核心知道如何执行特定的指令是由它实现的指令集（如 x86, x86_64, AltiVec, NEON）和指令集扩展（如 SSE, AVX, AVX-512）定义的。指令集扩展添加的许多操作都集中在 SIMD 指令上。

通过使用比处理的数据基本单元更大的寄存器和硬件，SIMD 指令允许在单个核上同时执行多个计算。使用 512 位寄存器，这样可以用一条机器指令执行 8 个 64 位计算。

图 16-2 在 CPU 硬件线程中执行 SIMD

```
h.parallel_for(1024, [=](id<1> k) {
    z[k] = x[k] + y[k];
});
```

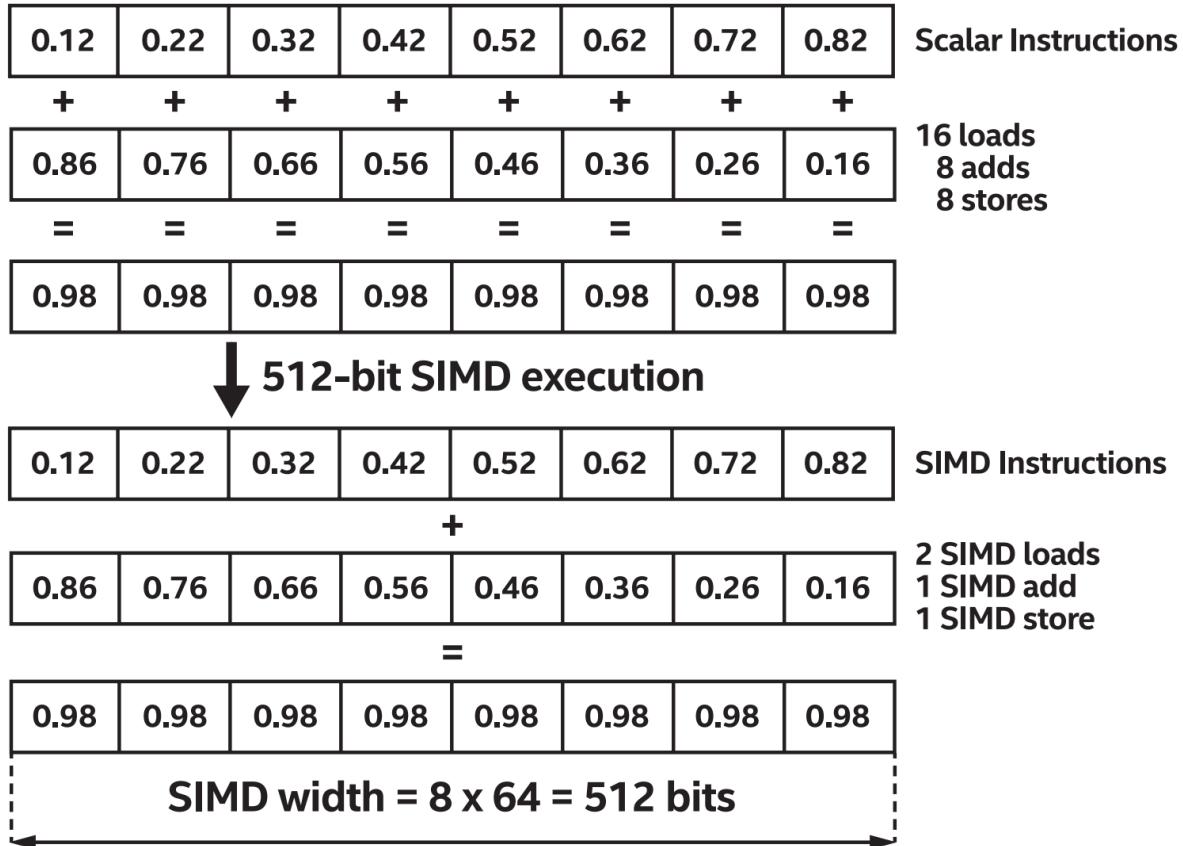


图 16-2 的这个例子可以带来 8 倍的加速。实际中，可能在达不到 8 倍，这是瓶颈可能不完全在计算上，也有部分在内存吞吐量上。通常，使用 SIMD 的性能优势取决于特定的场景。在一些情况下，甚至比更简单的非 SIMD 等效代码的性能更差。现代处理器上，如果知道何时以及如何应用（或让编译器应用）SIMD，就可以获得可观的收益。与所有性能优化一样，开发者应该在将目标机器投入生产之前，测试其性能收益。本章接下来的章节中有更多关于预期性能提高的细节。

具有 SIMD 单元的 cc-NUMA CPU 体系结构，构成了多核处理器的基础，可以以五种不同的方式利用指令级并行，如图 16-3 所示。

图 16-3 五种并行执行指令的方法

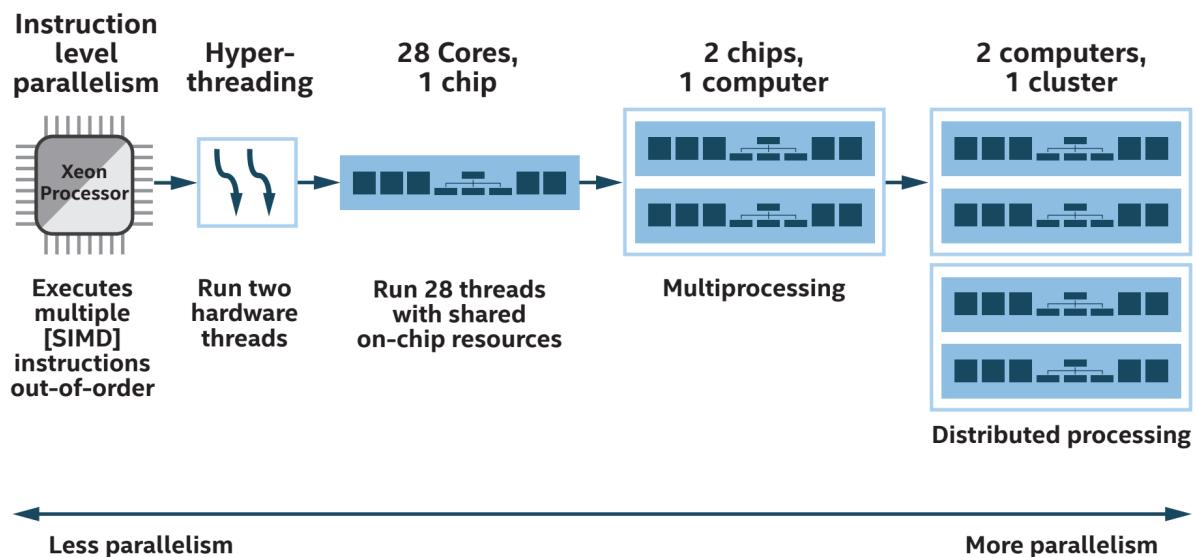


图 16-3 中，指令级并行可以通过标量指令的无序执行实现，也可以通过单个线程中的 SIMD (Single Instruction, Multiple Data) 数据并行实现。线程级并行可以通过在同一个核或不同规模的多个核上执行多个线程来实现。更具体地说，线程级并行性可以通过以下方式实现：

- 现代 CPU 体系结构允许一个核芯同时执行两个或多个线程的指令。
- 每个处理器中包含两个或多个多核架构。操作系统将每个执行核心视为具有所有相关执行资源的独立处理器。
- 处理器（芯片）级的多处理，可以通过独立的线程来完成。因此，处理器可以在应用中运行线程，在操作系统中运行另一个线程，或者可以在单个应用程序中运行并行线程。
- 分布式处理，可以通过在计算机集群上执行由多个线程组成的进程来完成，这些进程通常通过消息传递框架进行通信。

为了充分利用多核处理器资源，软件必须将工作负载分布到多个核的方式编写。这种方法利用了线程级并行性或简单的线程化。

随着多处理器计算机和具有超线程 (HT) 技术的多核处理器越来越多，将并行处理技术作为提高性能的标准实践是非常重要的。本章后面的部分将介绍 DPC++ 中的编码方法和性能调优技术，这些技术允许在多核 CPU 上实现最高性能。

与其他并行处理硬件（例如 GPU）一样，给 CPU 足够大的数据元素集来处理很重要。为了说明如何利用多级并行处理大量数据的重要性，请考虑一个简单的 C++ STREAM Triad 程序，如图

16-4 所示。

关于 STREAM Triad 工作负载的解释

STREAM Triad 工作负载 (www.cs.virginia.edu/stream) 是一个基准工作负载, CPU 供应商使用它来演示高度调优的性能。使用 STREAM Triad 内核来演示并行内核的代码生成, 以及通过本章描述的技术来实现显著提高性能的计划方式。STREAM Triad 是一个相对简单的工作负载, 但足以显示许多优化。

使用供应商提供的库!

当供应商提供函数库实现时, 使用它而不是将函数重新实现为并行内核!

图 16-4 STREAM Triad C++ 循环

```
1 // C++ STREAM Triad workload
2 // __restrict is used to denote no memory aliasing among arguments
3 template <typename T>
4 double triad(T* __restrict VA, T* __restrict VB,
5               T* __restrict VC, size_t array_size, const T scalar) {
6     double ts = timer_start()
7     for (size_t id = 0; id < array_size; id++) {
8         VC[id] = VA[id] + scalar * VB[id];
9     }
10    double te = timer_end();
11    return (te - ts);
12 }
```

STREAM Triad 循环可以在 CPU 上简单地执行, 使用单个 CPU 进行串行执行。好的 C++ 编译器会将执行循环向量化, 为具有 SIMD 硬件的 CPU 生成 SIMD 代码, 以便利用指令级的 SIMD 并行性。例如, 对于支持 AVX-512 的 Intel Xeon 处理器, Intel C++ 编译器生成 SIMD 代码, 如图 16-5 所示。编译器对代码的转换减少了执行时的循环迭代次数, 这是通过在运行时每个循环迭代执行更多的工作 (SIMD 宽度和展开的迭代)!

图 16-5 TREAM Triad C++ 循环的 AVX-512 代码

```
1 // STREAM Triad: SIMD code generated by the compiler, where zmm0, zmm1
2 // and zmm2 are SIMD vector registers. The vectorized loop is unrolled by 4
3 // to leverage the out-of-execution of instructions from Xeon CPU and to
4 // hide memory load and store latency
5
6 # %bb.0: # %entry
7 vbroadcastsd %xmm0, %zmm0 # broadcast "scalar" to SIMD reg zmm0
8 movq $-32, %rax
9 .p2align 4, 0x90
10 .LBB0_1: # %loop.19
11 # =>This Loop Header: Depth=1
```

```

12 vmovupd 256(%rdx,%rax,8), %zmm1 # load 8 elements from memory to zmm1
13 vfmadd213pd 256(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
14 # perform SIMD FMA for 8 data elements
15 # VC[id:8] = scalar*VB[id:8]+VA[id:8]
16 vmovupd %zmm1, 256(%rdi,%rax,8) # store 8-element result to mem from zmm1
17 # This SIMD loop body is unrolled by 4
18 vmovupd 320(%rdx,%rax,8), %zmm1
19 vfmadd213pd 320(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
20 vmovupd %zmm1, 320(%rdi,%rax,8)
21 vmovupd 384(%rdx,%rax,8), %zmm1
22 vfmadd213pd 384(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
23 vmovupd %zmm1, 384(%rdi,%rax,8)
24 vmovupd 448(%rdx,%rax,8), %zmm1
25 vfmadd213pd 448(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
26 vmovupd %zmm1, 448(%rdi,%rax,8)
27 addq $32, %rax
28 cmpq $134217696, %rax # imm = 0xFFFFE0
29 jb .LBB0_1

```

如图 16-5 所示，编译器能以两种方式利用指令级并行性。首先是通过 SIMD 指令的使用，利用指令级数据并行性，其中一条指令可以同时并行处理 8 个双精度数据元素（每个指令）。其次，基于硬件多路指令调度，编译器循环展开来获得（指令之间没有依赖关系）乱序执行是效果。

如果尝试在 CPU 上执行这个函数，可能会运行得很好——没有利用 CPU 的任何多核或线程能力，对于小数组来说已经足够好了。但是，当试图在 CPU 上使用大数组来执行这个函数，那性能很可能很差，因为单个线程只使用一个 CPU 核，当这个核的内存带宽饱和时就会出现性能瓶颈。

利用线程级别的并行性

为了提高 STREAM Triad 内核在 CPU 和 GPU 上的性能，可以通过将循环转换为 parallel_for 内核来计算。

STREAM Triad 内核可以将其提交到队列，并在 CPU 上并行执行。STREAM Triad DPC++ 并行内核的主体看起来就像在 CPU 上串行 C++ 中执行的 STREAM Triad 循环的主体，如图 16-6 所示。

图 16-6 DPC++ STREAM Triad 的 parallel_for 内核代码

```

1 constexpr int num_runs = 10;
2 constexpr size_t scalar = 3;
3
4 double triad(
5     const std::vector<double>& vecA,
6     const std::vector<double>& vecB,
7     std::vector<double>& vecC ) {
8
9     assert(vecA.size() == vecB.size() == vecC.size());
10    const size_t array_size = vecA.size();

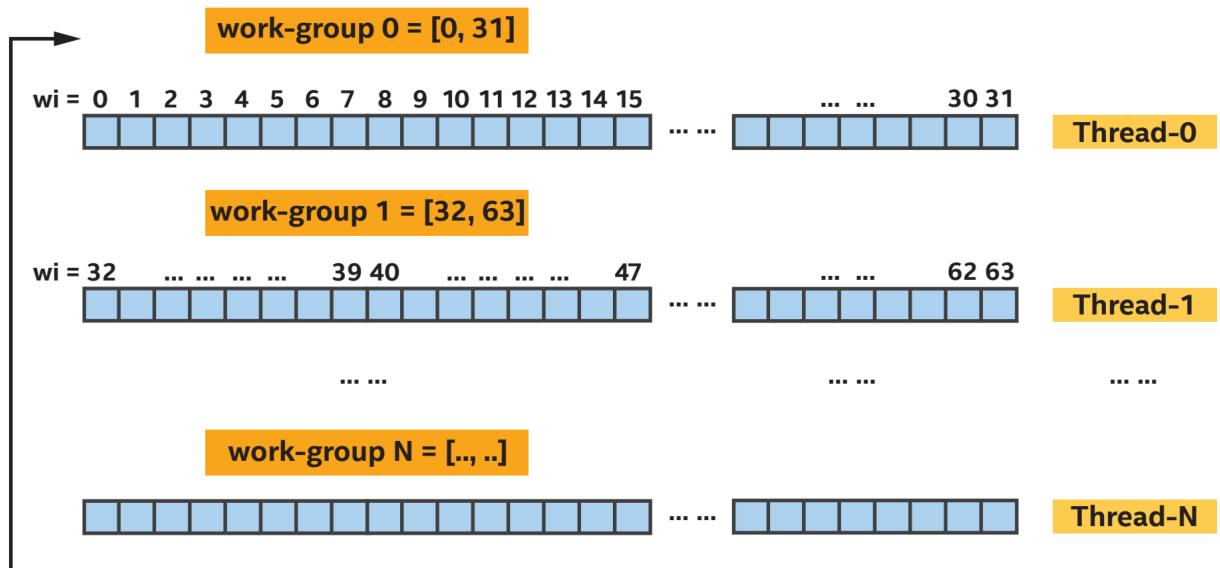
```

```

11 double min_time_ns = DBL_MAX;
12
13 queue Q{ property::queue::enable_profiling{} };
14 std::cout << "Running on device: " <<
15     Q.get_device().get_info<info::device::name>() << "\n";
16
17 buffer<double> bufA(vecA);
18 buffer<double> bufB(vecB);
19 buffer<double> bufC(vecC);
20
21 for (int i = 0; i < num_runs; i++) {
22     auto Q_event = Q.submit([&](handler& h) {
23         accessor A{ bufA, h };
24         accessor B{ bufB, h };
25         accessor C{ bufC, h };
26
27         h.parallel_for(array_size, [=](id<1> idx) {
28             C[idx] = A[idx] + B[idx] * scalar;
29         });
30     });
31
32     double exec_time_ns =
33         Q_event.get_profiling_info<info::event_profiling::command_end>() -
34         Q_event.get_profiling_info<info::event_profiling::command_start>();
35
36     std::cout << "Execution time (iteration " << i << ") [sec]: "
37         << (double)exec_time_ns * 1.0E-9 << "\n";
38     min_time_ns = std::min(min_time_ns, exec_time_ns);
39 }
40
41 return min_time_ns;
42 }
```

尽管并行内核非常类似于用带有循环的串行 STREAM Triad 函数，但在 CPU 上运行速度要快得多，因为 parallel_for 允许在多个内核上并行处理数组的不同元素。如图 16-7 所示，假设有一个系统，其中有一个插槽、四个内核和每个内核有两个超线程，有 1024 个双精度数据元素要处理。现实中，数据在包含 32 个数据元素的工作组中处理，这意味着有 8 个线程和 32 个工作组。工作组调度可以按循环顺序进行，即 thread-id = work-group-id mod 8。每个线程将执行四个工作组，每一轮可以并行执行 8 个工作组。注意，在本例中工作组是 DPC++ 编译器和运行时会隐式形成的。

图 16-7 STREAM Triad 内核并行的映射



注意，在 DPC++ 程序中不需要指定数据元素的分区，以及相应的处理器。这使得 DPC++ 可以灵活地选择，如何在特定的 CPU 上最好地并行内核。话虽如此，实现可以为程序员提供某种程度的控制，以便性能调优。

虽然 CPU 可能会带来相对较高的线程上下文切换和同步开销，但在处理器上开辟更多的线程是有益的，因为每个处理器核心提供了要执行的工作的选择。如果软件线程正在等待另一个线程产生数据，那么处理器可以切换到另一个准备运行的软件线程，而不会让处理器空闲。

选择如何绑定和调度线程

选择一种有效的方案来划分和调度线程之间的工作，对于在 CPU 和其他设备类型上调优非常重要。接下来的将描述这部分的技术。

线程的亲和力

线程亲和力指定特定线程可以在特定的 CPU 上执行。如果线程在多个核之间移动，性能可能会受到影响，例如：如果线程不在同一个核上执行，如果数据在核之间打乒乓球，则缓存的性能会变得很低。

DPC++ 运行时库支持通过环境变量 DPCPP_CPU CU_AFFINITY、DPCPP_CPU_PLACES、DPCPP_CPU_NUM_CUS 和 DPCPP_CPU_SCHEDULE 等将线程绑定到内核，但这都不是由 SYCL 定义。

首先是环境变量 DPCPP_CPU CU_AFFINITY。使用这些环境变量进行简单调优，并且对许多应用程序有很大影响。这个环境变量的描述如图 16-8 所示。

图 16-8 DPCPP_CPU CU AFFINITY 环境变量

DPCPP_CPU CU AFFINITY	描述
spread	按照循环顺序将连续线程绑定到从插槽 0 开始
close	以循环顺序将连续线程绑定到不同的超线程 (以线程 0 开始)

spread: $\text{boundHT} = (\text{tid mod numHT}) + (\text{tid mod numSocket}) \times \text{numHT}$

close: $\text{boundHT} = \text{tid mod}(\text{numSocket} \times \text{numHT})$

- tid 示软件线程标识符。
- boundHT 表示线程 tid 绑定到的超线程 (逻辑核芯)。
- numHT 表示每个插槽的超线程数。
- numSocket 表示系统中的插槽数量

假设在一个双核双插槽的超线程系统上，运行一个有 8 个线程的程序——换句话说，有 4 个核，总共有 8 个超线程要进行编程。图 16-9 展示了线程如何映射到不同 DPCPP_CPU CU AFFINITY 设置的超线程和核芯。

图 16-9 用超线程将线程映射到内核

DPCPP_CPU CU AFFINITY	socket0		socket1	
	core0	core1	core2	core3
spread	<T0, T4>	<T2, T6>	<T1, T5>	<T3, T7>
close	<T0, T1>	<T2, T3>	<T4, T5>	<T6, T7>

除了环境变量 DPCPP_CPU CU AFFINITY，还有其他支持 CPU 性能调优的环境变量：

- DPCPP_CPU_NUM_CUS = [n]，它设置用于内核执行的线程数。它的默认值是系统中的硬件线程数。
- DPCPP_CPU_PLACES = [sockets | numa_domains | cores | threads]，指定亲和性将被设置的位置，类似于 OpenMP 5.1 中的 OMP_PLACES。默认设置为“cores”。
- DPCPP_CPUSCHEDULE = [dynamic | affinity | static]，它指定了调度工作组的算法。默认设置是 dynamic。
 - dynamic: 启用 TBB auto_partitioner，通常可以平衡工作线程之间的负载。
 - affinity: 启用 TBB affinity_partitioner，可以改进缓存亲和性，并在将子范围映射到工作线程时使用相应的比例分割。
 - static: 启用 TBB static_partitioner，尽可能均匀地分布线程之间的负载。

TBB 使用粒度大小来控制工作拆分，默认粒度大小为 1，表示所有工作组都可以独立执行。更多信息可以在 spec.oneapi.com/versions/latest/elements/oneTBB/source/algorithms.html#partitioner 找到。

缺乏线程关联性调优并不一定性能会低。性能更多地取决于并行执行的线程总数，而不是线程和数据的关联和绑定程度。使用基准测试程序，是确定线程关联是否对性能有影响的一种方法。如图 16-1 所示，DPC++ STREAM Triad 代码在没有线程关联设置的情况下以较低的性能启动。通过控制亲和性设置和通过环境变量（输出如下所示）对软件线程进行静态调度，性能得到了改善：

```
export DPCPP_CPU_PLACES=numa_domains  
export DPCPP_CPU CU_AFFINITY=close
```

通过使用 numa_domains 对亲和性的位置进行设置，TBB 任务领域绑定到 numa 节点和插槽，并且任务均匀地分布在各个领域。一般情况下，环境变量 DPCPP_CPU_PLACES 建议与 DPCPP_CPU CU_AFFINITY 一起使用。这些环境变量设置可以在拥有 2 个插槽和 28 个双向超线程内核的 Skylake 服务器系统上实现了 30% 的性能提升，每个插槽中的核芯以 2.5GHz 运行。但是，我们还可以做得更好，可以进一步提高这个 CPU 的性能。

注意与内存的接触

示例中，初始化循环不是并行的，由主机线程串行执行，所有内存都与主机线程运行的任务相关联。其他任务随后的访问将连接到初始任务（用于初始化）的内存中的数据，这显然不符合性能要求。通过并行化初始化循环来控制跨任务与内存的第一次接触，可以在 STREAM Triad 内核上实现更高的性能，如图 16-10 所示。

图 16-10 STREAM Triad 并行初始化内核控制与内存的第一次接触

```
1 template <typename T>  
2 void init(queue &deviceQueue, T* VA, T* VB, T* VC, size_t array_size) {  
3     range<1> numItems{array_size};  
4  
5     buffer<T, 1> bufferA(VA, numItems);  
6     buffer<T, 1> bufferB(VB, numItems);  
7     buffer<T, 1> bufferC(VC, numItems);  
8  
9     auto queue_event = deviceQueue.submit([&](handler& cgh) {  
10         auto aA = bufA.template get_access<sycl_write>(cgh);  
11         auto aB = bufB.template get_access<sycl_write>(cgh);  
12         auto aC = bufC.template get_access<sycl_write>(cgh);  
13  
14         cgh.parallel_for<class Init<T>>(numItems, [=](id<1> wi) {  
15             aA[wi] = 2.0; aB[wi] = 1.0; aC[wi] = 0.0;  
16         });  
17     });  
18  
19     queue_event.wait();  
20 }
```

初始化代码中利用并行性可以提高内核在 CPU 上运行时的性能。在 Intel Xeon 处理器系统上，这样可以获得了 2 倍的性能增益。

本章最近的几节展示了通过利用线程级并行性，可以有效地利用 CPU 内核和超线程。然而，还需要利用 CPU 核心硬件中的 SIMD 向量级并行性，以实现最佳性能。

DPC++ 并行内核得益于跨核和超线程的线程级并行！

CPU 上的 SIMD 向量化

编写良好的 DPC++ 内核没有工作项依赖关系，就可以在 CPU 上高效地并行，还可以对 DPC++ 内核应用向量化，以利用 SIMD 硬件。实际上，CPU 处理器可以使用 SIMD 指令优化内存负载、存储和操作，因为大多数数据元素通常位于连续内存中，并且通过数据并行内核采用相同的控制流。例如，有 $a[i] = a[i] + b[i]$ 语句的内核中，通过在多个数据元素之间共享硬件逻辑，并将它们作为一个组执行，每个数据元素都以相同的指令流 load、load、add 和 store 操作，可以自然地映射到硬件的 SIMD 指令集。因此，一个指令可以同时处理多个数据元素。

由一条指令同时处理的数据元素的数量，有时称为指令或执行指令的处理器的向量长度（或 SIMD 宽度）。图 16-11 中，指令流以 4 路 SIMD 执行运行。

图 16-11 SIMD 的指令流

Serial execution				SIMD execution
work-0	work-1	work-2	work 3	vector sub-group
load r0, a[0]	load r0, a[1]	load r0, a[2]	load r0, a[3]	simdload vr0, a[0...3]
load r1, b[0]	load r1, b[1]	load r1, b[2]	load r1, b[3]	simdload vr1, b[0...3]
add r0, r1	add r0, r1	add r0, r1	add r0, r1	simddadd vr0, vr1
store a[0], r0	store a[1], r0	store a[2], r0	store a[3], r0	simdstore a[0...3], vr0

CPU 处理器并不是唯一实现 SIMD 指令集的处理器。其他处理器（如 GPU）实现 SIMD 指令以提高处理大型数据集的效率。与其他处理器类型相比，Intel Xeon CPU 处理器的区别是有三个固定大小的 SIMD 寄存器宽度（128 位 XMM、256 位 YMM 和 512 位 ZMM），而不是一个可变长度的 SIMD。当使用子工作组或向量类型编写具有 SIMD 并行性的 DPC++ 代码时，需要注意硬件中的 SIMD 宽度和 SIMD 向量寄存器的数量。

确保 SIMD 执行的合法性

DPC++ 执行模型确保 SIMD 执行可以应用于任何内核，以及每个工作组中的一组工作项（即一个子工作组）可以使用 SIMD 指令并发执行。有些实现可以选择使用 SIMD 指令在内核中执行循环，只有保留所有原始数据依赖关系，或者保留由编译器基于语义解析的数据依赖关系才有可能。

使用工作组内的 SIMD 指令，可以将单个 DPC++ 内核执行从单个工作项的处理转换为一组工作项。在 ND-Range 模型下，生成 SIMD 代码的编译器向量器选择了增长最快的（单位步幅）维度。实际上，要启用给定 ND-Range 的向量化，在同一子工作组中的任何两个工作项之间，不应该存在依赖关系，或者编译器需要在同一子工作组中保留工作项向前依赖关系。

当工作项的内核执行映射到 CPU 上的线程时，细粒度同步的代价很高，线程上下文切换的开销也很高。因此，为 CPU 编写 DPC++ 内核时，消除工作组内工作项之间的依赖对性能优化很重

要。另一种有效的方法是依赖子工作组中的工作项，如图 16-12 中“先读后写”的依赖。如果子工作组是在 SIMD 执行模型下执行的，那么编译器可以将内核中的子工作组栅栏忽略，在运行时不会产生同步成本。

图 16-12 使用子工作组向量化具有前向依赖性的循环

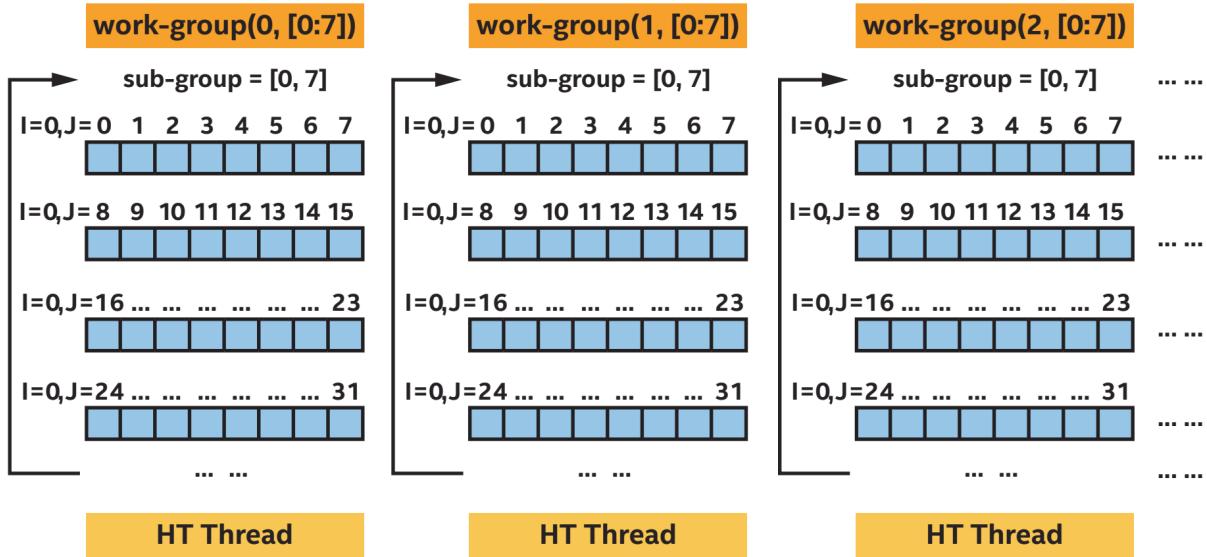
```
1 using namespace sycl::intel;
2
3 queue Q;
4 range<2> G = {n, w};
5 range<2> L = {1, w};
6
7 int *a = malloc_shared<int>(n*(n+1), Q);
8
9 for (int i = 0; i < n; i++)
10   for (int j = 0; j < n+1; j++) a[i*n + j] = i + j;
11
12 Q.parallel_for(nd_range<2>{G, L}, [=](nd_item<2> it)
13   [[ cl::intel_reqd_sub_group_size(w) ]] {
14
15   // distribute uniform "i" over the sub-group with 8-way
16   // redundant computation
17   const int i = it.get_global_id(0);
18   sub_group sg = it.get_sub_group();
19
20   for (int j = sg.get_local_id(0); j < n; j += w) {
21     // load a[i*n+j+1:8] before updating a[i*n+j:8] to preserve
22     // loop-carried forward dependence
23     auto va = a[i*n + j + 1];
24     sg.barrier();
25     a[i*n + j] = va + i + 2;
26   }
27   sg.barrier();
28 }
29 }).wait();
```

内核向量化 (向量长度为 8)，其 SIMD 执行如图 16-13 所示。工作组的规模为 (1,8)，内核的循环迭代分布在这些子工作组工作项上，并以 8 路 SIMD 并行方式执行。

本例中，如果内核中的循环控制性能，那么允许跨子工作组的 SIMD 向量化将有显著的性能改进。

使用并行处理数据元素的 SIMD 指令，是让内核的性能超出 CPU 内核和超线程数量的方法。

图 16-13 具有前向依赖关系的循环的 SIMD 向量化



SIMD 掩码和成本

实际应用中，我们可以期待条件语句，如 if 语句，条件表达式，如 $a = b > a?A: b$ ，迭代次数可变的循环，switch 语句等等。任何条件可能导致标量控制流不执行相同的代码路径，就像在 GPU(第 15 章)，可能会导致性能下降。SIMD 掩码是一组值为 1 或 0 的位，由内核中的条件语句生成。考虑一个例子， $A=1,2,3,4$, $B=3,7,8,1$ ，以及比较表达式 $A < B$ 。比较返回一个掩码，包含四个值 1,1,1,0，可以存储在硬件掩码寄存器中，以指示以后哪些通道的 SIMD 指令应该执行比较所保护(使能)的代码。

内核包含条件代码，与基于与每个数据元素相关联的掩码位 (SIMD 指令中的通道) 执行向量化的掩码指令。每个数据元素的掩码位与掩码寄存器中的位置相对应。

使用掩码可能会导致性能低于相应的非掩码代码。这可能是由于以下原因：

- 负载上附加了掩码操作
- 对目标的依赖

掩码是有成本的，所以只在必要时使用。当内核是具有执行范围内工作项显式分组的 ND-Range 内核时，在选择 ND-Range 工作组大小，通过最小化掩码成本来最大限度地提高 SIMD 效率时应格外小心。当工作组的大小不能被处理器的 SIMD 宽度均匀整除时，工作组的一部分可能会被内核忽略。

图 16-14 内核中使用的三种掩码

No Masking	Merge Masking	Zero Masking
vmulps zmm0, zmm6, zmm8	vmulps zmm0{k1}, zmm6, zmm8	vmulps zmm0{k1}{z}, zmm6, zmm8
vmulps zmm1, zmm7, zmm8	vmulps zmm1{k1}, zmm7, zmm8	vmulps zmm1{k1}{z}, zmm7, zmm8
Baseline	Slowdown 4x	Slowdown 1x

图 16-14 显示了如何使用合并掩码创建一个依赖于目标寄存器：

- 如果没有掩码，处理器每个周期执行两个乘法 (vmulps)。
- 合并掩码时，处理器每四个周期执行两次乘法，因为乘法指令 (vmulps) 保存在目的寄存器中，如图 16-17 所示。
- 零屏蔽不依赖于目标寄存器，因此在每个周期执行两个乘法 (vmulps)。

访问缓存对齐的数据比访问非对齐的数据具有更好的性能。地址在编译时未知，或者是未对齐的，这种情况下，可以剥离对内存的访问，使用掩码访问处理前几个元素，直到第一个对齐的地址，然后通过并行内核中的多版本控制技术处理未掩码的访问，随后是忽略剩余数。这种方法增加了代码大小，但从整体上改善了数据处理性能。

避免使用结构数组来提高 SIMD 效率

AOS(结构数组) 结构会影响 SIMD 的效率，也会为内存访问带来额外的带宽和延迟。硬件聚集-分散机制的存在并不能消除这种转换的需求——聚集-分散访问通常需要比连续负载更高的带宽和延迟。给定一个 AOS 数据布局 struct {float x; float y; float z; float w;} a[4]，考虑一个内核在它上面操作，如图 16-15 所示。

图 16-15 SIMD 在内核中聚集

```

1 cgh.parallel_for<class aos<T>>(numOfItems, [=](id<1> wi) {
2     x[wi] = a[wi].x; // lead to gather x0, x1, x2, x3
3     y[wi] = a[wi].y; // lead to gather y0, y1, y2, y3
4     z[wi] = a[wi].z; // lead to gather z0, z1, z2, z3
5     w[wi] = a[wi].w; // lead to gather w0, w1, w2, w3
6 });

```

当编译器沿着一组工作项向量化内核时，由于是非单位步长的内存访问，会导致 SIMD 收集指令的生成。例如， $a[0].x, a[1].x, a[2].x$ 和 $a[3].x$ 的跨距是 4，不是更有效的步幅 1。

w_3	z_3	y_3	x_3	w_2	z_2	y_2	x_2	w_1	z_1	y_1	x_1	w_0	z_0	y_0	x_0
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

内核中，通常可以通过消除对内存收集-分散操作的使用来实现更高效的 SIMD。一些代码可以从数据布局中获益，这些将以结构数组 (Array-of-Struct, AOS) 表示形式编写的数据结构转换为阵列结构 (Structure of Arrays, SOA)，在执行 SIMD 向量化时，为每个结构字段使用单独的数组以保持内存访问的连续性。例如，考虑这样一个 SOA 数据布局:struct float x[4]; float y[4]; float z[4]; float w[4]; a; 如下所示:

w_3	w_2	w_1	w_0	z_3	z_2	z_1	z_0	y_3	y_2	y_1	y_0	x_3	x_2	x_1	x_0
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

内核可以使用如图 16-16 所示的单位步长 (连续) 向量加载和存储数据，即使是向量化！

图 16-16 SIMD 操作单位跨距的内核

```

1 cgh.parallel_for<class aos<T>>(numOfItems, [=](id<1> wi) {
2     x[wi] = a.x[wi]; // lead to unit-stride vector load x[0:4]
3     y[wi] = a.y[wi]; // lead to unit-stride vector load y[0:4]
4     z[wi] = a.z[wi]; // lead to unit-stride vector load z[0:4]
5     w[wi] = a.w[wi]; // lead to unit-stride vector load w[0:4]
6 });

```

SOA 数据布局有助于在跨数组元素访问结构的一个字段时防止聚集，并帮助编译器对与工作项相关的连续数组元素上的内核进行向量化。考虑到使用这些数据结构的地方，希望在程序级别完成这些 AOS-to-SOA 或 AOSOA 的数据布局转换。仅在循环级别上执行此操作将涉及循环前后格式之间的转换。我们还可以依赖编译器对 AOS 数据布局进行向量加载和混洗优化，但要付出一定的代价。如果 SOA(或 AOS) 数据布局的成员具有向量类型，那么编译器向量化将执行第 11 章中描述的基于底层硬件的水平扩展或垂直扩展，以生成最佳代码。

数据类型对 SIMD 效率的影响

只要 C++ 开发者知道数据适合 32 位带符号的类型，就会使用整数数据类型，经常出现以下的代码

```

int id = get_global_id(0);
a[id] = b[id] + c[id];

```

假设 `get_global_id(0)` 的返回类型是 `size_t`(无符号整数，通常是 64 位)，这种转换降低了编译器的可优化性。

- 读取 `[get_global_id(0)]` 会导致 SIMD 单位跨距的向量负载。
- 读取 `[(int)get_global_id(0)]` 会导致形成非单元跨距的收集指令。

这种微妙的情况是由从 `size_t` 到 `int`(或 `uint`) 的数据类型转换行为(未指定的行为 C/C++ 标准中定义良好的转换行为)引起的，这主要是基于 C 语言进化的历史产物。具体来说，某些转换中的溢出是未定义的行为，这实际上允许编译器假定这种情况永远不会发生，并更积极地进行优化。图 16-17 显示了一些希望了解细节的示例。

图 16-17 整型值范围

<code>get_global_id(0)</code>	<code>a[(int)get_global_id(0)]</code>	<code>get_globalid(0)</code>	<code>a((uint)get_global_id(0))</code>
<code>0x7FFFFFFE</code>	<code>a[MAX_INT-1]</code>	<code>0xFFFFFFFFFE</code>	<code>a[MAX_UINT-1]</code>
<code>0x7FFFFFFF</code>	<code>a[MAX_INT (big positive)]</code>	<code>0xFFFFFFFFFF</code>	<code>a[MAX_UINT]</code>
<code>0x80000000</code>	<code>a[MIN_INT (big negative)]</code>	<code>0x100000000</code>	<code>a[0]</code>
<code>0x80000001</code>	<code>a[MIN_INT+1]</code>	<code>0x100000001</code>	<code>a[1]</code>

SIMD 收集/分散指令比 SIMD 单元跨矢量加载/存储操作慢。为了最佳的 SIMD 效率，避免聚集/分散对于应用程序来说是至关重要的，无论使用哪种编程语言。

大多数 SYCL `get_*_id()` 函数有相同的问题，尽管许多情况下适用于 MAX_INT，因为返回值是有界的（例如，一个工作组中的最大 id）。只要是合法的，DPC++ 编译器就会假定跨相邻工作项的单元跨内存地址，以避免聚集/分散。由于全局 id 和/或全局 id 的值可能溢出，编译器无法安全地生成单位跨距向量内存加载/存储操作，编译器将生成聚集/散点操作。

为用户提供最佳性能的理念下，DPC++ 编译器假定没有溢出，并且在实践中捕获真实状态，因此编译器可以生成最佳的 SIMD 代码以获得良好的性能。`D__SYCL_DISABLE_ID_TO_INT_CONV__` 是 DPC++ 编译器用来告诉编译器会发生溢出的宏，使用 `id` 查询向量的访问可能不安全，这可能会对性能产生很大影响。这个宏应该在不安全的情况下使用，以假定没有溢出发生。

使用 `single_task` 执行 SIMD

单个任务执行模型中，向量类型和函数相关的优化依赖于编译器。编译器和运行时可以自由地支持显式 SIMD 执行或在 `single_task` 内核中选择标量执行，结果取决于编译器实现。例如，DPC++ CPU 编译器将为向量类型生成 SIMD 指令。`vec` 加载、存储和 `swizzle` 函数将直接对向量变量执行操作，通知编译器数据元素正从内存中的相同（统一）位置开始访问连续数据，并能够请求优化的连续数据加载/存储。

图 16-18 `single_task` 内核中使用向量类型和混合操作

```
1 queue Q;
2 bool *resArray = malloc_shared<bool>(1, Q);
3 resArray[0] = true;
4
5 Q.single_task( [=](){
6     sycl::vec<int, 4> old_v = sycl::vec<int, 4>(000, 100, 200, 300);
7     sycl::vec<int, 4> new_v = sycl::vec<int, 4>();
8
9     new_v.rgba() = old_v.abgr();
10    int vals[] = {300, 200, 100, 000};
11
12    if (new_v.r() != vals[0] || new_v.g() != vals[1] ||
13        new_v.b() != vals[2] || new_v.a() != vals[3]) {
14        resArray[0] = false;
15    }
16 }) . wait();
```

如图 16-18 所示，执行单个任务时，声明了一个包含三个数据元素的 `vector`。使用 `old_v.abgr()` 执行混合操作。如果 CPU 为混合操作提供 SIMD 硬件指令，那么可以通过在程序中使用混合操作获得一些性能收益。

SIMD 向量化指南

CPU 处理器用不同的 SIMD 宽度实现 SIMD 指令集。这是一个实现细节，并且对在 CPU 上执行内核的应用程序是透明的，因为编译器可以确定要用特定的 SIMD 大小处理的一组有效数据元素，而不要求显式地使用 SIMD 指令。子工作组可以更直接地表示数据元素的分组，应受内核中 SIMD 执行的影响。

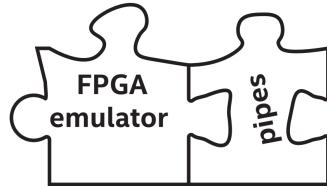
考虑到计算复杂度，选择最适合向量化的代码和数据布局最终可能会带来更高的性能收益。选择数据结构时，尽量选择数据布局、对齐方式和数据宽度，使最频繁执行的计算能够以对 SIMD 友好的方式访问内存，并具有最大的并行性。

总结

为了充分利用 CPU 上的线程级并行性和 SIMD 向量级并行性，需要牢记以下目标：

- 熟悉所有类型的 DPC++ 并行性和相应 CPU 的底层架构。
- 在最匹配硬件资源的线程级别上，不要增加或减少正确的并行度。使用供应商工具，如调试器和分析器，来帮助指导我们的调优工作，以实现这一点。
- 首先要注意线程关联性和内存对程序性能的影响。
- 使用数据布局、对齐和数据宽度设计数据结构，以便最频繁执行的计算能以 SIMD 友好的方式访问内存，并具有最大的 SIMD 并行性。
- 要注意平衡掩码和代码分支的成本。
- 使用清晰的编程风格，最大限度地减少潜在的内存混叠和副作用。
- 注意使用向量类型和接口的可扩展性限制。如果编译器将它们映射到 SIMD 指令，那么跨多代 CPU 和来自不同供应商的 CPU 的固定向量大小，可能无法很好地与 SIMD 寄存器宽度适配。

17 FPGA 编程



基于内核的编程最初是作为一种访问 GPU 的方式而流行起来的。由于现在已经在许多加速器中得到了推广，理解编程风格如何影响代码到 FPGA 的映射非常重要。

对大多数软件开发人员来说，对于可编程门阵列 (FPGA) 并不熟悉，部分原因是大多数桌面计算机在典型的 CPU 和 GPU 之外没有 FPGA。FPGA 的确很值得了开发人员来了解，因为它在很多应用中都有优势。和其他加速器一样，也需要问同样的问题，比如“什么时候使用 FPGA?”，“应用程序的哪些部分应该加载到 FPGA?”，以及“如何编写在 FPGA 上表现良好的代码?”

本章为我们提供了开始回答这些问题的知识，至少可以确定对 FPGA 是否感兴趣，并知道哪些构造可用于实现性能。我们可以阅读供应商文档来了解特定产品和工具链的详细信息。首先概述程序如何映射到 FPGA 等空间架构，然后讨论 FPGA 作为加速器的一些特性，最后介绍实现性能的编程结构。

本章的“如何使用 FPGA”一节适用于任何 FPGA。SYCL 允许供应商指定 CPU 和 GPU 之外的设备，但没有具体说明如何支持 FPGA。目前，对于特定的 FPGA 支持是 DPC++ 的优势，即可以使用 FPGA 选择器和管道。FPGA 选择器和管道是本章中使用的 DPC++ 扩展。我们希望厂商能够使用类似或兼容的方式来支持 FPGA，而 DPC++ 作为一个开源项目非常鼓励这种做法。

性能说明

与任何处理器或加速器一样，FPGA 设备因厂商而异，甚至因产品而异，因此对于一种设备的最佳实践可能并不适用于不同的设备。这一章的建议可能会使许多 FPGA 设备受益，不管是现在还是将来……

……为了实现特定 FPGA 的最佳性能，请始终参考供应商提供的文档!

FPGA 的工作原理

FPGA 通常归类为空间架构，其受益于与使用指令集体体系结构 (ISA) 的设备（包括大多数人更熟悉的 CPU 和 GPU），从而形成非常不同的编码风格和并行形式。为了理解 FPGA，我们将简要介绍基于 ISA 的加速器的一些思想。

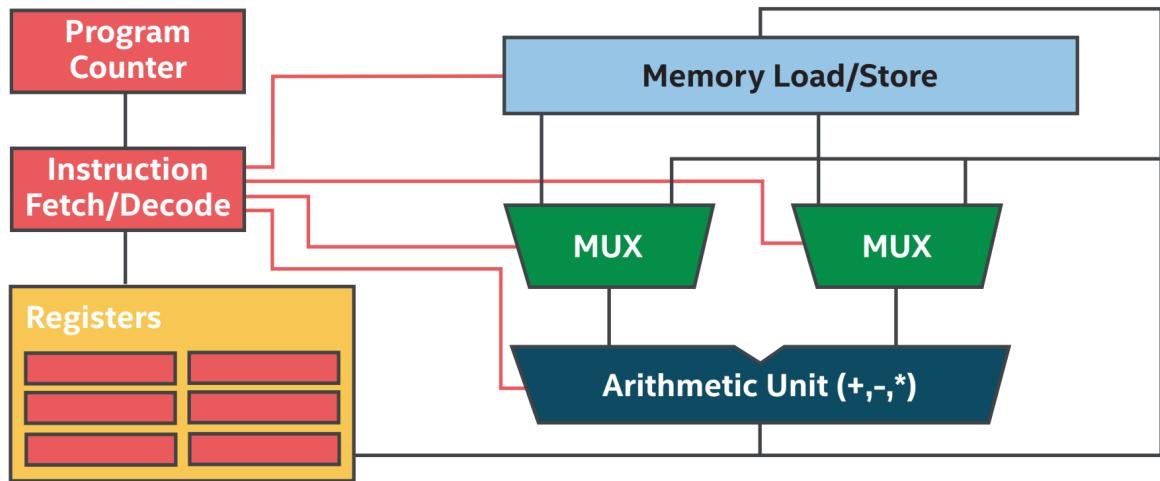
就目的而言，基于 ISA 的加速器设备可以执行许多不同的指令。这些指令通常比较简单，比如“从地址 A 的内存中加载数据”或“添加一组数字”。操作链串在一起形成程序，处理器就一条一条的执行指令。

基于 ISA 的加速器中，芯片的单个区域（或整个芯片）在每个时钟周期中执行来自程序的不同指令。这些指令在固定的硬件架构上执行，不同的硬件架构可以在不同的时间运行不同的指令，如图 17-1 所示。输入加法的内存负载单元可能与输入减法的内存负载单元相同，同样的算术单元可

能用于执行加法和减法指令。随着时间的推移，随着程序的执行，芯片上的硬件可以重用不同的指令。

图 17-1 简单的基于 ISA 的 (临时) 处理: 随着时间的推移重用硬件 (区域)

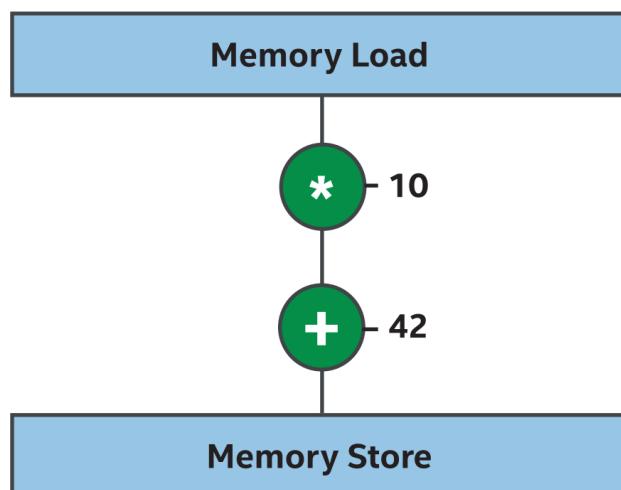
Simple SA-based Accelerator



由于空间架构不同，它们不是基于在共享硬件上执行各种指令的芯片，而是从相反的角度出发。程序空间在概念上把程序作为一个整体，并立即放在设备上执行，设备的不同区域在程序中执行不同的指令。空间架构中，专用硬件会接受到相应的指令，这些硬件可以与其他硬件同时执行（相同的时钟周期）。图 17-2 展示了这种思想，其为整个程序（本例中是一个非常简单的程序）的“空间”实现。

图 17-2 空间处理: 每个操作使用设备的不同区域

Simple Spatial Compute Machine



这个描述过于简单，但在空间架构、程序的不同部分的不同设备上执行，而不是随着时间变化而变化。

由于 FPGA 的不同区域可以编程执行不同的操作，一些与基于 ISA 的加速器相关联的硬件则没必要这样做。例如，图 17-2 显示不再需要指令获取或解码单元、程序计数器或寄存器文件。空间体系结构将一条指令的输出与另一条指令的输入相连接，而不是将数据存储在寄存器文件中，这就是为什么空间体系结构通常称为数据流体系结构。

介绍的 FPGA 映射时，出现了几个问题。首先，由于程序中的每条指令都占用设备空间面积一定的百分比，如果程序需要超过 100% 的面积会发生什么？一些解决方案提供资源共享机制，使更大的程序能够以性能成本来适应，但 FPGA 又有程序适应的概念。这些既是优点也是缺点：

- 好处：如果程序使用了 FPGA 上的大部分区域，并且在每个时钟周期中有足够的工作来保持所有硬件繁忙，由于极端的并行性，在设备上执行程序非常高效。更通用的体系可能在时钟周期中有大量未使用的硬件。使用 FPGA 时，可以为特定应用程序完美地定制面积，不会浪费。这种定制可以让应用程序通过大规模并行运行得更快，通常可以提高能源利用的效率。
- 缺点：大型程序可能需要调整和重组才能适应设备。编译器的资源共享特性可以帮助解决这个问题，但通常会使性能下降，从而降低使用 FPGA 的好处。基于 ISA 的加速器是非常有效的资源共享实现——FPGA 对计算有价值，主要是程序可以利用大多数可用区域。

极端的情况下，FPGA 上的资源共享解决方案会看起来像基于 ISA 的加速器，在可重构逻辑中构建。可重构逻辑导致相对于固定设计的开销大——因此，FPGA 通常不作为实现 ISA 的方法。当应用程序能够利用资源来实现高效的数据流算法时，FPGA 是最有利的，将在下一节中讨论这些算法。

管道并行性

图 17-2 中经常出现的另一个问题是，程序的空间实现如何与时钟频率相关，以及程序从开始到结束执行的速度。示例中，很容易看出数据可以很快地从内存中加载，执行乘法和加法运算，并将结果存储回内存中。随着程序变得越来越大，可能在 FPGA 设备上有成千上万的操作，所有的指令都要一个接一个地进行操作（操作通常取决于前一个操作产生的结果），考虑到每个操作带来的处理延迟，可能会花费大量的时间。

如图 17-3 所示的空间体系结构中，操作之间的中间结果会随着时间的推移而更新（传播）。例如，load 执行后将其结果传递给乘数器，乘数器的结果再传递给加法器，以此类推。一段时间后，中间数据一直传播到操作链的末端，最终结果可用或存储到内存中。

图 17-3 空间计算实现的传播时间

Simple Spatial Compute Machine

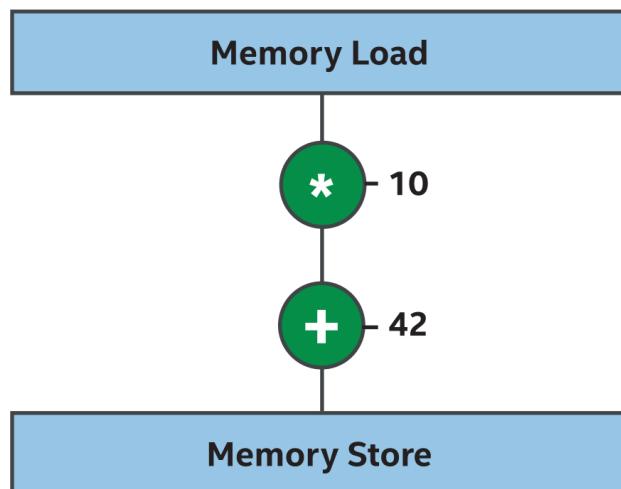
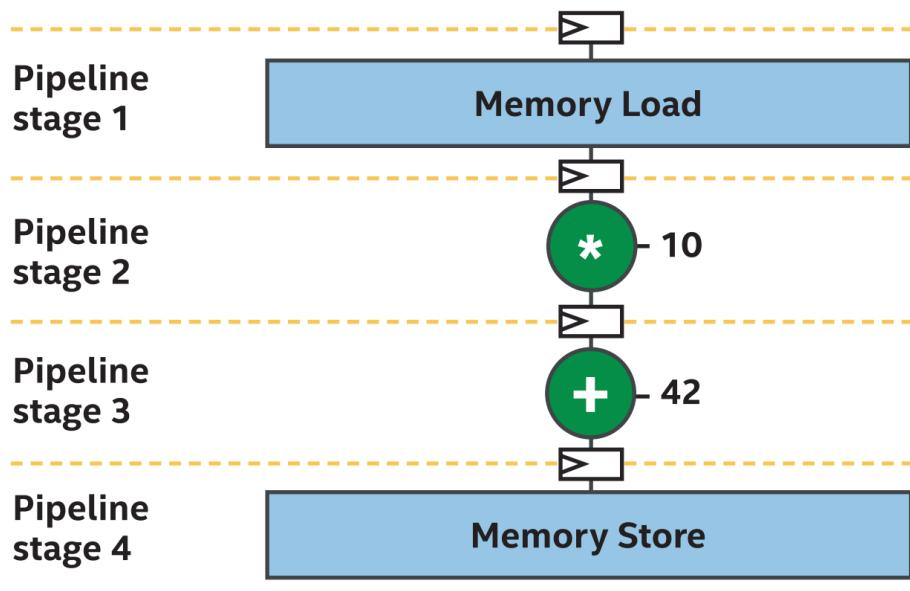


图 17-3 所示的空间实现非常低效，大多数硬件只在一小部分时间内执行有用的工作。大多数情况下，像乘法这样的操作要么等待加载中的新数据，要么保持其输出，以便让后续操作使用其结果。大多数空间编译器和实现通过流水线处理这种低效操作，这样单个程序的执行分散在多个时钟周期中。这是通过在一些操作之间插入寄存器（硬件中的数据存储原语）实现的，其中每个寄存器在一个时钟周期内保存一个二进制值。通过保存操作的输出结果，以便让下一个操作可以看到并对所保存的值进行操作，前一个操作可以自由地对不同的计算进行操作，而不会影响后续操作的输入。

算法流水线的目标是使每个操作（硬件单元）在每个时钟周期中处于繁忙状态。图 17-4 显示了前面简单示例的流水线实现。编译器会完成所有的流水线和平衡工作！我们讨论这个主题是为了在接下来的章节中理解如何用工作填充流水线。

图 17-4 计算流水线：各个阶段并行执行

Pipelined Spatial Compute



实现流水线化时，类似工厂装配线的方式就会让工作变得非常高效。每个管道阶段只执行总体工作的一小部分，执行得很快，然后立即开始处理下一个工作单元。管道从开始到结束处理单个计算需要许多时钟周期，但是管道可以同时计算不同数据上的不同计算实例。

当足够多的工作在流水线中执行，经过了足够的连续时钟周期，那么每个流水线阶段和程序中的操作都可以在时钟周期中执行有用的工作，这样整个空间设备同时执行工作。这是空间架构的力量——整个设备可以在任何时候并行执行。我们称之为流水线并行性。

流水线并行是 FPGA 上用来实现性能的并行的主要形式。

自动型流水线

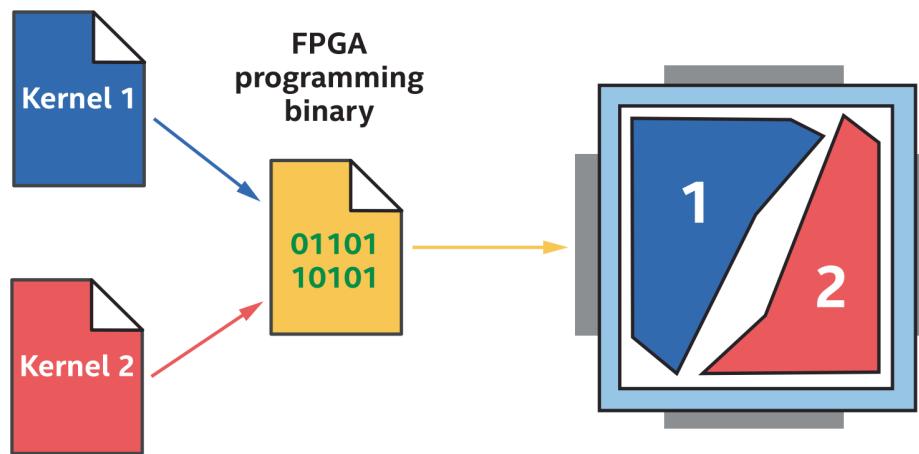
在用于 FPGA 的 DPC++ 的 Intel 实现中，以及用于 FPGA 的其他高级编程解决方案中，算法的流水线是由编译器自动执行。大致理解空间体系结构上的实现是很有用，这样就可以更容易地利用流水线并行性。流水线寄存器的使用平衡由编译器控制，而不是由开发控制。

真正的程序和算法通常有控制流（例如，if/else 结构），这会使程序的某些部分在某些时钟周期中处于非活动状态。FPGA 编译器通常会结合分支两边的硬件，尽可能减少浪费的空间面积，并在控制流发散期间最大化计算效率。这使得控制流分歧的成本大大降低，而且与其他方面（特别是向量化的体系结构）相比，开发上的关注点也更少。

消费核心——芯片“区域”

现有的实现中，DPC++ 应用程序中的每个内核都会生成空间管道，消耗 FPGA 的一些资源（可以将其看作是设备上的空间或区域），如图 17-5 所示。

图 17-5 同一个 FPGA 二进制文件中的多个内核：内核可以并发运行



内核在设备上使用自己的区域，不同的内核可以并发执行。当内核正在等待内存访问之类的操作，FPGA 上的其他内核可以继续执行，因为芯片上有其他独立的管道。这种思想形式上描述为内核之间的前向进程，是 FPGA 空间计算的关键。

何时使用 FPGA

与任何加速器架构一样，预测 FPGA 何时是正确选择，何时需要替代方案，通常取决于对架构、应用程序特征和系统瓶颈的了解。

海量任务

大多数现代的计算机加速器，想要获得良好的性能需要大量的工作。单个数据元素计算单个结果，对于加速器是没有加速效果的。这与 FPGA 一样，了解了 FPGA 编译器利用了流水线并行性，这一点就更加明显了。算法的流水化实现有很多阶段，通常是上千个甚至更多，每个阶段在任何时钟周期内都有不同的工作。如果没有足够的工作在占用流水线的大部分阶段，那么执行效率就会很低。

有多种方法可以在 FPGA 上生成工作来填充流水线阶段，我们将在接下来的章节中介绍这些方法。

自定义操作或操作宽度

FPGA 最初用来执行整数和位操作，并作为逻辑粘合剂，可以使其他芯片的接口工作。虽然 FPGA 已经发展为计算功能强大的设备，已不仅仅是粘合逻辑解决方案，但在位操作、自定义数据宽度或类型上的整数数学操作，以及对包头信息中的任意位字段的操作方面仍然非常高效。

本章末尾描述的 FPGA 的细粒度架构意味着可以有效地实现新的任意数据类型。例如，如果需要 33 位整数乘法器或 129 位加法器，FPGA 可以以很高的效率提供这些操作。由于这种灵活性，FPGA 通常用于快速发展的领域，例如：机器学习，其中数据宽度和操作的变化速度比内置在专用集成电路中要快。

标量数据流

从图 17-4 可以明显看出，FPGA 空间流水线的一个重要方面是，操作之间的中间数据不仅停留在芯片上（没有存储到外部内存），而且每个管道阶段之间的中间数据都有专用的存储寄存器。FPGA 的并行性来自于流水线，同时执行许多操作，每个操作在流水线的不同阶段。这与向量体系结构不同，在向量体系结构中，多个计算作为共享向量指令而执行。

空间流水线中并行性的标量性质，对于许多应用程序都很重要，即使在跨工作单元的数据依赖情况下仍然适用。可以在不损失性能的情况下处理这些数据依赖关系，我们将在本章后面讨论循环中的依赖关系时对此话题继续进行讨论。空间流水线对于不能破坏跨工作单元（如工作项）的数据依赖，并且可以进行细粒度通信。其他加速器的许多优化技术都侧重于打破这些依赖关系，或者通过子工作组等特性在可控范围内管理通信。相反，FPGA 可以很好地处理依赖与通信，应该考虑在这种模式的算法中使用 FPGA。

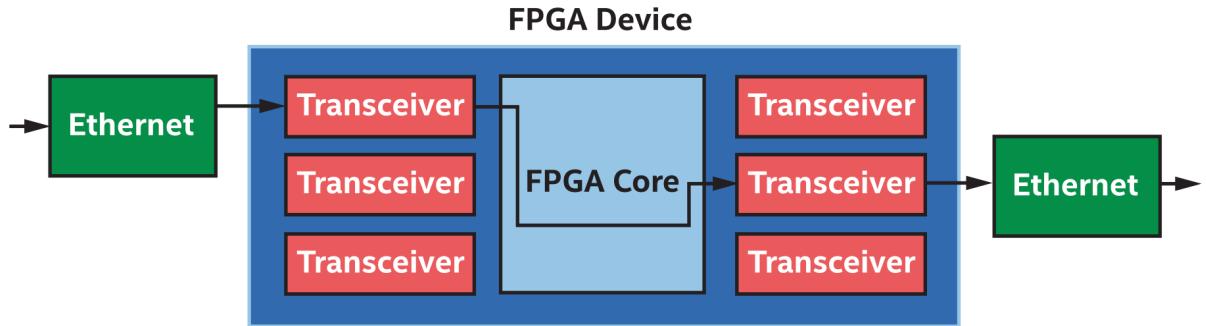
循环不是罪！

对于数据流体系结构的一个常见误解是，具有固定或动态迭代计数的循环会导致糟糕的数据流性能，因为不是简单的前馈流水。至少对于 Intel DPC++ 和 FPGA 工具链来说，这不正确。相反，循环迭代是在流水线中产生高占用率的一种好方法，而编译器是以重叠的方式执行多个循环迭代构建的。循环提供了一种简单的机制来保持流水线忙于工作！

低延迟和富连接

利用设备上丰富的输入和输出收发器的 FPGA 的更传统的使用同样适用于使用 DPC++ 的开发人员。例如，如图 17-6 所示，一些 FPGA 加速卡具有网络接口，可以将数据直接流进设备，进行处理后将结果直接流回网络。当需要最小化处理延迟、通过操作系统网络栈进行的处理太慢或需要加载时，通常会寻求这种系统。

图 17-6 低延迟 I/O 流:FPGA 连接网络数据和计算

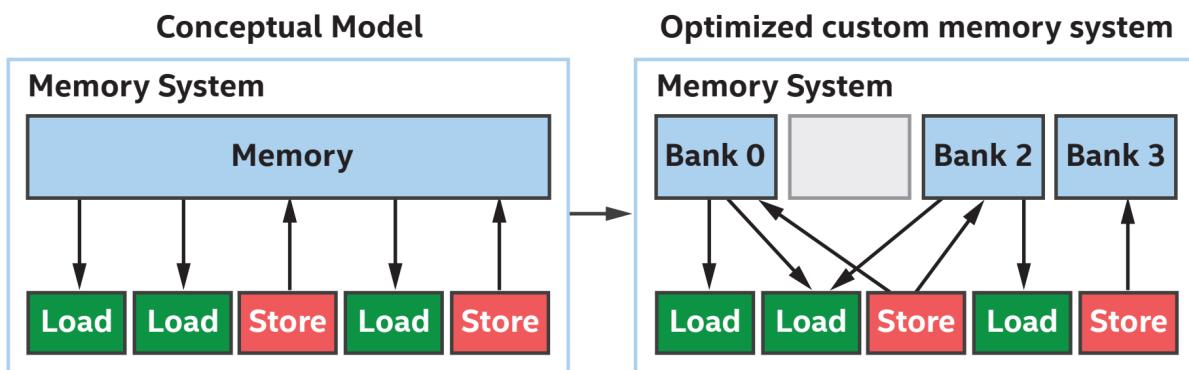


当考虑通过 FPGA 收发器直接输入/输出时，机会有很多，但选择确实取决于加速器上可用的东西。由于依赖于特定的加速卡和各种场景，除了在下一节中描述管道语言构造外，本章不深入研究这些程序。相反，我们应该阅读与特定加速卡相关的供应商文档，或者搜索与特定接口需求相匹配的加速卡。

定制的内存系统

FPGA 上的存储系统，私有存储器或工作组本地存储器，是由片上存储构建。每个内存系统都是为使用它的算法或内核的特定部分定制构建的。FPGA 具有显著的片上存储带宽，并且结合形成定制存储器，可以在具有非典型存储器访问模式和结构的应用程序上表现得很好。图 17-7 显示了在 FPGA 上实现内存系统时，编译器可以执行的一些优化。

图 17-7 FPGA 存储系统是由编译器为特定代码定制的



其他架构（如 GPU）具有固定的内存结构，这很容易使有经验的开发人员理解，但在许多情况下很难进行优化。例如，其他加速器的许多优化都集中在内存模式修改，以避免内存块冲突。如果

算法能够从自定义的内存结构中获益，比如每个内存块的访问端口数量不同，或者内存块数量异常，那么 FPGA 就有直接的优势。从概念上讲，两者的区别在于编写代码高效地使用固定的内存系统（大多数其他加速器），而让编译器定制内存系统，来提升特定代码的性能（FPGA）。

在 FPGA 上运行程序

在 FPGA 上运行内核有两个步骤（与提前编译一样）：

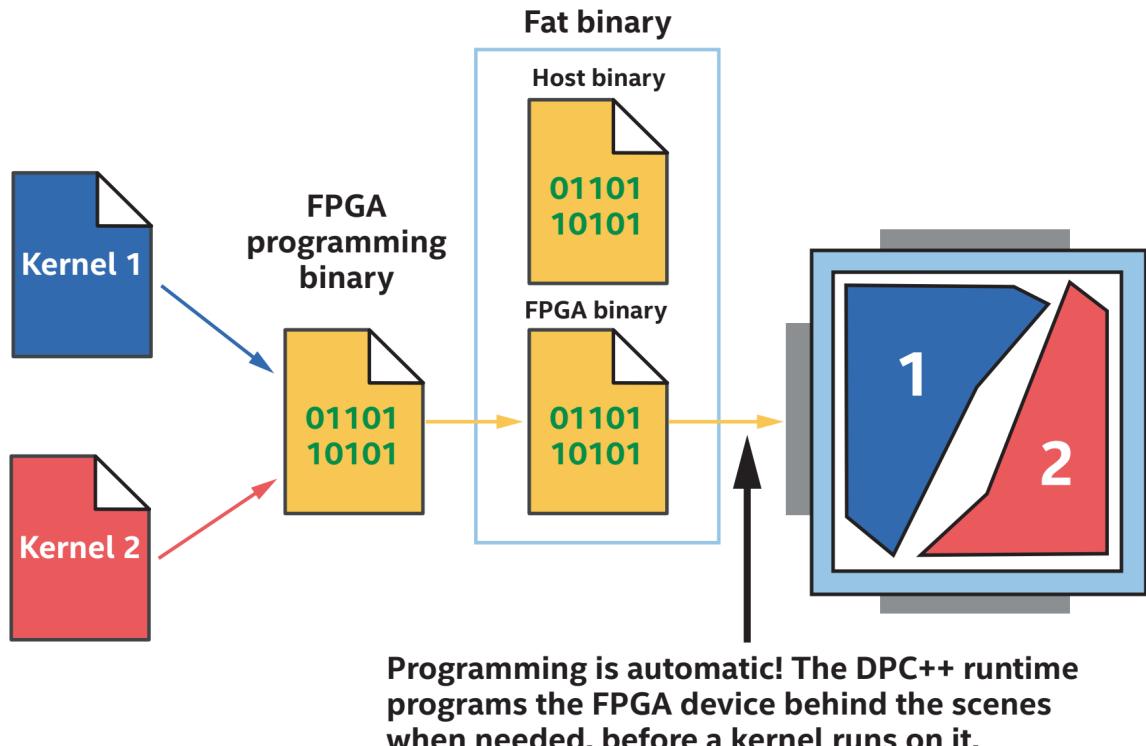
1. 将源代码编译成二进制文件，可以在感兴趣的硬件上运行
2. 运行时选择感兴趣的加速器

为了编译内核在 FPGA 硬件上运行，可以使用命令行：

```
dpcpp -fintelfpga my_source_code.cpp -Xhardware
```

这个命令告诉编译器将 my_source_code.cpp 中的所有内核转换为可以在 Intel FPGA 加速机上运行的二进制文件，然后打包到主机二进制文件中。当执行主机二进制文件时（例如，在 Linux 上运行 ./a.out），在执行提交的内核之前，运行时将根据需要为 FPGA 自动编程，如图 17-8 所示。

图 17-8 FPGA 在运行时的自动编程



FPGA 编程二进制文件嵌入在主机上运行和编译的 DPC++ 可执行文件中。FPGA 在后台进行自动配置。

当运行主程序并在 FPGA 上提交内核执行时，在内核开始执行之前可能会有一个延迟。重新提交内核执行不会出现相同的延迟，因为内核已经编程到设备中，并准备运行。

运行时选择 FPGA 设备在第 2 章中讨论。需要告诉主程序希望内核在哪里运行，通常会有多个加速器选项可用，比如 CPU 和 GPU，除了 FPGA。为了快速回顾在程序执行期间选择 FPGA 的方法，可以使用如图 17-9 所示的代码。

图 17-9 使用 fpga_selector 在运行时选择 FPGA

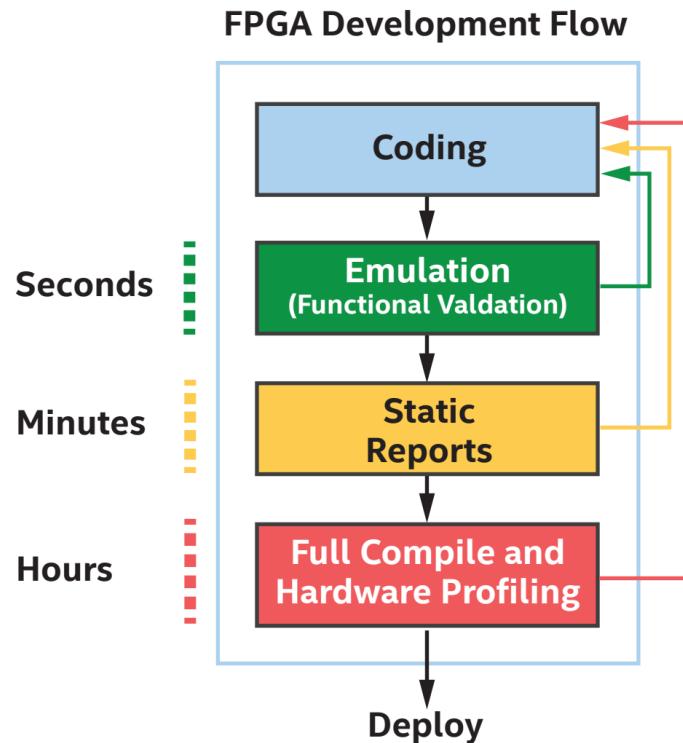
```
1 #include <CL/sycl.hpp>
2 #include <CL/sycl/intel/fpga_extensions.hpp> // For fpga_selector
3 using namespace sycl;
4
5 void say_device (const queue& Q) {
6     std::cout << "Device : "
7         << Q.get_device().get_info<info::device::name>()
8         << "\n";
9 }
10
11 int main() {
12     queue Q{ INTEL::fpga_selector{} };
13     say_device(Q);
14
15     Q.submit([&](handler &h){
16         h.parallel_for(1024, [=](auto idx) {
17             // ...
18         });
19     });
20
21     return 0;
22 }
```

编译时间

很多传言说，为 FPGA 编译设计可能需要很长时间，比基于 ISA 的加速器编译时间长得多。传言是真的！本章的最后概述了 FPGA 的细粒度架构元素，它们带来了 FPGA 的优点和计算密集型的编译（位置和路径优化），在某些情况下需要花费数小时。

从源代码到 FPGA 硬件执行的编译时间足够长，以至于我们不希望只在硬件中开发和迭代代码。FPGA 开发流程提供了几个阶段，这些阶段最小化了硬件编译的数量，使硬件编译的时间不受影响时，仍然具有生产力。图 17-10 显示了典型的阶段，大部分时间都花在提供快速周转和快速迭代的早期步骤上。

图 17-10 大多数验证和优化发生在冗长的硬件编译之前



编译器的渲染和静态报告是 DPC++ 中 FPGA 代码开发的基石。模拟器支持相关的扩展和执行模型，在主机上运行。因此，编译时间与期望从编译到 CPU 设备的时间相同，尽管看不到在实际 FPGA 硬件上执行所带来的性能提升。模拟器对于在应用程序中建立和测试功能正确性非常有用。

工具链可以快速生成静态报告，比如仿真。报告由编译器创建的 FPGA 结构，并由编译器识别的瓶颈。这两者都可以用来预测设计在 FPGA 硬件上运行时是否会有良好的性能，并用于优化代码。请阅读供应商的文档以获取有关报告的信息，这些报告通常会随着工具链的发布而不断改进（请参阅文档以获得最新和最伟大的特性！）供应商提供了详细的文档，说明如何根据报告进行解释和优化。这些信息将是另一本书的主题，所以不在这一章中详细讨论。

FPGA 仿真器

模拟主要用于从功能上调试应用程序，以确保它的行为符合预期并产生正确的结果。没有理由在编译时间较长的实际 FPGA 硬件上进行这种级别的开发。仿真流通过从 dpcpp 编译命令中删除-Xhardware 标志来激活，并且在我们的主机代码中使用 INTEL::fpga_emulator_selector，而不是使用 INTEL::fpga_selector。然后，使用以下命令进行编译

```
dpcpp -fintelfpga my_source_code.cpp
```

同时，将在运行时使用如图 17-11 所示的 FPGA 仿真器。通过使用 FPGA 模拟器选择器（使用主机处理器来模拟 FPGA），需要对实际的 FPGA 硬件进行冗长的编译之前，维护一个快速的开发和调试过程。

图 17-11 利用 FPGA 仿真器进行快速开发和调试

```
1 #include <CL/sycl.hpp>
2
3 #include <CL/sycl/intel/fpga_extensions.hpp> // For fpga_selector
4 using namespace sycl;
5
6 void say_device (const queue& Q) {
7     std::cout << "Device : "
8         << Q.get_device().get_info<info::device::name>() << "\n";
9 }
10
11 int main() {
12     queue Q{INTEL::fpga_emulator_selector{}};
13     say_device(Q);
14
15     Q.submit([&](handler &h){
16         h.parallel_for(1024, [=](auto idx) {
17             // ...
18         });
19     });
20
21     return 0;
22 }
```

如果经常在硬件和模拟器之间切换，可以在程序中使用宏在对设备选择器进行切换。如果需要，请查看供应商的文档和在线 FPGA DPC++ 代码示例。

FPGA 的 AOT(Ahead-of-Time) 编译

图 17-10 中的完全编译和硬件分析阶段在 SYCL 术语中是 AOT 编译。意味着将内核编译为设备二进制文件，是发生在最初编译程序时，而不是在将程序提交给要运行的设备时。在 FPGA 上，这一点特别重要，因为

1. 编译需要一段时间，这是运行程序时不希望发生的。
2. DPC++ 程序可能会在没有主机处理器的系统上执行。FPGA 二进制代码的编译过程得益于快速处理器和大量附加内存。AOT 编译可以轻松地选择编译发生的位置，而不是让它在部署程序的系统上运行。

在 FPGA 上使用 DPC++ 其实要经历很多!

传统的 FPGA 设计 (不使用高级语言) 可能非常复杂。除了编写内核之外，还有许多步骤，比如：构建和配置与芯片外存储器通信的接口，通过插入寄存器来关闭计时，这些寄存器需要使编译后的设计运行得足够快，以便与某些外设通信。DPC++ 解决了这一切，所以不需要知道任何关于传统 FPGA 设计的细节来实现工作应用程序！该工具将内核当作代码来优化和提高设备效率，然后自动处理与芯片外外设通信、关闭计时和设置驱动程序的所有细节。

与任何其他加速器一样，要在 FPGA 上实现峰值性能仍然需要详细的架构知识，但使用 DPC++ 从代码转移到工作设计的步骤，比传统 FPGA 流程要简单得多，生产率也更高。

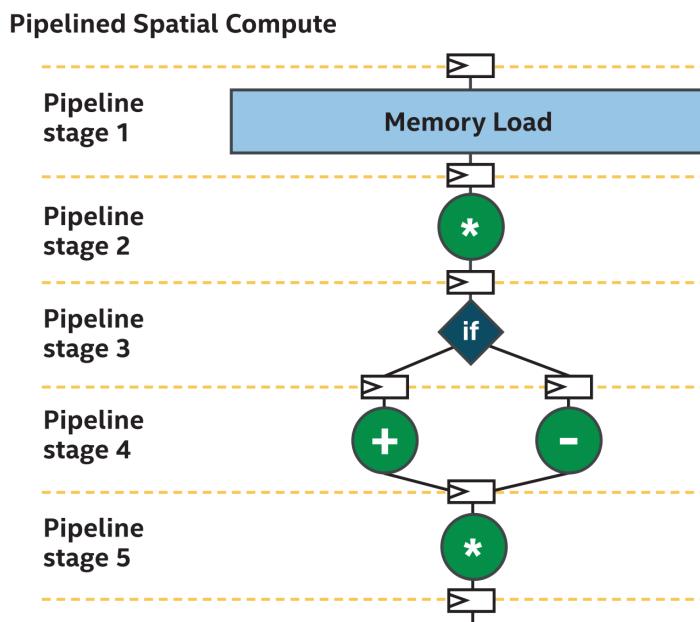
为 FPGA 编写内核函数

当决定为程序使用 FPGA 时，或者决定尝试一下 FPGA 时，了解如何编写代码以获得良好的性能很重要。本节会来了解一些重要概念，并包括一些常引起混淆的主题，以便更快入门。

并行性

我们了解了如何使用流水线并行在 FPGA 上有效地执行工作。简单的流水线示例如图 17-12 所示。

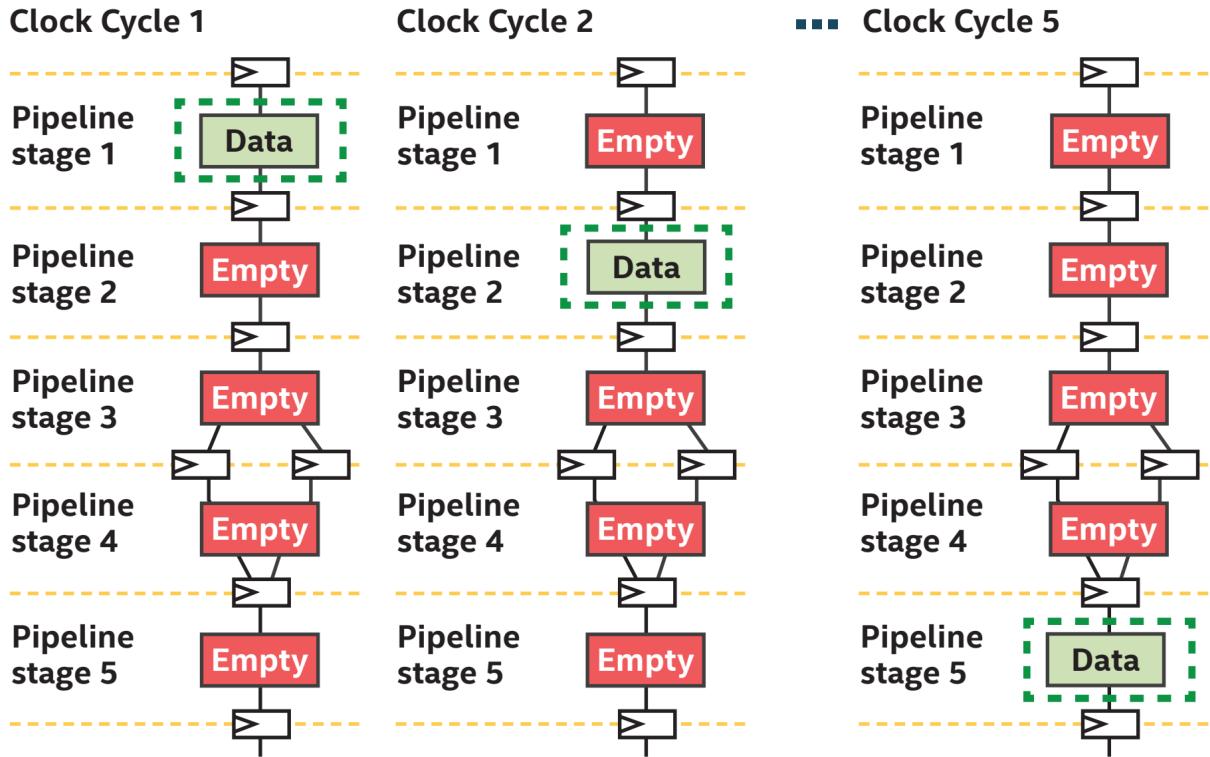
图 17-12 具有 5 个阶段的简单管道:6 个时钟周期来处理一个数据元素



这个过程中，有 5 个阶段。数据在每个时钟周期中从一个阶段移动到下一个阶段，所以在这个非常简单的例子中，从数据进入阶段 1 到从阶段 5 退出需要 6 个时钟周期。

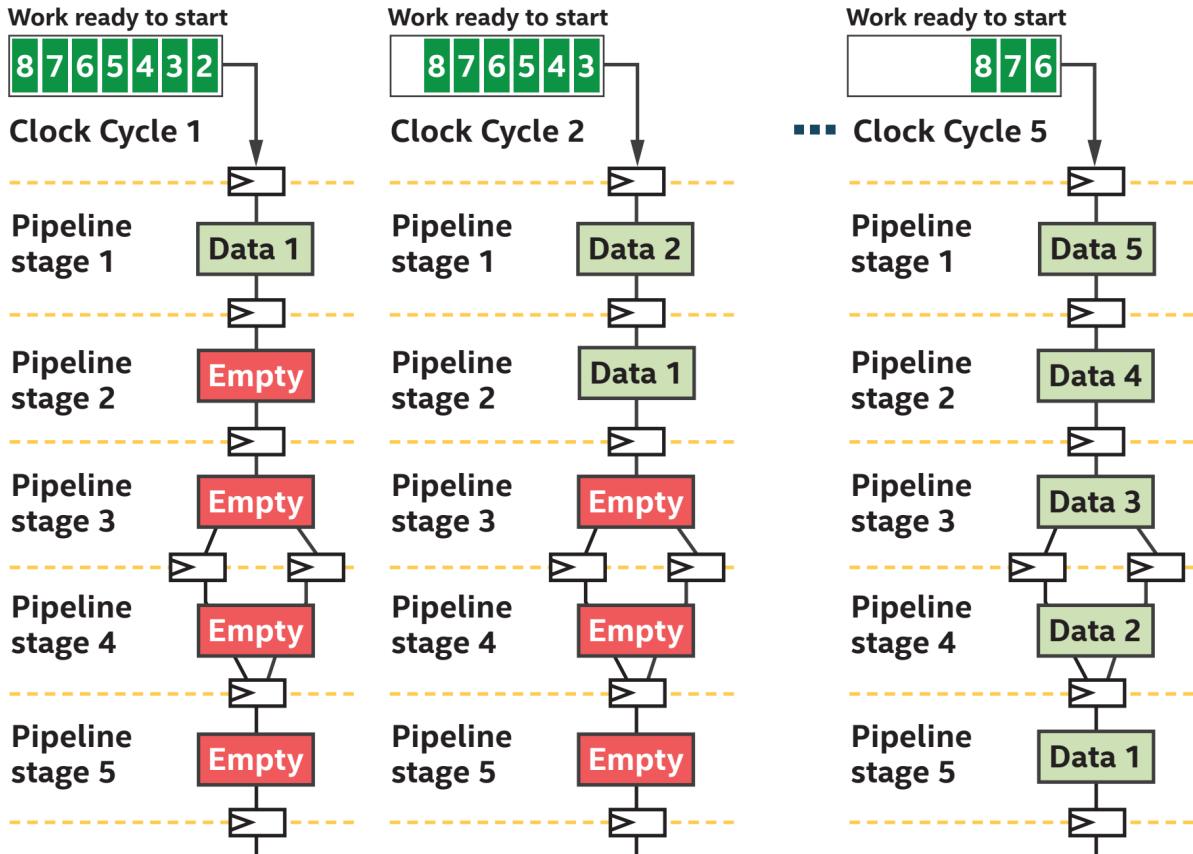
流水线的主要目标是使多个数据元素能够在不同阶段同时处理。为了明确这一点，图 17-13 展示了一个没有饱和 (在本例中只有一个数据元素) 的管道，这导致在大多数时钟周期中每个阶段都未使用。因为硬件大部分时间都是空闲的，所以这是对 FPGA 的浪费。

图 17-13 如果只处理单个工作元素，几乎不使用流水线



为了更好地利用流水线阶段，可以设想在为流水线提供数据的第一个阶段之前，有一个未启动的工作队列等待。每个时钟周期，管道可以从队列中消耗并启动一个工作，如图 17-14 所示。经过一些初始启动周期后，管道的每个阶段都被占用，并在每个时钟周期中做有用的工作，从而使 FPGA 资源得到有效利用。

图 17-14 当每个管道阶段都保持忙碌时，就会产生高效的利用



下面的两个部分介绍了一些方法，这些方法为队列向流水线输送做启动性工作：

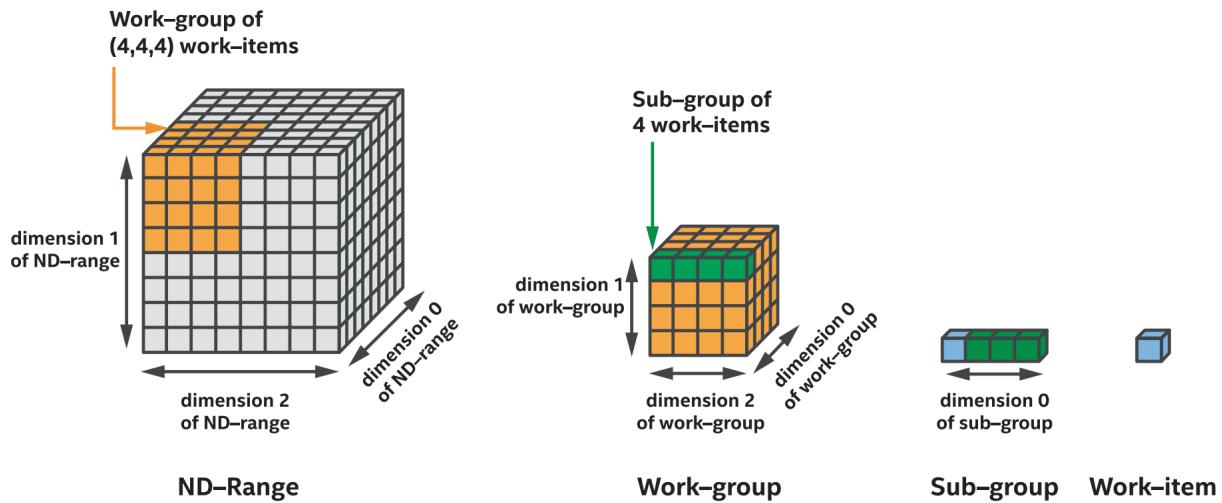
1. ND-Range 内核
2. 循环

选择将影响在 FPGA 上运行的内核的基本架构。某些情况下，算法很适合某种风格，而在其他情况下，开发者的偏好和经验决定了应该选择哪种方法。

使用 ND-Range 使流水线繁忙起来

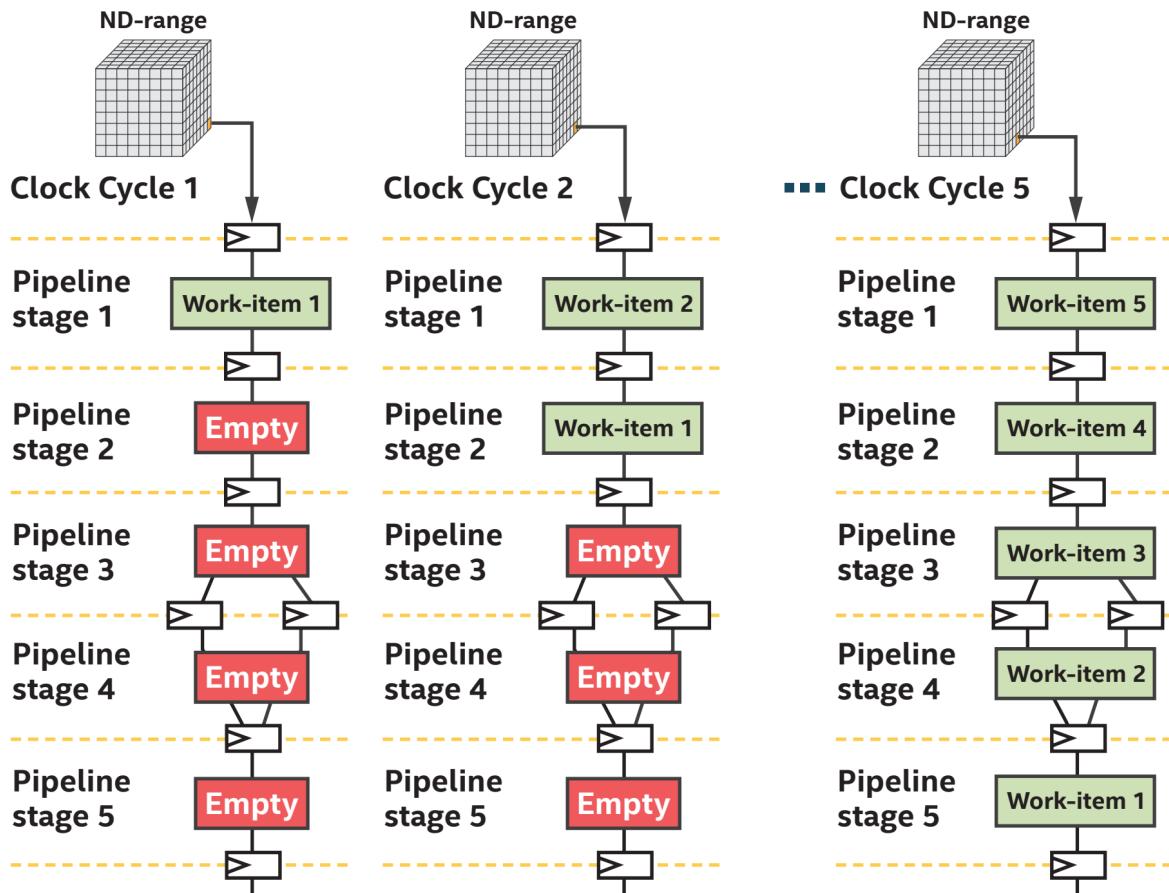
第 4 章描述了 ND-Range 分层执行模型。图 17-15 说明了关键概念:ND-Range 执行模型，其中有工作项的分层分组，工作项是内核定义的基本工作单元。这个模型最初是用来支持 GPU 的编程，其中工作项可以在执行模型层次结构的不同级别上并发执行。为了匹配 GPU 高效的工作类型，ND-Range 工作项在大多数应用程序中相互不通信。

图 17-15 ND-Range 执行模型: 工作项的分层分组



FPGA 流水线可以非常有效地使用 ND-Range 进行填充。FPGA 完全支持这种编程风格，可以将其看作如图 17-16 所示，每个时钟周期上，不同的工作项会进入第一阶段。

图 17-16 ND-Range 向流水线输送任务



什么时候应该使用工作项在 FPGA 上创建 ND-Range 内核以保持流水线占用呢？可以将算法或应用程序构建为独立的工作项，而这些工作项不需要经常通信（或者理想情况下根本不需要交流）

时，应该使用 ND-Range! 如果工作项确实需要经常通信，或者不考虑使用 DN-Range，那么循环（在下一节中描述）提供了一种高效的方式来表示算法。

如果可以构建算法，使工作项不需要太多（或根本不需要）交流，那么 ND-Range 是生成工作以保持流水线占满的好方法！

使用 ND-Range 输入流水线高效内核的一个例子是随机数生成器，在该序列中创建的数字独立于之前生成的数字。

图 17-17 展示了一个 ND-Range 内核，对 16×16×16 范围内的每个工作项进行了随机数生成。注意随机数生成函数如何将工作项 id 作为输入。

图 17-17 随机数生成器的多工作项 (16 × 16 × 16)

```
1 h.parallel_for({16,16,16}, [=](auto I) {  
2     output[I] = generate_random_number_from_ID(I);  
3 });
```

示例展示了使用 range 的 parallel_for，只指定了全局大小。可以交替使用带有 nd_range 的 parallel_for，其中指定了全局工作大小和本地工作组大小。FPGA 可以从片上资源实现工作组本地内存，所以只要有意义就可以随意使用工作组，要么因为想要工作组本地内存，要么因为有可用的工作组 id 可以简化代码。

并行随机数生成器

图 17-17 中的示例假设 generate_random_number_from_ID(I) 是一个随机数生成器。例如，如果 parallel_for 内的不同工作项执行这个函数，期望每个工作项创建不同的序列，每个序列都符合生成器所期望的任何分布。并行随机数生成器本身是一个很复杂的主题，因此可以使用库或通过块超前跳过算法等技术了解。

流水线无视数据依赖！

当一些工作项为向量架构（例如 GPU）编程时，一个挑战是在工作项之间没有通信的情况下是一个高效的算法。有些算法和应用程序很适合向量硬件，有些则不行。导致问题的常见原因是，由于数据依赖于在某种意义上相邻的其他计算，算法需要共享数据。子工作组通过工作项之间的通信来解决向量体系结构上的挑战，如第 14 章所述。

FPGA 对于不能分解成独立工作的算法起着重要的作用。FPGA 流水线没有跨工作项进行向量化，而是跨流水线阶段执行连续的工作项。这种并行性的实现，意味着可以在流水线中轻松有效地实现工作项（甚至是不同工作组中的工作项）之间的细粒度通信！

例子是随机数生成器它的输出 $N+1$ 取决于输出 N 是什么。这在两个输出之间创建了数据依赖关系，如果每个输出都是由 ND-Range 范围内的工作项生成的，那么工作项之间就存在着数据依赖关系，这可能需要在某些体系结构上进行复杂且代价高昂的同步。当按顺序编码这样的算法时，通常会写一个循环，其中迭代 $N+1$ 使用迭代 N 的计算，如图 17-18 所示。每次迭代都依赖于前一次迭代计算的状态。

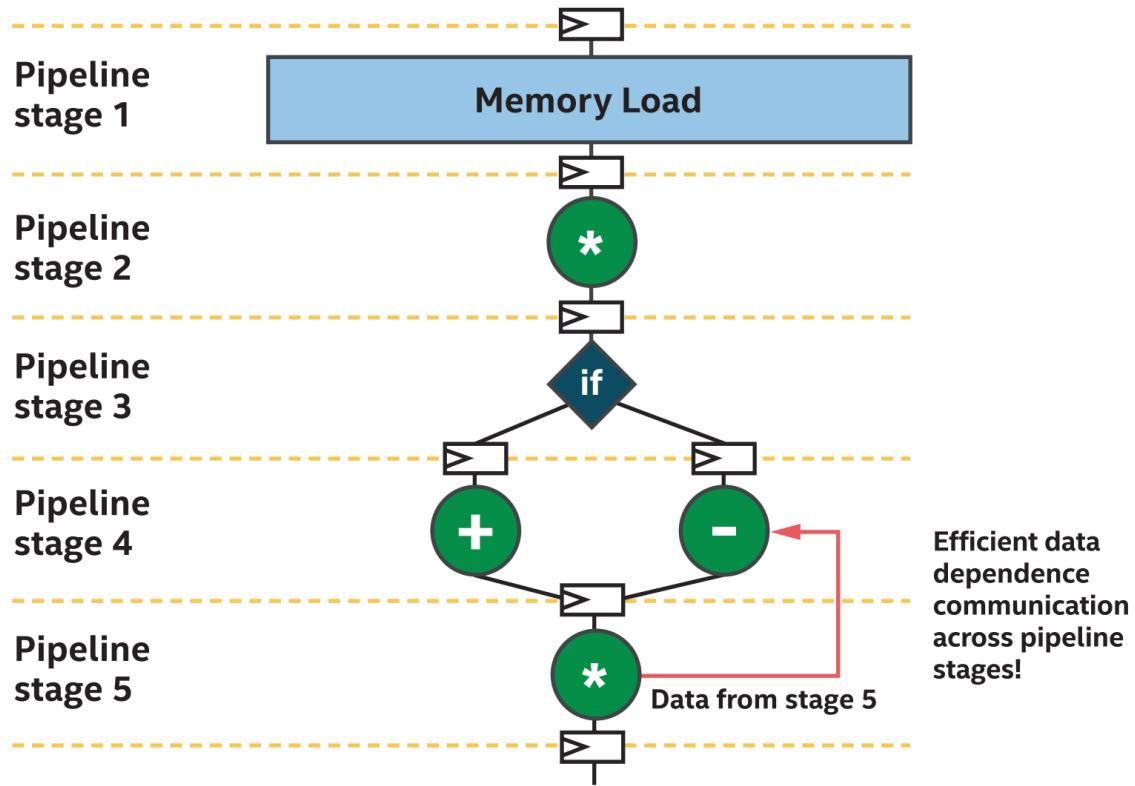
图 17-18 携带数据依赖 (状态) 的循环

```
1 int state = 0;
2 for (int i=0; i < size; i++) {
3     state = generate_random_number(state);
4     output[i] = state;
5 }
```

该方法可以非常有效地将结果向后传递到后续循环中开始工作的流水线中，和空间编译器实现围绕此模式的许多优化。图 17-19 展示了从第 5 阶段到第 4 阶段的数据反向通信的思想。空间管道不会跨工作项向量化。通过在管道中向后传递结果，支持高效的数据依赖通信！

图 17-19 向后通信使高效的数据依赖成为可能

Pipelined Spatial Compute



向后传递数据 (到管道中的早期阶段) 的能力是空间体系结构的关键，但是不清楚如何编写利用它的代码。有两种方法可以简化这种模式的表达：有两种方法可以简化这种模式的表达：

1. 循环
2. 带有内部管道的 ND-Range 内核

第二个选项是基于管道的，我们将在本章后面描述它。供应商文档提供了关于管道方法的更多细节，但是常用的还是循环，除非有其他原因。

空间流水线循环的实现

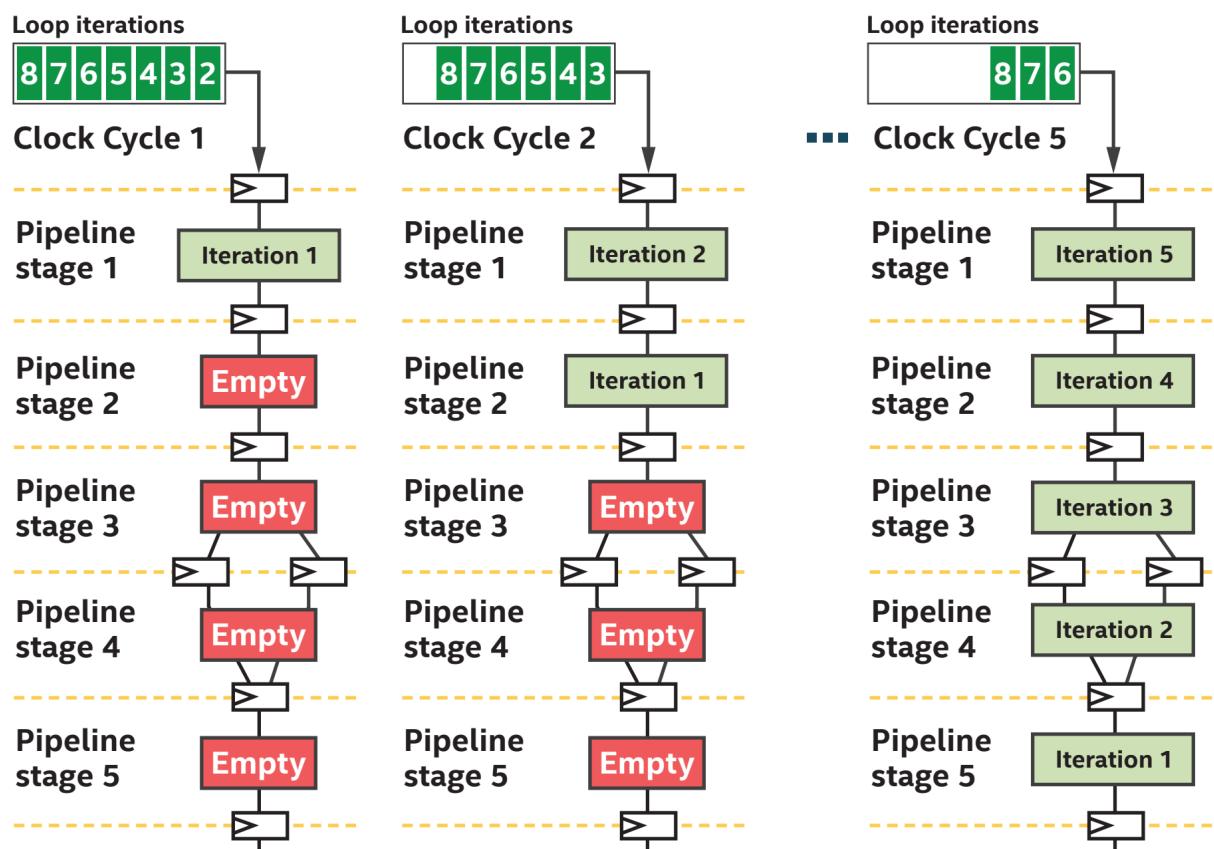
当编写具有数据依赖性的算法时，循环是一种自然的选择。循环经常表示迭代之间的依赖关系，即使在最基本的循环示例中，决定循环何时退出的计数器也会在迭代中执行（图 17-20 中的变量 i）。

图 17-20 带有两个循环依赖项（即 i 和 a）的循环

```
1 int a = 0;
2 for (int i=0; i < size; i++) {
3     a = a + i;
4 }
```

图 17-20 的简单循环中， $a = a + i$ 右边的 a 的值反映了前一次循环中存储的值，如果是第一次循环，则是初始值。当空间编译器实现循环时，可以使用循环的迭代来填充管道的各个阶段，如图 17-21 所示。请注意，现在准备开始的工作队列包含循环迭代，而不是工作项！

图 17-21 流水线阶段由循环的连续迭代提供



修改后的随机数生成器示例如图 17-22 所示。与基于工作项的 id 生成数字不同，如图 17-17 所示，生成器将先前计算的值将作为参数。

图 17-22 依赖于之前生成的值的随机数生成器

```
1 h.single_task([=]() {
2     int state = seed;
3     for (int i=0; i < size; i++) {
4         state = generate_incremental_random_number(state);
```

```
5     output[i] = state;
6 }
7});
```

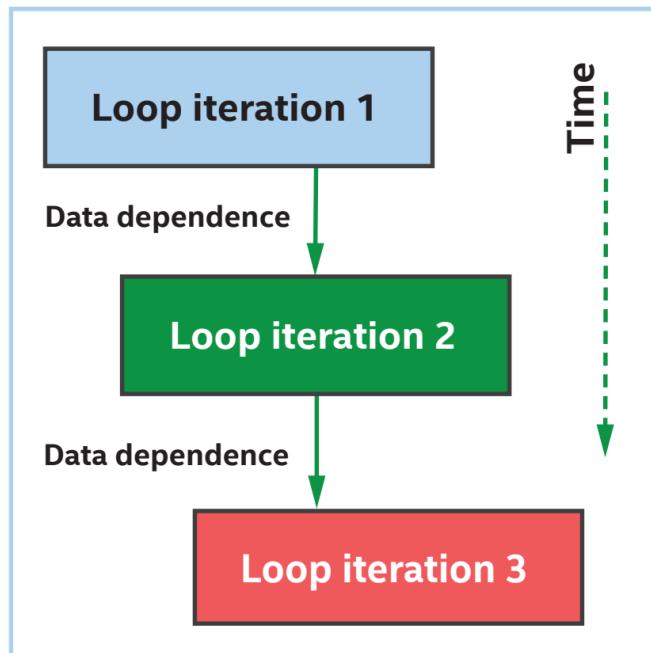
这个例子使用了 `single_task` 而不是 `parallel_for`，因为重复的工作是通过单个任务中的循环表示，所以没有理由在此代码中还包含多个工作项（通过 `parallel_for`）。`single_task` 内部的循环使得将之前计算的 `temp` 值传递给随机数生成函数的每次使用更加容易（编程方便）。

如图 17-22 所示的情况下，FPGA 可以有效地实现环路。许多情况下，它可以保持流水线完全占用的状态，或者至少可以通过报告告诉我们应该改变什么来增加占用率。考虑到这一点，如果用工作项替换循环迭代，那么同样的算法将更加难以描述，其中由一个工作项生成的值将需要传递给在增量计算中使用的另一个工作项。代码复杂性将迅速增加，特别是如果工作不能进行批处理，使每个工作项实际计算自己独立的随机数序列。

循环起始的间隔

从概念上讲，认为 C++ 中的循环迭代是一个接一个地执行，如图 17-23 所示。这是编程模型，也是思考循环的正确方式。实际上，编译器可以执行许多优化，只要程序的大部分行为没有明显的变化。不管编译器的优化如何，重要的是循环的执行看起来就像图 17-23 所示的那样。

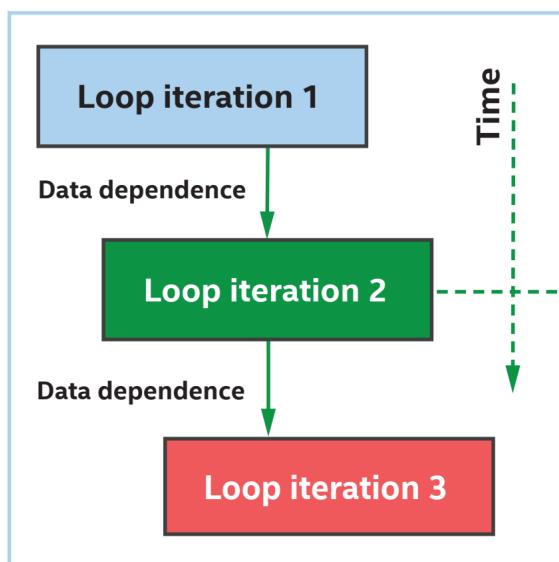
图 17-23 从概念上讲，循环迭代一个接一个地执行



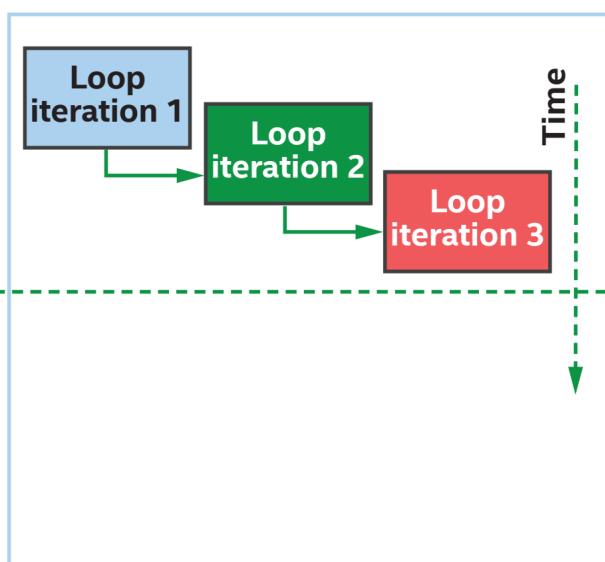
进入空间编译器透视图，图 17-24 显示了一个循环流水线优化，其中循环迭代的执行在时间上是重叠的。不同的迭代将彼此执行流水线的不同阶段，而各个阶段的数据依赖可以由编译器管理，以确保程序执行的迭代好似是顺序的（除了循环将更快地完成执行）。

图 17-24 循环流水线允许循环的迭代在流水线阶段之间重叠

Serial execution of loop

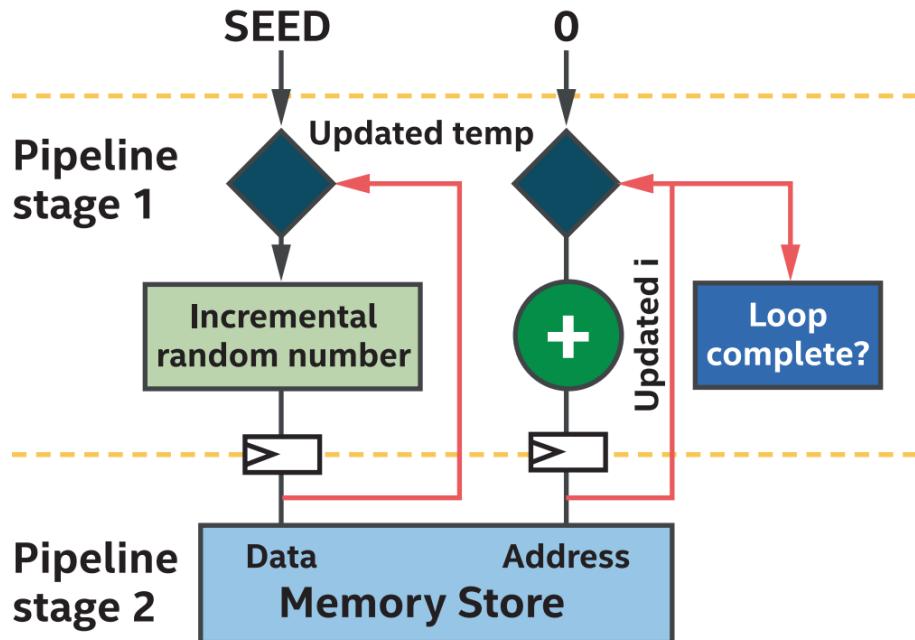


Loop pipelined execution



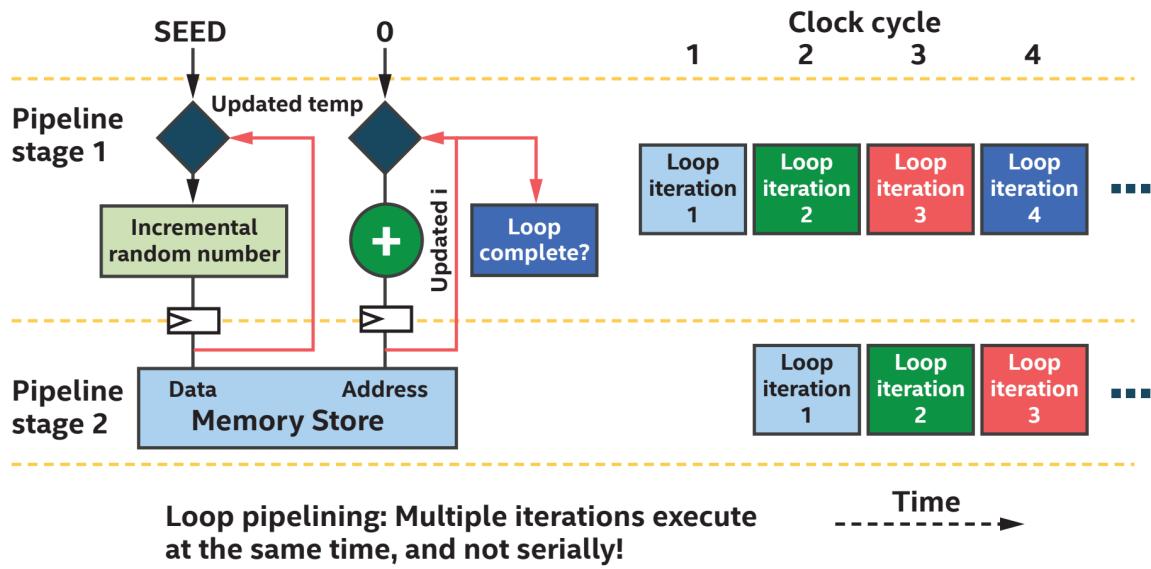
流水线中，当编译器决定这样做时，结果可以传递到更早的阶段，这样循环迭代中的许多结果可能在循环迭代完成所有工作之前就完成了计算。图 17-25 展示了这种思想，阶段 1 的结果在管道中向后反馈，允许未来的循环迭代在前一个迭代完成之前使用结果。

图 17-25 增量随机数生成器的流水线实现



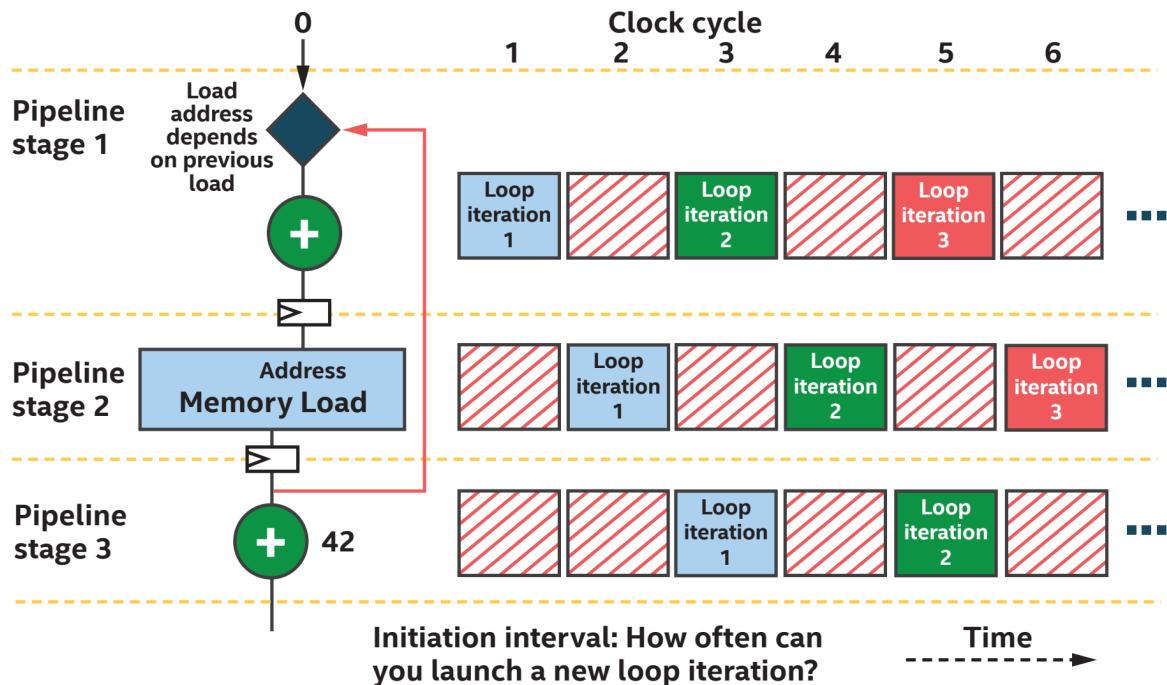
使用循环流水线，可以使循环的多次迭代重叠执行，即使使用循环携带的数据依赖项，循环迭代仍然可以用于用工作填充管道，从而实现高效利用。图 17-26 显示了循环迭代是如何在图 17-25 所示的同一个管道中的重叠执行。

图 17-26 循环流水线同时处理多个循环迭代的部分



实际算法中，不可能在每个时钟周期中启动新的循环迭代，因为数据依赖可能需要多个时钟周期来计算。如果内存查找（特别是片外内存）处于依赖计算的关键路径上，则经常会出现这种情况。结果是一个管道，每 N 个时钟周期只能启动一个新的循环迭代，将其称为 N 个周期的起始间隔 (II)。配置示例如图 17-27 所示。两个循环起始间隔 (II) 意味着一个新的循环迭代可以每秒钟开始一次，这导致管道阶段的次优占用。

图 17-27 管道阶段的次优占用



II 大于 1 会导致流水线的效率低下，因为每个阶段的平均占用率都降低了。从图 17-27 中可以看出，II=2 和管道阶段在很大比例（50%!）的时间内没有使用。有很多方法可以改善这种情况。

编译器执行优化以尽可能地减少 II，所以报告也会告诉我们每个循环的初始间隔，并告诉我们

为什么它大于 1。基于报告在循环中重新构造计算通常可以减少 II，可以进行编译器不允许的循环结构更改（因为是可观察的）。请阅读编译器报告，了解如何在特定情况下减少 II。

降低 II 大于 1 的低效率的另一种方法是通过嵌套循环，可以通过将具有 $II > 1$ 的内环迭代与外部循环迭代的交错来填充所有阶段。查看供应商文档和编译器报告，了解使用详细信息。

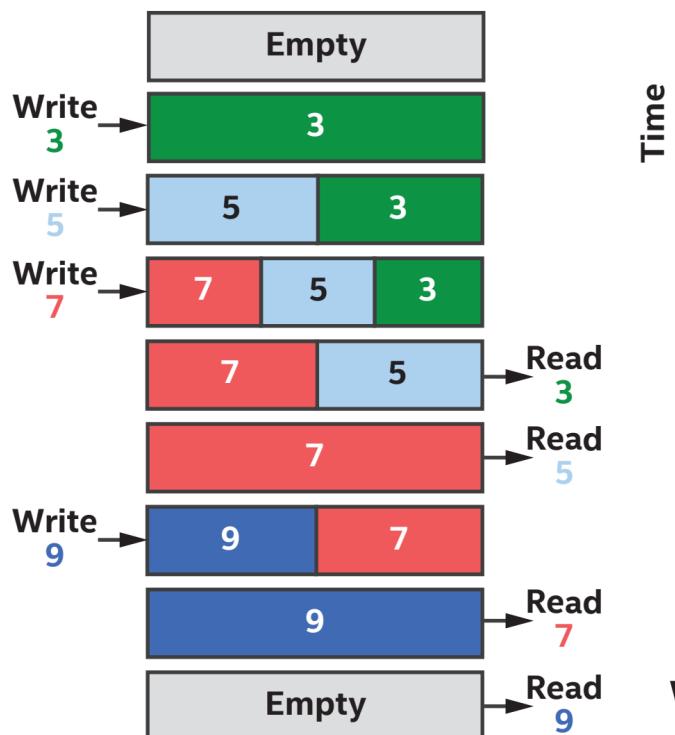
管道

空间和其他体系结构中的一个重要概念是先进先出 (FIFO) 缓冲区。FIFO 之所以重要有很多原因，但在考虑编程时，有两个原因：

1. 隐式控制信息与数据。这些信息告诉我们 FIFO 是空的还是满的，并且在将问题分解时非常有用。
2. FIFO 具有存储容量。在动态行为（如访问内存时的高可变延迟）时，可以更容易地实现性能。

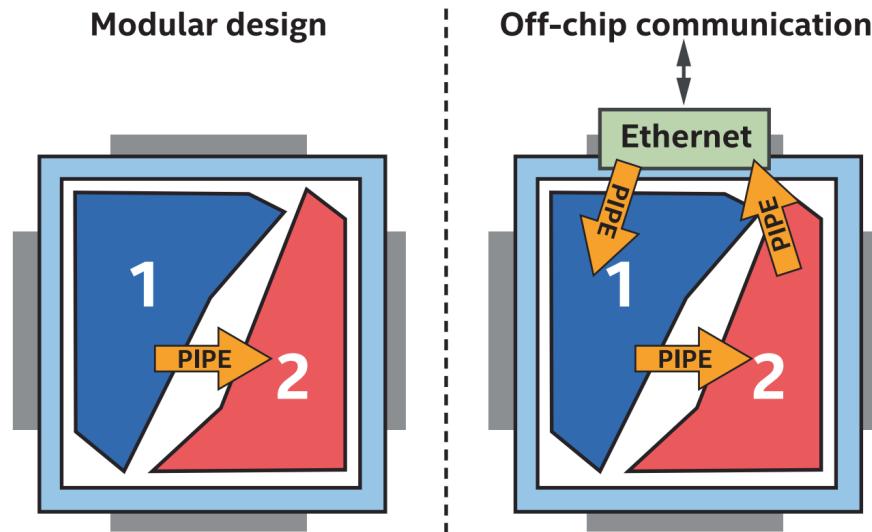
图 17-28 展示了一个简单的 FIFO 操作示例。

图 17-28 FIFO 的示例操作



FIFO 在 DPC++ 中通过管道的特性对外使用。编写 FPGA 程序时，应该关心管道的原因是，管道可以分解问题，以更模块化的方式关注开发和优化。还允许利用 FPGA 丰富的通信特性。如图 17-29 所示。

图 17-29 管道简化了模块化设计和对硬件外设的访问



记住 FPGA 内核可以同时存在于设备上 (芯片的不同区域)，在高效的设计中，内核的所有部分在每个时钟周期中都是活跃的。这意味着优化 FPGA 应用程序需要考虑内核的各个部分如何相互交互，而管道提供了一个抽象来简化这一过程。

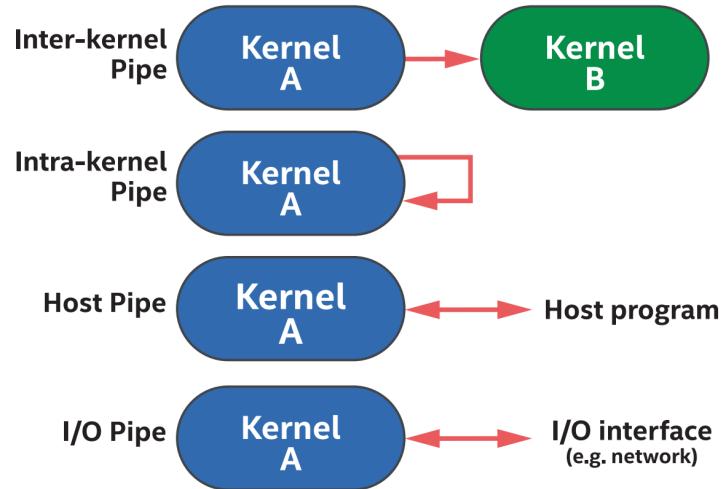
管道是使用 FPGA 上的片内存储器实现的 FIFO，因此允许在运行的内核之间和内部进行通信，而无需将数据移动到片外存储器。这可以提供廉价的通信，与管道 (空/全信号) 耦合的控制信息提供了轻量级的同步机制。

我们需要管道吗？

不使用管道也可以编写高效的内核。可以使用所有的 FPGA 资源，并在没有管道的情况下使用传统编程风格实现最大性能。但是对于大多数开发人员来说，编程和优化模块化空间设计更容易，而管道是实现这一目标的好方法。

如图 17-30 所示，共有四种类型的管道。本节的其余部分中，将介绍第一种类型（内核间管道），因为它们可以说明什么是管道，以及如何使用管道。管道还可以在单个内核中与主机或输入/输出外设通信。请查阅供应商文档，了解更多关于管道的形式和用途的信息。

图 17-30 DPC++ 中管道连接的类型



如图 17-31 所示。有两个内核通过管道进行通信，每个读或写操作都以 int 为单位。

图 17-31 两个内核之间的管道:(1)ND-range 和 (2) 单个任务和一个循环

```

1 // Create alias for pipe type so that consistent across uses
2 using my_pipe = pipe<class some_pipe, int>;
3
4 // ND-range kernel
5 Q.submit([&](handler& h) {
6     auto A = accessor(B_in, h);
7
8     h.parallel_for(count, [=](auto idx) {
9         my_pipe::write(A[idx]);
10    });
11 });
12
13 // Single_task kernel
14 Q.submit([&](handler& h) {
15     auto A = accessor(B_out, h);
16
17     h.single_task( [=]() {
18         for (int i=0; i < count; i++) {
19             A[i] = my_pipe::read();
20         }
21     });
22 });

```

图 17-31 中有几点需要观察。首先，两个内核使用管道相互通信。如果内核之间没有访问器或事件依赖，DPC++ 运行时将同时执行，允许通过管道而不是完整的 SYCL 内存缓冲区或 USM 进行通信。

管道使用基于类型的方法进行标识，其中每个管道都使用管道类型的参数化进行标识，如图 17-32 所示。管道类型的参数化标识了特定的管道。对相同管道类型的读或写是对相同的 FIFO。有三个模板参数一起定义了管道的类型，从而定义了管道的标识。

图 17-32 参数化的管道类型

```
1 template <typename name,
2     typename dataT,
3     size_t min_capacity = 0>
4 class pipe;
```

建议使用类型别名来定义管道类型，如图 17-31 中的第一行代码所示，以减少编程错误并提高代码可读性。

使用类型别名来标识管道，简化了代码并防止意外的创建管道。

管道有一个 `min_capacity` 参数，默认为 0，如果指定了，保证至少有一定数量的数据可以写入管道，而不会读出任何数据。此参数在以下情况下有用

1. 两个与管道通信的内核不会同时运行，需要管道中有足够的容量，让第一个内核在第二个内核开始运行并从管道中读取之前写入它的所有输出。
2. 如果内核突然生成或消耗数据，那么向管道添加容量可以提供内核隔离，将它们解耦。例如，产生数据的内核可以继续写（直到管道容量满了），即使消耗数据的内核很忙，还没有准备好消耗任何东西。这提供了相对于其他内核执行的灵活性，仅以 FPGA 上的一些内存资源为代价。

阻塞和非阻塞管道访问

像大多数 FIFO 接口一样，管道有两种类型的接口：阻塞和非阻塞。阻塞访问等待（阻塞/暂停执行！）操作成功，而非阻塞访问立即返回，并设置布尔值指示操作是否成功。

成功的定义很简单：如果正在从管道中读取数据，并且有可用数据可读（管道不为空），则读取成功。如果正在写入，而管道还没有满，则写入成功。图 17-33 显示了 `pipe` 类的两种访问成员函数形式。我们看到管道的成员函数允许对其进行写入或读取。对管道的访问可以是阻塞的，也可以是非阻塞的。

图 17-33 允许写入或读取管道的成员函数

```
1 // Blocking
2 T read();
3 void write( const T &data );
4
5 // Non-blocking
6 T read( bool &success_code );
7 void write( const T &data, bool &success_code );
```

阻塞访问和非阻塞访问都有用，这取决于程序试图实现什么。如果内核在从管道中读取数据之前不能做更多的工作，那么使用阻塞读取可能有意义。如果内核希望从一组管道中的任何一个读取数据，但不确定哪个管道可能有可用的数据，那么使用非阻塞调用从管道读取数据更有意义。内核

可以从管道中读取数据并处理数据 (如果有数据)，但如果管道是空的，它可以继续，尝试从可能有可用数据的下一个管道中读取数据。

有关管道的更多信息

这一章中，只能粗略地了解管道，以及了解了如何使用它们的基本知识。FPGA 供应商文档提供了更多的信息和在不同类型应用程序中使用的许例，因此如果认为管道与特定需求相关，应该先查看这些文档。

自定义内存系统

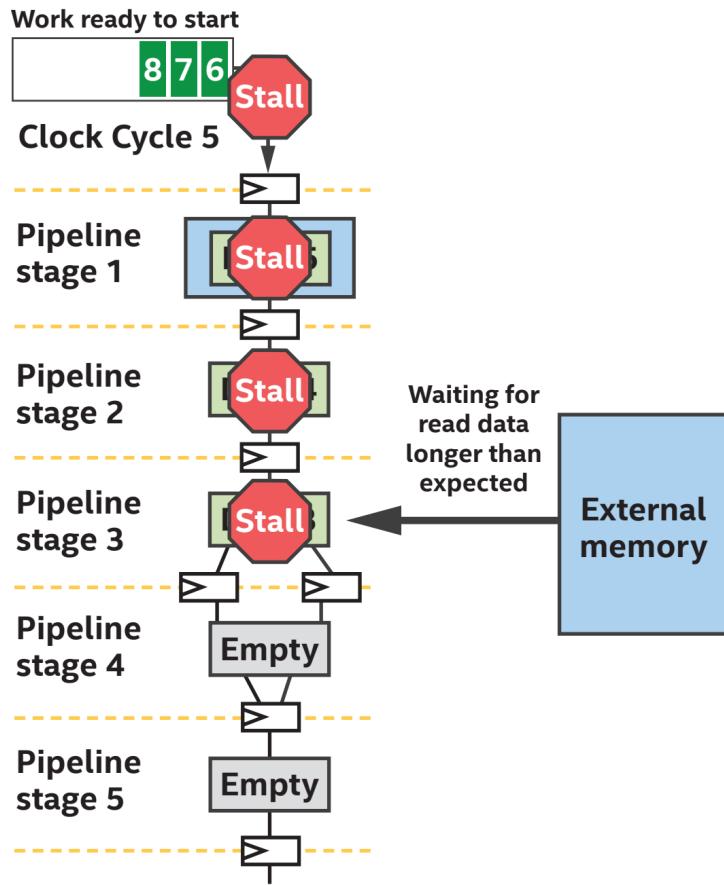
为大多数加速器编程时，大部分优化工作都花在提高内存访问的效率上。FPGA 设计也是如此，特别是当输入和输出数据通过片外存储器时。

FPGA 上的内存访问值得优化有两个主要原因：

1. 减少所需的带宽，特别是在带宽利用率低的情况下。
2. 修改内存的访问模式，避免流水线中的停顿。

流水线中，有必要简单谈一下暂停机制。编译器内置了读取或写入特定类型内存所需的时间的假设，并相应地优化和平衡了管道，在进程中隐藏了内存延迟。但是，如果以一种低效的方式访问内存，就会引入更长的延迟，并作为管道中的副产品停滞不前，因为等待，早期的阶段无法执行，管道阶段阻塞了 (例如，内存访问)。如图 17-34 所示，超过负载的管道停止工作。

图 17-34 内存停滞也会导致流水线阶段性停滞



可以在几个方面执行内存系统优化。编译器报告是了解编译器为实现了什么，以及哪些可能值得调整或改进的重要指南。这里列出了一些优化主题，以突出显示可供使用的自由度。优化通常可以通过显式控制和修改代码，让编译器推断想要的结构来实现。编译器静态报告和供应商文档，是内存系统优化的关键，有时在硬件执行期间与分析工具结合使用，以捕获实际的内存行为，用于验证或优化的最后阶段。

1. 静态合并: 编译器可以将内存访问合并为更小、更宽的访问。这降低了存储系统的复杂性，包括流水线中的负载或存储单元的数量、存储系统上的端口、仲裁网络的大小和复杂性，以及其他存储系统等。通常希望尽可能启用静态合并，这个可以通过编译器报告来确认。在内核中简化寻址逻辑有时就足以让编译器执行更为激进的静态合并，所以总是检查编译器是否推断出我们所期望的报告！
2. 内存访问: 编译器为内存访问创建加载或存储单元，这些单元适合被访问的内存技术（例如，片上、DDR、HBM）和从源代码推断的访问模式（例如，流、动态合并/扩展，或可能从特定大小的缓存中获益）。编译器报告告诉我们推断了什么，并允许修改或添加控件到我们的代码，以提高性能。
3. 内存结构: 内存系统（包括片上和片外）可以具有由编译器实现的内存块结构和许多优化。可以使用许多控件和模式修改来控制这些结构和调优空间实现。

一些相关的话题

与 FPGA 的开发菜鸟交谈时，会发现从高层次上理解组成设备的组件常常会有帮助，而且时钟频率容易混淆。

FPGA 构建块

为了帮助理解工具流 (特别是编译时)，有必要提到组成 FPGA 的构建块。构建块是 DPC++ 和 SYCL 的抽象，在程序开发中不起作用 (至少在使代码具有某些功能方面)。然而，它们的存在确实会影响空间架构的优化和工具流的开发，例如：为应用程序选择数据类型时，有时也会影响高级优化。

简化的 FPGA 设备由五个基本元素组成

1. 查询表: 一些二进制输入线产生二进制输出的基本块。相对于输入，输出通过写到查询表中使用。这些都是原始块，但在用于计算的现代 FPGA 上有许多 (数百万) 块。这些是大部分设计实现的基础!
2. 数学引擎: 对于普通的数学操作，如单精度定位点数的加法或乘法，FPGA 有专门的硬件使这些操作非常高效。FPGA 有数千个这样的块——有些设备有 8000 多个——这样至少每个时钟周期都可以并行执行这些浮点基元操作! 大多数 FPGA 将这些数学引擎命名为数字信号处理器 (DSP)。
3. 片上内存: 这是 FPGA 与其他加速器的区别，而内存有两种类型 (实际上更多):(1) 用于在操作和其他目的之间传输的寄存器，(2) 提供分布在设备上的随机访问内存的块内存。FPGA 可以有大约数百万个寄存器位和超过 10,000 个 20kbit 的 RAM 存储器块。由于每一个都可以在时钟周期中激活，因此当有效使用片上内存容量和带宽时，效果显著。
4. 外设接口:FPGA 的扩展部分是由灵活的收发器和输入/输出连接，允许与所有的外存储器、网络接口等进行通信。
5. 连接结构: 前文本中提到的 FPGA 中的每个元素都有很多，并且连接不是固定的。一个复杂的可编连接允许信号在 FPGA 的结构中，进行细粒度传递。

给定 FPGA 上每种特定类型的块的数量 (有些块以百万计) 和这些块的细粒度 (如查询表)，生成 FPGA 配置位流时看到的编译时间可能更有意义。不仅需要为每个细粒度资源分配功能，还需要在它们之间配置连接。很多编译时间来自于在优化开始之前，找到 FPGA 结构的第一个合法映射!

时钟频率

因为它是非常灵活和可配置的，与 CPU 或任何其他固定计算架构的同等设计相比，FPGA 运行的频率的可配置性会带来了一些开销，但这不是问题!FPGA 的空间架构大大弥补了时钟频率，因为有如此多的独立操作同时发生，分布在 FPGA 的区域。简单地说，由于可配置的设计，FPGA 的频率比其他架构低，但每个时钟周期发生更多的事件，从而平衡了频率。在对基准和加速器进行比较时，应该比较计算吞吐量 (例如每秒的操作数)，而不是原始频率。

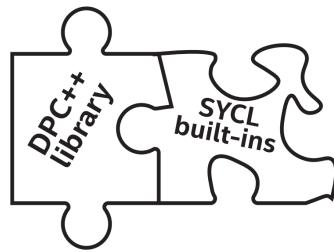
当 FPGA 上的资源利用率接近 100% 时，工作频率可能会下降，主要是设备上的信号连接资源过度使用的结果。有一些方法可以弥补这一点，通常是增加编译时间。但是对于大多数应用程序，最好避免在 FPGA 上使用超过 80-90% 的资源，除非愿意深入研究，避免频率下降。

经验法则: 尽量不要超过 FPGA 上任何资源的 90%，当然也不要超过多个资源的 90%。超过可能导致连接资源耗尽，从而降低工作频率。

总结

本章中，介绍了流水线如何将算法映射到 FPGA 的空间架构中。还讨论了一些概念，可以帮助我们决定 FPGA 对应用程序是否有用，并且可以帮助我们更快地启动和运行开发代码。从这个起点出发，需要仔细地阅览供应商编程和优化手册，并开始编写 FPGA 代码!FPGA 提供了在其他加速器上没有性能和启用的程序，所以我们应该把它们放在我们开发工具箱的最前面!

18 库



我们用整本书来推广代码的艺术，一些优秀的程序员已经编写了可以直接使用的代码，库是完成工作的最好方式。这并不是懒惰——这是一种比重塑他人工作更好的表现。这是一块值得拥有的拼图。

开源的 DPC++ 项目包括一些库。这些库可以帮助我们使用 `libstdc++`、`libc++` 和 MSVC 库函数，甚至在内核代码中。这些库是英特尔的 DPC++ 和 oneAPI 的一部分。这些库不绑定到 DPC++ 编译器，因此可以与任何 SYCL 编译器一起使用。

DPC++ 库为创建异构应用程序和解决方案的程序员提供了另一种选择。它的 API 基于熟悉的标准——C++ STL、Parallel STL (PSTL) 和 SYCL——为开发者提供高生产率的 API。这可以减少跨 CPU、GPU 和 FPGA 的编程工作，同时产生具有可移植性的高性能并行应用程序。

SYCL 标准定义了丰富的内置函数，这些函数为主机和设备代码提供了一些功能。DPC++ 和许多 SYCL 实现通过数学库实现了关键的内置函数。

本章讨论的库和内置函数与编译器无关。换句话说，它们同样适用于 DPC++ 编译器或 SYCL 编译器。`fpga_device_policy` 类用于 FPGA 支持的 DPC++ 特性。

由于在命名和功能上存在重叠，本章将从介绍 SYCL 内置函数开始。

内置函数

DPC++ 提供了一组关于各种数据类型的 SYCL 内置函数。内置函数可以在主机和设备上的 `sycl` 命名空间内使用，基于编译器选项（例如 DPC++ 编译器提供的 `-mfma`、`-fast-math` 和 `-ffp-contract=fast`）对目标设备提供低、中、高精度支持。主机和设备上的内置函数分为以下几类：

- 浮点数学函数 `asin`、`acos`、`log`、`sqrt`、`floor` 等，如图 18-2。
- 整数函数：`abs`、`max`、`min` 等，如图 18-3。
- 常用函数：`clamp`、`smoothstep` 等，如图 18-4。
- 几何函数：`cross`、`dot`、`distance` 等，如图 18-5。
- 关系函数：`isequal`、`isless`、`isfinite` 等，如图 18-6。

当函数是由 C++ `std` 库提供（如图 18-8 所示），并且有对应的 SYCL 内置函数，则 DPC++ 开发者可以使用其中任何一个。图 18-1 展示了 C++ `std::log` 函数和 SYCL 内置函数的用于主机和设备的 `sycl::log`，这两个函数产生相同的结果。本例中，内置函数 `sycl::isequal` 用于比较 `std::log` 和 `sycl::log` 的结果。

图 18-1 使用 `std::log` 和 `sycl::log`

```

1 constexpr int size = 9;
2 std::array<double, size> A;
3 std::array<double, size> B;
4
5 bool pass = true;
6
7 for (int i = 0; i < size; ++i) { A[i] = i; B[i] = i; }
8
9 queue Q;
10 range sz{size};
11
12 buffer<double> bufA(A);
13 buffer<double> bufB(B);
14 buffer<bool> bufP(&pass, 1);
15
16 Q.submit([&](handler &h) {
17     accessor accA{bufA, h};
18     accessor accB{bufB, h};
19     accessor accP{bufP, h};
20
21     h.parallel_for(size, [=](id<1> idx) {
22         accA[idx] = std::log(accA[idx]);
23         accB[idx] = sycl::log(accB[idx]);
24         if (!sycl::is_equal(accA[idx], accB[idx])) {
25             accP[0] = false;
26         }
27     });
28 });

```

除了 SYCL 中支持的数据类型外，DPC++ 设备库还提供了对 `std::complex` 数据类型，以及 C++ `std` 库中定义的相应数学函数的支持

内置函数中使用 `sycl::` 前缀

调用 SYCL 内置函数时，应该在名称前添加显式的 `sycl::`。对于当前的 SYCL 规范，即使使用了“`using namespace sycl;`”，也不能保证只调用 `sqrt()` 就能调用所有实现上的内置 SYCL。

使用 SYCL 内置函数时，应该始终在函数名称前面显式地加上 `sycl::`。不遵循此建议可能会导致不可移植的结果。

在应用程序中，内置函数的名称与非模板函数冲突，在许多实现中（包括 DPC++），内置函数将占上风，这是因为 C++ 的重载解析规则更喜欢非模板函数而不是模板函数。然而，当代码有一个与内置名称相同的函数名，那么为了保证可移植性，避免使用命名空间 `sycl`，或者确保没有冲突。否则，一些 SYCL 编译器将由于无法解决冲突而拒绝编译代码。

图 18-2 内置数学函数

Tf acos (Tf x)	Arc cosine	Tf lDEXP (Tf x,Ti k)	
Tf acosh (Tf x)	Inverse hyperbolic cosine	floatn lDEXP (floatn x, int k)	$x * 2^n$
Tf acospi (Tf x)	$\text{acos}(x) / \pi$	doublen lDEXP (doublen x , int k)	
Tf asin (Tf x)	Arc sine	Tf lgamma (Tf x)	Log gamma function.
Tf asinh (Tf x)	Inverse hyperbolic sine	Tf lgamma_r (Tf x,Ti*signp)	
Tf asinpi (Tf x)	$\text{asin}(x) / \pi$	Tf log (Tf)	N H Natural logarithm
Tf atan (Tf y_over_x)	Arc tangent	Tf log2 (Tf)	N H Base 2 logarithm
Tf atan2 (Tf y,Tf x)	Arc tangent of y / x	Tf log10 (Tf)	N H Base 10 logarithm
Tf atanh (Tf x)	Hyperbolic arc tangent	Tf log1p (Tf x)	$\ln(1.0 + x)$
Tf atanpi (Tf x)	$\text{atan}(x) / \pi$	Tf logb (Tf x)	Exponent of x
Tf atan2pi (Tf x,Tf y)	$\text{atan2}(y,x) / \pi$	Tf mad (Tf a,Tf b,Tf c)	Approximates $a * b + c$
Tf cbrt (Tf x)	Cube root	Tf maxmag (Tf x,Tf y)	Maximum magnitude of x and y
Tf ceil (Tf x)	Round to integer toward infinity	+Tf minmag (Tf x,Tf y)	Minimum magnitude of x and y
Tf copysign (Tf x,Tf y)	x with sign changed to sign of y	Tf modf (Tf x,Tf *iptr)	Decompose floating-point number
Tf cos (Tf x)	H Cosine	floatn nan (uintn nancode)	
Tf cosh (Tf x)	N Hyperbolic cosine	float nan (unsigned int nancode)	
Tf cosp (Tf x)	$\cos(\pi x)$	doublen nan (ulonglong nancode)	Quiet NaN(Return is scalar when nancode is scalar)
Tf divide (Tf x,Tf y)	*N x / y	doublen nan (longlong int nancode)	
Tf erfc (Tf x)	Complementary error function	Tf nextafter (Tf x,Tf y)	Next representable floating-point value after x in the direction of y
Tf erf (Tf x)	Calculates error function	Tf pow (Tf x,Tf y)	Compute x to the power of y
Tf exp (Tf x)	N H Exponential base e	Tf pown (Tf x,Ti y)	Compute x y,where y is an integer
Tf exp2 (Tf x)	N H Exponential base 2	Tf powr (Tf x,Tf y)	N Compute x y,where x is ≥ 0
Tf exp10 (Tf x)	N H Exponential base 10	Tf recip (Tf x)	*N H $1 / x$
Tf expm1 (Tf x)	N H $e^x - 1.0$	Tf remainder (Tf x,Tf y)	Floating point remainder
Tf fabs (Tf x)	Absolute value	Tf remquo (Tf x,Tf y,Ti *q)	Remainder and quotient
Tf fdim (Tf x,Tf y)	Positive difference between x and y	Tf rint (Tf)	Round to nearest even integer
Tf floor (Tf x)	Round to integer toward infinity	Tf rootn (Tf x,Ti y)	Compute x to the power of 1/y
Tf fma (Tf a,Tf b,Tf c)	Multiply and add,then round	Tf round (Tf x)	Integral value nearest to x rounding
Tf fmax (Tf x,Tf y)	Return y if $x < y$,otherwise it returns x	Tf rsqrt (Tf)	N H Inverse square root
Tf fmin (Tf x,Tf y)		Tf sin (Tf)	N H Sine
Tf fmod (Tf x,Tf y)	Modulus. Returns $x - y$ trunc(x/y)	*Tf sincos (Tf x,Tf *cosval)	Sine and cosine of x
floatn fract (floatn x, intn *iptr)	Fractional value in x	Tf sinh (Tf x)	Hyperbolic sine
float fract (float x, int *iptr)		Tf sinpi (Tf x)	$\sin(\pi x)$
doublen frexp (doublen x, intn *exp)	Extract mantissa and exponent	Tf sqrt (Tf x)	N H Square root
double frexp (double x, int *exp)		Tf tan (Tf x)	N H Tangent
Tf hypot (Tf x,Tf y)	Square root of $x^2 + y^2$	Tf tanh (Tf x)	Hyperbolic tangent
int logb (float x)		Tf tanpi (Tf x)	$\tan(\pi x)$
intn iLogb (Tf x)		Tf tgamma (Tf x)	Gamma function
int logb (double x)	Return exponent as an integer value	Tf trunc (Tf x)	Round to integer toward zero
intn logb (doublen x)			

For floatn, doublen, and intn: n is 2, 3, 4, 8, or 16.

Tf (genfloat in the spec) is type float, floatn, double, or doublen.

sTf (sgenfloat in the spec) is type float or double.

Ti (genint in the spec) is type int or intn.

N indicates that native variants are available.

*N indicates availability only in native forms.

H indicates availability in half-precision for devices with fp16 extension available.

图 18-3 内置整数函数

Tui abs (Ti x)	$ x $
Tui abs_diff (Ti x,Ti y)	$ x - y $ without modulo overflow
Ti add_sat (Ti x,Ti y)	$x + y$ and saturates the result
Ti hadd (Ti x,Ti y)	$(x + y) \gg 1$ without mod. overflow
Ti rhadd (Ti x,Ti y)	$(x + y + 1) \gg 1$
Ti clamp (Ti x,Ti min,Ti max)	$\min(\max(x,\text{min}),\text{max})$
Ti clamp (Ti x,Tsi min,Tsi max)	$\min(\max(x,\text{min}),\text{max})$
Ti clz (Ti x)	number of leading zero-bits in x; special case for $x == 0$: returns the size in bits of the type of x, or the component type of x if x is a vector type.
Ti ctz (Ti x)	Same as <code>clz()</code> but for trailing zero-bits
Ti mad_hi (Ti a,Ti b,Ti c)	<code>mul_hi(a,b) + c</code>
Ti mad_sat (Ti a,Ti b,Ti c)	$a * b + c$ and saturates the result
Ti max (Ti x,Ti y)	y if $x < y$, otherwise it returns x
Ti max (Ti x,Tsi y)	y if $x < y$, otherwise it returns x
Ti min (Ti x,Ti y)	y if $y < x$, otherwise it returns x
Ti min (Ti x,Tsi y)	y if $y < x$, otherwise it returns x
Ti mul_hi (Ti x,Ti y)	high half of the product of x and y
Ti popcount (Ti x)	Number of non-zero bits in x
Ti rotate (Ti v,Ti i)	$\text{result}[idx] = v[idx] \ll i[idx]$
Ti sub_sat (Ti x,Ti y)	$x - y$ and saturates the result
T popcount (T x)	Number of non-zero bits in x
shortn upsample (charn hi,ucharn lo)	$\text{result}[i] = ((\text{short})\text{hi}[i] \ll 8) \mid \text{lo}[i]$
ushortn upsample (ucharn hi,ucharn lo)	$\text{result}[i] = ((\text{ushort})\text{hi}[i] \ll 8) \mid \text{lo}[i]$
intn upsample (shortn hi,ushortn lo)	$\text{result}[i] = ((\text{int})\text{hi}[i] \ll 16) \mid \text{lo}[i]$
uintn upsample (ushortn hi,ushortn lo)	$\text{result}[i] = ((\text{uint})\text{hi}[i] \ll 16) \mid \text{lo}[i]$
longlongn upsample (intn hi,uintn lo)	$\text{result}[i] = ((\text{long})\text{hi}[i] \ll 32) \mid \text{lo}[i]$
ulonglongn upsample (uintn hi,uintn lo)	$\text{result}[i] = ((\text{ulong})\text{hi}[i] \ll 32) \mid \text{lo}[i]$
intn mad24 (intn x,intn y,intn z)	Multiply 24-bit integer values x,y,add 32-bit int. result to 32-bit integer z
uintn mad24 (uintn x,uintn y,uintn z)	Multiply 24-bit integer values x,y,add 32-bit int. result to 32-bit integer z
intn mul24 (intn x,intn y)	Multiply 24-bit integer values x and y
uintn mul24 (uintn x,uintn y)	Multiply 24-bit integer values x and y

T (gentype in the spec) is all integer and float types.

Ti (geninteger in spec) is all signed and unsigned integer types.

Tsi (sgeninteger in the spec) is all scalar integer types.

Tui (ugeninteger in the spec) is all unsigned integer types.

图 18-4 内置通用函数

Tf clamp (Tf x,Tf minval,Tf maxval) floatn clamp (floatn x,float minval,float maxval) doublen clamp (doublen x,double minval,doublen maxval)	Clamp x to range given by minval,maxval
Tf degrees (Tf radians)	radians to degrees
Tf max (Tf x,Tf y) Tff max (Tff x,float y) Tfd max (Tfd x,double y)	Max of x and y
Tf min (Tf x,Tf y) Tff min (Tff x,float y) Tfd min (Tfd x,double y)	Min of x and y
Tf mix (Tf x,Tf y,Tf a) Tff mix (Tff x,Tff y,float a) Tfd mix (Tfd x,Tfd y,double a)	Linear blend of x and y
Tf radians (Tf degrees)	degrees to radians
Tf step (Tf edge,Tf x) Tff step (float edge,Tff x) Tfd step (double edge,Tfd x)	0.0 if x < edge,else 1.0
Tf smoothstep (Tf edge0,Tf edge1,Tf x) Tff smoothstep (float edge0,float edge1,Tff x); Tfd smoothstep (double edge0,double edge1,Tfd x)	Step and interpolate
Tf sign (Tf x)	Sign of x

For floatn and doublen: n is 2, 3, 4, 8, or 16.

Tf (genfloat in the spec) is float, floatn, double, or doublen types.

Tff (genfloatf in the spec) is float or floatn types.

Tfd (genfloatd in the spec) is double or doublen types.

图 18-5 内置几何函数

T34 cross (T34 p0,T34 p1)	Cross product
T distance(Tn p0,Tn p1)	Vector distance
T dot(Tn p0,Tn p1)	Dot product
T length(Tn p)	Vector length
Tn normalize(Tn p)	Normal vector length 1
float fast_distance(floatn p0,floatn p1)	Vector distance
float fast_length(floatn p)	Vector length
floatn fast_normalize(floatn p)	Normal vector length 1

For Tn: n is 2, 3, 4, 8, or 16.

T34 means either T3 or T4.

T is type float, floatn, double, or doublen, consistently applied per function.

图 18-6 内置关系函数

Functions $F(\dots)$ can be: isequal isnotequal isgreater isgreaterequal isless islessequal islessgreater isordered isunordered	int $F(\text{float } x, \text{float } y)$ intn $F(\text{floatn } x, \text{floatn } y)$ long long $F(\text{double } x, \text{double } y)$ long longn $F(\text{doublen } x, \text{doublen } y)$
Functions $F(\dots)$ can be: isfinite isinf isnan isnormal signbit	int $F(\text{float})$ intn $F(\text{floatn})$ long long $F(\text{double})$ long longn $F(\text{doublen})$
int $\text{any}(\text{Ti } x)$	1 if MSB in component of x is set; else 0
int $\text{all}(\text{Ti } x)$	1 if MSB in all components of x are set else 0
T $\text{bitselect}(\text{T a, T b, T c})$	Each bit of result is corresponding bit of a if corresponding bit of c is 0
T $\text{select}(\text{T a, T b, Ti } c)$	For each component of a vector type, $\text{result}[i] = \text{if MSB of } c[i] \text{ is set? } b[i] : a[i]$ For scalar type, $\text{result} = c ? b : a$

For intn and longn: n is 2, 3, 4, 8, or 16.

T (gentype in the spec) is all signed, unsigned, float, double, scalar and vector types.

Ti (geninteger in spec) is signed and unsigned integer types.

Tui (ugeninteger in the spec) is all unsigned integer types.

DPC++ 库

DPC++ 库由以下组件组成:

- 经过测试的 C++ 标准 API——只需要包含相应的 C++ 标准头文件并使用 std 命名空间。
- 包含相应头文件使用并行 STL, 只需 #include <dpstd ...>, DPC++ 库就可使用 dpstd 名称空间。

DPC++ 中的标准 C++ API

DPC++ 库包含一组经过测试的标准 C++ API。许多 C++ 标准 API 的基本功能已经完成, 因此这些 API 可以在设备内核中使用。图 18-7 展示了如何在设备代码中使用 std::swap 的示例。

图 18-7 设备代码中使用 std::swap

```

1 class KernelSwap;
2 std::array<int,2> arr{8,9};
3 buffer<int> buf{arr};
4
5 {
6     host_accessor host_A(buf);
7     std::cout << "Before: " << host_A[0] << ", " << host_A[1] << "\n";
8 } // End scope of host_A so that upcoming kernel can operate on buf
9
10 queue Q;
11 Q.submit([&](handler &h) {
12     accessor A{buf, h};
13     h.single_task(==) {

```

```

14 // Call std::swap!
15     std::swap(A[0], A[1]);
16 }
17 });
18
19 host_accessor host_B(buf);
20 std::cout << "After: " << host_B[0] << ", " << host_B[1] << "\n";

```

可以使用以下命令来构建和运行程序 (假设它位于 stdswap.cpp 文件中):

```
dpcpp -std=c++17 stdswap.cpp -o stdswap.exe ./stdswap.exe
```

打印结果为:

```

8, 9
9, 8

```

图 18-8 列出了带有“Y”的 C++ 标准 API，以表明在编写本文时，这些 API 已经在用于 CPU、GPU 和 FPGA 设备的 DPC++ 内核中进行了测试。空白表示在本书出版时，不完全覆盖（不是所有三种设备类型）。在线 DPC++ 手册中也包含了一个表，并将随着时间的推移而更新——DPC++ 中的支持将继续扩大。

在 DPC++ 库中，一些 C++ std 函数是基于设备上内置函数的实现，以达到与 SYCL 版本相同的性能水平。

图 18-8 库支持 CPU/GPU/FPGA 覆盖 (本书出版时)

C++ standard API	libstdc++	libC++	MSVS
std::acos	Y		
std::acosh	Y		
std::add const	Y	Y	Y
std::add cv	Y	Y	Y
std::add volatile	Y	Y	Y
std::alignment_of	Y	Y	Y
std::array	Y	Y	Y
std::asin	Y		
std::fundamental	Y	Y	Y
std::is literal type	Y	Y	Y
std::is member pointer	Y	Y	Y
std::is move assignable	Y	Y	Y
std::is move constructible	Y	Y	Y
std::is object	Y	Y	Y
std::is pod	Y	Y	Y

std::exp	Y		
std::exp2	Y		
std::expm1	Y		
std::extent	Y	Y	Y
std::fdim	Y		
std::fmod	Y		
std::forward	Y	Y	Y
std::frexp	Y		
std::greater	Y	Y	Y
std::greater_equal	Y	Y	Y
std::hypot	Y		
std::ilogb	Y		
std::initializer_list	Y	Y	Y
std::integral_constant	Y	Y	Y
std::is arithmetic	Y	Y	Y
std::is assignable	Y	Y	Y
std::is base of	Y	Y	Y
std::is base of union	Y	Y	Y
std::is compound	Y	Y	Y
std::is const	Y	Y	Y
std::is constructible	Y	Y	Y

std::is fundamental	Y	Y	Y
std::is literal type	Y	Y	Y
std::is member pointer	Y	Y	Y
std::is move assignable	Y	Y	Y
std::is move constructible	Y	Y	Y
std::is object	Y	Y	Y
std::is pod	Y	Y	Y
std::asinh	Y		
std::assert	Y		Y
std::atan	Y		
std::atan2	Y		
std::atanh	Y		
std::binary negate	Y	Y	Y
std::binary search	Y	Y	Y
std::bit and	Y	Y	Y
std::bit not	Y	Y	Y
std::bit or	Y	Y	Y
std::bit xor	Y	Y	Y
std::cbrt	Y		
std::common type	Y	Y	Y
std::complex	Y		
std::conditional	Y	Y	Y
std::cos	Y		
std::cosh	Y		
std::decay	Y	Y	Y
std::declval	Y	Y	Y
std::divides	Y	Y	Y
std::enable if	Y	Y	Y
std::equal range	Y	Y	Y
std::equal to	Y	Y	Y
std::erf	Y		
std::erfc	Y		

std::move if noexcept	Y	Y	Y
std::multiplies	Y	Y	Y
std::negate	Y	Y	Y
std::nextafter	Y		
std::not equal to	Y	Y	Y
std::not1/2	Y	Y	Y
std::numeric limits	Y	Y	Y
std::pair	Y	Y	Y
std::plus	Y	Y	Y
std::pow	Y		
std::rank	Y	Y	Y
std::ratio	Y	Y	Y
std::ref/cref	Y	Y	Y

std::is copy assignable	Y	Y	Y
std::is copy constructible	Y	Y	Y
std::is default constructible	Y	Y	Y
std::is destructible	Y	Y	Y
std::is empty	Y	Y	Y
std::is same	Y	Y	Y
std::is scalar	Y	Y	Y
std::is signed	Y	Y	Y
std::is standard layout	Y	Y	Y
std::is trivial	Y	Y	Y
std::is trivially assignable	Y	Y	Y
std::is trivially constructible	Y	Y	Y
std::is trivially copyable	Y	Y	Y
std::is unsigned	Y	Y	Y
std::is volatile	Y	Y	Y
std::ldexp	Y		
std::less	Y	Y	Y
std::less equal	Y	Y	Y
std::lgamma	Y		
std::log	Y		
std::log10	Y		
std::log1p	Y		
std::log2	Y		
std::logb	Y		
std::logical and	Y	Y	Y
std::logical not	Y	Y	Y
std::logical or	Y	Y	Y
std::lower bound	Y	Y	Y
std::minus	Y	Y	Y
std::modf	Y		
std::modulus	Y	Y	Y
std::move	Y	Y	Y

std::reference wrapper	Y	Y	Y
std::remainder	Y		
std::remove all extents	Y	Y	Y
std::remove const	Y	Y	Y
std::remove cv	Y	Y	Y
std::remove extent	Y	Y	Y
std::remove volatile	Y	Y	Y
std::remquo	Y		
std::sin	Y		
std::sinh	Y		
std::sqrt	Y		
std::swap	Y	Y	Y

在 libstdc++ (GNU) 与 gcc 7.4.0 和 libc++ (LLVM) 与 clang 10.0 和 MSVC 标准 C++ 库与 Microsoft Visual Studio 2017(主机 CPU) 支持的标准 C++ API。

在 Linux 上, GNU libstdc++ 是 DPC++ 编译器的默认 C++ 标准库, 因此不需要编译或链接选项。如果我们想要使用 libc++, 请使用编译选项 -stdlib=libc++ -fno-diagnostics 来使用 libc++, 不要包含系统中的 C++ std 头文件。DPC++ 编译器已经在 Linux 上的 DPC++ 内核中使用 libc++ 进行了验证, 但是 DPC++ 运行时需要用 libc++ 而不是 libstdc++ 重新构建。详情请参见 <https://intel.github.io/llvmdocs/GetStartedGuide.html#build-dpc-toolchain-with-libc-library>。所以, libc++ 不是推荐使用的 C++ 标准库。

在 FreeBSD 上, libc++ 是默认的标准库, -stdlib=libc++ 选项不是必需的。更多详情请登录 <https://libcxx.llvm.org/docs/UsingLibcxx.html>。在 Windows 上, 只能使用 MSVC c++ 库。

执行策略	含义
seq	串行执行
unseq	同步执行 SIMD。此策略要求在 SIMD 中安全执行所有函数。
par	由多个线程并行执行。
par_unseq	合并了 unseq 与 par 的特性

为了实现跨架构的可移植性，如果 std 函数在图 18-8 中没有标记“Y”，在编写设备函数时需要注意是否可移植！

DPC++ 的 Parallel STL

Parallel STL 是 C++ 标准库算法的实现，支持执行策略，如 ISO/IEC 14882:2017 标准，通常称为 C++17。现有的实现还支持 Parallelism TS version 2 中指定的未排序执行策略，并在 C++ 工作组 P1001R1 中为下一版本的 C++ 标准提出了该策略。

当使用算法和执行策略时，如果没有特定于 C++17 的标准库实现，则指定命名空间 std::execution，否则指定命名空间 `pstd::execution`。

对于任何已实现的算法，都可以将 seq、unseq、par 或 par_unseq 中的值作为调用算法的第一个参数传递，以指定所需的执行策略。这些策略的含义如下：

对 DPC++ 的 Parallel STL 进行了扩展，支持使用特殊执行策略的 DPC++ 设备。DPC++ 执行策略指定了并行 STL 算法运行的位置和方式，继承了标准 C++ 的执行策略。封装了一个 SYCL 设备或队列，并允许设置可选的内核名称。DPC++ 执行策略可以与所有支持 C++17 标准执行策略的算法一起使用。

DPC++ 执行策略

目前，DPC++ 库只支持并行未串行策略 (par_unseq)。使用 DPC++ 执行策略，有三个步骤：

1. 在代码中添加 `#include <dpstd/execution>`。
2. 通过提供标准策略类型、作为模板参数唯一内核名的类型（可选）和以下构造函数参数之一来创建策略对象：
 - SYCL 队列
 - SYCL 设备
 - SYCL 设备选择器
 - 具有不同内核名称的已存在策略对象
3. 将创建的策略对象传递给 Parallel STL 算法。

`dpstd::execution::default_policy` 对象是预定义的 `device_policy`，使用默认的内核名和默认队列创建。这可以用于创建自定义策略对象，或者在调用算法时直接传递（如果默认选择足够的话）。

图 18-9 显示了使用 `using` 命名空间 `dpstd::execution` 的示例，引用策略类和函数。

图 18-9 创建执行策略

```

1 auto policy_b =
2   device_policy<parallel_unsequenced_policy, class PolicyB>
3   {sycl::device{sycl::gpu_selector{}}};
4 std::for_each(policy_b, ...);
5
6 auto policy_c =
7   device_policy<parallel_unsequenced_policy, class Policy >
8   {sycl::default_selector{}};
9 std::for_each(policy_c, ...);
10
11 auto policy_d = make_device_policy<class PolicyD>(default_policy);
12 std::for_each(policy_d, ...);
13
14 auto policy_e = make_device_policy<class PolicyE>(sycl::queue{});
15 std::for_each(policy_e, ...);

```

FPGA 的执行策略

`fpga_device_policy` 是一个 DPC++ 策略类，用于在 FPGA 硬件上实现性能更好的并行算法。在 FPGA 硬件或 FPGA 仿真设备上运行应用程序时，可以使用该策略：

1. FPGA 实际设备运行时设置 `_PSTL_FPGA_DEVICE` 宏，以及在 FPGA 仿真器上运行时设置 `_PSTL_FPGA_EMU` 宏。
2. 在代码中添加 `#include <dpstd/execution>`。
3. 通过为内核名和展开因子（参见第 17 章）提供类类型作为模板参数（两个都是可选的）和以下构造函数参数之一来创建策略对象：
 - 为 FPGA 选择器构造的 SYCL 队列（任何其他设备类型的行为都未定义）
 - 具有不同内核名称和/或展开因子的 FPGA 策略对象
4. 将创建的策略对象传递给 Parallel STL 算法。

`fpga_device_policy` 的默认构造函数创建一个对象，其中包含为 FPGA 选择器构造的 SYCL 队列。如果定义了 `_PSTL_FPGA_EMU`，则为 FPGA 仿真器选择器构造的 SYCL 队列。

`dpstd::execution::fpga_policy` 是 `fpga_device_policy` 类的预定义对象，使用默认的内核名称和默认的展开因子创建。使用它来创建定制的策略对象，或者在调用算法时直接使用。

图 18-10 中的代码假设 `using namespace dpstd::execution;`，用于策略，而 `using namespace sycl;`，用于队列和设备选择器。

指定策略的展开因子可以在算法的实现中展开循环，默认值为 1。了解如何选择更好的值，可以回顾第 17 章。

图 18-10 FPGA 使用执行策略

```

1 auto fpga_policy_a = fpga_device_policy<class FPGAPolicyA>{};
2
3 auto fpga_policy_b = make_fpga_policy(queue{intel::fpga_selector{}});

```

```
4  
5 constexpr auto unroll_factor = 8;  
6 auto fpga_policy_c =  
7 make_fpga_policy<class FPGAPolicyC, unroll_factor>(fpga_policy);
```

使用 DPC++ Parallel STL

为了使用 DPC++ Parallel STL，需要通过添加以下行集的子集来包含 Parallel STL 头文件。头包含哪些文件，取决于所要使用的算法：

- #include <dpstd/algorithm>
- #include <dpstd/numeric>
- #include <dpstd/memory>

dpstd::begin 和 dpstd::end 是特殊的辅助函数，允许将 SYCL 缓冲区传递给 Parallel STL 算法。这些函数接受 SYCL 缓冲区，并返回一个未指定类型的满足以下要求的对象：

- 可拷贝构造，可拷贝赋值，并且支持比较操作符 == 和 !=。
- 以下表达式是有效的：a+n, a-n, a-b，其中 a 和 b 是该类型的对象，n 是一个整数值。
- 有一个没有参数的 get_buffer，会返回 dpstd::begin 和 dpstd::end 的 SYCL 缓冲区。

要使用这些辅助函数，先将 #include <dpstd/iterators> 添加到代码中。图 18-11 和 18-12 中的代码，使用 std::fill 函数作为使用开始/结束协助器的示例。

图 18-11 使用 std::fill

```
1 #include <dpstd/execution>  
2 #include <dpstd/algorithm>  
3 #include <dpstd/iterators>  
4  
5 sycl::queue Q;  
6 sycl::buffer<int> buf { 1000 };  
7  
8 auto buf_begin = dpstd::begin(buf);  
9 auto buf_end = dpstd::end(buf);  
10  
11 auto policy = dpstd::execution::make_device_policy<class fill>( Q );  
12 std::fill(policy, buf_begin, buf_end, 42);  
13 // each element of vec equals to 42
```

减少主机和设备之间的数据复制

并行 STL 算法可以用普通（主机端）迭代器调用，如图 18-11 中的代码示例所示。

本例中，将创建临时 SYCL 缓冲区，并将数据复制到该缓冲区。设备上的临时缓冲区处理完成后，数据复制回主机。建议直接使用现有的 SYCL 缓冲区，以减少主机和设备之间的数据移动，以及避免创建和销毁缓冲区的不必要的开销。

图 18-12 使用默认策略的 std::fill

```
1 #include <dpstd/execution>
2 #include <dpstd/algorithms>
3
4 std::vector<int> v( 1000000 );
5 std::fill(dpstd::execution::default_policy, v.begin(), v.end(), 42);
6 // each element of vec equals to 42
```

图 18-13 展示了一个示例，对提供的搜索序列中的每个值执行输入序列的二进制搜索。作为搜索序列的第 i 个元素的结果，将一个搜索值是否在输入序列中找到的布尔值，赋给结果序列的第 i 个元素。该算法返回的迭代器指向赋值序列的最后一个元素的下一个位置。该算法假定输入序列已提供了比较排序，如果没有提供比较器，则使用操作符 $<$ 进行元素比较。

前面描述的复杂性强调了应该尽可能利用库函数，而不是自己编写算法实现，这可能需要大量的调试和调优时间。库作者通常都是设备架构的专家，他们可能能看到一些非公开的信息，所以当优化的库可用时，应该先使用。

图 18-13 所示的代码示例演示了使用 DPC++ 并行 STL 算法时的三个步骤：

- 创建 DPC++ 的迭代器。
- 从现有策略创建命名策略。
- 调用并行算法

图 18-13 中的示例使用 dpstd::binary_search 算法根据我们的设备选择在 CPU、GPU 或 FPGA 上执行二叉搜索。

图 18-13 使用 binary_search

```
1 #include <dpstd/execution>
2 #include <dpstd/algorithms>
3 #include <dpstd/iterator>
4
5 buffer<uint64_t, 1> kB{ range<1>(10) };
6 buffer<uint64_t, 1> vB{ range<1>(5) };
7 buffer<uint64_t, 1> rB{ range<1>(5) };
8
9 accessor k{kB};
10 accessor v{vB};
11
12 // create dpc++ iterators
13 auto k_beg = dpstd::begin(kB);
14 auto k_end = dpstd::end(kB);
15 auto v_beg = dpstd::begin(vB);
16 auto v_end = dpstd::end(vB);
17 auto r_beg = dpstd::begin(rB);
18
19 // create named policy from existing one
20 auto policy = dpstd::execution::make_device_policy<class bSearch>
```

```

21 (dpstd::execution::default_policy);
22
23 // call algorithm
24 dpstd::binary_search(policy, k_beg, k_end, v_beg, v_end, r_beg);
25
26 // check data
27 accessor r{rB};
28 if ((r[0] == false) && (r[1] == true) &&
29 (r[2] == false) && (r[3] == true) && (r[4] == true)) {
30     std::cout << "Passed.\nRun on "
31     << policy.queue().get_device().get_info<info::device::name>()
32     << "\n";
33 } else
34     std::cout << "failed: values do not match.\n";

```

并行 STL 与 USM

下面的例子描述了并行 STL 算法与 USM 结合使用的两种方法:

- 通过 USM 指针
- 通过 USM 分配器

如果有 USM 分配器，可以将指向分配开始和结束的指针传递给并行算法。要确保执行策略和分配器本身是为相同的队列或上下文创建的，以避免在运行时出现未定义行为。

如果相同的内存要由多个算法处理，可以使用有序队列，或者显式地等待每个算法完成后再在下一个算法中使用相同的内存(这是使用 USM 时的操作顺序)。也需要等待操作完成后才能访问主机上的数据，如图 18-14 所示。

或者，可以使用 std::vector 和 USM 分配器，如图 18-15 所示。

图 18-14 使用带有 USM 指针的并行 STL

```

1 #include <dpstd/execution>
2 #include <dpstd/algorithm>
3
4 sycl::queue q;
5 const int n = 10;
6 int* d_head = static_cast<int*>(
7     sycl::malloc_device(n * sizeof(int),
8         q.get_device(),
9         q.get_context()));
10
11 std::fill(dpstd::execution::make_device_policy(q),
12           d_head, d_head + n, 78);
13 q.wait();
14
15 sycl::free(d_head, q.get_context());

```

图 18-15 使用并行 STL 与 USM 分配器

```
1 #include <dpstd/execution>
2 #include <dpstd/algorithms>
3
4 sycl::queue Q;
5 const int n = 10;
6 sycl::usm_allocator<int, sycl::usm::alloc::shared>
7         alloc(Q.get_context(), Q.get_device());
8 std::vector<int, decltype(alloc)> vec(n, alloc);
9
10 std::fill(dpstd::execution::make_device_policy(Q),
11             vec.begin(), vec.end(), 78);
12 Q.wait();
```

错误处理与 DPC++ 的执行策略

如第 5 章所述，DPC++ 错误处理模型支持两种类型的错误。对于同步错误，运行时抛出异常，而异步错误只在程序执行期间，会指定用户处理程序错误的时间。

对于使用 DPC++ 策略执行的并行 STL 算法，处理所有错误（同步或异步）是调用者的责任。具体地说

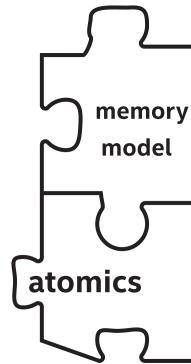
- 算法不会显式抛出异常。
- 运行时在主机 CPU 上抛出的异常（包括 DPC++ 同步异常），并传递给调用者。
- Parallel STL 不处理 DPC++ 异步错误，因此必须由程序处理。

要处理 DPC++ 异步错误，必须用错误处理程序对象创建与 DPC++ 策略关联的队列。预定义的策略对象（`default_policy` 和其他）没有错误处理程序，因此如果需要处理异步错误，应该创建自己的策略。

总结

DPC++ 库是 DPC++ 编译器的伙伴，帮助我们为异构应用程序的某些部分提供解决方案，为常用功能和并行模式使用预构建和调优库。DPC++ 库允许在内核中显式使用 C++ STL API，通过并行 STL 算法扩展简化了跨体系结构编程，并通过自定义迭代器增加了并行算法的成功应用。除了支持常用库（`libstdc++`、`libc++`、MSVS）之外，DPC++ 还提供了对 SYCL 内置函数的支持。本章概述了利用库的好处（而不是自己编写所有代码的方法），并且应该在任何可行的地方使用这种方法来简化应用程序开发，并实现更好的性能。

19 内存模型和原子操作



如果想成为优秀的并行开发者，必须了解内存一致性。它是拼图的关键部分，帮助我们确保数据在需要时就在需要的地方。这一章阐明了需要掌握的关键，以确保程序正常运行。

对于内存进行并发更新来说，对编程语言的内存（一致性）模型有基本的了解是必要的（这些更新是来自同一个内核中的多个工作项，多个设备，或者两者都是）。无论内存是如何分配的，选择使用缓冲区还是 USM 分配，本章的内容对我们来说都很重要。

前面的章节中，我们重点讨论了简单内核的开发，实例要么操作完全独立的数据，要么使用结构化通信模式共享数据，这些模式可以使用语言和/或库直接表示。当要编写更复杂和更现实的内核时，可能会遇到这样的情况：程序实例可能需要以更不结构化的方式进行通信——要完成可移植的高效程序，理解内存模型如何与 DPC++ 语言特性和硬件功能相关联是正确设计的前提。

标准 C++ 的内存一致性模型足以编写在主机设备上执行的应用程序，但 DPC++ 对其进行了修改，以解决在编写异构系统时的复杂性，以及在讨论程序实例时不能清晰地映射到 C++ 线程概念时可能出现的复杂性。

- 系统中设备可以访问哪些类型的内存分配：使用缓冲区和 USM。
- 内核执行期间防止不安全的并发内存访问（数据竞争）：使用栅栏和原子操作。
- 启用执行相同内核的程序实例之间的安全通信，以及不同设备之间的安全通信：使用组内栅栏、内存栅栏、原子操作、内存序和内存域。
- 防止与期望不兼容的优化：使用组内栅栏、内存栅栏、原子操作、内存序和内存域。
- 启用依赖于开发者的优化：使用内存序和内存域。

内存模型是个复杂的主题，可以根据兴趣进行了解——处理器架构师关心的是让处理器和加速器尽可能高效地执行代码！本章中，我们努力的打破了这种复杂性，并强调了关键概念和语言特性。本章开启了一条道路，不仅了解内存模型的内部和外部，并且享受并行编程。如果这里的描述和示例代码存在问题，强烈推荐访问本章末尾列出的网站或参考 C++、SYCL 和 DPC++ 语言规范。

内存模型中有什么？

本节详述了编程语言包含内存模型的动机，并介绍了并行开发者应该了解的几个核心概念：

- 数据竞争和同步
- 计算和内存栅栏
- 原子操作

- 内存序

要理解这些概念在 C++、SYCL 和 DPC++ 中的表达和用法，必须从高层次来考虑这些概念。具有丰富的并行编程经验，特别是使用 C++ 的读者，可以跳过前面的内容。

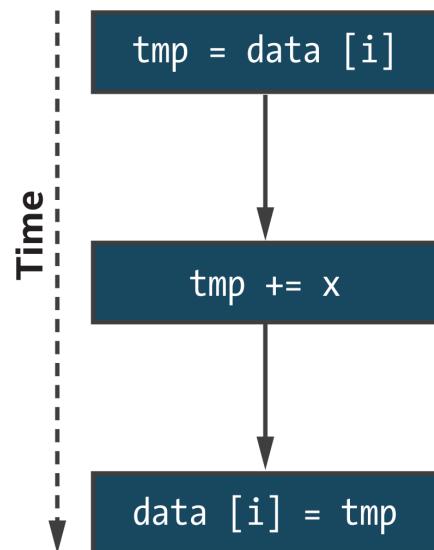
数据竞争和同步

在程序中编写的操作不会直接映射到单个硬件指令或微操作。简单的加法操作，例如 `data[i] += x`，可以分解成一系列的指令或微操作：

1. 将 `data[i]` 加载到一个临时内存中（寄存器）。
2. 计算将 `x` 添加到 `data[i]` 中。
3. 将结果存储回 `data[i]`。

这不是在开发应用程序时需要担心的事情——添加的三个阶段将按照顺序执行，如图 19-1 所示。

图 19-1 `data[i] += x` 的连续执行分为三个独立的操作



切换到并行应用程序开发带来的复杂性：如果有多个操作同时应用于相同的数据，如何确定数据的一致性？考虑图 19-2 所示的情况，其中 `data[i] += x` 的两次执行交织在一起。如果两次执行，使用了不同的 `i` 值，则应用程序将正确执行。如果使用相同的 `i` 值，会从内存中加载相同的值，并且其中一个结果会被另一个覆盖！这只是调度它们的操作之一，程序的行为取决于哪个程序实例首先到达——这就是数据竞争。

图 19-2 可能的 `data[i] += x` 交错并发执行

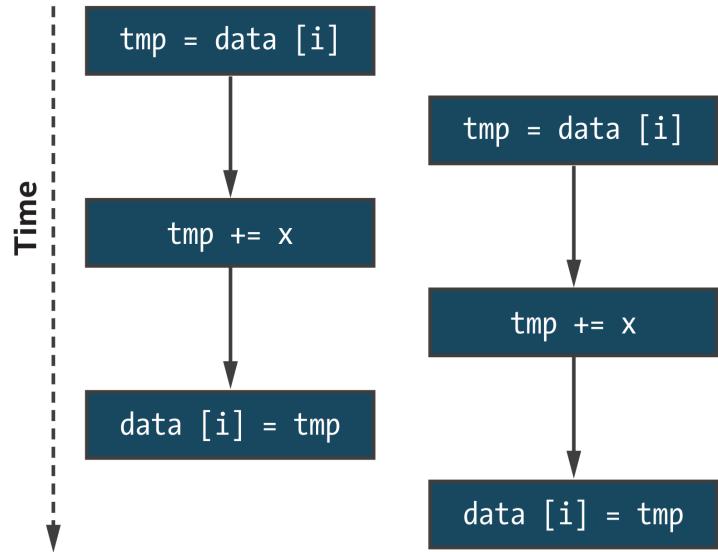


图 19-3 中的代码和图 19-4 中的输出展示了并发极易发生。如果 M 大于或等于 N，则每个程序实例中 j 的值是唯一的；如果不是，j 的值将会冲突，更新可能会丢失。我们说可能丢失是因为包含数据竞争的程序，仍然可能在某个时间点产生正确的答案（取决于实现和硬件如何安排工作）。无论编译器还是硬件都不可能知道这个程序要做什么，也不可能知道在运行时 N 和 M 的值是多少——作为开发者，有责任了解的程序是否包含数据竞争，以及是否对执行顺序敏感。

图 19-3 包含数据竞争的内核

```

1 int* data = malloc_shared<int>(N, Q);
2 std::fill(data, data + N, 0);
3
4 Q.parallel_for(N, [=](id<1> i) {
5     int j = i % M;
6     data[j] += 1;
7 }).wait();
8
9 for (int i = 0; i < N; ++i) {
10    std::cout << "data [" << i << "] = " << data[i] << "\n";
11 }
```

图 19-4 图 19-3 中的代码输出示例，用于小值 N 和 M

$N = 2, M = 2:$

data [0] = 1

data [1] = 1

$N = 2, M = 1:$

data [0] = 1

data [1] = 0

通常，开发大规模并行应用程序时，不应该关注单个工作项执行的确切顺序——可能有数百（或数千！）个工作项同时执行，而特定的顺序将对可扩展性和性能产生负面影响。相反，我们的重点应该是开发可移植的正确执行的应用程序，可以通过向编译器（和硬件）提供关于程序实例何时共享数据、共享发生时，需要什么保证，以及哪些执行顺序是合法的信息来实现这一点。

大规模并行应用程序不应该关注单个工作项执行的顺序！

计算和内存栅栏

防止同一组中工作项之间的数据竞争的方法是，使用工作组栅栏和适当的内存栅栏在不同程序实例之间引入同步。可以使用工作组栅栏对 $\text{data}[i]$ 的更新进行排序，如图 19-5 所示，图 19-6 给出了示例内核的更新版本。请注意，因为工作组栅栏不会同步不同组中的工作项，所以只有将自己限制在单个工作组中时，示例才能保证正确执行！

图 19-5 $\text{data}[i] += x$ 的两个实例被栅栏隔开

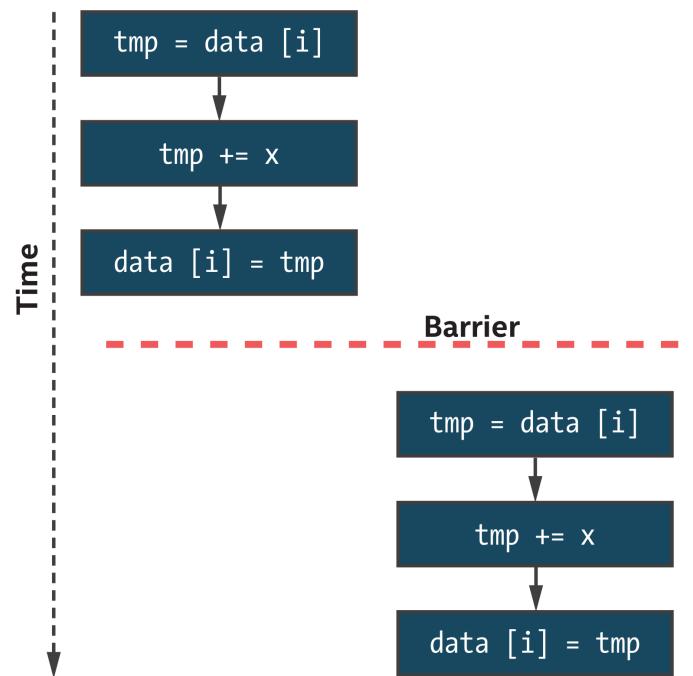


图 19-6 使用栅栏避免数据竞争

```
1 int* data = malloc_shared<int>(N, Q);
2 std::fill(data, data + N, 0);
3
4 // Launch exactly one work-group
5 // Number of work-groups = global / local
6 range<1> global{N};
7 range<1> local{N};
8
9 Q.parallel_for(nd_range<1>{global, local}, [=](nd_item<1> it) {
10     int i = it.get_global_id(0);
11     int j = i % M;
12     for (int round = 0; round < N; ++round) {
13         // Allow exactly one work-item update per round
14         if (i == round) {
15             data[j] += 1;
16         }
17         it.barrier();
18     }
19 }).wait();
20
21 for (int i = 0; i < N; ++i) {
22     std::cout << "data [" << i << "] = " << data[i] << "\n";
23 }
```

尽管使用栅栏实现此模式没什么问题，但不鼓励这样做——强制组中的工作项按顺序和特定的顺序执行，这可能导致在负载不平衡的情况下长时间的存在。还可能引入比严格要求更多的同步——如果不同的程序实例碰巧使用了不同的 *i* 值，仍需要在组内栅栏上进行同步。

栅栏同步是一种有用的工具，确保工作组或子工作组中的所有工作项在进入下一个阶段之前完成内核的某部分，但是对于细粒度的（可能依赖于数据的）同步来说，栅栏同步过重了。对于更通用的同步模式，必须使用原子操作。

原子操作

原子操作允许在不引入数据竞争的情况下并发访问内存位置。当多个原子操作访问同一内存时，保证不会重叠。如果只有一些访问具有原子性，这个保证就没意义，开发者有责任确保不使用具有不同原子性的操作，去并发访问相同的数据。

同一个内存位置同时混合原子和非原子操作会导致未定义行为的发生！

如果使用原子操作来表示加法，结果可能如图 19-8 所示——每个更新都是不可分割的工作块，并且程序将产生正确的结果。对应的代码如图 19-7 所示——在本章后面重新讨论 atomic_ref 类及其模板参数的含义。

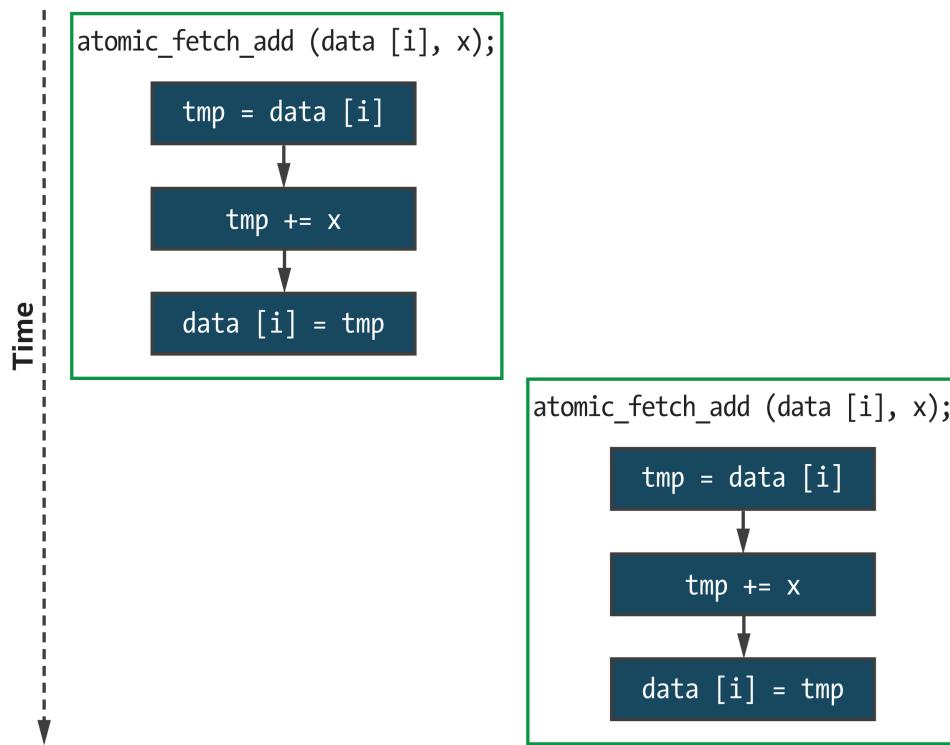
图 19-7 使用原子操作避免数据竞争

```

1 int* data = malloc_shared<int>(N, Q);
2 std::fill(data, data + N, 0);
3
4 Q.parallel_for(N, [=](id<1> i) {
5     int j = i % M;
6     atomic_ref<int, memory_order::relaxed, memory_scope::system,
7         access::address_space::global_space> atomic_data(data[j]);
8     atomic_data += 1;
9 }).wait();
10
11 for (int i = 0; i < N; ++i) {
12     std::cout << "data [" << i << "] = " << data[i] << "\n";
13 }

```

图 19-8 与原子操作交错执行的 $\text{data}[i] += x$



需要注意的是，这仍然只是一个可能的执行顺序。使用原子操作可以保证更新不重叠（如果两个实例使用相同的 i 值），但不能保证两个实例中哪一个会先执行。更重要的是，不能保证这些原子操作相对于不同程序实例中的任何非原子操作的执行顺序。

内存序

即使在顺序应用程序中，优化编译器和硬件可以对操作进行地重新排序。换句话说，应用程序的行为必须与开发者编写的程序完全一样即可。

然而，这种假设还不足以推断并行程序的执行。现在有两个需要担心的重排来源：编译器和硬件可能会在每个顺序的程序实例中重新排序语句的执行，而程序实例本身可能以任何（可能是交错

的)顺序执行。为了设计和实现程序实例之间的安全通信协议，需要能够约束这种重新排序。为编译器提供我们想要的内存顺序的信息，可以防止与应用程序的预期行为不兼容的重排优化。

常用的三种内存序

1. 自由的内存序
2. 获取-释放或释放-获取的内存序
3. 顺序一致的内存序

在自由内存排序下，内存操作可以不受任何限制地重排。自由内存模型最常见的用法是增加共享变量（例如，一个计数器，直方图计算的一个值数组）。

获取-释放内存排序下，程序实例释放一个原子变量，而另一个程序实例获取同一个原子变量，这两个程序实例充当同步点，并保证释放实例之前对内存的任何写操作对获取实例可见。通常，可以考虑原子操作将其他内存操作释放到其他程序实例，或者获取其他程序实例的内存操作。如果想通过内存程序实例对之间传递值，就需要一个内存模型，这可能比想象的更常见。当程序获得锁时，通常会执行一些额外的计算，并在最终释放锁之前修改内存——只有锁会自动更新，但是我们希望由锁保护的内存更新能够避免数据竞争。这种行为依赖于获取-释放内存排序，使用自由内存序无法实现锁。

顺序一致的内存序下，获取-释放顺序仍然有效，但所有原子操作的全局顺序一致。这种内存序行为是三种方法中最直观的，也是最接近开发顺序应用程序时的习惯。有了顺序一致性，对程序实例组（而不是成对）之间的通信进行推理就会变得容易得多。

设计可移植的并行应用程序时，必须了解编程模型和设备的组合支持哪些内存序。明确地描述应用程序所需的内存序，可以确保当需要的行为不受支持时，可以预见地失败（例如，在编译时），并避免做出不安全的假设。

内存模型

目前为止，本章已经介绍了理解内存模型所需的概念。本章的其余部分详细解释了内存模型，包括

1. 如何表达内核的内存序要求
2. 如何查询指定设备支持的内存
3. 内存模型如何对待不相交的地址空间和多个设备
4. 内存模型如何与计算栅栏、内存栅栏和原子交互
5. 缓冲区和 USM 之间使用原子操作有何不同

内存模型是基于标准 C++ 的内存模型，但在一些重要方面有所不同。这些差异反映了我们的长期愿景，即 DPC++ 和 SYCL 应该有助于预告未来的 C++ 标准：类的默认行为和命名与 C++ 标准库紧密一致，目的是扩展标准 C++ 功能，而不是限制它。

图 19-9 中的表格总结了在标准 C++(C++11, C++14, C++17, C++20) 与 SYCL 和 DCP++ 中不同的内存模型概念是如何作为语言特性使用的。C++14, C++17 和 C++20 标准还包括了一些对 C++ 实现影响的说明。这些说明不应该影响程序代码，所以这里不讨论它们。

图 19-9 比较标准 C++ 和 SYCL/DPC++ 内存模型

Feature	Standard C++	SYCL / DPC++
Atomic Objects	<code>std::atomic</code>	Not available.
Atomic References	<code>std::atomic_ref</code> (C++20 onwards)	<code>sycl::atomic_ref</code>
Memory Orders	<code>relaxed</code> <code>consume</code> <code>acquire</code> <code>release</code> <code>scq_rel</code> <code>seq_cst</code>	<code>relaxed</code> <code>acquire</code> <code>release</code> <code>scq_rel</code> <code>seq_cst</code>
Memory Scopes	Not available. Behavior of atomics and fences matches DPC++ system scope.	<code>work_item</code> <code>sub_group</code> <code>work_group</code> <code>device</code> <code>system</code>
Fences	<code>std::atomic_thread_fence</code>	<code>sycl::atomic_fence</code>
Barriers	<code>std::barrier</code> (C++20 onwards)	<code>nd_item::barrier</code> <code>sub_group::barrier</code>
Address Spaces	All memory is in a single (host) address space.	Host Device (Global) Device (Local) Device (Private) Shared (USM)

memory_order 枚举类

内存模型通过 `memory_order` 枚举类的 6 个值表示不同的内存序，这些值可以作为参数提供给内存栅栏和原子操作。为一个操作提供一个内存序参数，告诉编译器相对于该操作的所有其他内存操作 (任何地址) 需要以什么内存序，如下所述：

- `memory_order::relaxed`

读写操作可以在操作之前或之后重排，没有任何限制。没有顺序保证。

- `memory_order::acquire`

操作之后出现的读和写操作必须在该操作之后出现 (也就是说，不能在操作之前重排)。

- `memory_order::release`

操作之前出现的读操作和写操作必须在程序的操作之前发生 (即，不能在操作之后重排)，并且前面的写操作保证对其他程序实例可见，这些程序实例已经通过相应的获取操作 (即，使用相同的变量和 `memory_order::acquire` 或 `barrier` 函数的原子操作)。

- `memory_order::acq_rel`

操作既是获取也是释放。读和写操作不能围绕操作重排，前面的写操作必须像前面描述的 `memory_order::release` 一样可见。

- `memory_order::seq_cst`

该操作分别作为获取、释放或两者都起作用，具体取决于它是读操作、写操作或读-改-写操作。以顺序一致的内存序执行操作。

对于每个操作所支持的内存顺序有几个限制。图 19-10 中的表格总结了有效的组合。

图 19-10 使用 `memory_order` 支持原子操作

Functions	Supported <code>memory_order</code> Values				
	relaxed	acquire	release	acq_rel	seq_cst
load	✓	✓	✗	✗	✓
store	✓	✗	✓	✗	✓
exchange					
compare_exchange_*	✓	✓	✓	✓	✓
fetch_*					
fence	✓	✓	✓	✓	✓

加载操作不会将值写入内存，因此与释放语义不兼容。类似地，存储操作不从内存中读取值，因此与获取语义不兼容。其余的读-改-写原子操作和内存栅栏与所有内存序兼容。

C++ 中的内存序

C++ 内存模型还包括 `memory_order::consume`，其行为与 `memory_order::acquire` 类似。然而，C++17 标准不鼓励使用它，注意到它的定义正在修订。它在 DPC++ 中的实现会推迟到未来的版本。

`memory_scope` 枚举类

标准的 C++ 内存模型假设应用程序，在具有单个地址空间的单个设备上执行。这两种假设都不适用于 DPC++ 应用程序：应用程序的不同部分在不同的设备上执行（例如，一个主机设备和一个或多个加速器设备）；每个设备有多个地址空间（即，私有，本地和全局）；并且每个设备的全局地址空间可能是断开的（取决于 USM 支持）。

为了解决这个问题，DPC++ 扩展了 C++ 内存顺序的概念，以包括原子操作的范围，表示给定内存顺序约束应用的最小工作集。作用域集合是通过 `memory_scope` 枚举类定义的：

- `memory_scope::work_item`
内存序约束仅应用于调用工作项。这个作用域只对映像操作有用，因为工作项中的所有其他操作已经保证按程序顺序执行。
- `memory_scope::sub_group, memory_scope::work_group`
内存序约束仅应用于与调用工作项相同的子工作组或工作组中的工作项。
- `memory_scope::device`
内存序约束仅适用于在与调用工作项相同的设备上执行的工作项。
- `memory_scope::system`
内存排序约束适用于系统中的所有工作项。

除了设备功能所施加的限制，所有内存作用域都是所有原子操作和内存栅栏操作的有效参数。但在以下三种情况之一中，作用域参数可能会自动降级为较窄的作用域：

1. 当原子操作更新了工作组本地内存中的值，则任何比 `memory_scope::work_group` 范围更宽的作用域都会缩小（因为本地内存只对同一个工作组中的工作项可见）。
2. 当设备不支持 USM，指定 `memory_scope::system` 等同于 `memory_scope::device`（因为缓冲区不能被多个设备同时访问）。
3. 当原子操作使用 `memory_order::relaxed`，则没有顺序保证，并且内存作用域参数会被忽略。

查询设备能力

为了确保与 SYCL 以前版本支持的设备的兼容性，并最大化可移植性，DPC++ 支持 OpenCL 1.2 设备和其他可能无法支持完整 C++ 内存模型的硬件（例如，某些类型的嵌入式设备）。DPC++ 提供设备查询来帮助我们推断系统中，可用设备支持的内存序和内存范围：

- `atomic_memory_order_capabilities`
`atomic_fence_order_capabilities`
返回特定设备上原子操作和内存栅栏操作支持的所有内存序列表。所有设备都必须支持 `memory_order::relaxed`，并且主机设备必须支持所有内存序。
- `atomic_memory_scope_capabilities`
`atomic_fence_scope_capabilities`
返回特定设备上原子操作和 fence 操作支持的所有内存作用域的列表。所有设备都必须至少支持 `memory_order::work_group`，并且主机设备必须支持所有内存作用域。

一开始可能很难记住哪些内存顺序和作用域支持哪些功能和设备功能的组合。在实践中，可以通过以下两种开发方法来避免这种复杂性：

1. 开发具有顺序一致性和系统栅栏的应用程序。
性能调优期间，只考虑采用不那么严格的内存序。
2. 开发具有自由一致性和工作组栅栏的应用程序。
只有在需要正确性时，才考虑采用更严格的内存序和更广泛的作用域。

第一种方法确保所有原子操作和栅栏语义匹配标准 C++ 的默认行为。这是最简单、最不容易出错的选项，但是具有性能和可移植性最差的特征。

第二种方法更符合 SYCL 以前版本和 OpenCL 等语言的默认行为。虽然更复杂——因为它要求我们更加熟悉不同的内存序和作用域——确保我们编写的大部分 DPC++ 代码可以在任何设备上运行，而不会造成性能损失。

栅栏

目前为止，本书中所有之前使用的栅栏都忽略了内存序和作用域的问题，而是依赖于默认行为。

DPC++ 中的每一个工作组栅栏对于调用工作项可访问的所有地址空间都起到了获取-释放的作用，并且使得前面的写操作至少对同一组中的所有其他工作项可见。这确保了一组工作项在栅栏之后的内存一致性，这与我们对同步的直观理解（以及同步的定义——与 C++ 中的关系）一致。

`atomic_fence` 函数提供了比这更细粒度的控制，允许工作项以指定的内存顺序和范围执行栅栏。DPC++ 的未来版本中，工作组栅栏可能同样接受一个可选参数来调整与栅栏相关的获取-释放栅栏的内存作用域

DPC++ 中的原子操作

DPC++ 支持对各种数据类型的多种原子操作。所有设备都保证支持通用操作（例如，加载、存储、算术操作符）的原子版本，以及实现无锁算法所需的原子比较和交换操作。该语言为所有基本整数、浮点数和指针类型定义了这些操作——所有设备都必须支持 32 位类型的这些操作，但 64 位类型的支持是可选的。

atomic 类

C++11 中的 `std::atomic` 类提供了一个创建和操作原子变量的接口。原子类的实例拥有数据，不能移动或复制，只能使用原子操作进行更新。这些限制大大减少了不正确使用类和引入未定义行为的机会。它们也阻止了类在 DPC++ 内核中使用——在主机上创建原子对象并将它们传输到设备上是不可能的！我们可以在宿主代码中继续使用 `std::atomic`，但是尝试在设备内核中使用将导致编译错误。

原子类在 SYCL 2020 和 DPC++ 中已弃用

SYCL 1.2.1 规范包括一个 `cl::sycl::atomic` 类，它基于 C++11 中的 `std::atomic` 类。这两个类的接口有一些不同，最值得注意的是 SYCL 1.2.1 版本不拥有自己的数据，默认情况下使用宽松的内存序。

DPC++ 完全支持 `cl::sycl::atomic` 类，但是为了避免混淆，不建议使用它。我们建议使用 `atomic_ref` 类（将在下一节中讨论）代替它。

atomic_ref 类

C++20 中的 `std::atomic_ref` 类为原子操作提供了一个替代接口，它比 `std::atomic` 提供了更大的灵活性。这两个类之间最大的区别是 `std::atomic_ref` 的实例并不拥有数据，而是从现有的非原子变量构造而来。创建原子引用实际上相当于一个承诺，即引用的变量只在引用的生命周期内进行原子访问。这些正是 DPC++ 所需要的语义，因为它们允许在主机上创建非原子数据，将数据传

输到设备，并且只有在它传输之后才将其视为原子数据。因此，DPC++ 内核中使用的 atomic_ref 类是基于 std::atomic_ref 的。

我们说基于，因为类的 DPC++ 版本包括三个额外的模板参数，如图 19-11 所示。

图 19-11 atomic_ref 类的构造函数和静态成员

```
1 template <typename T,
2     memory_order DefaultOrder,
3     memory_scope DefaultScope,
4     access::address_space AddressSpace>
5 class atomic_ref {
6 public:
7     using value_type = T;
8     static constexpr size_t required_alignment =
9         /* implementation-defined */;
10    static constexpr bool is_always_lock_free =
11        /* implementation-defined */;
12    static constexpr memory_order default_read_order =
13        memory_order_traits<DefaultOrder>::read_order;
14    static constexpr memory_order default_write_order =
15        memory_order_traits<DefaultOrder>::write_order;
16    static constexpr memory_order default_read_modify_write_order =
17        DefaultOrder;
18    static constexpr memory_scope default_scope = DefaultScope;
19
20    explicit atomic_ref(T& obj);
21    atomic_ref(const atomic_ref& ref) noexcept;
22};
```

正如前面所讨论的，不同 DPC++ 设备的功能是不同的。为 DPC++ 的原子类选择一个默认行为比较困难：默认为标准 C++ 行为（即，memory_order::seq_cst，memory_scope::system）限制代码只能在最有能力的设备上执行；另一方面，在迁移现有 C++ 代码时，打破 C++ 约定并默认使用最小公分母（即 memory_order::relaxed，memory_scope::work_group）可能会导致意外行为。DPC++ 采用的设计提供了一种折中方案，允许我们将所需的默认行为定义为对象类型的一部分（使用 DefaultOrder 和 DefaultScope 模板参数）。其他排序和作用域可以作为运行时参数提供给特定的原子操作——DefaultOrder 和 DefaultScope 只影响那些没有或不能覆盖默认行为的操作（例如，当使用像 += 这样的简写操作符时）。模板的最后一个参数表示被引用对象分配的地址空间。

原子引用根据所引用对象的类型为不同的操作提供支持。所有类型支持的基本操作如图 19-12 所示，提供了原子地将数据移动到内存和从内存中移动数据的能力。

图 19-12。使用 atomic_ref 对所有类型进行操作

```
1 void store(T operand,
2     memory_order order = default_write_order,
3     memory_scope scope = default_scope) const noexcept;
4 T operator=(T desired) const noexcept; // equivalent to store
5
```

```

6 T load(memory_order order = default_read_order,
7     memory_scope scope = default_scope) const noexcept;
8 operator T() const noexcept; // equivalent to load
9
10 T exchange(T operand,
11    memory_order order = default_read_modify_write_order,
12    memory_scope scope = default_scope) const noexcept;
13
14 bool compare_exchange_weak(T &expected, T desired,
15    memory_order success,
16    memory_order failure,
17    memory_scope scope = default_scope) const noexcept;
18
19 bool compare_exchange_weak(T &expected, T desired,
20    memory_order order = default_read_modify_write_order,
21    memory_scope scope = default_scope) const noexcept;
22
23 bool compare_exchange_strong(T &expected, T desired,
24    memory_order success,
25    memory_order failure,
26    memory_scope scope = default_scope) const noexcept;
27
28 bool compare_exchange_strong(T &expected, T desired,
29    memory_order order = default_read_modify_write_order,
30    memory_scope scope = default_scope) const noexcept;

```

对整数和浮点类型对象的原子引用扩展了可用原子操作集，以包括算术操作，如图 19-13 和 19-14 所示。设备必须支持原子浮点类型，不管是否具有对硬件中浮点原子的支持，而且许多设备都希望使用原子比较交换来模拟原子浮点加法。这种模拟是 DPC++ 中提供性能和可移植性的重要部分，可以自由地在算法需要的地方使用浮点原子——结果代码将正确工作，并将受益于浮点原子硬件的改进而无需任何修改！

图 19-13 仅对整数类型使用 atomic_ref 的附加操作

```

1 Integral fetch_add(Integral operand,
2     memory_order order = default_read_modify_write_order,
3     memory_scope scope = default_scope) const noexcept;
4
5 Integral fetch_sub(Integral operand,
6     memory_order order = default_read_modify_write_order,
7     memory_scope scope = default_scope) const noexcept;
8
9 Integral fetch_and(Integral operand,
10    memory_order order = default_read_modify_write_order,
11    memory_scope scope = default_scope) const noexcept;
12
13 Integral fetch_or(Integral operand,
14     memory_order order = default_read_modify_write_order,

```

```

15     memory_scope scope = default_scope) const noexcept;
16
17 Integral fetch_min(Integral operand,
18     memory_order order = default_read_modify_write_order,
19     memory_scope scope = default_scope) const noexcept;
20
21 Integral fetch_max(Integral operand,
22     memory_order order = default_read_modify_write_order,
23     memory_scope scope = default_scope) const noexcept;
24
25 Integral operator++(int) const noexcept;
26 Integral operator--(int) const noexcept;
27 Integral operator++() const noexcept;
28 Integral operator--() const noexcept;
29 Integral operator+=(Integral) const noexcept;
30 Integral operator-=(Integral) const noexcept;
31 Integral operator&=(Integral) const noexcept;
32 Integral operator|=(Integral) const noexcept;
33 Integral operator^=(Integral) const noexcept;

```

图 196-4 仅用于浮点类型的 atomic_ref 的附加操作

```

1 Floating fetch_add(Floating operand,
2     memory_order order = default_read_modify_write_order,
3     memory_scope scope = default_scope) const noexcept;
4
5 Floating fetch_sub(Floating operand,
6     memory_order order = default_read_modify_write_order,
7     memory_scope scope = default_scope) const noexcept;
8
9 Floating fetch_min(Floating operand,
10    memory_order order = default_read_modify_write_order,
11    memory_scope scope = default_scope) const noexcept;
12
13 Floating fetch_max(Floating operand,
14    memory_order order = default_read_modify_write_order,
15    memory_scope scope = default_scope) const noexcept;
16
17 Floating operator+=(Floating) const noexcept;
18 Floating operator-=(Floating) const noexcept;

```

缓冲区中使用原子操作

如上一节所讨论的，在 DPC++ 中没有办法分配原子数据，也不能在主机和设备之间移动。将原子操作与缓冲区结合使用，必须创建非原子数据缓冲区，以便将其传输到设备，然后通过原子引用访问该数据。

图 19-15 通过显式创建的 atomic_ref 访问缓冲区

```
1 Q.submit([&](handler& h) {
2     accessor acc{buf, h};
3     h.parallel_for(N, [=](id<1> i) {
4         int j = i % M;
5         atomic_ref<int, memory_order::relaxed, memory_scope::system,
6             access::address_space::global_space> atomic_acc(acc[j]);
7         atomic_acc += 1;
8     });
9 });
```

图 19-15 中的代码是在 DPC++ 中使用显式创建的原子引用对象表示原子性的示例。缓冲区存储普通整数，需要具有读和写权限的访问器。然后，可以为每个数据访问创建 atomic_ref 实例，使用 `+=` 操作符作为 `fetch_add` 成员函数进行替代。

如果想在同一个内核中混合对缓冲区的原子和非原子访问，这个模式是有用的，以避免在不需要原子操作时产生性能开销。如果已知缓冲区中只有一个内存位置的子集将被多个工作项并发访问，那么只需要在访问那个子集时使用原子引用即可。或者，如果已知同一个工作组中的工作项仅在内核的一个阶段（即两个工作组栅栏之间）同时访问本地内存，那么只需要在该阶段使用原子引用。

有时，我们很乐意为每个访问支付原子性的开销，要么是因为为了正确性，要么是因为关心生产力而不是性能。对于这种情况，DPC++ 为声明访问器必须始终使用原子操作提供了一种简写，如图 19-16 所示。

图 19-16 通过原子访问器隐式创建的 atomic_ref 访问缓冲区

```
1 buffer buf(data);
2
3 Q.submit([&](handler& h) {
4     atomic_accessor acc(buf, h, relaxed_order, system_scope);
5     h.parallel_for(N, [=](id<1> i) {
6         int j = i % M;
7         acc[j] += 1;
8     });
9 });
```

缓冲区像以前一样存储普通整数，但我们将常规访问器替换为特殊的 `atomic_accessor` 类型。这样的原子访问器自动使用原子引用包装的每个成员，从而简化了内核代码。

最好是直接使用原子引用类，还是通过访问器使用。我们的建议是在原型设计和初始开发期间从访问器开始（只是简单），只有在性能调优期间（例如，如果分析显示原子性操作是性能瓶颈）或者原子性只在定义良好的内核阶段才需要（例如，在本章后面的直方图代码中可见）。

统一共享内存中使用原子

如图 19-17（从图 19-7 中复制）所示，可以用与缓冲区完全相同的方式从 USM 中存储的数据构造原子引用。实际上，此代码和图 19-15 中所示的代码之间的唯一区别是 USM 代码不需要缓冲区或访问器。

图 19-17 通过显式创建的 atomic_ref 实现 USM 分配

```
1 q.parallel_for(range<1>(N), [=](size_t i) {
2     int j = i % M;
3     atomic_ref<int, memory_order::relaxed, memory_scope::system,
4         access::address_space::global_space> atomic_data(data[j]);
5     atomic_data += 1;
6 }) . wait();
```

没有办法只使用标准的 DPC++ 特性，来模仿原子访问器为 USM 指针提供的简写语法的方法。我们希望未来的 DPC++ 版本能够在 C++23 中，为的 mspan 类提供缩写。

实际中的原子操作

原子操作的用法广泛和多样，以至于不可能在本书中提供每种用法的示例。我们展示了两个具有代表性的例子，它们具有广泛的适用性：

1. 计算直方图
2. 实现设备范围内的同步

计算直方图

图 19-18 中的代码演示了如何使用自由原子操作和工作组栅栏来计算直方图。内核被栅栏分为三个阶段，每个阶段都有自己的原子性需求。这个栅栏既是一个同步点，又是一个获取-释放栅栏——这确保了一个阶段中的任何读和写，对后面阶段中工作组的所有工作项都是可见的。

第一阶段将某个工作组本地内存的内容设置为零。每个工作组中的工作项不能按照设计争用条件更新工作组本地内存中的位置，并且不需要原子性。

第二阶段将部分直方图结果累加到本地内存中。同一个工作组中的工作项可以更新工作组本地内存中的相同位置，但是同步可以推迟到该阶段的结束——可以使用 memory_order::relaxed 和 memory_scope::work_group 来满足原子性需求。

第三阶段将部分直方图结果存储在全局内存中。保证同工作组中的工作项从工作组本地内存中的位置读取，但是会更新全局内存中的相同位置——不再需要工作组本地内存的原子性，可以像以前一样使用 memory_order::relaxed 和 memory_scope::system 来满足全局内存的原子性要求。

图 19-18 使用不同内存空间中的原子操作计算直方图

```
1 // Define shorthand aliases for the types of atomic needed by this kernel
2 template <typename T>
3 using local_atomic_ref = atomic_ref<
4     T,
5     memory_order::relaxed,
6     memory_scope::work_group,
7     access::address_space::local_space>;
8
9 template <typename T>
10 using global_atomic_ref = atomic_ref<
```

```

11 T,
12 memory_order::relaxed,
13 memory_scope::system,
14 access::address_space::global_space>;
15
16 Q.submit([&](handler& h) {
17     auto local = local_accessor<uint32_t, 1>{B, h};
18     h.parallel_for(
19         nd_range<1>{num_groups * num_items, num_items}, [=](nd_item<1> it){
20             // Phase 1: Work-items co-operate to zero local memory
21             for (int32_t b=it.get_local_id(0); b < B; b+=it.get_local_range(0)){
22                 local[b]=0;
23             }
24
25             it.barrier(); // Wait for all to be zeroed
26
27             // Phase 2: Work-groups each compute a chunk of the input
28             // Work-items co-operate to compute histogram in local memory
29             auto grp=it.get_group();
30
31             const auto [group_start, group_end] = distribute_range(grp, N);
32             for (int i = group_start + it.get_local_id(0); i < group_end;
33                  i +=it.get_local_range(0)){
34
35                 int32_t b = input[i] % B;
36                 local_atomic_ref<uint32_t>(local[b])++;
37             }
38             it.barrier(); // Wait for all local histogram updates to complete
39
40             // Phase 3: Work-items co-operate to update global memory
41             for (int32_t b = it.get_local_id(0); b < B; b +=it.get_local_range(0)){
42
43                 global_atomic_ref<uint32_t>(histogram[b]) += local[b];
44             }
45         });
46     }).wait();

```

实现设备范围内的同步

第 4 章警告不要编写跨工作组同步工作项的内核。然而，我们完全期望本章的几个示例能在原子操作之上实现对设备范围的同步。

设备范围内的同步目前是不可移植的，最好留给专业开发者使用。语言的未来版本将解决这个问题。

本节中讨论的代码是危险的，不应该期望在所有设备上工作，因为在调度和并发保证方面可能存在差异。原子操作提供的内存序保证与前进进度保证是正交的；在编写本文时，SYCL 和 DPC++

中的工作组调度完全是由实现定义的。讨论执行模型和调度保证所需的概念和术语是目前学术研究领域的热门，DPC++ 的未来版本预计将在此工作的基础上提供额外的调度查询和控制。

图 19-19 展示了一个设备级锁存器 (一次性栅栏) 的简单实现，图 19-20 展示了一个简单的使用示例。每个工作组选择一个单独的工作项，以发出该组到达的信号，并使用自旋环等待其他组，而其他工作项使用工作组栅栏等待所选的工作项。正是这种自旋环使得设备范围的同步不安全，当工作组还没有开始执行，或者当前执行的工作组没有平衡地调度，代码可能会死锁。

如果没有进度保证，单独依赖内存序来实现同步原语可能会导致死锁！

要使代码正确工作，必须满足以下三个条件：

1. 为了保证生成正确的内存栅栏，原子操作必须使用严格的内存序。
2. ND-Range 中的每个工作组必须能够执行，避免单个工作组在循环中自旋，从而使尚未增加计数器的工作组挨饿。
3. 设备必须能够同时执行 ND-Range 中的所有工作组，以确保 ND-Range 中的所有工作组最终达到门闩处。

图 19-19 原子引用的顶部构建一个简单的设备级闩锁

```
1 struct device_latch {
2     using memory_order = intel::memory_order;
3     using memory_scope = intel::memory_scope;
4
5     explicit device_latch(size_t num_groups) :
6         counter(0), expected(num_groups) {}
7
8     template <int Dimensions>
9     void arrive_and_wait(nd_item<Dimensions>& it) {
10        it.barrier();
11        // Elect one work-item per work-group to be involved
12        // in the synchronization
13        // All other work-items wait at the barrier after the branch
14        if (it.get_local_linear_id() == 0) {
15            atomic_ref<
16                size_t,
17                memory_order::acq_rel,
18                memory_scope::device,
19                access::address_space::global_space> atomic_counter(counter);
20
21            // Signal arrival at the barrier
22            // Previous writes should be visible to
23            // all work-items on the device
24            atomic_counter++;
25
26            // Wait for all work-groups to arrive
27            // Synchronize with previous releases by
```

```

28     // all work-items on the device
29     while (atomic_counter.load() != expected) {}
30   }
31   it.barrier();
32 }
33 size_t counter;
34 size_t expected;
35 };

```

图 19-20 使用图 19-19 中的设备级闩锁

```

1 // Allocate a one-time-use device_latch in USM
2 void* ptr = sycl::malloc_shared(sizeof(device_latch), Q);
3 device_latch* latch = new (ptr) device_latch(num_groups);
4 Q.submit([&](handler& h) {
5   h.parallel_for(R, [=](nd_item<1> it) {
6     // Every work-item writes a 1 to its location
7     data[it.get_global_linear_id()] = 1;
8
9     // Every work-item waits for all writes
10    latch->arrive_and_wait(it);
11
12    // Every work-item sums the values it can see
13    size_t sum = 0;
14    for (int i = 0; i < num_groups * items_per_group; ++i) {
15      sum += data[i];
16    }
17    sums[it.get_global_linear_id()] = sum;
18  });
19 }).wait();
20 free(ptr, Q);

```

虽然这段代码不能保证可移植性，但强调两个关键点：1)DPC++ 具有足够的表达能力，可以实现特定于设备的调优，有时需要以牺牲可移植性为代价；2)DPC++ 包含了实现高级同步例程所需的构建块，这可能会包含在语言的未来版本中。

总结

本章提供了内存模型和原子类的高级介绍。理解如何使用（以及如何不使用）这些类是正确开发、可移植和高效并行程序的关键。

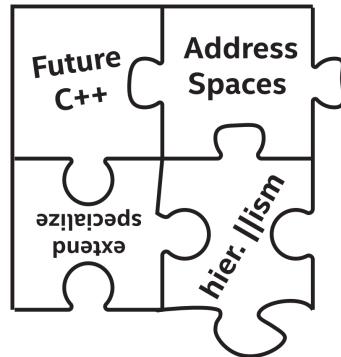
内存模型是非常复杂的主题，重点是为编写真实的应用程序建立基础。如果需要更多的信息，下面有一些专门介绍内存模型的网站、书籍和讲座。

更多信息

- A. Williams, C++ Concurrency in Action: Practical Multithreading, Manning, 2012, 978-1933988771

- H.Sutter, “atomic<> Weapons: The C++ Memory Model and Modern Hardware”, <https://herbsutter.com/weapons-the-c-memory-model-and-modern-hardware/>
- H-J.Boehm, “Temporarily discourage memory_order_consume,” <http://wg21.link/p0371>
- C++ Reference, “`std::atomic`,” <https://en.cppreference.com/w/cpp/atomic/atomic>
- C++ Reference, “`std::atomic_ref`,” https://en.cppreference.com/w/cpp/atomic/atomic_ref

20 结语：DPC++ 的未来方向



现在让我们感受一下平静，因为我们终于了解了使用 SYCL 和 DPC++ 编程的所有知识。所有的谜团都已解开。

但还是要注意，这本书是在 SYCL 和 DPC++ 刚出现的时候写的。随着第一个 DPC++ 规范和 SYCL 2020 临时规范的发布，这是一个快速发展的时期，我们努力确保代码示例，与开源 DPC++ 编译器编译的时候（2020 年第三季）和执行广泛的硬件上时，出版了本书。但是，本结语中显示的未来代码到 2020 年年中还不能使用任何编译器进行编译。

结语中，我们对未来进行了展望。我们的水晶球可能有点难以解读。

这本书的绝大部分内容将会流传很长时间。也就是说，这是一个热门的区域，而且正在发生的变化可能会破坏我们已经了解的一些知识。这包括一些最初作为供应商扩展出现的项目，后来纳入了规范（比如子工作组和 USM）。如此多的新特性将成为下一个 SYCL 标准的一部分，但这也使得讨论这些特性变得复杂：我们应该将这些特性称为供应商扩展、SYCL 的实验/临时特性，还是 SYCL 的一部分？

这个结语提供了一个即将到来的 DPC++ 特性的先睹所快，我们对这些特性感到非常兴奋。但在本书出版时，这些特性还没有完全完成。不保证本结语中的代码示例可以编译：一些可能已经与本书之后发布的 SYCL 或 DPC++ 编译器兼容，而另一些可能需要经过一些语法调整后才编译。一些特性可能作为扩展发布或合并到未来的标准中，而其他特性可能无限期地保持实验特性。随着本书的发展，GitHub 存储库中的代码样本可能会更新，以使用新的语法。同样地，我们将为这本书提供勘误表。我们建议检查这两个地方的更新（代码库和图书勘误表链接可以在第 1 章中找到）。

与 C++20 和 C++23 对齐

保持 SYCL、DPC++ 和 ISO C++ 之间的紧密一致有两个优点。首先，使 SYCL 和 DPC++ 能够利用标准 C++ 的最新和最伟大的特性来提高开发人员的生产力。其次，增加了 SYCL 或 DPC++ 中引入的异构编程特性，以及影响标准 C++ 未来发展方向（例如，executor）的机会。

SYCL 1.2.1 是基于 C++11 的，而对 SYCL 2020 和 DPC++ 接口的许多最大的改进都是在 C++14（例如，通用 Lambda）和 C++17（例如，类模板参数演绎-CTAD）中引入的语言特性中才可能实现的。

C++20 规范是在 2020 年发布的（当时我们正在写这本书！）它包括一些已经被 DPC++ 和 SYCL 预先采用的特性（例如，`std::atomic_ref`, `std::bit_cast`），随着我们走向 SYCL 的下一个官方版本（2020 之后的临时版本）和 DPC++ 的下一个版本，我们期望与 C++20 更紧密地结合。例

如，C++20 以 std::latch 和 std::barrier 的形式引入了一些额外的线程同步例程；我们已经在第 19 章探讨了如何使用类似的接口来定义设备范围的栅栏，并且在 C++20 的新语法中重新检查子工作组和工作组的栅栏也是有意义的

C++23 的工作已经开始了，因为规范还没有最终定稿，所以在 SYCL 或 DPC++ 规范中采用任何这些特性都将是一个错误——这些特性可能会在进入 C++23 之前发生重大变化，导致难以修复的不兼容性。然而，有许多正在讨论的特性可能会改变未来 SYCL 和 DPC++ 程序的形式和行为。最令人兴奋的提议特性之一是 mdspan，这是一个非拥有的数据内存，它为指针提供多维数组语法，并提供一个 AccessorPolicy 作为控制对底层数据访问的扩展点。这些语义与 SYCL 访问器的语义非常相似，mdspan 将使访问器类语法能够用于缓冲区和 USM 分配，如图 EP-1 所示。

图 EP-1 使用 mdspan 将类似访问器的索引附加到 USM 指针

```
1 queue Q;
2 constexpr int N = 4;
3 constexpr int M = 2;
4 int* data = malloc_shared<int>(N * M, Q);
5 stdex::mdspan<int, N, M> view{data};
6 Q.parallel_for(range<2>{N, M}, [=](id<2> idx) {
7     int i = idx[0];
8     int j = idx[1];
9     view(i, j) = i * M + j;
10 }).wait();
```

希望 mdspan 成为标准 C++ 只是时间问题。同时，我们建议感兴趣的读者尝试参考 Kokkos 项目的开源产品质量参考实现。

另一个令人兴奋的特性是 std::simd 类模板它试图为 C++ 中的显式向量并行提供可移植的接口。采用这个接口将在第 11 章中描述的两种不同的向量类型使用之间提供明确的区别：使用向量类型来方便程序员，使用向量类型来进行底层性能调优。同一种语言中同时支持 SPMD 和 SIMD 编程风格也提出了一些有趣的问题：应该如何声明内核使用哪种风格，是否能够在同一个内核中混合和匹配样式？希望未来的供应商扩展能够探索这些问题，因为在标准化之前，供应商会在这个领域进行试验。

地址空间

正如前面章节中所看到的，简单的代码会因为内存空间的存在变得复杂。我们可以自由地使用常规 C++ 指针，但有时候需要使用 multi_ptr 类，并显式地指定期望支持的地址空间。

许多现代体系结构通过为所谓的通用地址空间提供硬件支持来解决这个问题，指针可以指向任何内存空间中的内存，因此我们（和编译器！）可以利用运行时查询，特化不同内存空间需要不同处理的情况下的代码（例如，访问工作组本地内存可能使用不同的指令）。对泛型地址空间的支持在其他编程语言中已经可用，比如 OpenCL，预计 SYCL 的未来版本将采用默认泛型代替推理规则。

这一更改将极大地简化许多代码，并使 multi_ptr 类成为可选的性能调优特性，而不是正确性所必需的特性。图 EP-2 显示了一个使用现有地址空间编写的简单类，图 EP-3 和 EP-4 显示了通过引入通用地址空间可以实现的两种可选设计。

图 EP-2 类中存储指向特定地址空间的指针

```
1 // Pointers in structs must be explicitly decorated with address space
2 // Supporting both address spaces requires a template parameter
3 template <access::address_space AddressSpace>
4 struct Particles {
5     multi_ptr<float, AddressSpace> x;
6     multi_ptr<float, AddressSpace> y;
7     multi_ptr<float, AddressSpace> z;
8 };
```

图 EP-3 类中存储指向泛型地址空间的指针

```
1 // Pointers in structs default to the generic address space
2 struct Particles {
3     float* x;
4     float* y;
5     float* z;
6 };
```

图 EP-4 类中存储带有可选地址空间的指针

```
1 // Template parameter defaults to generic address space
2 // User of class can override address space for performance tuning
3 template <access::address_space AddressSpace =
4 access::address_space::generic_space>
5
6 struct Particles {
7     multi_ptr<float, AddressSpace> x;
8     multi_ptr<float, AddressSpace> y;
9     multi_ptr<float, AddressSpace> z;
10 };
```

扩展与更特化的机制

第 12 章引入了一组查询，使主机能够在运行时提取有关设备的信息。这些查询允许针对特定设备调优工作组大小等运行时参数进行查询，并允许将实现不同算法的不同内核分派到不同类型的设备。

未来的版本预计将使用编译时查询来扩充这些运行时查询，允许基于实现是否理解供应商扩展而对代码进行特化。图 EP-5 显示了如何使用预处理器来检测编译器是否支持特定的供应商扩展

图 EP-5 使用 #ifdef 检查 Intel 子工作组的编译器扩展支持情况

```
1 #ifdef SYCL_EXT_INTEL_SUB_GROUPS
2 sycl::ext::intel::sub_group sg = it.get_sub_group();
3 #endif
```

我们还计划引入编译时查询，使内核能够根据目标设备的属性（我们称之为方面）进行特化（例如，设备类型、对特定扩展的支持、工作组本地内存的大小、编译器选择的子工作组大小）。这些的常量表达式，目前在 C++ 中不存在——在编译主机代码时不一定是 `constexpr`，但在目标设备时就变成了 `constexpr`。用于公开设备 `constexpr` 的机制仍在设计中。我们希望 SYCL 2020 临时规范引入的专用常量化特性，并且形式和行为上类似于图 EP-6 中所示。

图 EP-6 内核编译时基于设备方面特化内核代码

```
1 h.parallel_for(..., [=](item<1> it) {
2     if devconstexpr (this_device().has<aspect::cpu>()) {
3         /* Code specialized for CPUs */
4     }
5     else if devconstexpr (this_device().has<aspect::gpu>()) {
6         /* Code specialized for GPUs */
7     }
8 });
```

分层并行

第 4 章中提到的，我们认为旧版本 SYCL 中的分层并行是一种实验性特性，在使用新的语言特性时，预计会比基本的数据并行和 ND-Range 内核慢一些。

DPC++ 和 SYCL 2020 中有很多新的语言特性，其中一些与分层并行不兼容（例如子工作组、组算法、归约减）。消除这种差异将有助于提高程序员的工作效率，并为一些简单的情况提供了更紧凑的语法。图 EP-7 中的代码显示了将归约支持扩展到层次并行的可能，从而实现层次归约：每个工作组计算一个总和，内核作为一个整体计算所有工作组中所有总和的最大值。

图 EP-7 使用层次并行进行层次归约

```
1 h.parallel_for_work_group(N, reduction(max, maximum<>()),
2 [=](group<1> g, auto& max) {
3     float sum = 0.0f;
4     g.parallel_for_work_item(M, reduction(sum, plus<>()),
5     [=](h_item<1> it, auto& sum) {
6         sum += data[it.get_global_id()];
7     });
8     max.combine(sum);
9 });
```

第 4 章中简要提到的分层并行性的另一个方面是实现的复杂性。将嵌套的并行性映射到加速器并不是 SYCL 或 DPC++ 特有的挑战，这个主题是很多人感兴趣和研究的课题。随着在实现分层并行性和不同设备的能力方面的经验积累，我们期望 SYCL 和 DPC++ 中的语法能够与标准保持一致。

总结

关于 SYCL 和 DPC++ 已经有很多令人兴奋的事情了，而这仅仅是开始！我们（作为一个社区）还有很长的路要走，我们需要持续不断的努力来提炼异构编程的能力，并设计新的语言特性，在性能、可移植性和生产力之间达到理想的平衡。

我们需要你的帮助！如果 SYCL 或 DPC++ 缺少您喜欢的 C++（或任何其他编程语言）特性，请联系我们。我们可以共同塑造 SYCL、DPC++ 和 ISO C++ 的未来方向。

更多信息

- Khronos SYCL Registry, www.khronos.org/registry/SYCL/
- J. Hoberock et al., “C++ 的执行提案,” <http://wg21.link/p0443>
- H. Carter Edwards et al., “mdspan: 非拥有多维数组引用,” <http://wg21.link/p0009>
- D. Hollman et al., “生产型 mdspan 实现,” <https://github.com/kokkos/mdspan>