

数据并行 C++

使用 C++ 和 SYCL 编程加速系统

杨丰

目录

1 介绍	21
1.1 阅读书籍，而不是标准说明书	21
1.2 SYCL 2020 和 DPC++	22
1.3 为什么不使用 CUDA?	22
1.4 为什么使用带有 SYCL 的标准 C++?	23
1.5 获取支持 SYCL 的 C++ 编译器	23
1.6 Hello, world! 和 SYCL 程序剖析	24
1.7 队列和操作	25
1.8 一切都与并行性有关	25
1.8.1 吞吐量	26
1.8.2 延迟	26
1.8.3 并行思维	26
1.8.4 阿姆达尔和古斯塔夫森	26
1.8.5 规模效应	27
1.8.6 异构系统	27
1.8.7 数据并行编程	29
1.9 带有 SYCL 的 C++ 的关键属性	29
1.9.1 单源	29
1.9.2 主机	30
1.9.3 设备	30
1.9.4 Kernel 代码	30
1.9.5 异步执行	30

目录	2
----	---

1.9.6 当我们犯错误时的竞争条件	33
1.9.7 死锁	35
1.9.8 C++ Lambda 表达式	35
1.9.9 功能可移植性和性能可移植性	38
1.10 并发与并行	39
1.11 总结	40
2 代码执行位置	41
2.1 单源	41
2.1.1 主机代码	42
2.1.2 设备代码	43
2.2 选择设备	44
2.3 方法 #1: 在任何类型的设备上运行	44
2.3.1 队列	45
2.3.2 当任何设备都可以时将队列绑定到设备	49
2.4 方法 #2: 使用 CPU 设备进行开发、调试和部署	49
2.5 方法 #3: 使用 GPU (或其他加速器)	52
2.5.1 加速器设备	52
2.5.2 设备选择器	52
2.6 方法 #4: 使用多个设备	56
2.7 方法 #5: 自定义 (非常具体) 的设备选择	57
2.7.1 根据设备方面 (Aspect) 进行选择	57
2.7.2 通过自定义选择器进行选择	58
2.8 在设备上创建任务	59
2.8.1 任务图简介	60
2.8.2 设备代码在哪里?	61
2.8.3 行动	64
2.8.4 主机任务	66
2.9 总结	68
3 数据管理	69
3.1 介绍	69
3.2 数据管理问题	70
3.3 本地设备与远程设备	70

3.4	管理多个内存	70
3.4.1	显式数据移动	71
3.4.2	隐式数据	71
3.4.3	选择正确的策略	72
3.5	USM、Buffer 和 Images	72
3.6	统一共享内存	72
3.6.1	通过指针访问内存	73
3.6.2	USM 和数据移动	74
3.7	Buffer	77
3.7.1	创建 Buffer	77
3.7.2	访问 Buffer	77
3.7.3	访问模式	78
3.8	对数据的使用进行排序	79
3.8.1	有序队列	82
3.8.2	无序队列	82
3.9	选择数据管理策略	89
3.10	Handler 类：关键成员	91
3.11	总结	93
4	表达并行性	94
4.1	Kernel 内的并行性	94
4.2	循环与 Kernel	95
4.3	多维 Kernel	96
4.4	语言特性概述	97
4.4.1	将 Kernel 与主机代码分离	97
4.5	不同形式的并行 Kernel	98
4.6	基础数据并行 Kernel	99
4.6.1	了解基本数据并行 Kernel	100
4.6.2	编写基本数据并行 Kernel	101
4.6.3	基本数据并行 Kernel 的详细信息	103
4.7	显式 ND 范围 Kernel	105
4.7.1	了解显式 ND 范围并行 Kernel	106
4.7.2	编写显式 ND 范围数据并行 Kernel	107
4.7.3	显式 ND 范围数据并行 Kernel 的详细信息	112

4.8 将计算映射到 Work-Items	116
4.8.1 一对映射	116
4.8.2 多对映射	116
4.9 选择 Kernel 形式	117
4.10 总结	119
5 错误处理	120
5.1 安全第一	120
5.2 错误类型	120
5.3 让我们创建一些错误!	122
5.3.1 同步错误	122
5.3.2 异步错误	123
5.4 应用程序错误处理策略	124
5.4.1 忽略错误处理	124
5.4.2 同步错误处理	125
5.4.3 异步错误处理	127
5.4.4 异步处理程序	128
5.4.5 处理程序的调用	130
5.5 设备上的错误	131
5.6 总结	131
6 统一共享内存	132
6.1 为什么要使用 USM?	132
6.2 分配类型	132
6.2.1 设备分配	133
6.2.2 主机分配	133
6.2.3 共享分配	133
6.3 分配内存	134
6.3.1 我们需要知道什么?	134
6.3.2 多种风格	135
6.3.3 释放内存	141
6.3.4 分配示例	142
6.4 数据管理	143
6.4.1 初始化	143

6.4.2 数据移动	143
6.5 查询	149
6.6 还有一件事	151
6.7 总结	152
7 Buffers	153
7.1 Buffers	153
7.1.1 Buffer 创建	154
7.1.2 Buffer 属性	158
7.1.3 我们可以用 Buffer 做什么?	159
7.2 访问器	160
7.2.1 访问器创建	163
7.2.2 我们可以用访问器做什么?	167
7.3 总结	168
8 调度 Kernel 和数据移动	169
8.1 什么是图调度?	169
8.2 SYCL 中的 Graph 如何工作	170
8.2.1 命令组行动	170
8.2.2 命令组如何声明依赖关系	170
8.2.3 例子	171
8.2.4 命令组的各个部分何时执行?	179
8.3 数据移动	179
8.3.1 显式数据移动	179
8.3.2 隐式数据移动	180
8.4 与主机同步	181
8.5 总结	182
9 通讯与同步	184
9.1 Work-Groups 和 Work-Items	184
9.2 高效通讯的基石	185
9.2.1 通过屏障 (Barriers) 进行同步	185
9.2.2 Work-Groups 本地内存	187
9.3 使用 Work-Groups Barrier 和本地内存	188

9.3.1 ND 范围 Kernel 中的 Work-Groups Barrier 和本地内存	191
9.4 Sub-Groups	194
9.4.1 通过 Sub-Groups Barrier 进行同步	195
9.4.2 在 Sub-Groups 内交换数据	195
9.4.3 完整 Sub-Groups ND 范围 Kernel 示例	198
9.5 组函数和组算法	199
9.5.1 广播 Broadcast	199
9.5.2 投票 Votes	199
9.5.3 洗牌 Shuffles	200
9.6 总结	202
10 定义 Kernel	204
10.1 为什么用三种方式来表示 Kernel?	205
10.2 作为 Lambda 表达式的 Kernel	205
10.2.1 Kernel Lambda 表达式的元素	206
10.2.2 识别 Kernel Lambda 表达式	208
10.3 Kernel 作为命名函数对象	209
10.3.1 Kernel 命名函数对象的元素	210
10.4 Kernel 包中的 Kernel	211
10.5 与其他 API 的互操作性	215
10.6 总结	216
11 向量和数学数组	217
11.1 向量类型的歧义	217
11.2 我们对于 SYCL 向量类型的心智模型	218
11.3 数学数组 (marray)	219
11.4 矢量 (vec)	221
11.4.1 加载和存储	221
11.4.2 与后端本机向量类型的互操作性	222
11.4.3 Swizzle 操作	223
11.5 向量类型如何执行	225
11.5.1 向量作为便利类型	226
11.5.2 作为 SIMD 类型的向量	229
11.6 总结	230

12 设备信息和 Kernel 特化	231
12.1 是否有 GPU?	231
12.2 细化 Kernel 代码使其更加规范	232
12.3 如何枚举设备和功能	233
12.3.1 Aspect	236
12.3.2 自定义设备选择器	238
12.3.3 好奇: get_info<>	239
12.3.4 更好奇: 详细的枚举代码	241
12.3.5 非常好奇: get_info 加上 has()	242
12.4 设备信息描述符	242
12.5 设备特定的 Kernel 信息描述符	242
12.6 细节: “正确性”的细节	242
12.6.1 设备查询	243
12.6.2 Kernel 查询	244
12.7 具体内容: “调整/优化”的具体内容	245
12.7.1 设备查询	245
12.7.2 Kernel 查询	245
12.8 运行时与编译时属性	245
12.9 Kernel 特化	246
12.10 总结	248
13 实用技巧	249
13.1 获取代码示例和编译器	249
13.2 在线资源	249
13.3 平台模型	249
13.3.1 多架构二进制文件	250
13.3.2 编译模型	251
13.4 上下文: 需要了解的重要事项	253
13.5 将 SYCL 添加到现有 C++ 程序	254
13.6 使用多个编译器时的注意事项	255
13.7 调试	255
13.7.1 调试死锁和其他同步问题	257
13.7.2 调试 Kernel 代码	257
13.7.3 调试运行时故障	258

13.7.4 队列分析和由此产生的计时功能	260
13.7.5 跟踪和分析工具接口	263
13.8 初始化数据并访问 Kernel 输出	264
13.9 多个翻译单元	272
13.9.1 多个翻译单元的性能影响	272
13.10 当匿名 Lambda 需要名称时	273
13.11 总结	273
14 常见的并行模式	274
14.1 理解模式	274
14.1.1 映射 Map	275
14.1.2 模版 Stencil	277
14.1.3 归约 Reduction	278
14.1.4 扫描 Scan	279
14.1.5 打包和拆包	280
14.2 使用内置函数和库	282
14.2.1 SYCL 归约库	282
14.2.2 集体算法	287
14.3 直接编程	290
14.3.1 映射 Map	291
14.3.2 模版 Stencil	291
14.3.3 归约 Reduction	294
14.3.4 扫描 Scan	295
14.3.5 打包和拆包	298
14.4 总结	301
14.4.1 了解更多信息	301
15 GPU 编程	302
15.1 性能注意事项	302
15.2 GPU 的工作原理	302
15.2.1 GPU 构建模块	303
15.2.2 更简单的处理器（但数量更多）	304
15.2.3 简化的控制逻辑（SIMD 指令）	308
15.2.4 切换工作以隐藏延迟	312

15.3 将 Kernel 卸载到 GPU	314
15.3.1 SYCL 运行时库	314
15.3.2 GPU 软件驱动程序	314
15.3.3 GPU 硬件	315
15.3.4 当心卸载成本!	316
15.4 GPU Kernel 最佳实践	317
15.4.1 访问全局内存	317
15.4.2 访问 Work-Groups 本地内存	320
15.4.3 通过 Sub-Groups 完全避免本地内存	322
15.4.4 使用小数据类型优化计算	322
15.4.5 优化数学函数	323
15.4.6 特化功能和扩展	323
15.5 总结	324
15.5.1 了解更多信息	324
16 CPU 编程	326
16.1 性能注意事项	326
16.2 多核 CPU 的基础知识	327
16.3 SIMD 硬件基础知识	329
16.4 利用线程级并行性	333
16.4.1 线程亲和力洞察	336
16.4.2 注意第一次接触内存	338
16.5 CPU 上的 SIMD 矢量化	339
16.5.1 确保 SIMD 执行合法性	340
16.5.2 SIMD 掩蔽和成本	342
16.5.3 避免结构数组以提高 SIMD 效率	343
16.5.4 数据类型对 SIMD 效率的影响	345
16.5.5 使用 single_task 执行 SIMD	346
16.6 总结	348
17 FPGA 编程	349
17.1 性能注意事项	349
17.2 如何看待 FPGA	350
17.2.1 管道并行性	350

目录	10
----	----

17.2.2 Kernel 消耗芯片“区域”	350
17.3 何时使用 FPGA	350
17.3.1 很多很多的工作	350
17.3.2 自定义操作或操作宽度	350
17.3.3 标量数据流	350
17.3.4 低延迟和丰富的连接性	350
17.3.5 定制内存系统	350
17.4 在 FPGA 上运行	350
17.4.1 编译时间	350
17.4.2 FPGA 仿真器	350
17.4.3 FPGA 硬件编译“提前”进行	350
17.5 为 FPGA 编写 Kernel	350
17.5.1 暴露并行性	350
17.5.2 使用 ND 范围保持管道繁忙	350
17.5.3 管道不介意数据依赖性!	350
17.5.4 循环的空间管道实现	350
17.5.5 循环启动间隔	350
17.5.6 管道	350
17.5.7 定制内存系统	350
17.6 一些结束语	350
17.6.1 FPGA 构建模块	350
17.6.2 时钟频率	350
17.7 概括	350
18 标准库	352
18.1 内置函数	352
18.1.1 使用带有内置函数的 <code>sycl::</code> 前缀	355
18.2 C++ 标准库	355
18.3 oneAPI DPC++ 库 (oneDPL)	358
18.3.1 SYCL 执行策略	359
18.3.2 将 oneDPL 与 Buffer 结合使用	360
18.3.3 将 oneDPL 与 USM 结合使用	363
18.3.4 使用 SYCL 执行策略进行错误处理	365
18.4 总结	366

19 内存模型和原子	367
19.1 内存模型中有什么?	368
19.1.1 数据竞争和同步	368
19.1.2 Barrier 和栅栏	372
19.1.3 原子操作	374
19.1.4 内存序	375
19.2 内存模型	377
19.2.1 memory_order 枚举类	378
19.2.2 memory_scope 枚举类	379
19.2.3 查询设备能力	380
19.2.4 Barrier 和栅栏	381
19.2.5 SYCL 中的原子操作	382
19.2.6 将原子与 Buffer 一起使用	386
19.2.7 将原子与统一共享内存结合使用	388
19.3 在现实生活中使用原子	388
19.3.1 计算直方图	389
19.3.2 实现设备范围的同步	390
19.4 概括	393
19.4.1 了解更多信息	393
20 后端互操作性	394
20.1 什么是后端互操作性?	394
20.2 后端互操作性何时有用?	395
20.2.1 将 SYCL 添加到现有代码库	396
20.2.2 将现有库与 SYCL 结合使用	398
20.3 使用 Kernel 的后端互操作性	401
20.3.1 与 API 定义的 Kernel 对象的互操作性	402
20.3.2 与非 SYCL 源语言的互操作性	404
20.4 后端互操作性提示和技巧	406
20.4.1 为特定后端选择设备	406
20.4.2 小心上下文!	408
20.4.3 访问低级 API 特性	408
20.4.4 对其他后端的支持	408
20.5 总结	408

目录	12
----	----

21 迁移 CUDA 代码	410
21.1 CUDA 和 SYCL 之间的设计差异	410
21.1.1 多个目标与单个设备目标	410
21.1.2 对齐 C++ 与扩展 C++	412
21.2 CUDA 和 SYCL 之间的术语差异	412
21.3 共同点和不同点	413
21.3.1 执行模型	413
21.3.2 内存模型	418
21.3.3 其他差异	420
21.4 CUDA 中的功能尚未在 SYCL 中提供!	421
21.4.1 全局变量	422
21.4.2 协作组	422
21.4.3 矩阵乘法硬件	423
21.5 移植工具和技术	423
21.5.1 使用 dpct 和 SYCLomatic 迁移代码	423
21.5.2 运行 dpct	424
21.5.3 检查 dpct 输出	426
21.6 总结	427
21.7 了解更多信息	427
22 SYCL 未来发展方向	428
22.1 与 C++11、C++14 和 C++17 更紧密地结合	428
22.2 采用 C++20、C++23 及其他版本的功能	429
22.3 混合 SPMD 和 SIMD 编程	430
22.4 地址空间	431
22.5 特化机制	432
22.6 编译时属性	433
22.7 总结	433
22.8 更多信息	433

序言

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

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

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

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

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

第二版

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

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

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

第 20 章和第 21 章是受本书第一版读者鼓励而添加的内容。

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

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

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

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

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

本书的结构

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

第 1-4 章：奠定基础

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

第一章通过涵盖新的或值得我们刷新的核心概念奠定了第一个基础。

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

第 5-12 章：构建基础

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

第 13-21 章：SYCL 实践提示/建议

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

结语：对未来的推测

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

我们祝您在学习通过 SYCL 使用 C++ 时一切顺利。

前言

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

计算机硬件开发是由我们解决更大、更复杂问题的需求驱动的，但这些硬件进步在很大程度上是无用的，除非像你我这样的程序员拥有允许我们实现我们的想法并以合理的努力利用可用能力的语言。有许多令人惊叹的硬件示例，使用它们的第一个解决方案通常是专有的，因为它可以节省时间，而不必为委员会就标准达成一致而烦恼。然而，在计算史上，它们最终总是以供应商锁定告终——无法与允许开发人员针对任何硬件和共享代码的开放标准竞争——因为最终全球社区和生态系统的资源远远大于任何单个供应商，更不用说开放软件标准如何推动硬件竞争了。

在过去的几年里，我的团队非常荣幸地通过开发 GROMACS（世界上使用最广泛的科学 HPC 代码之一）为塑造新兴的 SYCL 生态系统做出了贡献。我们需要我们的代码在世界上每台超级计算机以及我们的笔记本电脑上运行。虽然我们不能承受性能损失，但我们也依赖于成为更大社区的一部分，其他团队在我们依赖的库上投入精力，那里有可用的开放编译器，以及我们可以招募人才的地方。自本书第一版以来，SYCL 已发展成为这样一个社区；除了几个供应商提供的编译器之外，我们现在还有一个针对所有硬件的主要社区驱动的实现^{0.1}，并且全球有数千名开发人员分享经验、为培训活动做出贡献并参与论坛。开源的杰出力量——无论是应用程序、编译器还是开放标准——是我们可以深入了解、学习、借用和扩展。正如我们反复从 Intel 主导的 LLVM 实现中的代码^{0.2}、海德堡大学社区驱动的实现以及其他几个代码中学习一样，您可以使用我们的公共存储库^{0.3} 来比较大型生产代码库中的 CUDA 和 SYCL 实现，或者借用满足您需求的解决方案 - 因为当您这样做时，您正在帮助进一步扩展我们的社区。

也许令人惊讶的是，数据并行编程作为一种范式可以说比消息传递通信或显式多线程等经典解决方案要容易得多，但它对我们这些在专注于硬件和显式数据放置的旧范式中度过了几十年的人来说带来了特殊的挑战。在小规模上，明确决定如何在少数几个进程之间移动数据对我们来说是微不

^{0.1}Community-driven implementation from Heidelberg University: tinyurl.com/HeidelbergSYCL

^{0.2}DPC++ compiler project: github.com/intel/llvm

^{0.3}GROMACS: gitlab.com/gromacs/gromacs/

足道的，但随着问题扩展到数千个单元，在不引入错误或让硬件闲置等待数据的情况下管理复杂性成为一场噩梦。使用 SYCL 进行数据并行编程解决了这个问题，它主要要求我们显式表达算法的固有并行性，但一旦我们这样做了，编译器和驱动程序将主要处理数以万计的功能单元的数据局部性和调度。为了在数据并行编程中取得成功，重要的是不要将计算机视为执行一个程序的单个单元，而是将其视为独立工作的单元的集合。

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

这在理论上听起来不错，但 SYCL 在实践中的可移植性如何？我们的应用程序是一个代码库的示例，它的优化非常具有挑战性，因为数据访问模式是随机的，每个步骤中要处理的数据量是有限的，我们需要实现每秒数千次迭代，并且我们都受到内存的限制带宽、浮点和整数运算——它与简单数据并行问题截然相反。我们花了二十多年的时间为多种 GPU 架构编写汇编 SIMD 指令和本机实现，我们第一次接触 SYCL 时遇到了适应差异和向驱动程序和编译器开发人员报告性能回归的痛苦。然而，截至 2023 年春季，我们的 SYCL Kernel 不仅可以通过单个代码库，甚至可以通过单个预编译的二进制文件在所有 GPU 架构上实现 80-100% 的本机性能。

SYCL 还很年轻，并且拥有快速发展的生态系统。虽然还有一些东西尚未成为该语言的一部分，但 SYCL 是独一无二的，它是唯一可成功针对所有现代硬件的性能可移植标准。无论您是想要学习并行编程的初学者、对数据并行编程感兴趣的开发人员，还是需要将 100,000 行专有 API

代码移植到开放标准的维护者，这第二版都是您需要成为的唯一一本书这个社区的一部分。

致谢

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

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

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

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

第一版通过 GitHub 收到了反馈^{0.4}，我们确实对其进行了审核，但我们并不总是及时予以确认（想象一下六位合著者都在想“你这样做了，对吗？”）。我们确实从这些反馈中受益匪浅，并且我们相信我们已经解决了本版本示例和文本中的所有反馈。Jay Norwood 是在评论和帮助我们方面最多产的人——所有作者都非常感谢 Jay！其他反馈贡献者包括 Oscar Barenys、Marcel Breyer、Jeff Donner、Laurence Field、Michael Firth、Piotr Fusik、Vincent Mierlak 和 Jason Mooneyham。无论我们是否记得您的名字，我们都感谢所有提供反馈并帮助我们通过 SYCL 完善 C++ 教学的人。

对于这一版本，一些志愿者不知疲倦地阅读了手稿并提供了富有洞察力的反馈，对此我们深表感谢。这些审稿人包括 Aharon Abramson, Thomas Applencourt, Rod Burns, Joe Curley, Jessica Davies, Henry Gabb, Zheming Jin, Rakshith Krishnappa, Praveen Kundurthy, Tim Lewis, Eric Lindahl, Gregory Lueck, Tony Mongkolsmai, Ruyman Reyes Castro, Andrew Richards, Sanjiv Shah, Neil Trevett, 和 Georg Viehöver.

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

^{0.4}github.com/apress/data-parallel-CPP

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

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

谢谢大家，我们希望您发现这本书对您的努力非常有价值。

1 介绍

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

被誉为“计算机架构的新黄金时代”^{1.1}，我们面临着计算设备丰富多样性带来的巨大机遇。我们需要不依赖于任何单一供应商或架构的便携式软件开发能力，以便充分发挥加速计算的潜力。

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

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

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

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

1.1 阅读书籍，而不是标准说明书

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

^{1.1}A New Golden Age for Computer Architecture by John L. Hennessy, David A. Patterson; Communications of the ACM, February 2019, Vol. 62 No. 2, Pages 48-60.

没有一本书可以一次性解释所有事情。因此，本章所做的事情是其他章节所不会做的：代码示例包含一些编程结构，这些编程结构在后面的章节中才会得到解释。我们不应该沉迷于完全理解第一章中的编码示例，并相信每一章都会变得更好。

1.2 SYCL 2020 和 DPC++

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

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

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

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

1.3 为什么不使用 CUDA？

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

传奇人物 John L. Hennessy 和 David A. Patterson 所著的《计算机架构的新黄金时代》。我们认为这是一篇必读的文章。

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

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

1.4 为什么使用带有 SYCL 的标准 C++?

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

1.5 获取支持 SYCL 的 C++ 编译器

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

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

DPC++ 编译器有多个发行版，增加了额外的工具和库，可作为大型项目的一部分提供，为异构系统提供广泛的支持，其中包括库、调试器和其他工具，称为 oneAPI 项目。oneAPI 工具（包括 DPC++ 编译器）可免费获取 (www.oneapi.io/implements)。

1.6 Hello, world! 和 SYCL 程序剖析

```
1. #include <iostream>
2. #include <sycl/sycl.hpp>
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.
9. const auto sz = secret.size();
10.
11. int main() {
12.     queue q;
13.
14.     char* result = malloc_shared<char>(sz, q);
15.     std::memcpy(result, secret.data(), sz);
16.
17.     q.parallel_for(sz, [=](auto& i) {
18.         result[i] -= 1;
19.     }).wait();
20.
21.     std::cout << result << "\n";
22.     free(result, q);
23.     return 0;
24. }
```

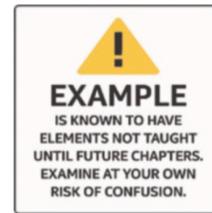


图 1.1: 你好数据并行编程。

图 1-1 显示了 SYCL 程序示例。编译并运行它会打印以下内容：

Hello, world! (以及一些通过运行它来体验的附加文本)

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

- 第 3 行让我们避免一遍又一遍地编写 `sycl::`。
- 第 12 行实例化一个针对特定设备的工作请求队列 (第 2 章)。
- 第 14 行为与设备共享的数据创建分配 (第 3 章)。
- 第 15 行将秘密字符串复制到设备内存中，Kernel 将在其中对其进行处理。

- 第 17 行将工作排队到设备（第 4 章）。
- 第 18 行是唯一将在设备上运行的代码行。所有其他代码都在主机（CPU）上运行。

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

1.7 队列和操作

第 2 章讨论队列和操作，但现在我们可以从简单的解释开始。队列是允许应用程序直接在设备上完成工作的唯一连接。可以将两种类型的操作放入队列中：(a) 要执行的代码和 (b) 内存操作。要执行的代码通过 single_task 或 parallel_for 表示（如图 1-1 中使用）。内存操作执行主机和设备之间的复制操作或填充操作以初始化内存。仅当我们寻求比自动为我们完成的控制更多的控制时，我们才需要使用内存操作。这些都将在本书后面从第 2 章开始讨论。现在，我们应该意识到队列是允许我们命令设备的连接，并且我们有一组可用的操作来放入队列以执行代码和移动数据。了解请求的操作无需等待即可放入队列也非常重要。主机在将操作提交到队列中后，继续执行程序，而设备最终将异步执行通过队列请求的操作。

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

操作异步发生。

1.8 一切都与并行性有关

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

1.8.1 吞吐量

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

1.8.2 延迟

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

1.8.3 并行思维

成功的并行程序员在编程中使用这两种技术。这是我们寻求并行思考的开始。

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

1.8.4 阿姆达尔和古斯塔夫森

阿姆达尔定律由超级计算机先驱 Gene Amdahl 在 1967 年提出，是一个预测使用多个处理器时理论上最大加速的公式。Amdahl 感叹并行性的最大增益仅限于 $(1/(1-p))$ ，其中 p 是并行运行的程序的比例。如果我们只并行运行三分之二的程序，那么该程序最多可以加速 3 倍。我们绝对需要深入理解这个概念！发生这种情况是因为无论我们使三分之二的程序运行得有

多快，另外三分之一仍然需要相同的时间才能完成。即使我们添加 100 个 GPU，性能也只能提高 3 倍。

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

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

1.8.5 规模效应

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

1.8.6 异构系统

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

2 章开始讨论我们的主应用程序（主机代码）如何将工作（计算）引导到异构系统中的特定设备。

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

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

虽然异构从技术角度描述了系统，但使我们的硬件和软件复杂化的原因是为了获得更高的性能。因此，加速计算一词在异构系统或其组件的营销中很流行。我们想强调的是，不能保证加速。只有当我们做得正确时，异构系统的编程才会加速我们的应用程序。这本书可以帮助我们教会我们如何正确地做事！

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

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

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

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

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

1.8.7 数据并行编程

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

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

1.9 带有 SYCL 的 C++ 的关键属性

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

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

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

1.9.1 单源

程序是单源的，这意味着同一个翻译单元^{1.2} 既包含定义要在设备上执行的计算 Kernel 的代码，也包含协调这些计算 Kernel 的执行的主机代码。第 2 章首先更详细地介绍此功能。如果我们愿意，我们仍然可以将程序源分为不同的文件和主机和设备代码的翻译单元，但关键是我们不必这样做！

^{1.2}我们可以只说“文件”，但这在这里并不完全正确。翻译单元是编译器的实际输入，由源文件经过 C 预处理器处理为内联头文件和扩展宏后生成。

1.9.2 主机

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

1.9.3 设备

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

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

1.9.4 Kernel 代码

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

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

一切仍然有效，并且我们不需要进行不同的编程。

1.9.5 异步执行

设备的代码被指定为 Kernel。这个概念并不是带有 SYCL 的 C++ 独有的：它是其他卸载加速语言（包括 OpenCL 和 CUDA）的核心概念。虽然它与面向循环的方法（例如通常与 OpenMP 目标卸载一起使用）不同，但

它可能类似于最内层循环中的代码主体，而不需要程序员显式编写循环嵌套。

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

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

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

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

Kernel: 矢量加法 (DAXPY)

```

1. ! Fortran loop
2. do i = 1, n
3.   z(i) = alpha * x(i) + y(i)
4. end do

1. // C/C++ loop
2. for (int i=0;i<n;i++) {
3.   z[i] = alpha * x[i] + y[i];
4. }

1. // SYCL kernel
2. q.parallel_for(range{n}, [=](id<1> i) {
3.   z[i] = alpha * x[i] + y[i];
4. }).wait();

```

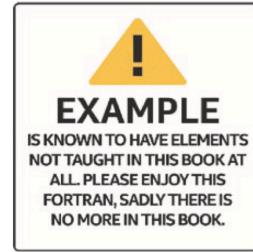


图 1.2: Fortran、C/C++ 和 SYCL 中的 DAXPY 计算。

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

异步执行

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

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

第 3 章开始讨论将我们的程序视为异步任务图，第 8 章极大地扩展了

这个概念。

1.9.6 当我们犯错误时的竞争条件

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



图 1.3: 添加竞争条件来说明异步的要点。

在我们的第一个代码示例（图 1-1）中，我们专门在第 19 行执行了“等待”，以防止第 21 行在结果可用之前写出结果中的值。我们必须牢记这种异步行为。在同一代码示例中还做了另一件微妙的事情 - 第 15 行使用 std::memcpy 来加载输入。由于 std::memcpy 在主机上运行，因此第 17 行及后续行在第 15 行完成之前不会执行。读完第 3 章后，我们可能会想将其更改为使用 q.memcpy（使用 SYCL）。我们已经在图 1-3 的第 7 行中做到了这一点。由于这是一个队列提交，因此无法保证它将在第 9 行之前执行。这会产生竞争条件，这是一种并行编程错误。当程序的两个部分在没有协调的情况下访问相同的数据时，就会出现竞争条件。由于我们希望使用第 7 行写入数据，然后在第 9 行中读取数据，因此我们不希望在第 7 行完成之前执行第 9 行！这样的竞争条件将使我们的程序变得不可预测——我们的程序可能会在不同的运行和不同的系统上得到不同的结果。解决此问题的方法是在第 7 行末尾添加.wait() 来显式等待 q.memcpy 完成，然后再继续。这不是最佳解决方案。我们可以使用事件依赖来解决这个问题（第 8 章）。将队列创建为有序队列还会在 memcpy 和 parallel_for 之间添加隐式依赖关系。作为替代方案，在第 7 章中，我们将看到如何使用缓冲区和访问器编程风格来让 SYCL 管理依赖性并自动等待我们。

注 8 (竞争条件并不总是导致程序失败) 一位精明的读者注意到，图 1-3 中的代码在他们尝试过的每个系统上都没有失败。使用带有 `partition_max_sub_devices==0`

的 *Gpu* 并没有失败，因为它是一个小型 *Gpu*，在 *memcpy* 完成之前无法运行 *parallel_for*。不管怎样，代码是有缺陷的，因为竞争条件存在，即使它不会普遍导致运行时失败。我们称之为一场竞赛——有时我们赢，有时我们输。此类编码缺陷可能会一直处于休眠状态，直到编译和运行时环境的正确组合导致可观察到的故障为止。

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

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

为了帮助检测程序（包括 *Kernel*）中的数据竞争条件，Intel Inspector（可与前面“获取 DPC++ 编译器”中提到的 oneAPI 工具一起使用）等工具可能会有所帮助。此类工具使用的复杂方法通常不适用于所有设备。检测竞争条件的最佳方法可能是让所有 *Kernel* 在 CPU 上运行，这可以在开发工作期间作为调试技术来完成。这个调试技巧在第 2 章中作为 Method#2 进行了讨论。

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

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

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

1.9.7 死锁

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

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

死锁可能很难调试。我们将在本章末尾的“并发与并行”部分重新讨论这一点。

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

1.9.8 C++ Lambda 表达式

现代 C++ 的一个被并行编程技术大量使用的功能是 *lambda 表达式*。Kernel（在设备上运行的代码）可以用多种方式表达，最常见的一种是 *lambda 表达式*。第 10 章讨论了 Kernel 可以采用的所有各种形式，包括 *lambda 表达式*。在这里，我们回顾了 C++ *lambda 表达式*以及有关用于定义 Kernel 的一些注释。在我们在中间的章节中了解了有关 *SYCL* 的更多信息之后，第 10 章将扩展 Kernel 方面的内容。

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

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

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

其中

- *capture-list* 是一个以逗号分隔的捕获列表。我们通过在捕获列表中列出变量名称来按值捕获变量。我们通过在变量前面加上 & 符号来通过

引用捕获变量，例如 `&v`。还有一些适用于所有作用域内自动变量的简写：`[=]` 用于捕获在正文中按值使用的所有自动变量和按引用捕获当前对象，`[&]` 用于捕获在正文中使用的所有自动变量 `body` 以及当前对象的引用，并且 `[]` 不捕获任何内容。对于 SYCL，始终使用 `[=]`，因为不允许通过引用捕获变量以在 Kernel 中使用。根据 C++ 标准，全局变量不会在 lambda 中捕获。非全局静态变量可以在 Kernel 中使用，但前提是它们是 `const`。这里提到的一些限制允许 Kernel 在不同的设备架构和实现中保持一致的行为。

- `params` 是函数参数的列表，就像命名函数一样。SYCL 提供参数来标识正在调用 Kernel 来处理的元素：这可以是唯一的 id（一维）或 2D 或 3D id。这些将在第 4 章中讨论。
- `ret` 是返回类型。如果未指定 `->ret`，则从 `return` 语句推断。缺少 `return` 语句或返回没有值，意味着返回类型为 `void`。SYCL Kernel 必须始终具有 `void` 的返回类型，因此我们不应该使用此语法来指定 Kernel 的返回类型。
- `body` 是函数体。对于 SYCL Kernel，该 Kernel 的内容有一些限制（请参阅本章前面的“Kernel 代码”部分）。

```
int i = 1, j = 10, k = 100, l = 1000;

auto lambda = [i, &j](int k0, int& l0) -> int {
    j = 2 * j;
    k0 = 2 * k0;
    l0 = 2 * l0;
    return i + j + k0 + l0;
};

print_values(i, j, k, l);
std::cout << "First call returned " << lambda(k, l)
    << "\n";
print_values(i, j, k, l);
std::cout << "Second call returned " << lambda(k, l)
    << "\n";
print_values(i, j, k, l);
```

图 1.4: C++ 代码中的 Lambda 表达式。

```
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 1.5: 图 1-4 中 *lambda* 表达式演示代码的输出。

图 1-4 显示了一个 C++ lambda 表达式，它通过值捕获一个变量 i，通过引用捕获另一个变量 j。它还具有一个参数 k0 和另一个通过引用接收的参数 l0。运行该示例将产生如图 1-5 所示的输出。

```
class Functor {
public:
    Functor(int i, int &j) : my_i{i}, my_jRef{j} {}

    int operator()(int k0, int &l0) {
        my_jRef = 2 * my_jRef;
        k0 = 2 * k0;
        l0 = 2 * l0;
        return my_i + my_jRef + k0 + l0;
    }

private:
    int my_i;
    int &my_jRef;
};
```

图 1.6: 函数对象而不是 *lambda* 表达式（第 10 章中有更多介绍）。

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

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

1.9.9 功能可移植性和性能可移植性

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

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

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

虽然我们作为程序员自己负责设计有效的性能可移植性计划，但 SYCL

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

1.10 并发与并行

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

请考虑《Sun Microsystems 多线程编程指南》中的这些定义：^{1.3}

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

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

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

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

但有时，我们确实关心。我们思维中的错误可能会产生灾难性的影响（例如“死锁”）。想象一下，我们对上一段的示例进行了修改，使得任务（X、

^{1.3}The authors are fans of this programming guide's coverage of the fundamentals that never go away. It is online at docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf.

Y 或 Z) 执行的最后一件事是“等待所有任务完成”。如果任务数量永远不会超过硬件的限制，我们的程序就会运行得很好。但是，如果我们将任务分成批次，那么第一批中的任务将永远等待。不幸的是，这意味着我们的应用程序永远不会完成。

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

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

1.11 总结

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

2 代码执行位置

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

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

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

2.1 单源

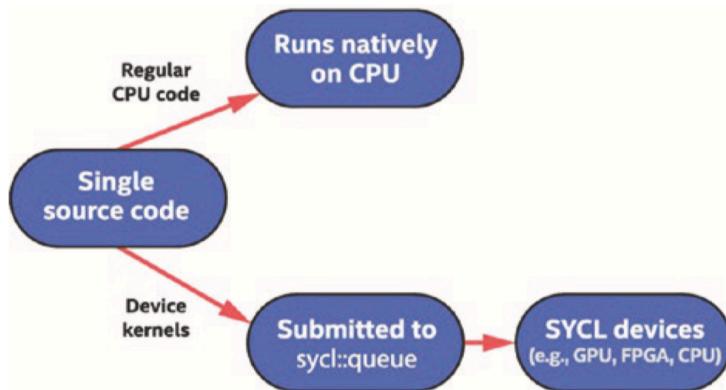


图 2.1：单源代码包含主机代码（在 CPU 上运行）和设备代码（在 SYCL 设备上运行）

```

#include <array>
#include <iostream>
#include <sycl/sycl.hpp>
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;
}

```

图 2.2: 简单的 SYCL 程序

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

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

2.1.1 主机代码

应用程序包含 C++ 主机代码，由动作系统在其上启动应用程序的 CPU 执行。主机代码是应用程序的主干，它定义和控制可用设备的工作分配。它

也是我们定义应由 SYCL 运行时管理的数据和依赖项的接口。

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

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

2.1.2 设备代码

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

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

1. **它从主机代码异步执行。** 主机程序向设备提交设备代码，只有当所有执行依赖性都得到满足时，运行时才会跟踪并启动该工作（更多内容将在第 3 章中介绍）。主机程序执行在设备上启动提交的工作之前进行，从而提供了设备上的执行与主机程序执行异步的属性，除非我们明确地将两者绑定在一起。作为这种异步执行的副作用，只有主机程序通过我们在后面的章节中介绍的各种机制（例如主机访问器和阻塞队列等待动作）强制执行开始，才能保证设备上的工作开始。
2. **对设备代码进行限制，使其能够在加速器设备上编译并实现性能。** 例如，设备代码中不支持动态内存分配和运行时类型信息 (RTTI)，因为它们会导致许多加速器的性能下降。第 10 章详细介绍了一小部分设备代码限制。
3. **SYCL 定义的一些函数和查询仅在设备代码中可用，** 因为它们只在那里有意义，例如，工作项标识符查询允许设备代码的执行实例查询其在更大的数据并行范围中的位置（描述第 4 章）。

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

2.2 选择设备

为了探索让我们控制设备代码执行位置的机制，我们将看五个用例：

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

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

方法 #3：将设备代码分派到 GPU 或其他加速器。

方法 #4：将设备代码分派到一组异构设备，例如 GPU 和 FPGA。

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

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

2.3 方法 #1：在任何类型的设备上运行

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

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

2.3.1 队列

```
class queue {
public:
    // Create a queue associated with a default
    // (implementation chosen) device.
    queue(const property_list & = {});

    queue(const async_handler &, const property_list & = {});

    // Create a queue using a DeviceSelector.
    // A DeviceSelector is a callable that ranks
    // devices numerically. There are a few SYCL-defined
    // device selectors available such as
    // cpu_selector_v and gpu_selector_v.
    template <typename DeviceSelector>
    explicit queue(const DeviceSelector &deviceSelector,
                   const property_list &propList = {});

    // Create a queue associated with an explicit device to
    // which the program already holds a reference.
    queue(const device &, const property_list & = {});

    // Create a queue associated with a device in a specific
    // SYCL context. A device selector may be used in place
    // of a device.
    queue(const context &, const device &,
          const property_list & = {});
};
```

图 2.3: 队列类的某些构造函数的简化定义

```
class queue {
public:
    // Submit a command group to this queue.
    // The command group may be a lambda expression or
    // function object. Returns an event reflecting the status
    // of the action performed in the command group.
    template <typename T>
    event submit(T);

    // Wait for all previously submitted actions to finish
    // executing.
    void wait();

    // Wait for all previously submitted actions to finish
    // executing. Pass asynchronous exceptions to an
    // async_handler function.
    void wait_and_throw();
};
```

图 2.4: 简化了队列类中一些关键成员函数的定义

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

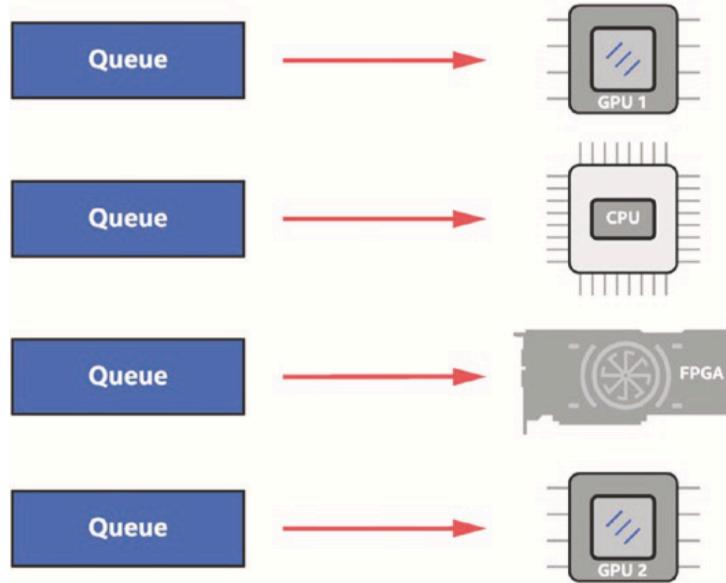


图 2.5: 队列绑定到单个设备。提交到队列的工作在该设备上执行

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

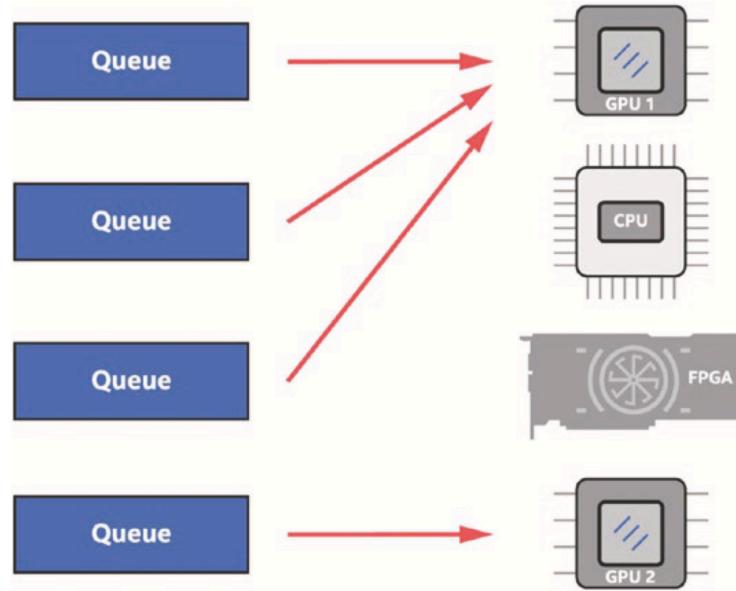


图 2.6: 多个队列可以绑定到单个设备

可以按照我们希望的应用程序架构或编程风格的任何方式在程序中创建多个队列。例如，可以创建多个队列以分别与不同的设备绑定或由主机程序中的不同线程使用。多个不同的队列可以绑定到单个设备（例如 GPU），并且向这些不同队列的提交将导致在设备上执行组合工作。图 2-6 显示了一个示例。相反，正如我们之前提到的，一个队列不能绑定到多个设备，因为请求执行动作的位置不能有任何歧义。例如，如果我们想要一个能够跨多个设备负载平衡工作的队列，那么我们可以在代码中创建该抽象。

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

2.3.2 当任何设备都可以时将队列绑定到设备

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

int main() {
    // Create queue on whatever default device that the
    // implementation chooses. Implicit use of
    // default_selector_v
    queue q;

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

    return 0;
}

Sample Outputs (one Line per run depending on system):
Selected device: NVIDIA GeForce RTX 3060
Selected device: AMD Radeon RX 5700 XT
Selected device: Intel(R) Data Center GPU Max 1100
Selected device: Intel(R) FPGA Emulation Device
Selected device: AMD Ryzen 5 3600 6-Core Processor
Selected device: Intel(R) UHD Graphics 770
Selected device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Selected device: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
many more possible... these are only examples
```

图 2.7: 通过队列的默认构造隐式默认设备选择器

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

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

2.4 方法 #2: 使用 CPU 设备进行开发、调试和部署

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

使用 CPU 设备进行代码开发有几个优点：

1. 在没有任何加速器的功能较差的系统上开发设备代码：一种常见用途是在本地系统上开发和测试设备代码，然后部署到 HPC 集群进行性能测试和优化。
2. 使用非加速器工具调试设备代码：加速器通常通过较低级别的 API 公开，这些 API 可能没有主机 CPU 可用的先进调试工具。考虑到这一点，CPU 设备通常支持使用开发人员熟悉的标准工具进行调试。
3. 如果没有其他设备可用，则进行备份，以保证设备代码可以正常执行：CPU 设备可能不以性能为主要目标，或者可能与 Kernel 代码优化的架构不匹配，但通常可以考虑作为功能备份，以确保设备代码始终可以在任何应用程序中执行。

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

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

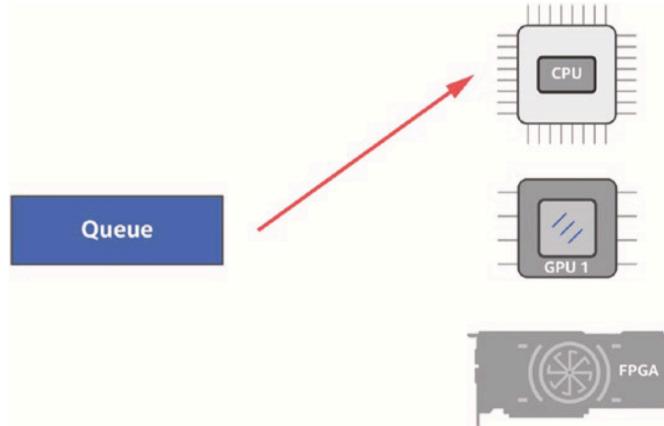


图 2.8: CPU 设备可以像任何加速器一样执行设备代码

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

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

int main() {
    // Create queue to use the CPU device explicitly
    queue q{cpu_selector_v};

    std::cout << "Selected device: "
        << q.get_device().get_info<info::device::name>()
        << "\n";
    std::cout
        << " -> Device vendor: "
        << q.get_device().get_info<info::device::vendor>()
        << "\n";

    return 0;
}

Example Output:
Selected device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
-> Device vendor: Intel(R) Corporation
```

图 2.9: 使用 `cpu_selector_v` 选择主机设备

应用程序可以通过将 `cpu_selector_v` 显式传递给队列构造函数来选择

创建绑定到 CPU 设备的队列，如图 2-9 所示。

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

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

2.5 方法 #3：使用 GPU（或其他加速器）

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

2.5.1 加速器设备

在 SYCL 规范的术语中，有几组广泛的加速器类型：

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

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

2.5.2 设备选择器

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

有四个内置选择器适用于各种常见的设备。

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

<code>default_selector_v</code>	Any device of the implementation's choosing
<code>cpu_selector_v</code>	Select a device that identifies itself as a CPU in device queries
<code>gpu_selector_v</code>	Select a device that identifies itself as a GPU in device queries
<code>accelerator_selector_v</code>	Select a device that identifies itself as an "accelerator," which includes FPGAs

`ext::intel::fpga_selector_v` Select a device that identifies itself as an FPGA

可以使用内置选择器之一构造队列，例如

`queue myQueue{ gpu_selector_v{} };`

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

int main() {
    // Create queue bound to an available GPU device
    queue q{gpu_selector_v};

    std::cout << "Selected device: "
        << q.get_device().get_info<info::device::name>()
        << "\n";
    std::cout
        << " -> Device vendor: "
        << q.get_device().get_info<info::device::vendor>()
        << "\n";

    return 0;
}
```

Example Output:

Selected device: AMD Radeon RX 5700 XT
 -> Device vendor: AMD Corporation

图 2.10: GPU 设备选择器示例

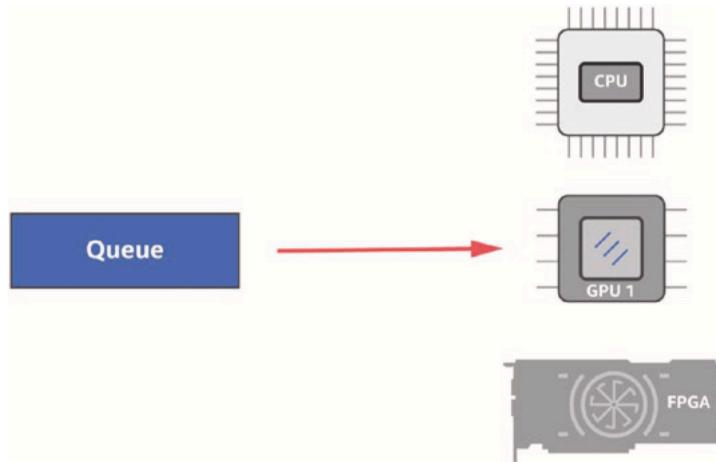


图 2.11: 绑定到应用程序可用的 *GPU* 设备的队列

```

#include <iostream>
#include <string>
#include <sycl/ext/intel/fpga_extensions.hpp> // For fpga_selector_v
#include <sycl/sycl.hpp>
using namespace sycl;

void output_dev_info(const device& dev,
                     const std::string& selector_name) {
    std::cout << selector_name << ": Selected device: "
    << dev.get_info<info::device::name>() << "\n";
    std::cout << "          -> Device vendor: "
    << dev.get_info<info::device::vendor>() << "\n";
}

int main() {
    output_dev_info(device{default_selector_v},
                    "default_selector_v");
    output_dev_info(device{cpu_selector_v}, "cpu_selector_v");
    output_dev_info(device{gpu_selector_v}, "gpu_selector_v");
    output_dev_info(device{accelerator_selector_v},
                    "accelerator_selector_v");
    output_dev_info(device{ext::intel::fpga_selector_v},
                    "fpga_selector_v");

    return 0;
}

Example Output:
default_selector_v: Selected device: Intel(R) UHD Graphics [0x9a60]
                   -> Device vendor: Intel(R) Corporation
cpu_selector_v: Selected device: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
                 -> Device vendor: Intel(R) Corporation
gpu_selector_v: Selected device: Intel(R) UHD Graphics [0x9a60]
                 -> Device vendor: Intel(R) Corporation
accelerator_selector_v: Selected device: Intel(R) FPGA Emulation Device
                        -> Device vendor: Intel(R) Corporation
fpga_selector_v: Selected device: pac_a10 : Intel PAC Platform (pac_ee00000)
                  -> Device vendor: Intel Corp

```

图 2.12: 来自各种类别的设备选择器的示例设备标识输出，并演示设备选择器不仅可用于构建队列（在本例中为构造设备类实例）

图 2-10 显示了使用 GPU 选择器的完整示例，

图 2-11 显示了队列与可用 GPU 设备的相应绑定。

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

当设备选择失败时

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

2.6 方法 #4: 使用多个设备

```
#include <iostream>
#include <sycl/ext/intel/fpga_extensions.hpp> // For fpga_selector_v
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
    queue my_gpu_queue(gpu_selector_v);
    queue my_fpga_queue(ext::intel::fpga_selector_v);

    std::cout << "Selected device 1: "
        << my_gpu_queue.get_device()
            .get_info<info::device::name>()
        << "\n";

    std::cout << "Selected device 2: "
        << my_fpga_queue.get_device()
            .get_info<info::device::name>()
        << "\n";

    return 0;
}

Example Output:
Selected device 1: Intel(R) UHD Graphics [0x9a60]
Selected device 2: pac_a10 : Intel PAC Platform (pac_ee00000)
```

图 2.13: 为 *GPU* 和 *FPGA* 设备创建队列

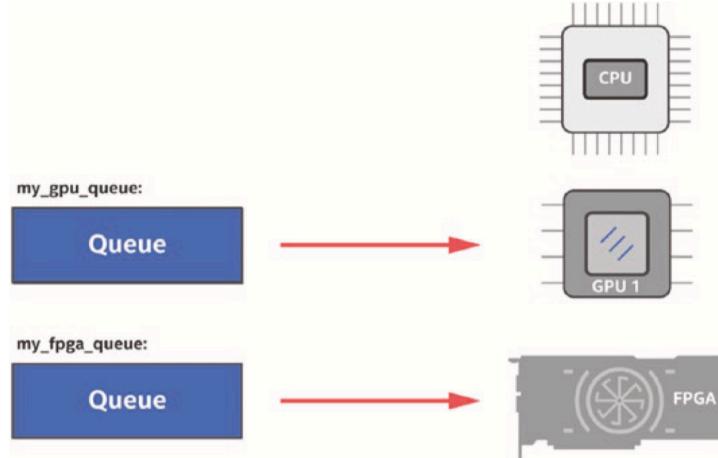


图 2.14: *GPU + FPGA* 设备选择器示例：一个队列绑定到 *GPU*, 另一个队列绑定到 *FPGA*

如图 2-5 和 2-6 所示，我们可以在一个应用程序中构造多个队列。我们可以将这些队列绑定到单个设备（队列的工作总和集中到单个设备）、多个设备或这些设备的某种组合。图 2-13 提供了一个示例，创建一个绑定到 GPU 的队列和另一个绑定到 FPGA 的队列。相应的映射如图 2-14 所示。

2.7 方法 #5：自定义（非常具体）的设备选择

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

2.7.1 根据设备方面 (Aspect) 进行选择

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

```

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

int main() {
    // In the aspect_selector form taking a comma seperated
    // group of aspects, all aspects must be present for a
    // device to be selected.
    queue q1{aspect_selector(aspect::fp16, aspect::gpu)};

    // In the aspect_selector form that takes two vectors, the
    // first vector contains aspects that a device must
    // exhibit, and the second contains aspects that must NOT
    // be exhibited.
    queue q2{aspect_selector(
        std::vector{aspect::fp64, aspect::fp16},
        std::vector{aspect::gpu, aspect::accelerator})};

    std::cout
        << "First selected device is: "
        << q1.get_device().get_info<info::device::name>()
        << "\n";

    std::cout
        << "Second selected device is: "
        << q2.get_device().get_info<info::device::name>()
        << "\n";

    return 0;
}

Example Output:
First selected device is: Intel(R) UHD Graphics [0x9a60]
Second selected device is: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz

```

图 2.15: Aspect selector

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

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

2.7.2 通过自定义选择器进行选择

当现有方面不足以选择特定设备时，可以定义自定义设备选择器。这样的选择器只是一个 C++ 可调用的（例如，函数或 lambda），它接受 const

Device& 作为参数，并返回特定设备的整数分数。SYCL 运行时在可以找到的所有可用根设备上调用选择器，并选择选择器返回最高分数的设备（该分数必须为非负数才能进行选择）。

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

设备评分机制

我们有很多选项来创建与特定设备相对应的整数分数，例如：

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

```
int my_selector(const device &dev) {
    if (dev.get_info<info::device::name>().find("pac_a10") !=
        std::string::npos &&
        dev.get_info<info::device::vendor>().find("Intel") !=  
        std::string::npos) {
        return 1;
    }
    return -1;
}
```

Example Output:

```
Selected device is: pac_a10 : Intel PAC Platform (pac_ee00000)
```

图 2.16: 特定英特尔 Arria FPGA 加速板的自定义选择器

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

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

2.8 在设备上创建任务

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

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

2.8.1 任务图简介

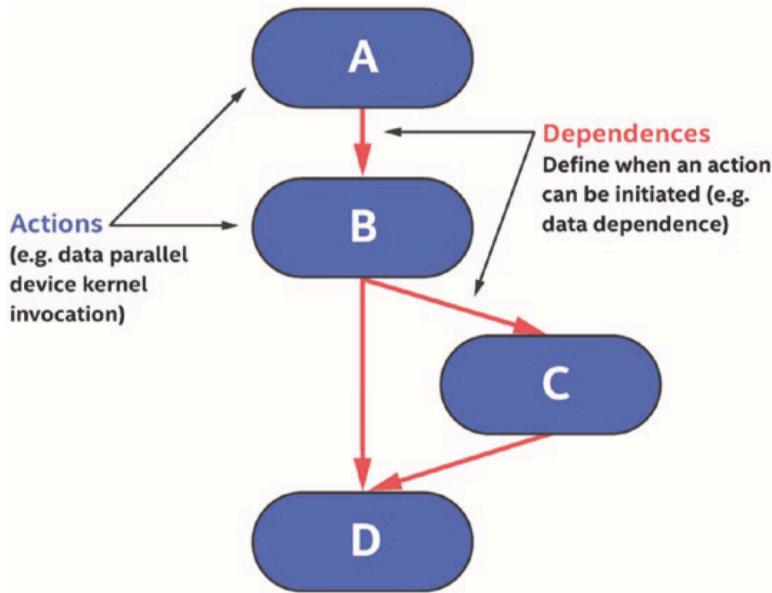


图 2.17: 任务图定义要在一个或多个设备上执行的操作（从主机程序异步执行），以及确定操作何时可以安全执行的依赖关系

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

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

中进行更详细的介绍。

2.8.2 设备代码在哪里?

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

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



图 2.18: 提交设备代码

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

```
class handler {
public:
    // Specify event(s) that must be complete before the action
    // defined in this command group executes.
    void depends_on(std::vector<event> & events);

    // Guarantee that the memory object accessed by the accessor
    // is updated on the host after this action executes.
    template <typename AccessorT>
    void update_host(AccessorT acc);

    // Submit a memset operation writing
    // to the specified pointer.
    // Return an event representing this operation.
    event memset(void *ptr, int value, size_t count);

    // Submit a memcpy operation copying from src to dest.
    // Return an event representing this operation.
    event memcpy(void *dest, const void *src, size_t count);

    // Copy to/from an accessor and host memory.
    // Accessors are required to have appropriate correct
    // permissions. Pointer can be a raw pointer or
    // shared_ptr.
    template <typename SrcAccessorT, typename DestPointerT>
    void copy(SrcAccessorT src, DestPointerT dest);

    template <typename SrcPointerT, typename DestAccessorT>
    void copy(SrcPointerT src, DestAccessorT dest);

    // Copy between accessors.
    // Accessors are required to have appropriate correct
    // permissions.
    template <typename SrcAccessorT, typename DestAccessorT>
    void copy(SrcAccessorT src, DestAccessorT dest);

    // Submit different forms of kernel for execution.
    template <typename KernelName, typename KernelType>
    void single_task(KernelType kernel);

    template <typename KernelName, typename KernelType,
             int Dims>
    void parallel_for(range<Dims> num_work_items,
                     KernelType kernel);

    template <typename KernelName, typename KernelType, int Dims>
    void parallel_for(nd_range<Dims> execution_range,
                     KernelType kernel);

    template <typename KernelName, typename KernelType, int Dims>
    void parallel_for_work_group(range<Dims> num_groups,
                                 KernelType kernel);

    template <typename KernelName, typename KernelType, int Dims>
    void parallel_for_work_group(range<Dims> num_groups,
                                 range<Dims> group_size,
                                 KernelType kernel);
};

};
```

图 2.19: 处理程序类中成员函数的简化定义

```

class queue {
public:
    // Submit a memset operation writing to the specified
    // pointer. Return an event representing this operation.
    event memset(void* ptr, int value, size_t count);

    // Submit a memcpy operation copying from src to dest.
    // Return an event representing this operation.
    event memcpy(void* dest, const void* src, size_t count);

    // Submit different forms of kernel for execution.
    // Return an event representing the kernel operation.
    template <typename KernelName, typename KernelType>
    event single_task(KernelType kernel);

    template <typename KernelName, typename KernelType,
              int Dims>
    event parallel_for(range<Dims> num_work_items,
                      KernelType kernel);

    template <typename KernelName, typename KernelType,
              int Dims>
    event parallel_for(nd_range<Dims> execution_range,
                      KernelType kernel);

    // Submit different forms of kernel for execution.
    // Wait for the specified event(s) to complete
    // before executing the kernel.
    // Return an event representing the kernel operation.
    template <typename KernelName, typename KernelType>
    event single_task(const std::vector<event>& events,
                      KernelType kernel);

    template <typename KernelName, typename KernelType,
              int Dims>
    event parallel_for(range<Dims> num_work_items,
                      const std::vector<event>& events,
                      KernelType kernel);

    template <typename KernelName, typename KernelType,
              int Dims>
    event parallel_for(nd_range<Dims> execution_range,
                      const std::vector<event>& events,
                      KernelType kernel);
};


```

图 2.20: 简化了队列类中成员函数的定义，这些函数充当处理程序类中等效函数的简写表示法

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

2.8.3 行动

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

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

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

Work Type	Actions (handler class methods)	Summary
Device code execution	single_task	Execute a single instance of a device function.
	parallel_for	Multiple forms are available to launch device code with different combinations of work sizes.
Explicit memory operation	copy	Copy data between locations specified by accessor, pointer, and/or shared_ptr. The copy occurs as part of the SYCL task graph (described later), including dependence tracking.
	update_host	Trigger update of host data backing of a buffer object.
	fill	Initialize data in a buffer to a specified value.

图 2.21：触发设备代码，以及显式的内存操作

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

注 13 命令组中最多只能有一个操作，例如 *Kernel* 启动或显式内存操作。

代码在未来异步执行的想法是作为主机程序的一部分在 CPU 上运行的代码与将来在满足依赖性时运行的设备代码之间的关键区别。命令组通常

包含每个类别的代码，其中定义依赖关系的代码作为主机程序的一部分运行（以便运行时知道依赖关系是什么），而设备代码则在满足依赖关系后运行。

```
#include <array>
#include <iostream>
#include <sycl/sycl.hpp>
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;
}
```

The diagram uses curly braces on the right side of the code to group different parts. A blue brace groups the entire main function body, labeled 'Host code'. A green brace groups the output statement and the submission of the parallel_for loop, labeled 'Immediate code to set up task graph node'. A red brace groups the parallel_for loop itself, labeled 'Device code runs in the future when dependences are met'.

图 2.22: 提交设备代码

图 2-22 中有三类代码：

1. 主机代码：驱动应用程序，包括创建和管理数据缓冲区以及将工作提交到队列以在任务图中形成新节点以进行异步执行。
2. 命令组内的主机代码：此代码在执行主机代码的处理器上运行，并在提交调用返回之前立即执行。例如，此代码通过创建访问器来设置节点依赖性。任何任意 CPU 代码都可以在这里执行，但最佳实践是将其限制为配置节点依赖项的代码。
3. 动作：图 2-21 中列出的任何动作都可以包含在命令组中，它定义了将来满足节点要求时异步执行的工作（由 (2) 设置）。

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

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

2.8.4 主机任务

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

主机任务在某些程序中很重要，原因有二：

1. 可以包含任意 C++，甚至 `std::cout` 或 `printf`。这对于轻松调试、与 OpenCL 等较低级别 API 的互动作性或在现有代码中逐步启用加速器非常重要。
2. 主机任务作为任务图的一部分异步执行，而不是与主机程序同步执行。尽管主机程序可以启动附加线程或使用其他任务并行方法，但主机任务与 SYCL 运行时的依赖性跟踪机制集成。当设备和主机代码需要分散时，这非常方便，并且可能会带来更高的性能。

```

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

int main() {
    queue q;
    int* A = malloc_shared<int>(N, q);

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

    // Initialize values in the shared allocation
    auto eA = q.submit([&](handler& h) {
        h.parallel_for(N, [=](auto& idx) { A[idx] = idx; });
    });

    // Use a host task to output values on the host as part of
    // task graph. depends_on is used to define a dependence
    // on previous device code having completed. Here the host
    // task is defined as a lambda expression.
    q.submit([&](handler& h) {
        h.depends_on(eA);
        h.host_task([=]() {
            for (int i = 0; i < N; i++)
                std::cout << "host_task @ " << i << " = " << A[i]
                << "\n";
        });
    });

    // Wait for work to be completed in the queue before
    // accessing the shared data in the host program.
    q.wait();

    for (int i = 0; i < N; i++)
        std::cout << "main @ " << i << " = " << A[i] << "\n";

    free(A, q);
}

return 0;
}

Example Output:
Selected device: NVIDIA GeForce RTX 3060
host_task @ 0 = 0
host_task @ 1 = 1
host_task @ 2 = 2
host_task @ 3 = 3
main @ 0 = 0
main @ 1 = 1
main @ 2 = 2
main @ 3 = 3

```

图 2.23: 简单的 `host_task` 例子

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

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

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

2.9 总结

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

3 数据管理

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

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

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

我们需要控制数据的移动，本章将介绍实现这一目标的选项。

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

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

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

3.1 介绍

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

在本章中，我们概述了数据管理的各种机制。我们介绍了统一共享内存

和数据管理的 Buffer 抽象，并描述了 Kernel 执行和数据移动之间的关系。

3.2 数据管理问题

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

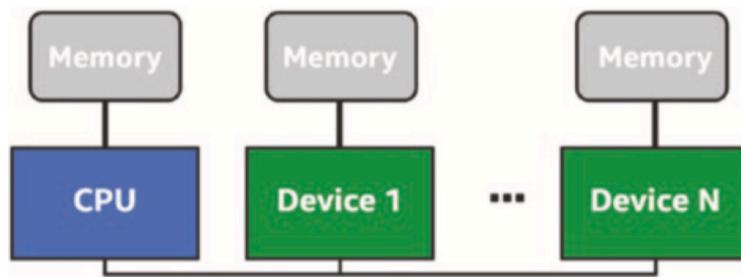


图 3.1: 多个离散存储器

3.3 本地设备与远程设备

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

3.4 管理多个内存

管理多个内存大致可以通过两种方式完成：显式地通过我们的程序或隐式地通过 SYCL 运行时库。每种方法都有其优点和缺点，我们可以根据

情况或个人喜好选择其中一种。

3.4.1 显式数据移动

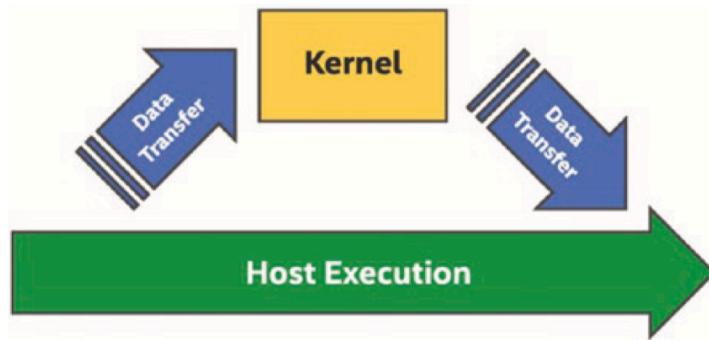


图 3.2: 数据移动和 *Kernel* 执行

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

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

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

3.4.2 隐式数据

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

隐式数据移动的优点是，应用程序无需花费太多精力即可利用直接连接到设备的更快内存。所有繁重的工作都是由运行时自动完成的。这也减少

了在程序中引入错误的机会，因为运行时将自动识别何时必须执行数据传输以及必须传输多少数据。

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

3.4.3 选择正确的策略

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

3.5 USM、Buffer 和 Images

管理内存有三个抽象：统一共享内存 (USM)、Buffer 和 Images。USM 是一种基于指针的方法，C/C++ 程序员应该熟悉。USM 的优点之一是更容易与现有的操作指针的 C++ 代码集成。Buffer（由 Buffer 模板类表示）描述一维、二维或三维数组。它们提供了可以在主机或设备上访问的内存的抽象视图。Buffer 不由程序直接访问，而是通过访问器对象使用。Images 充当一种特殊类型的 Buffer，提供特定于 Images 处理的额外功能。此功能包括对特殊 Images 格式的支持、使用采样器对象读取 Images 等等。Buffer 和 Images 是强大的抽象，可以解决许多问题，但重写现有代码中的所有接口以接受 Buffer 或访问器可能非常耗时。由于 Buffer 和 Images 的接口基本相同，因此本章的其余部分将仅关注 USM 和 Buffer。

3.6 统一共享内存

USM 是我们可用于数据管理的一种工具。USM 是一种基于指针的方法，使用 malloc 或 new 分配数据的 C 和 C++ 程序员应该熟悉它。USM 简化了移植大量使用指针的现有 C/C++ 代码的过程。支持 USM 的设备支持统一的虚拟地址空间。拥有统一的虚拟地址空间意味着主机上的 USM 分

配例程返回的任何指针值都将是设备上的有效指针值。我们不必手动转换主机指针来获取“设备版本”——我们在主机和设备上看到相同的指针值。

USM 的更详细描述可以在第 6 章中找到。

3.6.1 通过指针访问内存

Allocation Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	✗	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	can migrate back and forth

图 3.3: USM allocation types

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

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

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

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

3.6.2 USM 和数据移动

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

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

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    std::array<int, N> host_array;
    int *device_array = malloc_device<int>(N, q);

    for (int i = 0; i < N; i++) host_array[i] = N;

    // We will learn how to simplify this example later
    q.submit([&](handler &h) {
        // copy host_array to device_array
        h.memcpy(device_array, &host_array[0], N * sizeof(int));
    });
    q.wait();

    q.submit([&](handler &h) {
        h.parallel_for(N, [=](id<1> i) { device_array[i]++; });
    });
    q.wait();

    q.submit([&](handler &h) {
        // copy device_array back to host_array
        h.memcpy(&host_array[0], device_array, N * sizeof(int));
    });
    q.wait();

    free(device_array, q);
    return 0;
}
```

图 3.4: USM explicit data movement

图 3-4 包含一个在设备分配上运行的 Kernel。在 Kernel 使用 memcpy()

操作执行之前和之后，数据会在 host_array 和 device_array 之间复制。调用队列上的 wait() 可确保在 Kernel 执行之前完成到设备的复制，并确保在数据复制回主机之前 Kernel 已完成。我们将在本章后面学习如何消除这些调用。

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

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;
    int *host_array = malloc_host<int>(N, q);
    int *shared_array = malloc_shared<int>(N, q);

    for (int i = 0; i < N; i++) {
        // Initialize host_array on host
        host_array[i] = i;
    }

    // We will learn how to simplify this example later
    q.submit([&](handler &h) {
        h.parallel_for(N, [=](id<1> i) {
            // access shared_array and host_array on device
            shared_array[i] = host_array[i] + 1;
        });
    });
    q.wait();

    for (int i = 0; i < N; i++) {
        // access shared_array on host
        host_array[i] = shared_array[i];
    }

    free(shared_array, q);
    free(host_array, q);
    return 0;
}
```

图 3.5: USM implicit data movement

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

3.7 Buffer

为数据管理提供的另一个抽象是 Buffer 对象。Buffer 是一种数据抽象，表示给定 C++ 类型的一个或多个对象。Buffer 对象的元素可以是标量数据类型（例如 int、float 或 double）、向量数据类型（第 11 章）或用户定义的类或结构。SYCL 2020 定义了一个新概念“设备可复制”，它扩展了可简单复制的概念，并添加了允许类型集。特别是，如果常见 C++ 类（例如 std::array、std::pair、std::tuple 或 std::span）中的模板化类型本身是设备可复制的，那么使用这些类型构建的那些 C++ 类特化也是设备可复制的。在将数据类型与 Buffer 一起使用之前，请注意您的数据类型是设备可复制的！

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

Buffer 的更详细描述可以在第 7 章中找到。

3.7.1 创建 Buffer

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

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

3.7.2 访问 Buffer

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

确保安排数据移动。

3.7.3 访问模式

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    std::array<int, N> my_data;
    for (int i = 0; i < N; i++) my_data[i] = 0;

    {
        queue q;
        buffer my_buffer(my_data);

        q.submit([&](handler &h) {
            // create an accessor to update
            // the buffer on the device
            accessor my_accessor(my_buffer, h);

            h.parallel_for(N, [=](id<1> i) { my_accessor[i]++; });
        });

        // create host accessor
        host_accessor host_accessor(my_buffer);

        for (int i = 0; i < N; i++) {
            // access myBuffer on host
            std::cout << host_accessor[i] << " ";
        }
        std::cout << "\n";
    }

    // myData is updated when myBuffer is
    // destroyed upon exiting scope
    for (int i = 0; i < N; i++) {
        std::cout << my_data[i] << " ";
    }
    std::cout << "\n";
}
```

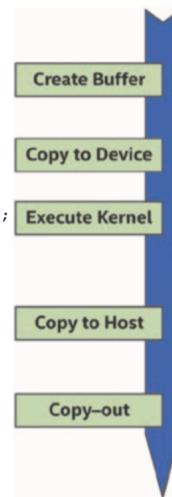


图 3.6: *Buffer and accessors*

Access Mode	Description
read	Read-only access.
write	Write-only access. Previous contents are not discarded in case of partial writes.
read_write	Read and write access.

图 3.7: *Buffer access modes*

创建访问器时，我们可以通知运行时我们将如何使用它来提供更多优化信息。我们通过指定访问模式来做到这一点。访问模式在图 3-7 中描述的 `access_mode` 枚举类中定义。在图 3-6 所示的代码示例中，访问器 `my_accessor` 是使用默认访问模式 `access_mode::read_write` 创建的。这让运行时知道我们打算通过 `my_accessor` 读取和写入 Buffer。访问模式是运行时优化隐式数据移动的方式。例如，`access_mode::read` 告诉运行时，在该 Kernel 开始执行之前，数据需要在设备上可用。如果 Kernel 仅通过访问器读取数据，则无需在 Kernel 完成后将数据复制回主机，因为我们没有修改它。同样，`access_mode::write` 让运行时知道我们将修改 Buffer 的内容，并且可能需要在计算结束后将结果复制回来。

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

3.8 对数据的使用进行排序

Kernel 可以被视为提交执行的异步任务。这些任务必须提交到队列，并安排它们在设备上执行。在许多情况下，Kernel 必须按特定顺序执行，以便计算出正确的结果。如果要获得正确结果需要任务 A 先于任务 B 执行，则称任务 A 和任务 B 之间存在依赖关系。^{3.1}

^{3.1}请注意，您可能会看到“dependence”和“dependences”有时在其他文本中拼写为“dependency”和“dependencies”。它们的意思是一样的，但我们倾向于在几篇关于数据流分析的重要

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

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

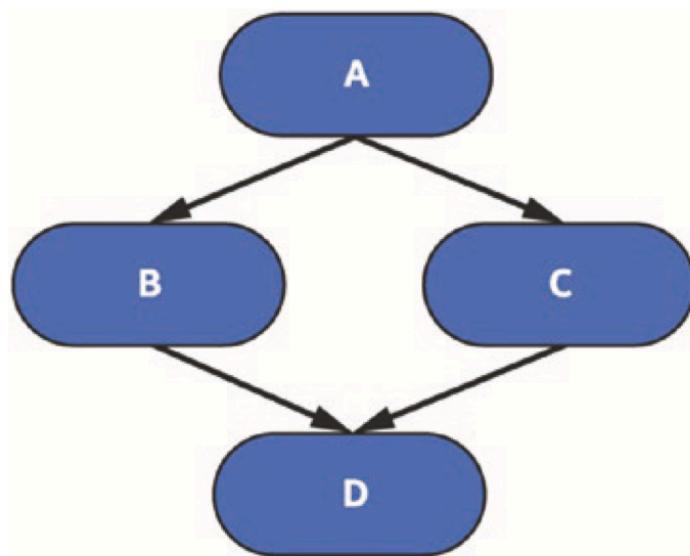


图 3.8: 简单的任务图

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

论文中使用的拼写。请参阅 <https://dl.acm.org/doi/pdf/10.1145/75277.75280> 和 <https://dl.acm.org/doi/pdf/10.1145/113446.113449>。

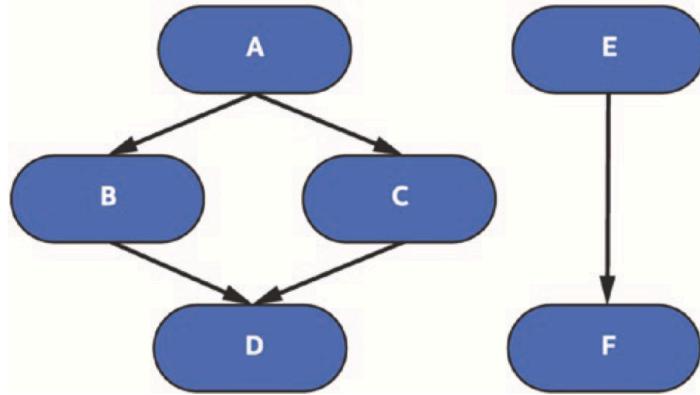


图 3.9: 具有不相交依赖关系的任务图

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

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

3.8.1 有序队列

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

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

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

    return 0;
}
```

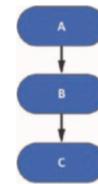


图 3.10: In-order queue usage

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

3.8.2 无序队列

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

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

与事件的显式依赖关系 任务之间的显式依赖关系类似于我们看到的示例（图 3-8），其中任务 A 必须在任务 B 之前执行。以这种方式表达依赖关系侧重于基于发生的计算而不是计算访问的数据的显式排序。请注意，表达计算

之间的依赖关系主要与使用 USM 的代码相关，因为使用 Buffer 的代码通过访问器表达大多数依赖关系。在图 3-4 和 3-5 中，我们只是告诉队列等待所有先前提交的任务完成，然后再继续。相反，我们可以通过事件对象来表达任务依赖性。将命令组提交到队列时，`submit()` 方法返回一个事件对象。这些事件可以通过两种方式使用。

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

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

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

int main() {
    queue q;

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

    return 0;
}
```

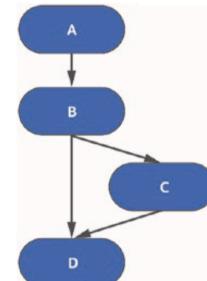


图 3.11: Using events and depends_on

Dependence Type	Description
Read-after-Write (RAW)	Occurs when task B needs to read data computed by task A.
Write-after-Read (WAR)	Occurs when task B writes over data after it has been read by task A.
Write-after-Write(WAW)	Occurs when task B also writes over data written by task A.

图 3.12: *Three forms of data dependences*

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

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    std::array<int, N> a, b, c;
    for (int i = 0; i < N; i++) {
        a[i] = b[i] = c[i] = 0;
    }

    queue q;

    // We will learn how to simplify this example later
    buffer a_buf{a};
    buffer b_buf{b};
    buffer c_buf{c};

    q.submit([&](handler &h) {
        accessor a(a_buf, h, read_only);
        accessor b(b_buf, h, write_only);
        h.parallel_for( // computeB
            N, [=](id<1> i) { b[i] = a[i] + 1; });
    });

    q.submit([&](handler &h) {
        accessor a(a_buf, h, read_only);
        h.parallel_for( // readA
            N, [=](id<1> i) {
                // Useful only as an example
                int data = a[i];
            });
    });

    q.submit([&](handler &h) {
        // RAW of buffer B
        accessor b(b_buf, h, read_only);
        accessor c(c_buf, h, write_only);
        h.parallel_for( // computeC
            N, [=](id<1> i) { c[i] = b[i] + 2; });
    });

    // read C on host
    host_accessor host_acc_c(c_buf, read_only);
    for (int i = 0; i < N; i++) {
        std::cout << host_acc_c[i] << " ";
    }
    std::cout << "\n";
    return 0;
}
```

图 3.13: Read-after-Write

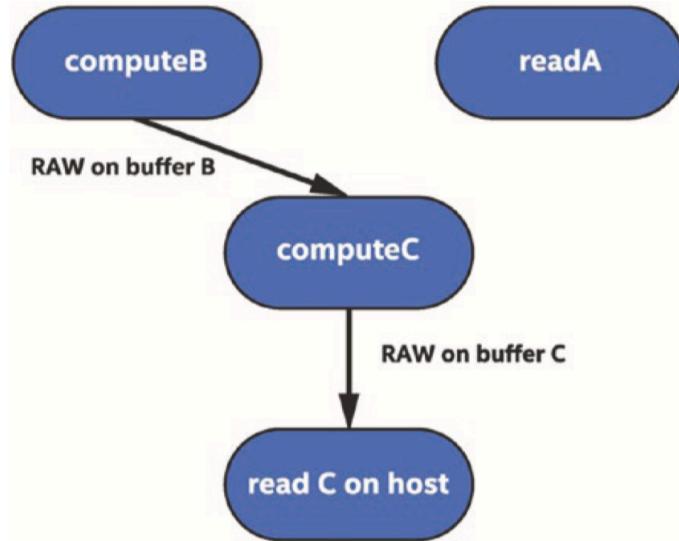


图 3.14: RAW task graph

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

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

Kernel readA 还为 Buffer `a_buf` 创建一个只读访问器。由于 Kernel readA 是在 Kernel computeB 之后提交的，因此这会创建 Read-afterRead (RAR) 场景。然而，RAR 不会对运行时施加额外的限制，并且 Kernel 可以自由地以任何顺序执行。事实上，运行时可能更喜欢在 Kernel computeB 之前执行 Kernel readA，甚至同时执行两者。两者都需要将 Buffer `a_buf` 复制到设备，但 Kernel computeB 还需要复制 Buffer `b_buf`，以防任何现有值不被 computeB 覆盖并且可能被以后的 Kernel 使用。这意味着运行时可以在 Buffer `b_buf` 的数据传输发生时执行 Kernel readA，并且还表明即使

Kernel 仅写入 Buffer，Buffer 的原始内容仍可能被移动到设备，因为无法保证 Buffer 中的所有值都将由 Kernel 写入（请参阅第 7 章了解允许我们在这些情况下进行优化的标签）。

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

```

#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    std::array<int, N> a, b;
    for (int i = 0; i < N; i++) {
        a[i] = b[i] = 0;
    }

    queue q;
    buffer a_buf{a};
    buffer b_buf{b};

    q.submit([&](handler &h) {
        accessor a(a_buf, h, read_only);
        accessor b(b_buf, h, write_only);
        h.parallel_for( // computeB
            N, [=](id<1> i) { b[i] = a[i] + 1; });
    });

    q.submit([&](handler &h) {
        // WAR of buffer A
        accessor a(a_buf, h, write_only);
        h.parallel_for( // rewriteA
            N, [=](id<1> i) { a[i] = 21 + 21; });
    });

    q.submit([&](handler &h) {
        // WAW of buffer B
        accessor b(b_buf, h, write_only);
        h.parallel_for( // rewriteB
            N, [=](id<1> i) { b[i] = 30 + 12; });
    });

    host_accessor host_acc_a(a_buf, read_only);
    host_accessor host_acc_b(b_buf, read_only);
    for (int i = 0; i < N; i++) {
        std::cout << host_acc_a[i] << " " << host_acc_b[i]
            << " ";
    }
    std::cout << "\n";
    return 0;
}

```

图 3.15: Write-after-Read and Write-after-Write

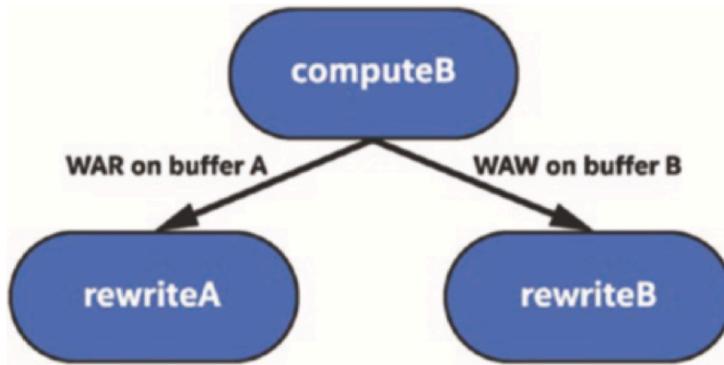


图 3.16: *WAR and WAW task graph*

在图 3-15 和 3-16 中, 我们再次执行三个 Kernel: computeB、rewriteA 和 rewriteB。KernelcomputeB 再次读取 Buffer a_buf 并写入 Buffer b_buf, KernelrewriteA 写入 Buffer a_buf, KernelrewriteB 写入 Buffer b_buf。理论上, Kernel rewriteA 可以比 KernelcomputeB 更早执行, 因为在 Kernel 准备好之前需要传输的数据较少, 但它必须等到 KernelcomputeB 完成之后, 因为对 Buffer a_buf 存在 WAR 依赖性。

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

3.9 选择数据管理策略

为我们的应用程序选择正确的数据管理策略很大程度上取决于个人喜好。事实上, 我们可能会从一种策略开始, 随着我们的项目成熟而转向另一

种策略。然而，有一些有用的指南可以帮助我们选择满足我们需求的策略。

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

如果我们决定从一开始就完全控制所有数据移动，那么我们要从使用 USM 设备分配的显式数据移动开始。我们只需要确保在主机和设备之间添加所有必要的副本即可！

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

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

3.10 Handler 类: 关键成员

```

class handler {
    ...
    // Specifies event(s) that must be complete before the
    // action defined in this command group executes.
    void depends_on({event / std::vector<event> & });

    // Enqueues a memcpy from Src to Dest.
    // Count bytes are copied.
    void memcpy(void* Dest, const void* Src, size_t Count);

    // Enqueues a memcpy from Src to Dest.
    // Count elements are copied.
    template <typename T>
    void copy(const T* Src, T* Dest, size_t Count);

    // Enqueues a memset operation on the specified pointer.
    // Writes the first byte of Value into Count bytes.
    void memset(void* Ptr, int Value, size_t Count);

    // Enques a fill operation on the specified pointer.
    // Fills Pattern into Ptr Count times.
    template <typename T>
    void fill(void* Ptr, const T& Pattern, size_t Count);

    // Submits a kernel of one work-item for execution.
    template <typename KernelName, typename KernelType>
    void single_task(KernelType KernelFunc);

    // Submits a kernel with NumWork-items work-items for
    // execution.
    template <typename KernelName, typename KernelType,
              int Dims>
    void parallel_for(range<Dims> NumWork - items,
                     KernelType KernelFunc);

    // Submits a kernel for execution over the supplied
    // nd_range.
    template <typename KernelName, typename KernelType,
              int Dims>
    void parallel_for(nd_range<Dims> ExecutionRange,
                     KernelType KernelFunc);
    ...
};


```

图 3.17: Simplified definition of the non-accessor members of the handler class

```

class handler {
    ...
    // Specifies event(s) that must be complete before the
    // action. Copy to/from an accessor.
    // Valid combinations:
    // Src: accessor, Dest: shared_ptr
    // Src: accessor, Dest: pointer
    // Src: shared_ptr Dest: accessor
    // Src: pointer Dest: accessor
    // Src: accessor Dest: accessor
    template <typename T_Src, typename T_Dst, int Dims,
              access::mode AccessMode,
              access::target AccessTarget,
              access::placeholder IsPlaceholder =
              access::placeholder::false_t>
    void copy(accessor<T_Src, Dims, AccessMode,
              AccessTarget, IsPlaceholder> Src,
              shared_ptr_class<T_Dst> Dst);
    void copy(shared_ptr_class<T_Src> Src,
              accessor<T_Dst, Dims, AccessMode, AccessTarget,
              IsPlaceholder>
              Dst);
    void copy(accessor<T_Src, Dims, AccessMode, AccessTarget,
              IsPlaceholder> Src,
              T_Dst *Dst);
    void copy(const T_Src *Src,
              accessor<T_Dst, Dims, AccessMode, AccessTarget,
              IsPlaceholder> Dst);
    template <typename T_Src, int Dims_Src,
              access::mode AccessMode_Src,
              access::target AccessTarget_Src, typename T_Dst,
              int Dims_Dst, access::mode AccessMode_Dst,
              access::target AccessTarget_Dst,
              access::placeholder IsPlaceholder_Src =
              access::placeholder::false_t,
              access::placeholder IsPlaceholder_Dst =
              access::placeholder::false_t>
    void copy(accessor<T_Src, Dims_Src, AccessMode_Src,
              AccessTarget_Src, IsPlaceholder_Src> Src,
              accessor<T_Dst, Dims_Dst, AccessMode_Dst,
              AccessTarget_Dst, IsPlaceholder_Dst> Dst);

    // Provides a guarantee that the memory object accessed by
    // the accessor is updated on the host after this action
    // executes.
    template <typename T, int Dims, access::mode AccessMode,
              access::target AccessTarget,
              access::placeholder IsPlaceholder =
              access::placeholder::false_t>
    void update_host(accessor<T, Dims, AccessMode,
                      AccessTarget, IsPlaceholder> Acc);
    ...
};

```

图 3.18: Simplified definition of the accessor members of the handler class

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

一个密切相关的类，队列类，在第 2 章末尾有类似的解释。

3.11 总结

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

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

本章概述了统一共享内存和 Buffer。我们在第 6 章中更详细地探讨了 USM 的所有模式和行为。第 7 章更深入地探讨了 Buffer，包括创建 Buffer 和控制其行为的所有不同方法。第 8 章回顾了控制 Kernel 执行和数据移动顺序的队列调度机制。

4 表达并行性

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

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

- 为什么有不止一种方式来表达并行性？
- 我应该使用哪种表达并行性的方法？
- 关于执行模型我到底需要了解多少？

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

4.1 Kernel 内的并行性

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

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

注 14 保证功能便携性并不等于保证高性能！

支持的设备存在很大的多样性，我们必须记住，不同的架构是针对不同的用例设计和优化的。每当我们希望在特定设备上实现最高水平的性能时，

无论我们使用哪种编程语言，我们都应该始终期望需要一些额外的手动优化工作！此类特定于设备的优化的示例包括针对特定缓存大小的阻塞、选择分摊调度开销的工作粒度大小、利用专用指令或硬件单元，以及最重要的是选择适当的算法。其中一些示例将在第 15、16 和 17 章中重新讨论。

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

4.2 循环与 Kernel

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

```
for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i];
}
```

图 4.1: 将向量加法表示为串行循环

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

并行循环的确切含义有些模糊（由于不同并行编程语言和运行时对该术语的重载），但许多常见的并行循环结构表示应用于顺序循环的编译器转换。这种编程模型使我们能够编写顺序循环，然后才提供有关如何安全地并行执行不同迭代的信息。这些模型非常强大，与其他最先进的编译器优化集

成良好，并极大地简化了并行编程，但并不总是鼓励我们在开发的早期阶段考虑并行性。

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

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

图 4.2: 循环重写（在伪代码中）为并行 Kernel

图 4.2 显示了使用伪代码重写为 Kernel 的简单循环示例。该 Kernel 中的并行机会是清晰明确的：Kernel 可以由任意数量的实例并行执行，并且每个实例独立地应用于单独的数据块。通过将此操作编写为 Kernel，我们断言并行运行是安全的（并且理想情况下应该并行运行）。

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

注 15 我们越早将思维从并行循环转移到 Kernel，就越容易使用 C++ 和 SYCL 编写有效的并行程序。

4.3 多维 Kernel

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

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

一个重要的考虑因素是哪个维度是连续的或单位步幅（即，多维空间中的哪些位置在一维映射中彼此相邻）。SYCL 中与并行性相关的所有多维量都使用相同的约定：维度从 0 到 N-1 进行编号，其中维度 N-1 对应于连

续维度。无论多维数量被写为列表（例如，在构造函数中）或类支持多个下标运算符，此编号都从左到右应用（从左侧的维度 0 开始）。此约定与标准 C++ 中多维数组的行为一致。

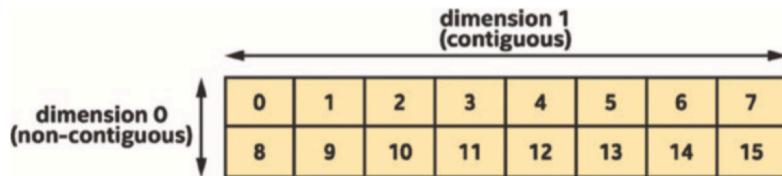


图 4.3: 映射到线性索引的大小 $(2, 8)$ 的二维范围

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

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

4.4 语言特性概述

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

4.4.1 将 Kernel 与主机代码分离

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

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

注 16 (Lambda 表达式不被认为是有害的) 为了开始使用 SYCL，无需完全理解 C++ 规范中有关 lambda 表达式的所有内容 - 我们需要知道的是

lambda 表达式的主体代表 *Kernel*, 并且（按值）捕获的变量将是作为参数传递给 *Kernel*。

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

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

4.5 不同形式的并行 *Kernel*

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

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

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

第三种形式提供了一种实验性的替代语法，用于使用类似于嵌套并行循环的语法来表达 ND 范围 *Kernel*。第三种形式称为分层数据并行，指的是用户源代码中出现的嵌套结构的层次结构。编译器对此语法的支持仍然不成熟，并且许多 *SYCL* 实现不能像其他两种形式那样有效地实现分层数据并行 *Kernel*。语法也不完整，因为 *SYCL* 有许多与分层 *Kernel* 不兼容或无法访问的性能支持功能。*SYCL* 中的分层并行性正在更新过程中，并且 *SYCL* 规范包含一条注释，建议新代码在该功能准备就绪之前不要使用分层并行性；为了与本说明的精神保持一致，本书的其余部分仅教授基本的和 ND 范围的并行性。

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

4.6 基础数据并行 Kernel

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

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

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

4.6.1 了解基本数据并行 Kernel

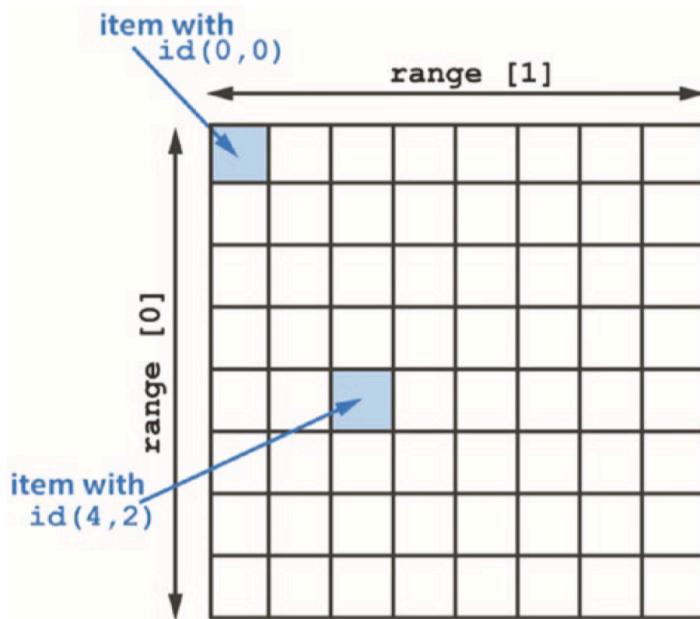


图 4.4: 基本并行 Kernel 的执行空间, 显示为 64 个“项”的 2D 范围

基本并行 Kernel 的执行空间被称为它的执行范围, 并且 Kernel 的每个实例被称为一个“项”。图 4-4 对此进行了示意性表示。

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

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

4.6.2 编写基本数据并行 Kernel

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

图 4.5: 用 *parallel_for* 表达向量加法核

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

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

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

图 4.6: 用 *parallel_for* 表达矩阵加法核

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

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

```

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

图 4.7: 用 *parallel_for* 表达平方矩阵的朴素矩阵乘法核

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

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

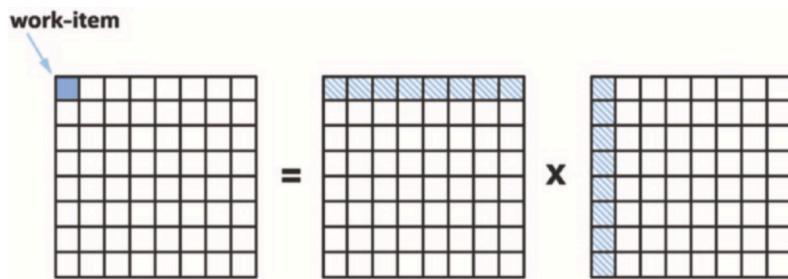


图 4.8: 将矩阵乘法工作映射到执行范围内的“项”

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

4.6.3 基本数据并行 Kernel 的详细信息

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

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

```
template <int Dimensions = 1>
class range {
public:
    // Construct a range with one, two or three dimensions
    range(size_t dim0);
    range(size_t dim0, size_t dim1);
    range(size_t dim0, size_t dim1, size_t dim2);

    // Return the size of the range in a specific dimension
    size_t get(int dimension) const;
    size_t &operator[](int dimension);
    size_t operator[](int dimension) const;

    // Return the product of the size of each dimension
    size_t size() const;

    // Arithmetic operations on ranges are also supported
};
```

图 4.9: range 类的简化定义

图 4-9 显示了范围类的简化定义, 显示了构造函数和查询其范围的各种方法。

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

```
template <int Dimensions = 1>
class id {
public:
    // Construct an id with one, two or three dimensions
    id(size_t dim0);
    id(size_t dim0, size_t dim1);
    id(size_t dim0, size_t dim1, size_t dim2);

    // Return the component of the id in a specific dimension
    size_t get(int dimension) const;
    size_t &operator[](int dimension);
    size_t operator[](int dimension) const;

    // Arithmetic operations on ids are also supported
};
```

图 4.10: *id* 类的简化定义

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

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

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

item 类 “项”代表 Kernel 函数的单个实例, 封装了 Kernel 的执行范围和该范围内的实例索引 (分别使用范围和 *id*)。与 *range* 和 *id* 一样, 它的维数必须在编译时已知。

```

template <int Dimensions = 1, bool WithOffset = true>
class item {
public:
    // Return the index of this item in the kernel's execution
    // range
    id<Dimensions> get_id() const;
    size_t get_id(int dimension) const;
    size_t operator[](int dimension) const;

    // Return the execution range of the kernel executed by
    // this item
    range<Dimensions> get_range() const;
    size_t get_range(int dimension) const;

    // Return the offset of this item (if WithOffset == true)
    id<Dimensions> get_offset() const;

    // Return the linear index of this item
    // e.g. id(0) * range(1) * range(2) + id(1) * range(2) +
    // id(2)
    size_t get_linear_id() const;
};


```

图 4.11: *item* 类的简化定义

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

4.7 显式 ND 范围 Kernel

并行 Kernel 的第二种形式用“项”属于组的执行范围替换基本数据并行 Kernel 的平坦执行范围。这种形式最适合我们想要在 Kernel 中表达某些局部性概念的情况。为不同类型的组定义和保证不同的行为，使我们能够更深入地了解和/或控制如何将工作映射到特定的硬件平台。

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

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

4.7.1 了解显式 ND 范围并行 Kernel

ND 范围 Kernel 的执行范围分为 Work-Groups、Sub-Groups 和 Work-Items。ND-range 表示总的执行范围, 它被划分为统一大小的 Work-Groups (即, Work-Groups 大小必须在每个维度上精确地除以 ND-range 大小)。每个 Work-Groups 可以根据实施进一步划分为 Sub-Groups。了解为 Work-Items 和每种类型的组定义的执行模型是编写正确且可移植的程序的重要组成部分。

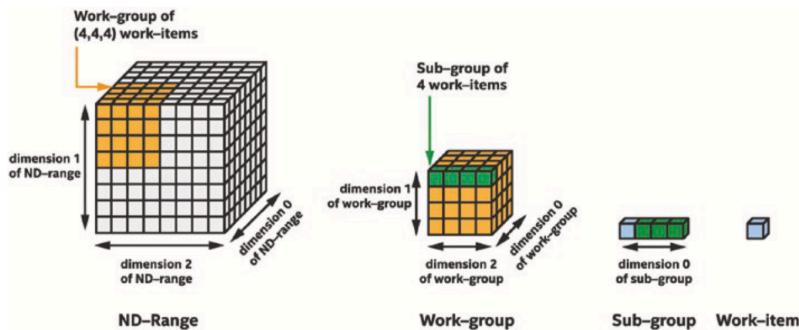


图 4.12: 三维 ND 范围分为 Work-Groups、Sub-Groups 和 Work-Items

图 4-12 显示了大小为 $(8, 8, 8)$ 的 ND 范围分为 8 个大小为 $(4, 4, 4)$ 的 Work-Groups 的示例。每个 Work-Groups 包含 16 个一维 Sub-Groups, 每组有 4 个 Work-Items。请特别注意维度的编号: Sub-Groups 始终是一维的, 因此 ND 范围和 Work-Groups 的维度 2 成为 Sub-Groups 的维度 0。

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

在本章中, 我们仅关注 ND 范围执行模型在通用目标平台方面的语义保证, 并且我们不会涵盖其到任何一个平台的映射。有关 GPU、CPU 和

FPGA 的硬件映射和性能建议的详细信息, 请分别参阅第 15、16 和 17 章。

4.7.2 编写显式 ND 范围数据并行 Kernel

Work-Items Work-Items 代表核函数的各个实例。在没有其他分组的情况下, Work-Items 可以按任何顺序执行, 并且不能相互通信或同步, 除非通过对全局内存的原子内存操作 (参见第 19 章)。

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

1. Work-Groups 中的 Work-Items 可以访问 Work-Groups 本地内存, 该内存可能会映射到某些设备上的专用快速内存 (请参阅第 9 章)。
2. Work-Groups 中的 Work-Items 可以使用 Work-GroupsBarrier 进行同步, 并使用 Work-Groups 内存栅栏保证内存一致性 (参见第 9 章)。
3. Work-Groups 中的 Work-Items 可以访问组功能, 提供通用通信例程 (参见第 9 章) 和组算法的实现, 提供通用并行模式的实现, 例如归约和扫描 (参见第 14 章)。

Work-Groups 中 Work-Items 的数量通常在运行时为每个 Kernel 配置, 因为最佳分组将取决于可用并行度 (即 ND 范围的大小) 和目标设备的属性。我们可以使用设备类的查询函数确定特定设备支持的每个 Work-Groups 的最大 Work-Items 数 (参见第 12 章), 并且我们有责任确保每个 Kernel 请求的 Work-Groups 大小已验证。

Work-Groups 执行模型中有一些微妙之处值得强调。

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

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

注 17 (在 Work-Groups 中思考) *Work-Groups* 在许多方面与其他编程模型中的任务概念相似（例如，线程构建块）：任务可以按任何顺序执行（由调度程序控制）；超额订阅带有任务的机器是可能的（甚至是可取的）；尝试在一组任务中实现 *Barrier* 通常不是一个好主意（因为它可能非常昂贵或与调度程序不兼容）。如果我们已经熟悉了基于任务的编程模型，我们可能会发现将 *Work-Groups* 视为数据并行任务是有用的。

Sub-Groups 在许多现代硬件平台上，*Work-Groups* 中的 *Work-Items* 子集（称为 Sub-Groups）在附加调度保证的情况下执行。例如，Sub-Groups 中的 *Work-Items* 可以由于编译器向量化而同时执行，和/或 Sub-Groups 本身可以以强大的前进进度保证来执行，因为它们被映射到独立的硬件线程。

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

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

与 *Work-Groups* 一样，Sub-Groups 内的 *Work-Items* 可以同步、保证内存一致性或通过组函数和组算法执行常见的并行模式。然而，Sub-Groups 没有 *Work-Groups* 本地存储器的等价物（即，没有 Sub-Groups 本地存储器）。相反，Sub-Groups 中的 *Work-Items* 可以使用组算法的子集（俗称“SHUFFLE”操作）直接交换数据，无需显式内存操作（第 9 章）。

注 18 (为什么是“SHUFFLE”?) *OpenCl*、*CUDA* 和 *SPIR-V* 等语言中的“shuffle”操作的名称中都包含“shuffle”（例如，*sub_group_shuffle*、*__shfl* 和 *OpGroupNonUniformShuffle*）。*SYCl* 采用不同的命名约定，以避免与 C++ 中定义的 *std::shuffle* 函数混淆（该函数随机重新排序范围的内容）。

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

与 Work-Groups 一样，Sub-Groups 中的 Work-Items 不需要提供任何特定的前进进度保证 - 实现可以自由地顺序执行 Sub-Groups 中的每个 Work-Items，并且仅在 Work-Items 发生变化时才在 Work-Items 之间切换。遇到 Sub-Groups 集体函数。然而，在某些设备上，Work-Groups 内的所有 Sub-Groups 都保证最终执行（取得进展），这是多种生产者-消费者模式的基石。这是当前实现定义的行为，因此如果我们希望 Kernel 保持可移植性，我们就不能依赖 Sub-Groups 来取得进展。我们期望 SYCL 的未来版本能够提供描述 Sub-Groups 进度保证的设备查询。

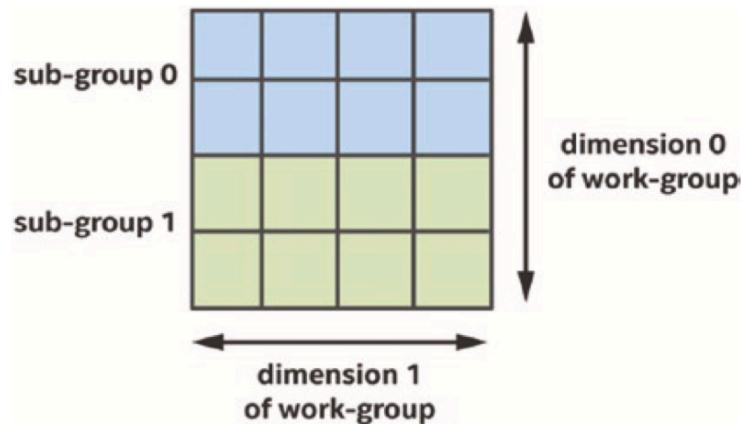


图 4.13：一种可能的 Sub-Groups 映射，其中 Sub-Groups 大小允许大于 Work-Groups 的最高编号（连续）维度的范围，因此 Sub-Groups 看起来是“环绕”

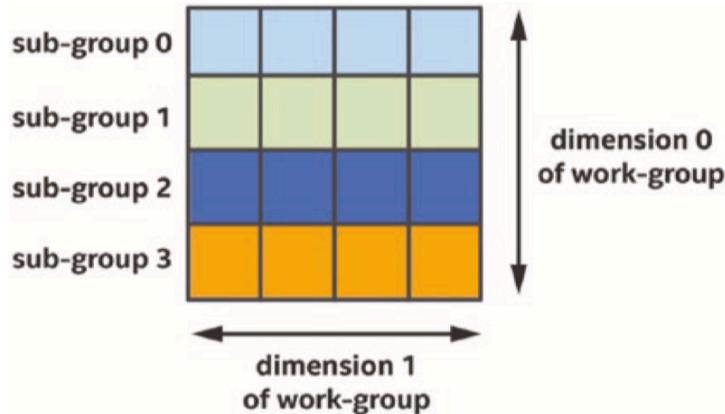


图 4.14: 另一种可能的 Sub-Groups 映射, 其中 Sub-Groups 大小不允许大于 Work-Groups 的最高编号 (连续) 维度的范围

当为特定设备编写 Kernel 时, Work-Items 到 Sub-Groups 的映射是已知的, 并且我们的代码通常可以利用此映射的属性来提高性能。然而, 一个常见的错误是假设因为我们的代码可以在一台设备上运行, 所以它也可以在所有设备上运行。图 4-13 和 4-14 仅显示了将范围为 4, 4 的多维 Kernel 中的 Work-Items 映射到 Sub-Groups (最大 Sub-Groups 大小为 8) 时的两种可能性。图 4-13 生成两个包含 8 个 Work-Items 的 Sub-Groups, 而图 4-14 中的映射生成四个包含 4 个 Work-Items 的 Sub-Groups!

SYCL 当前不提供查询 Work-Items 如何映射到 Sub-Groups 的方法, 也不提供请求特定映射的机制。使用 Sub-Groups 编写可移植代码的最佳方法是使用一维 Work-Groups 或多维 Work-Groups, 其中最高编号的维度可被 Kernel 所需的 Sub-Groups 大小整除。

注 19 (在 Sub-Groups 中思考) 如果我们来自一个需要我们考虑显式矢量化的编程模型, 那么将每个 Sub-Groups 视为一组打包到 SIMD 寄存器中的 Work-Items 可能会很有用, 其中 Sub-Groups 中的每个 Work-Items 对应于 SIMD 通道。当多个 Sub-Groups 同时飞行并且设备保证它们会向前推进时, 这种心智模型扩展到将每个 Sub-Groups 视为并行执行的单独向量指令流。

```

range global{N, N};
range local{B, B};
h.parallel_for(nd_range{global, local},
    [=](nd_item<2> it) {
        int j = it.get_global_id(0);
        int i = it.get_global_id(1);

        for (int k = 0; k < N; ++k) {
            c[j][i] += a[j][k] * b[k][i];
        }
    });

```

图 4.15: 用 ND 范围 parallel_for 表达朴素矩阵乘法核

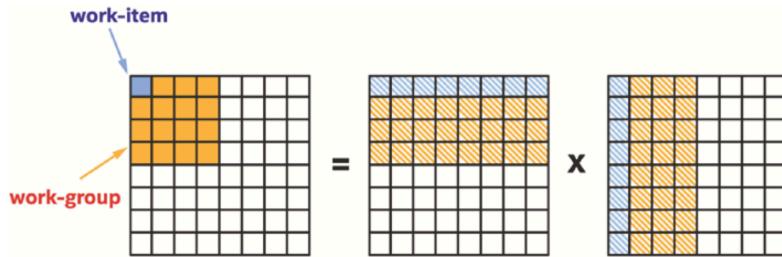


图 4.16: 将矩阵乘法映射到 Work-Groups 和 Work-Items

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

到目前为止，我们的矩阵乘法示例依赖于硬件缓存来优化同一 Work-Groups 中的 Work-Items 对 A 和 B 矩阵的重复访问。此类硬件缓存在传统 CPU 架构上很常见，并且在 GPU 架构上变得越来越常见，但一些架构已经明确管理可以提供更高性能（例如，通过更低延迟）的“暂存器”内存。ND 范围 Kernel 可以使用本地访问器来描述应放置在 Work-Groups 本地内

存中的分配，然后实现可以自由地将这些分配映射到特殊内存（如果存在）。该 Work-Groups 本地内存的使用将在第 9 章中介绍。

4.7.3 显式 ND 范围数据并行 Kernel 的详细信息

与基本数据并行 Kernel 相比，ND 范围数据并行 Kernel 使用不同的类：range 被 nd_range 替换，item 被 nd_item 替换。还有两个新类，代表 Work-Items 可能所属的不同类型的组：与 Work-Groups 相关的功能封装在 group 类中，与 Sub-Groups 相关的功能封装在 sub_group 类中。

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

```
template <int Dimensions = 1>
class nd_range {
public:
    // Construct an nd_range from global and work-group local
    // ranges
    nd_range(range<Dimensions> global,
              range<Dimensions> local);

    // Return the global and work-group local ranges
    range<Dimensions> get_global_range() const;
    range<Dimensions> get_local_range() const;

    // Return the number of work-groups in the global range
    range<Dimensions> get_group_range() const;
};
```

图 4.17: *nd_range* 类的简化定义

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

nd_item 类 `nd_item` 是“项”的 ND 范围形式，再次封装了 Kernel 的执行范围以及该范围内“项”的索引。`nd_item` 与 `item` 的不同之处在于如何查询和表示其在范围中的位置，如图 4-18 中简化的类定义所示。例如，我们可以使用 `get_global_id()` 函数查询（全局）ND 范围中的“项”索引，或者使用 `get_local_id()` 函数查询“项”在其（本地）父 Work-Groups 中的索引。

```
template <int Dimensions = 1>
class nd_item {
public:
    // Return the index of this item in the kernel's execution
    // range
    id<Dimensions> get_global_id() const;
    size_t get_global_id(int dimension) const;
    size_t get_global_linear_id() const;

    // Return the execution range of the kernel executed by
    // this item
    range<Dimensions> get_global_range() const;
    size_t get_global_range(int dimension) const;

    // Return the index of this item within its parent
    // work-group
    id<Dimensions> get_local_id() const;
    size_t get_local_id(int dimension) const;
    size_t get_local_linear_id() const;

    // Return the execution range of this item's parent
    // work-group
    range<Dimensions> get_local_range() const;
    size_t get_local_range(int dimension) const;

    // Return a handle to the work-group
    // or sub-group containing this item
    group<Dimensions> get_group() const;
    sub_group get_sub_group() const;
};
```

图 4.18: `nd_item` 类的简化定义

nd_item 类还提供了用于获取描述“项”所属组和 Sub-Groups 的类句柄的函数。这些类提供了用于查询 ND 范围中“项”索引的替代接口。

group 类 `group` 类封装了与 Work-Groups 相关的所有功能，简化的定义如图 4-19 所示。

```

template <int Dimensions = 1>
class group {
public:
    // Return the index of this group in the kernel's
    // execution range
    id<Dimensions> get_id() const;
    size_t get_id(int dimension) const;
    size_t get_linear_id() const;

    // Return the number of groups in the kernel's execution
    // range
    range<Dimensions> get_group_range() const;
    size_t get_group_range(int dimension) const;

    // Return the number of work-items in this group
    range<Dimensions> get_local_range() const;
    size_t get_local_range(int dimension) const;
};

```

图 4.19: *group* 类的简化定义

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

```

void body(group& g);

h.parallel_for(nd_range{global, local}, [=](nd_item<1> it) {
    group<1> g = it.get_group();
    range<1> r = g.get_local_range();
    ...
    body(g);
});

```

图 4.20：使用 *group* 类提高可读性

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

sub_group 类 sub_group 类封装了与 Sub-Groups 相关的所有功能，简化的定义如图 4-21 所示。与 Work-Groups 不同，sub_group 类是访问 Sub-Groups 功能的唯一方法；它的功能在 nd_item 中没有重复。

```
class sub_group {
public:
    // Return the index of the sub-group
    id<1> get_group_id() const;

    // Return the number of sub-groups in this item's parent
    // work-group
    range<1> get_group_range() const;

    // Return the index of the work-item in this sub-group
    id<1> get_local_id() const;

    // Return the number of work-items in this sub-group
    range<1> get_local_range() const;

    // Return the maximum number of work-items in any
    // sub-group in this item's parent work-group
    range<1> get_max_local_range() const;
};
```

图 4.21: *sub_group* 类的简化定义

请注意，有单独的函数用于查询当前 Sub-Groups 中的 Work-Items 数以及 Work-Groups 内任何 Sub-Groups 中的最大 Work-Items 数。这些是否不同以及如何不同取决于具体设备的 Sub-Groups 实现方式，但其目的是反映编译器目标 Sub-Groups 大小与运行时 Sub-Groups 大小之间的任何

差异。例如，非常小的 Work-Groups 可以包含比编译时 Sub-Groups 大小更少的 Work-Items，或者可以使用不同大小的 Sub-Groups 来处理不能被 Sub-Groups 大小整除的 Work-Groups 和维度。

4.8 将计算映射到 Work-Items

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

4.8.1 一对映射

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

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

4.8.2 多对映射

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

编写这种形式的 Kernel 需要进行两处更改：

1. Kernel 必须接受一个描述工作总量的参数。
2. Kernel 必须包含一个将工作分配给 Work-Items 的循环。

图 4-22 给出了此类 Kernel 的一个简单示例。请注意，Kernel 内部的循环有一种稍微不寻常的形式 - 起始索引是 Work-Items 在全局范围内的索引，步幅是 Work-Items 的总数。这种数据到 Work-Items 的循环调度确保循环的所有 N 次迭代都将由 Work-Items 执行，而且线性 Work-Items 访问连续的内存位置（以改进缓存局部性和矢量化行为）。工作可以类似地跨组或各个组中的 Work-Items 进行分配，以进一步改善局部性。

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

图 4.22: 具有独立数据和执行范围的 Kernel

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

4.9 选择 Kernel 形式

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

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

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

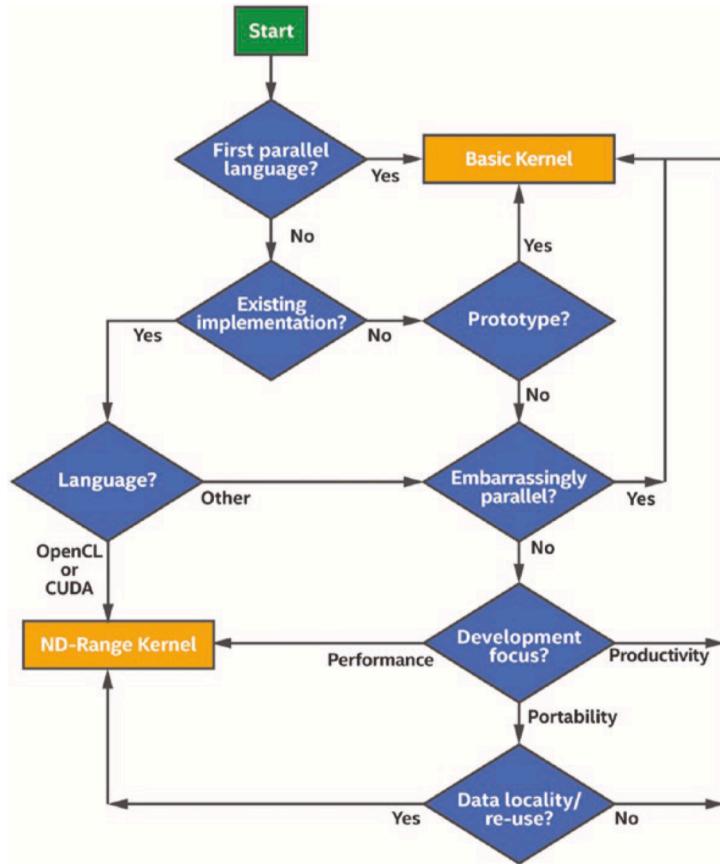


图 4.23: 帮助为我们的 Kernel 选择正确的形式

第一个指南是图 4-23 所示的流程图，它根据以下条件选择 Kernel 形式：

1. 我们是否有并行编程的经验
2. 无论我们是从头开始编写新代码还是移植用不同语言编写的现有并行程序
3. 我们的 Kernel 是否是高度并行的，或者在 Kernel 函数的不同实例之间重用数据
4. 我们是否在 SYCL 中编写新 Kernel 是为了最大限度地提高性能、提

高代码的可移植性，还是因为它提供了比低级语言更高效的表达并行性的方法。

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

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

4.10 总结

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

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

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

5 错误处理

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

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

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

5.1 安全第一

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

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

注 20 如果我们不添加任何错误管理代码并且发生错误，我们仍然会看到程序异常终止，这表明需要进行更深入的研究。

5.2 错误类型

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

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

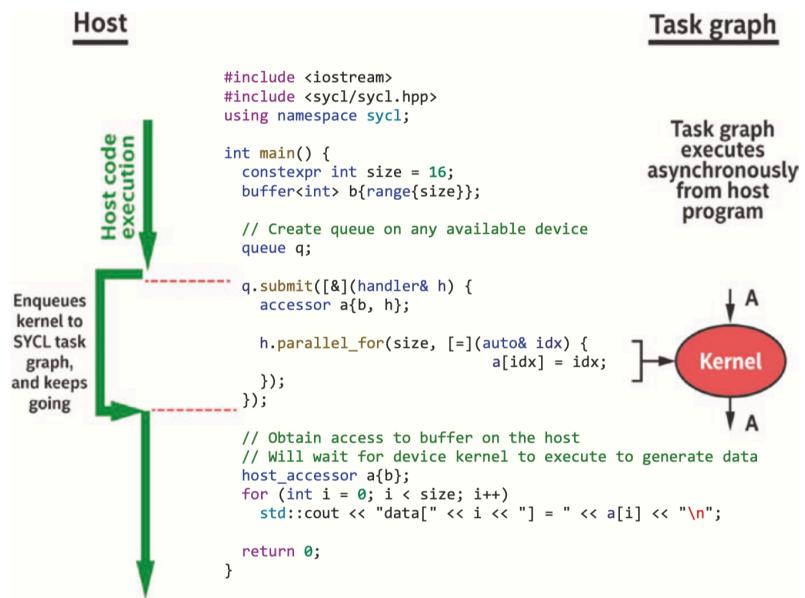


图 5.1: 主机程序和任务图执行的分离

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

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

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

异步错误发生在图 5-1 右侧的部分，只有在执行任务图中的操作时才

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

5.3 让我们创建一些错误！

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

5.3.1 同步错误

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
    buffer<int> b{range{16}};

    // ERROR: Create sub-buffer larger than size of parent
    // buffer. An exception is thrown from within the buffer
    // constructor.
    buffer<int> b2(b, id{8}, range{16});

    return 0;
}

Example Output:
terminate called after throwing an instance of 'sycl::_V1::invalid_object_error'
  what(): Requested sub-buffer size exceeds the size of the parent buffer -30
(PI_ERROR_INVALID_VALUE)
Aborted
```

图 5.2: 创建同步错误

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

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

5.3.2 异步错误

```

#include <sycl/sycl.hpp>
using namespace sycl;

// Our example asynchronous handler function
auto handle_async_error = [] (exception_list elist) {
    for (auto &e : elist) {
        try {
            std::rethrow_exception(e);
        } catch (...) {
            std::cout << "Caught SYCL ASYNC exception!!\n";
        }
    }
};

void say_device(const queue &Q) {
    std::cout << "Device : "
        << Q.get_device().get_info<info::device::name>()
        << "\n";
}

class something_went_wrong {}; // Example exception type

int main() {
    queue q{cpu_selector_v, handle_async_error};
    say_device(q);

    q.submit([](handler &h) {
        h.host_task([]() { throw(something_went_wrong{}); });
    }).wait();

    return 0;
}

Example output:
Device : Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Caught SYCL ASYNC exception!!

```

图 5.3: 创建异步错误

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

5-3 中所示的主机任务示例只是一种可能性，而不是异步错误的要求。

5.4 应用程序错误处理策略

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

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

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

5.4.1 忽略错误处理

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

```
#include <iostream>

class something_went_wrong {};

int main() {
    std::cout << "Hello\n";
    throw(something_went_wrong{});
}

Example output:
Hello
terminate called after throwing an instance of 'something_went_wrong'
Aborted
```

图 5.4: C++ 中未处理的异常

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

```
#include <iostream>

int main() {
    std::cout << "Hello\n";
    std::terminate();
}
```

Example output:

```
Hello
terminate called without an active exception
Aborted
```

图 5.5: `std::terminate` 在未处理 SYCL 异步异常时调用

图 5-5 显示了调用 `std::terminate` 的示例输出，这将是我们的应用程序中未处理的 SYCL 异步错误的结果。我们可以使用此代码来测试特定操作系统在这种情况下会报告什么。

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

5.4.2 同步错误处理

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

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

图 5.6: 专门捕获 *sycl::exception* 的模式

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

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

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
    try {
        buffer<int> b{range{16}};

        // ERROR: Create sub-buffer larger than size of parent
        // buffer. An exception is thrown from within the buffer
        // constructor.
        buffer<int> b2(b, id{8}, range{16});

    } catch (sycl::exception &e) {
        // Do something to output or handle the exception
        std::cout << "Caught synchronous SYCL exception: "
               << e.what() << "\n";
        return 1;
    } catch (std::exception &e) {
        std::cout << "Caught std exception: " << e.what()
               << "\n";
        return 2;
    } catch (...) {
        std::cout << "Caught unknown exception\n";
        return 3;
    }

    return 0;
}
```

Example output:

```
Caught synchronous SYCL exception: Requested sub-buffer
size exceedsthe size of the parent buffer -30
(PI_ERROR_INVALID_VALUE)
```

图 5.7: 从代码块中捕获异常的模式

图 5-7 中提供了一个稍微更完整的示例，其中处理了其他类别的异常。

5.4.3 异步错误处理

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

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

5.4.4 异步处理程序

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

```
// Our simple asynchronous handler function
auto handle_async_error = [](exception_list elist) {
    for (auto& e : elist) {
        try {
            std::rethrow_exception(e);
        } catch (sycl::exception& e) {
            std::cout << "ASYNC EXCEPTION!!\n";
            std::cout << e.what() << "\n";
        }
    }
};
```

图 5.8: 定义为 *lambda* 的异步处理程序实现示例

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

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

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

```
queue my_queue{ gpu_selector_v, handle_async_error };
```

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

```
context my_context{handle_async_error };
```

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

注 21 在定义异步处理程序时，大多数开发人员应该在队列上定义它们，除非出于其他原因已经显式管理上下文。

```
// Our simple asynchronous handler function
auto handle_async_error = [](exception_list elist) {
    for (auto& e : elist) {
        try {
            std::rethrow_exception(e);
        } catch (sycl::exception& e) {
            // Print information about the asynchronous exception
        } catch (...) {
            // Print information about non-sycl::exception
        }
    }

    // Terminate abnormally to make clear to user that
    // something unhandled happened
    std::terminate();
};

Example output:
Device : Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
terminate called without an active exception
Aborted
```

图 5.9: 默认异步处理程序的行为示例

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

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

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

虽然异步处理程序在内部做什么由我们决定，但一个常见的错误是打印一条错误消息（可能会在程序中的其他消息的噪音中错过），然后完成处理程序函数。除非我们制定了错误管理原则，允许我们恢复已知的程序状态并确信继续执行是安全的，否则我们应该考虑在异步处理程序函数中终止

应用程序。这减少了检测到错误但无意中允许应用程序继续执行的程序出现错误结果的机会。在许多程序中，一旦我们检测到异步异常，异常终止是首选结果。

注 22 如果未建立全面的错误恢复和管理机制，请考虑在输出有关错误的信息后终止异步处理程序中的应用程序。

5.4.5 处理程序的调用

异步处理程序由运行时在特定时间调用。错误发生时不会立即报告，因为如果是这种情况，错误管理和安全应用程序编程（特别是多线程）将变得更加困难和昂贵（例如，主机和设备之间的额外同步）。相反，异步处理程序会在以下特定时间被调用：

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

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

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

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

5.5 设备上的错误

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

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

5.6 总结

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

6 统一共享内存

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

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

6.1 为什么要使用 USM?

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

6.2 分配类型

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

设备可能不支持所有类型（甚至任何类型）的 USM 分配。

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

Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	✗	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	Can migrate between host and device

图 6.1: USM 分配类型

6.2.1 设备分配

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

6.2.2 主机分配

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

6.2.3 共享分配

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

自动迁移可以通过多种方式实现。无论运行时选择哪种方式实现共享分配，它们通常都会付出延迟增加的代价。通过设备分配，我们可以准确地知道需要复制多少内存，并可以安排复制尽早开始。自动迁移机制无法预见未来，并且在某些情况下，在 Kernel 尝试访问数据之前不会开始移动数据。然后，Kernel 必须等待或阻塞，直到数据移动完成才能继续执行。在其他情

况下，运行时可能无法确切知道 Kernel 将访问多少数据，并且可能会保守地移动比所需数量更大的数据，这也会增加 Kernel 的延迟。

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

6.3 分配内存

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

6.3.1 我们需要知道什么？

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

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

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

最后，设备分配（以及一些共享分配）还要求我们指定哪个设备将为分配提供内存。这很重要，因为我们不想超额订阅设备的内存（除非设备能够

支持这一点——我们将在本章后面讨论数据迁移时详细介绍这一点)。USM 分配例程可以通过添加这些额外参数来区别于它们的 C++ 类似例程。

6.3.2 多种风格

有时，试图用单一选项取悦所有人被证明是一项不可能完成的任务，就像有些人喜欢咖啡而不是茶，或者 emacs 而不是 vi 一样。如果我们问程序员分配接口应该是什么样子，我们会得到几个不同的答案。USM 拥抱这种选择的多样性，并提供几种不同风格的分配接口。这些不同的风格是 C 风格、C++ 风格和 C++ 分配器风格。我们现在将讨论每一个并指出它们的相似点和不同点。

```

// Named Functions
void *malloc_device(size_t size, const device &dev,
                     const context &ctxt);
void *malloc_device(size_t size, const queue &q);
void *aligned_alloc_device(size_t alignment, size_t size,
                           const device &dev,
                           const context &ctxt);
void *aligned_alloc_device(size_t alignment, size_t size,
                           const queue &q);

void *malloc_host(size_t size, const context &ctxt);
void *malloc_host(size_t size, const queue &q);
void *aligned_alloc_host(size_t alignment, size_t size,
                        const context &ctxt);
void *aligned_alloc_host(size_t alignment, size_t size,
                        const queue &q);

void *malloc_shared(size_t size, const device &dev,
                    const context &ctxt);
void *malloc_shared(size_t size, const queue &q);
void *aligned_alloc_shared(size_t alignment, size_t size,
                           const device &dev,
                           const context &ctxt);
void *aligned_alloc_shared(size_t alignment, size_t size,
                           const queue &q);

// Single Function
void *malloc(size_t size, const device &dev,
             const context &ctxt, usm::alloc kind);
void *malloc(size_t size, const queue &q, usm::alloc kind);
void *aligned_alloc(size_t alignment, size_t size,
                    const device &dev, const context &ctxt,
                    usm::alloc kind);
void *aligned_alloc(size_t alignment, size_t size,
                    const queue &q, usm::alloc kind);

```

图 6.2: C 风格的 USM 分配函数

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

我们可以将这种分配方式进一步分为两类：命名函数和单一函数。这两

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

如果不简要提及对齐，我们就无法继续前进。每个版本的 malloc 也有一个 aligned_alloc 对应项。malloc 函数返回与我们设备的默认行为一致的内存。成功时，它将返回一个具有有效对齐方式的合法指针，但在某些情况下，我们可能更愿意手动指定对齐方式。在这些情况下，我们应该使用 aligned_alloc 变体之一，它也要求我们指定所需的分配对齐方式。标准对齐是 2 的幂次。值得注意的是，在许多设备上，分配最大程度地对齐以对应于硬件的功能，因此，虽然我们可能要求分配为 4、8、16 或 32 字节对齐，但实际上我们可能会看到更大的分配空间。对齐可以满足我们的要求，然后是一些。

```
// Named Functions
template <typename T>
T *malloc_device(size_t Count, const device &Dev,
                 const context &Ctxt);
template <typename T>
T *malloc_device(size_t Count, const queue &Q);
template <typename T>
T *aligned_alloc_device(size_t Alignment, size_t Count,
                        const device &Dev,
                        const context &Ctxt);
template <typename T>
T *aligned_alloc_device(size_t Alignment, size_t Count,
                        const queue &Q);

template <typename T>
T *malloc_host(size_t Count, const context &Ctxt);
template <typename T>
T *malloc_host(size_t Count, const queue &Q);
template <typename T>
T *aligned_alloc_host(size_t Alignment, size_t Count,
                      const context &Ctxt);
template <typename T>
T *aligned_alloc_host(size_t Alignment, size_t Count,
                      const queue &Q);

template <typename T>
T *malloc_shared(size_t Count, const device &Dev,
                 const context &Ctxt);
template <typename T>
T *malloc_shared(size_t Count, const queue &Q);
template <typename T>
T *aligned_alloc_shared(size_t Alignment, size_t Count,
                        const device &Dev,
                        const context &Ctxt);
template <typename T>
T *aligned_alloc_shared(size_t Alignment, size_t Count,
                        const queue &Q);

// Single Function
template <typename T>
T *malloc(size_t Count, const device &Dev,
          const context &Ctxt, usm::alloc Kind);
template <typename T>
T *malloc(size_t Count, const queue &Q, usm::alloc Kind);
template <typename T>
T *aligned_alloc(size_t Alignment, size_t Count,
                 const device &Dev, const context &Ctxt,
                 usm::alloc Kind);
template <typename T>
T *aligned_alloc(size_t Alignment, size_t Count,
                 const queue &Q, usm::alloc Kind);
```

图 6.3: C++ 风格的 USM 分配函数

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

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

```
template <typename T, usm::alloc AllocKind,
          size_t Alignment = 0>
class usm_allocator {
public:
    using value_type = T;
    using propagate_on_container_copy_assignment =
        std::true_type;
    using propagate_on_container_move_assignment =
        std::true_type;
    using propagate_on_container_swap = std::true_type;

public:
    template <typename U>
    struct rebind {
        typedef usm_allocator<U, AllocKind, Alignment> other;
    };

    usm_allocator() = delete;
    usm_allocator(const context& syclContext,
                  const device& syclDevice,
                  const property_list& propList = {});
    usm_allocator(const queue& syclQueue,
                  const property_list& propList = {});
    usm_allocator(const usm_allocator& other);
    usm_allocator(usm_allocator&&) noexcept;
    usm_allocator& operator=(const usm_allocator&);
    usm_allocator& operator=(usm_allocator&&);

    template <class U>
    usm_allocator(usm_allocator<U, AllocKind,
                  Alignment> const&) noexcept;

    /// Allocate memory
    T* allocate(size_t count);

    /// Deallocate memory
    void deallocate(T* Ptr, size_t count);

    /// Equality Comparison
    ///
    /// Allocators only compare equal if they are of the same
    /// USM kind, alignment, context, and device
    template <class U, usm::alloc AllocKindU,
              size_t AlignmentU>
    friend bool operator==(
        const usm_allocator<T, AllocKind, Alignment>&,
        const usm_allocator<U, AllocKindU, AlignmentU>&);

    /// Inequality Comparison
    /// Allocators only compare unequal if they are not of the
    /// same USM kind, alignment, context, or device
    template <class U, usm::alloc AllocKindU,
              size_t AlignmentU>
    friend bool operator!=(
        const usm_allocator<T, AllocKind, Alignment>&,
        const usm_allocator<U, AllocKindU, AlignmentU>&);
};
```

图 6.4: C++ 分配器风格的 USM 分配函数

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

6.3.3 释放内存

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

6.3.4 分配示例

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    // Allocate N floats

    // C-style
    float *f1 = static_cast<float *>(malloc_shared(
        N * sizeof(float), q.get_device(), q.get_context()));

    // C++-style
    float *f2 = malloc_shared<float>(N, q);

    // C++-allocator-style
    usm_allocator<float, usm::alloc::shared> alloc(q);
    float *f3 = alloc.allocate(N);

    // Free our allocations
    free(f1, q.get_context());
    free(f2, q);
    alloc.deallocate(f3, N);

    return 0;
}
```

图 6.5: 三种风格的分配方式

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

6.4 数据管理

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

6.4.1 初始化

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

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

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

6.4.2 数据移动

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

策略的选择与我们的硬件支持或我们想要使用的 USM 分配类型有关。

```
#include <array>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    std::array<int, N> host_array;
    int* device_array = malloc_device<int>(N, q);
    for (int i = 0; i < N; i++) host_array[i] = N;

    q.submit([&](handler& h) {
        // copy host_array to device_array
        h.memcpy(device_array, &host_array[0], N * sizeof(int));
    });
    q.wait(); // needed for now (we learn a better way later)

    q.submit([&](handler& h) {
        h.parallel_for(N, [=](id<1> i) { device_array[i]++; });
    });
    q.wait(); // needed for now (we learn a better way later)

    q.submit([&](handler& h) {
        // copy device_array back to host_array
        h.memcpy(&host_array[0], device_array, N * sizeof(int));
    });
    q.wait(); // needed for now (we learn a better way later)

    free(device_array, q);
    return 0;
}
```

图 6.6: USM 显式数据移动示例

显式的 USM 提供的第一个策略是显式数据移动（图 6-6）。

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

显式数据移动的最常见用法是在 USM 中的设备分配之间进行复制，因为它们在主机上无法访问。必须插入显式数据复制确实需要我们付出努力。

此外，它可能是错误的来源：副本可能会被意外省略、复制的数据量可能不正确、或者源或目标指针可能不正确。

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

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

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    int* host_array = malloc_host<int>(N, q);
    int* shared_array = malloc_shared<int>(N, q);
    for (int i = 0; i < N; i++) host_array[i] = i;

    q.submit([&](handler& h) {
        h.parallel_for(N, [=](id<1> i) {
            // access shared_array and host_array on device
            shared_array[i] = host_array[i] + 1;
        });
    });
    q.wait();

    free(shared_array, q);
    free(host_array, q);
    return 0;
}
```

图 6.7: USM 隐式数据移动示例

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

工作是确保数据在使用时在正确的位置可用。

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

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

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

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

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

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

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

细粒度控制 当设备支持共享分配的按需迁移时，在当前未驻留的位置访问内存后，会发生数据移动。但是，Kernel 在等待数据移动完成时可能会停止。它执行的下一条语句甚至可能会导致发生更多数据移动，并给 Kernel 执行带来额外的延迟。

```
#include <sycl/sycl.hpp>
using namespace sycl;

// Appropriate values depend on your HW
constexpr int BLOCK_SIZE = 42;
constexpr int NUM_BLOCKS = 2500;
constexpr int N = NUM_BLOCKS * BLOCK_SIZE;

int main() {
    queue q;
    int *data = malloc_shared<int>(N, q);
    int *read_only_data = malloc_shared<int>(BLOCK_SIZE, q);

    for (int i = 0; i < N; i++) {
        data[i] = -i;
    }

    // Never updated after initialization
    for (int i = 0; i < BLOCK_SIZE; i++) {
        read_only_data[i] = i;
    }

    // Mark this data as "read only" so the runtime can copy
    // it to the device instead of migrating it from the host.
    // Real values will be documented by your backend.
    int HW_SPECIFIC_ADVICE_RO = 0;
    q.mem_advise(read_only_data, BLOCK_SIZE,
                  HW_SPECIFIC_ADVICE_RO);
    event e = q.prefetch(data, BLOCK_SIZE * sizeof(int));

    for (int b = 0; b < NUM_BLOCKS; b++) {
        q.parallel_for(range{BLOCK_SIZE}, e, [=](id<1> i) {
            data[b * BLOCK_SIZE + i] += read_only_data[i];
        });
        if ((b + 1) < NUM_BLOCKS) {
            // Prefetch next block
            e = q.prefetch(data + (b + 1) * BLOCK_SIZE,
                            BLOCK_SIZE * sizeof(int));
        }
    }
    q.wait();

    free(data, q);
    free(read_only_data, q);
    return 0;
}
```

图 6.8: 通过 *prefetch* 和 *mem_advise* 进行细粒度控制

SYCL 为我们提供了一种修改自动迁移机制性能的方法。它通过定义两个函数来实现这一点: *prefetch* 和 *mem_advise*。图 6-8 显示了每种方法的

简单用法。这些函数让我们向运行时提供有关 Kernel 如何访问数据的提示，以便运行时可以选择在 Kernel 尝试访问数据之前开始移动数据。请注意，此示例使用队列快捷方式方法，直接在队列对象上调用 parallel_for，而不是在传递给 submit 方法（命令组）的 lambda 内部调用。

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

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

6.5 查询

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

```

#include <sycl/sycl.hpp>
using namespace sycl;
namespace dinfo = info::device;
constexpr int N = 42;

template <typename T>
void foo(T data, id<1> i) {
    data[i] = N;
}

int main() {
    queue q;
    auto dev = q.get_device();
    auto ctxt = q.get_context();
    bool usm_shared = dev.has(aspect::usm_shared_allocations);
    bool usm_device = dev.has(aspect::usm_device_allocations);
    bool use_USM = usm_shared || usm_device;

    if (use_USM) {
        int *data;
        if (usm_shared) {
            data = malloc_shared<int>(N, q);
        } else /* use device allocations */ {
            data = malloc_device<int>(N, q);
        }
        std::cout << "Using USM with "
                << ((get_pointer_type(data, ctxt) ==
                      usm::alloc::shared)
                     ? "shared"
                     : "device")
                << " allocations on "
                << get_pointer_device(data, ctxt)
                    .get_info<dinfo::name>()
                << "\n";
        q.parallel_for(N, [=](id<1> i) { foo(data, i); });
        q.wait();
        free(data, q);
    } else /* use buffers */ {
        buffer<int, 1> data{range{N}};
        q.submit([&](handler &h) {
            accessor a(data, h);
            h.parallel_for(N, [=](id<1> i) { foo(a, i); });
        });
        q.wait();
    }
    return 0;
}

```

图 6.9: 对 USM 指针和设备的查询

USM 中的指针查询回答两个问题。第一个问题是“这个指针指向什么类

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

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

Aspect	Description
aspect::usm_device_allocations	This device supports device allocations
aspect::usm_host_allocations	This device supports host allocations
aspect::usm_atomic_host_allocations	This device supports host allocations that may be modified atomically by the device
aspect::shared_allocations	This device supports shared allocations
aspect::atomic_shared_allocations	This device supports shared allocations and the host and device may concurrently access and atomically modify shared allocations
aspect::usm_system_allocations	This device supports using allocations made with the system allocator on the device

图 6.10: USM 设备 Aspect

6.6 还有一件事

我们还没有介绍另一种形式的 USM。我们在本章中描述的 USM 形式都需要使用特殊的分配函数。虽然不是一个巨大的负担，但这代表了传统 C++ 代码的变化，传统 C++ 代码使用 malloc 或 new 运算符形式的系统分配器。虽然当今的某些设备（例如 CPU）可能不需要此要求，但大多数加速器设备仍然需要它。因此，我们以更大的可移植性为名描述了如何使用 USM 分配函数。然而，我们相信我们很快就会看到更多支持使用系统分配

器的加速器设计。此类设备将极大地简化程序，使程序员无需担心分配正确类型的 USM 内存或在适当的时间复制正确的数据。从某种意义上说，我们可以将最终的系统分配器支持视为 USM 的最终演进——它将提供共享 USM 分配的好处，而不需要使用特殊的分配函数。

6.7 总结

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

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

在下一章中，我们将介绍数据管理的第二种策略：Buffer。

7 Buffers

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

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

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

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

7.1 Buffers

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

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

图 7.1: *Buffer* 类定义

Buffer 类是一个具有三个模板参数的模板类，如图 7-1 所示。第一个模板参数是 Buffer 将包含的对象的类型。此类型必须是设备可复制的，这扩展了 C++ 定义的普通可复制的概念。可简单复制的类型可以安全地逐字节复制，而无需使用任何特殊的复制或移动构造函数。设备可复制类型将此概念递归地扩展到某些 C++ 类型，例如 std::pair 或 std::tuple。下一个模板参数是描述 Buffer 维数的整数。最后的模板参数是可选的，默认值通常是使用的值。此参数指定一个 C++ 样式的分配器类，用于在主机上执行 Buffer 所需的任何内存分配。首先，我们将研究创建 Buffer 对象的多种方法。

7.1.1 Buffer 创建

在下图中，我们展示了创建 Buffer 对象的几种方法。让我们浏览一下示例并查看每个实例。

```
// Create a buffer of 2x5 ints using the default allocator
buffer<int, 2, buffer_allocator<int>> b1{range<2>{2, 5}};

// Create a buffer of 2x5 ints using the default allocator
// and CTAD for range
buffer<int, 2> b2{range{2, 5}};

// Create a buffer of 20 floats using a
// default-constructed std::allocator
buffer<float, 1, std::allocator<float>> b3{range{20}};

// Create a buffer of 20 floats using a passed-in
// allocator
std::allocator<float> myFloatAlloc;
buffer<float, 1, std::allocator<float>> b4{range(20),
                                             myFloatAlloc};
```

图 7.2: 创建 Buffer, 第一部分

我们在图 7-2 中创建的第一个 Buffer b1 是一个包含 10 个整数的二维 Buffer。我们显式传递所有模板参数，甚至显式传递 buffer_allocator<T> 的默认值作为分配器类型。由于 buffer_allocator 也是模板化类型，因此我们必须显式地专门化它，就像通过指定 buffer_allocator<int> 来专门化 Buffer 一样。然而，使用现代 C++，我们可以更简洁地表达这一点。Buffer b2 也是使用默认分配器的十个整数的二维 Buffer。这里我们利用 C++17 的类模板参数推导（CTAD）来自动推断模板参数。CTAD 是一个全有或全

无的工具，它必须要么推断出类的每个模板参数，要么不推断出其中任何一个。在本例中，我们利用这样一个事实：我们用一个带有两个参数的范围来初始化 b2 来推断它是一个二维范围。分配器模板参数有一个默认值，因此我们在创建 Buffer 时不需要显式列出它。

使用 Buffer b3，我们创建一个 20 个浮点数的 Buffer，并使用默认构造的 std::allocator 来分配主机上任何必要的内存。当将自定义分配器类型与 Buffer 一起使用时，我们通常希望将实际的分配器对象传递给 Buffer 来使用，而不是默认构造的分配器对象。Buffer b4 展示了如何执行此操作，在调用其构造函数的范围之后获取分配器对象。

对于示例中的前四个 Buffer，我们让 Buffer 分配它需要的任何内存，并且在创建数据时不使用任何值初始化该数据。使用 Buffer 来有效包装现有的 C++ 分配是一种常见模式，这些分配可能已经使用数据进行了初始化。我们可以通过将初始值源传递给 Buffer 构造函数来做到这一点。这样做可以让我们做几件事，我们将在下一个示例中看到。

```
// Create a buffer of 4 doubles and initialize it from a
// host pointer
double myDoubles[4] = {1.1, 2.2, 3.3, 4.4};
buffer b5{myDoubles, range{4}};

// Create a buffer of 5 doubles and initialize it from a
// host pointer to const double
const double myConstDbls[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
buffer b6{myConstDbls, range{5}};

// Create a buffer from a shared pointer to int
auto sharedPtr = std::make_shared<int>(42);
buffer b7{sharedPtr, range{1}};
```

图 7.3: 创建 Buffer, 第二部分

在图 7-3 中，Buffer b5 创建一个包含四个双精度数的一维 Buffer。除了指定 Buffer 大小的范围之外，我们还将指向 C 数组 myDoubles 的主机指针传递给 Buffer 构造函数。这里我们可以充分利用 CTAD 来推断 Buffer 的所有模板参数。我们传递的主机指针指向双精度数，它为我们提供了 Buffer 的数据类型。维数是从一维范围自动推断的，一维范围本身是推断出来的，因为它仅由一个数字创建。最后，使用默认分配器，因此我们不必指定它。

传递主机指针有一些我们应该注意的后果。通过传递指向主机内存的

指针，我们向运行时保证在 Buffer 的生命周期内不会尝试访问主机内存。这不是（也不能）由 SYCL 实施强制执行。我们有责任确保我们不违反本合同。我们不应该在 Buffer 处于活动状态时尝试访问此内存的原因之一是，Buffer 可能会选择使用主机上的不同内存来表示 Buffer 内容，这通常是出于优化原因。如果这样做，这些值将从主机指针复制到这个新内存中。如果后续 Kernel 修改 Buffer，则原始主机指针将不会反映更新的值，直到某些指定的同步点。我们将在本章后面详细讨论数据何时写回到主机指针。

Buffer b6 与 Buffer b5 非常相似，但有一个主要区别。这次，我们使用指向 const double 的指针来初始化 Buffer。这意味着我们只能通过主机指针读取值而不能写入它们。然而，本例中 Buffer 的类型仍然是 double，而不是 const double，因为推导指南没有考虑常量性。这意味着 Buffer 可以由 Kernel 写入，但在 Buffer 过期后我们必须使用不同的机制来更新主机（本章稍后将介绍）。

Buffer 也可以使用 C++ 共享指针对象进行初始化。如果我们的应用程序已经使用共享指针，这非常有用，因为这种初始化方法将正确计算引用并确保内存不会被释放。Buffer b7 创建一个包含单个整数的 Buffer，并使用共享指针对其进行初始化。

```
// Create a buffer of ints from an input iterator
std::vector<int> myVec;
buffer b8{myVec.begin(), myVec.end()};
buffer b9{myVec};

// Create a buffer of 2x5 ints and 2 non-overlapping
// sub-buffers of 5 ints.
buffer<int, 2> b10{range{2, 5}};
buffer b11{b10, id{0, 0}, range{1, 5}};
buffer b12{b10, id{1, 0}, range{1, 5}};
```

图 7.4: 创建 Buffer, 第三部分

容器通常用于现代 C++ 应用程序，示例包括 std::array、std::vector、std::list 或 std::map。我们可以通过两种不同的方式使用容器初始化一维 Buffer。第一种方式，如图 7-4 中的 Buffer b8 所示，使用输入迭代器。我们将两个迭代器传递给 Buffer 构造函数，而不是主机指针，一个代表数据的开头，另一个代表数据的结尾。Buffer 的大小计算为通过递增起始迭代器

直到等于结束迭代器而返回的元素数。这对于实现 C++ InputIterator 接口的任何数据类型都很有用。如果为 Buffer 提供初始值的容器对象也是连续的，那么我们可以使用更简单的形式来创建 Buffer。Buffer b9 只需将向量传递给构造函数即可从向量创建 Buffer。Buffer 的大小由用于初始化它的容器的大小决定，Buffer 数据的类型来自容器数据的类型。使用这种方法创建 Buffer 很常见，并且建议在 std::vector 和 std::array 等容器中使用这种方法。

Buffer 创建的最后一个示例说明了 Buffer 类的另一个功能。可以创建一个子 Buffer，它是一个 Buffer 的另一个 Buffer 的视图。子 Buffer 需要三件事：对父 Buffer 的引用、基本索引和子 Buffer 的范围。无法从子 Buffer 创建子 Buffer。可以从同一个 Buffer 创建多个子 Buffer，并且它们可以自由重叠。Buffer b10 的创建方式与 Buffer b2 完全相同，Buffer b2 是一个二维整数 Buffer，每行有 5 个整数。接下来，我们从 Buffer b10 创建两个子 Buffer，子 Buffer b11 和 b12。子 Buffer b11 从索引 (0,0) 开始，包含第一行中的每个元素。类似地，子 Buffer b12 从索引 (1,0) 开始，包含第二行中的每个元素。这会产生两个不相交的子 Buffer。由于子 Buffer 不重叠，因此不同的 Kernel 可以同时对不同的子 Buffer 进行操作，但我们在下一章中详细讨论调度执行图和依赖关系。

7.1.2 Buffer 属性

```

queue q;
int my_ints[42];

// Create a buffer of 42 ints
buffer<int> b{range(42)};

// Create a buffer of 42 ints, initialize with a host
// pointer, and add the use_host_pointer property
buffer b1{my_ints,
           range(42),
           {property::buffer::use_host_ptr{}));

// Create a buffer of 42 ints, initialize with a host
// pointer, and add the use_mutex property
std::mutex myMutex;
buffer b2{my_ints,
           range(42),
           {property::buffer::use_mutex{myMutex}}};
// Retrieve a pointer to the mutex used by this buffer
auto mutexPtr =
    b2.get_property<property::buffer::use_mutex>()
    .get_mutex_ptr();
// Lock the mutex until we exit scope
std::lock_guard<std::mutex> guard{*mutexPtr};

// Create a context-bound buffer of 42 ints, initialized
// from a host pointer
buffer b3{
    my_ints,
    range(42),
    {property::buffer::context_bound{q.get_context()}}};

```

图 7.5: *Buffer* 属性

还可以使用改变其行为的特殊属性来创建 Buffer。在图 7-5 中，我们将通过一个示例介绍三个不同的可选 Buffer 属性，并讨论如何使用它们。请注意，这些属性在大多数代码中相对不常见。

use_host_ptr 在 Buffer 创建期间可以选择指定的第一个属性是 use_host_ptr。如果存在，此属性要求 Buffer 不在主机上分配任何内存，并且在 Buffer 构造中传递或指定的任何分配器都将被有效忽略。相反，Buffer 必须使用传递给构造函数的主机指针指向的内存。请注意，这并不要求设备使用相同的内

存来保存 Buffer 的数据。设备可以自由地将 Buffer 的内容缓存在其附加内存中。另请注意，仅当主机指针传递给构造函数时才可以使用此属性。当程序想要完全控制所有主机内存分配时，此选项非常有用，例如，它允许程序员尝试最小化应用程序的内存占用量。

在图 7-5 的示例中，我们创建了一个 Buffer b，正如我们在前面的示例中看到的那样。接下来我们创建 Buffer b1 并使用指向 myInts 的指针对其进行初始化。我们还传递了 use_host_ptr 属性，这意味着 Buffer b1 将仅使用 myInts 指向的内存，而不会在主机上分配任何额外的临时存储。

use_mutex 下一个属性 use_mutex 涉及 Buffer 和主机代码之间的细粒度内存共享。Buffer b2 是使用此属性创建的。该属性获取对互斥对象的引用，稍后可以从 Buffer 中查询该对象，如示例中所示。此属性还需要将主机指针传递给构造函数，它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更新值。在运行时保证主机指针看到 Buffer 的最新值之前，我们无法锁定互斥体。虽然这可以与 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 Buffer 仍处于活动状态时访问 Buffer 内的数据，并且无需使用主机访问器机制（稍后介绍）。一般来说，除非我们有特定原因使用互斥体，否则应首选主机访问器机制，特别是因为无法保证成功锁定互斥体以及数据可供主机代码使用之前需要多长时间。

context_bound 在我们的示例中，最终属性显示在 Buffer b3 的创建中。在这里，我们的 42 个整数的 Buffer 是使用 context_bound 属性创建的。该属性采用对上下文对象的引用。通常，Buffer 可以在任何设备或上下文上自由使用。但是，如果使用此属性，则会将 Buffer 锁定到指定的上下文。尝试在另一个上下文中使用 Buffer 将导致运行时错误。例如，这可以通过识别 Kernel 可能被提交到错误队列的情况来调试程序。实际上，我们并不期望在许多程序中看到此属性，并且在任何上下文中的任何设备上访问 Buffer 的能力是 Buffer 抽象最强大的属性之一（此属性撤销了这一点）。

7.1.3 我们可以用 Buffer 做什么？

使用 Buffer 对象可以完成很多事情。我们可以查询 Buffer 的特征，确定 Buffer 被破坏后是否以及在何处将任何数据写回主机内存，或者将 Buffer 重

新解释为具有不同特征的 Buffer。然而，不能做的一件事是直接访问 Buffer 表示的数据。相反，我们必须创建访问器对象来访问数据，我们将在本章后面学习所有这些内容。

可以查询 Buffer 的示例包括 Buffer 的范围、它表示的数据元素的总数以及存储其元素所需的字节数。我们还可以查询 Buffer 正在使用哪个分配器对象以及该 Buffer 是否是子 Buffer。

当 Buffer 被破坏时更新主机内存是使用 Buffer 时需要考虑的一个重要方面。根据 Buffer 的创建方式，主机内存可能会也可能不会在 Buffer 销毁后使用计算结果进行更新。如果创建 Buffer 并从指向非常量数据的主机指针进行初始化，则当 Buffer 被销毁时，同一指针将使用最新数据进行更新。然而，还有一种方法可以更新主机内存，无论 Buffer 是如何创建的。`set_final_data` 方法是 buffer 的模板方法，它可以接受原始指针、C++ `OutputIterator` 或 `std::weak_ptr`。当 Buffer 被破坏时，Buffer 包含的数据将使用提供的位置写入主机。请注意，如果 Buffer 是从主机指针创建并初始化为非常量数据，则就像使用该指针调用 `set_final_data` 一样。从技术上讲，原始指针是 `OutputIterator` 的特殊情况。如果传递给 `set_final_data` 的参数是 `std::weak_ptr`，则当指针已过期或已被删除时，数据不会写入主机。是否发生回写也可以通过 `set_write_back` 方法来控制。

7.2 访问器

Buffer 表示的数据不能通过 Buffer 对象直接访问。相反，我们必须创建允许我们安全访问 Buffer 数据的访问器对象。访问器通知运行时我们要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位于正确的位置。这是一个非常强大的概念，特别是与部分基于数据依赖性来调度 Kernel 执行的任务图结合使用时。

访问器对象是从模板化访问器类实例化的。该类有五个模板参数。第一个参数是正在访问的数据的类型。这应该与相应 Buffer 存储的数据类型相同。同样，第二个参数描述数据和 Buffer 的维度，默认值为 1。

Mode	Description
read	Read-only access
write	Write-only access preserving previous contents
read_write	Read and write access

图 7.6: 访问模式

接下来的三个模板参数对于访问器来说是唯一的。第一个是访问模式。访问模式描述了我们打算如何在程序中使用访问器。图 7-6 列出了可能的模式。我们将在第 8 章中了解如何使用这些模式来命令 Kernel 的执行和执行数据移动。如果没有指定或自动推断，访问模式参数确实有一个默认值。如果我们没有另外指定，访问器将默认对非 `const` 数据类型使用 `read_write` 访问模式，对 `const` 数据类型默认使用 `read` 访问模式。这些默认值始终是正确的，但提供更准确的信息可能会提高运行时执行优化的能力。当开始应用程序开发时，简单地不指定访问模式是安全且简洁的，然后我们可以根据应用程序的性能关键区域的分析来细化访问模式。

Target	Description
device	Access a buffer via device global memory
host_task	Access a buffer from a host task

图 7.7: 访问目标

下一个模板参数是访问目标。Buffer 是数据的抽象，不描述数据的存储位置和方式。访问目标描述了我们正在访问数据的位置。图 7-7 列出了两个可能的访问目标。

当使用带有 SYCL 的 C++ 时，只有两个目标：`device` 和 `host_task`。默认模板值为 `device`，这意味着我们打算访问设备上的 Buffer 数据。这是合理的，因为访问器最常用于设备上的操作，例如 Kernel 或数据传输。另一个访问目标是 `host_task`，当主机任务需要访问 Buffer 的数据时使用它。

设备可能有不同类型的可用存储器。特别是，许多设备都具有某种快速本地内存，可以在工作组中的多个工作项之间共享。SYCL 的早期版本对本地内存有特殊的访问目标，但 SYCL 2020 以不同的方式处理它。我们将在第 9 章中学习如何使用工作组本地内存。早期版本的 SYCL 还为主机提供了

特殊的访问目标（在主机任务之外，这是 SYCL 2020 的新增功能）。它已被新的 `host_accessor` 类取代，该类提供对主机代码中 Buffer 数据的访问。但是，访问将在 `host_accessor` 的生命周期内保持有效。鉴于当 `host_accessor` 有效时 Buffer 被锁定到主机，因此应特别注意限制 `host_accessor` 对象的范围。

最终的模板参数控制访问器是否是占位符访问器。这不是程序员可能直接设置的参数，通常是通过使用构造函数调用来创建访问器来推断的。占位符访问器是在命令组外部声明的访问器，但旨在用于访问 Kernel 内设备上的数据。一旦我们查看了访问器创建的示例，我们就会明白占位符访问器与非占位符访问器的区别。

虽然可以使用其 `get_access` 方法从 Buffer 对象中提取访问器，但直接创建（构造）它们更简单。这是我们将在接下来的示例中使用的样式，因为它非常易于理解并且紧凑。

7.2.1 访问器创建

```

#include <cassert>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;
    // Create 3 buffers of 42 ints
    buffer<int> a_buf{range{N}};
    buffer<int> b_buf{range{N}};
    buffer<int> c_buf{range{N}};
    accessor pc{c_buf};

    q.submit([&](handler &h) {
        accessor a{a_buf, h};
        accessor b{b_buf, h};
        accessor c{c_buf, h};
        h.parallel_for(N, [=](id<1> i) {
            a[i] = 1;
            b[i] = 40;
            c[i] = 0;
        });
    });
    q.submit([&](handler &h) {
        accessor a{a_buf, h};
        accessor b{b_buf, h};
        accessor c{c_buf, h};
        h.parallel_for(N,
                      [=](id<1> i) { c[i] += a[i] + b[i]; });
    });
    q.submit([&](handler &h) {
        h.require(pc);
        h.parallel_for(N, [=](id<1> i) { pc[i]++; });
    });

    host_accessor result{c_buf};
    for (int i = 0; i < N; i++) {
        assert(result[i] == N);
    }
    return 0;
}

```

图 7.8: 创建简单的访问器

图 7-8 显示了一个示例程序，其中包含我们开始使用访问器所需的一切。在此示例中，我们有三个 Buffer A、B 和 C。我们提交到队列的第一

个并行任务为每个 Buffer 创建访问器，并定义一个 Kernel，该 Kernel 使用这些访问器用一些值初始化 Buffer。每个访问器都是通过对其将访问的 Buffer 的引用以及我们提交到队列的命令组定义的处理程序对象来构造的。这有效地将访问器绑定到我们作为命令组的一部分提交的 Kernel。常规访问器是设备访问器，因为默认情况下它们的目标是存储在设备内存中的全局 Buffer。这是最常见的用例。

我们提交的第二个任务还定义了三个 Buffer 访问器。然后，我们使用第二个 Kernel 中的这些访问器将 Buffer A 和 B 的元素添加到 Buffer C 中。由于第二个任务操作的数据与第一个任务相同，因此运行时将在第一个任务完成后执行此任务。我们将在下一章详细了解这一点。

第三个任务展示了如何使用占位符访问器。在我们创建 Buffer 之后，访问器 pC 在图 7-8 示例的开头声明。请注意，构造函数不会传递处理程序对象，因为我们没有要传递的处理程序对象。这让我们可以提前创建一个可重用的访问器对象。但是，为了在 Kernel 中使用此访问器，我们需要在提交期间将其绑定到命令组。我们使用处理程序对象的 require 方法来完成此操作。一旦我们将占位符访问器绑定到命令组，我们就可以像使用任何其他访问器一样在 Kernel 中使用它。

最后，我们创建一个 host_accessor 对象，以便在主机上读回计算结果。请注意，这与我们在 Kernel 中使用的类型不同。请注意，此示例中的主机访问器结果也不采用处理程序对象，因为我们再次没有要传递的处理程序对象。主机访问器的特殊类型还可以让我们消除它们与占位符的歧义。主机访问器的一个重要方面是，构造函数仅在数据可在主机上使用时完成，这意味着主机访问器的构造可能会花费很长时间。构造函数必须等待任何生成要复制的数据的 Kernel 完成执行以及复制本身完成。一旦主机访问器构建完成，就可以安全地使用它直接在主机上访问的数据，并且我们可以保证在主机上可以使用最新版本的数据。

虽然这个例子是完全正确的，但我们在创建访问器时并没有说明我们打算如何使用它们。相反，我们对 Buffer 中的非常量 int 数据使用默认访问模式，即 read_write。这可能过于保守，并且可能会在操作之间产生不必要的依赖性或多余的数据移动。如果运行时有更多关于我们计划如何使用我们创建的访问器的信息，它可能会做得更好。然而，在我们讨论执行此操作的示例之前，我们应该首先介绍另一个工具 - 推导标签。

Tag value	access_mode::	target::
read_only	read	device
read_write	read_write	device
write_only	write	device
read_only_host_task	read	host_task
read_write_host_task	read_write	host_task
write_only_host_task	write	host_task

图 7.9: 推导标签

推导标签是一种表达访问者所需的访问模式和目标组合的紧凑方式。使用推导标签时，它会作为参数传递给访问器的构造函数。可能的标签如图 7-9 所示。当使用标签参数构造访问器时，C++ CTAD 可以正确推断出所需的访问模式和目标，从而提供一种简单的方法来覆盖这些模板参数的默认值。我们还可以手动指定所需的模板参数，但标签提供了一种更简单、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。

```

#include <cassert>
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    // Create 3 buffers of 42 ints
    buffer<int> buf_a{range{N}};
    buffer<int> buf_b{range{N}};
    buffer<int> buf_c{range{N}};

    accessor pc{buf_c};

    q.submit([&](handler &h) {
        accessor a{buf_a, h, write_only, no_init};
        accessor b{buf_b, h, write_only, no_init};
        accessor c{buf_c, h, write_only, no_init};
        h.parallel_for(N, [=](id<1> i) {
            a[i] = 1;
            b[i] = 40;
            c[i] = 0;
        });
    });
    q.submit([&](handler &h) {
        accessor a{buf_a, h, read_only};
        accessor b{buf_b, h, read_only};
        accessor c{buf_c, h, read_write};
        h.parallel_for(N,
                      [=](id<1> i) { c[i] += a[i] + b[i]; });
    });
    q.submit([&](handler &h) {
        h.require(pc);
        h.parallel_for(N, [=](id<1> i) { pc[i]++; });
    });

    host_accessor result{buf_c, read_only};

    for (int i = 0; i < N; i++) {
        assert(result[i] == N);
    }
    return 0;
}

```

图 7.10: 使用指定用法创建访问器

让我们以前面的示例为例，重写它以添加扣除标签。这个新的改进示例

如图 7-10 所示。

我们首先声明 Buffer，如图 7-8 所示。我们还创建了稍后将使用的占位符访问器。现在让我们看看提交到队列的第一个任务。之前，我们通过传递对 Buffer 的引用和命令组的处理程序对象来创建访问器。现在，我们向构造函数调用添加两个额外的参数。第一个新参数是扣除标签。由于该 Kernel 正在为 Buffer 写入初始值，因此我们使用 write_only 推导标签。这让运行时知道该 Kernel 正在生成新数据并且不会从 Buffer 读取。

第二个新参数是一个可选的访问器属性，类似于我们在本章前面看到的 Buffer 的可选属性。我们传递的属性 no_init 让运行时知道可以丢弃 Buffer 的先前内容。这很有用，因为它可以让运行时消除不必要的数据移动。在此示例中，由于第一个任务是写入 Buffer 的初始值，因此运行时无需在 Kernel 执行之前将未初始化的主机内存复制到设备。no_init 属性对于本例很有用，但不应该用于读取-修改-写入的情况或仅更新 Buffer 中的某些值的 Kernel。

我们提交到队列的第二个任务与之前相同，但现在我们向访问器添加推导标签。在这里，我们将标签 read_only 添加到访问器 aA 和 aB 中，让运行时知道我们只会通过这些访问器读取 Buffer A 和 B 的值。第三个访问器 aC 获得 read_write 推导标签，因为我们将 A 和 B 的元素之和累加到 C 中。我们在示例中显式使用该标签以保持一致，但这是不必要的，因为默认访问模式是 read_write。

在我们使用占位符访问器的第三个任务中保留了默认用法。这与我们在图 7-8 中看到的简化示例保持不变。我们的最终访问器（主机访问器结果）现在在创建时会收到一个推导标签。由于我们只读取主机上的最终值，因此我们将 read_only 标记传递给构造函数。如果我们以破坏主机访问器的方式重写程序，则启动另一个对 Buffer C 进行操作的 Kernel 将不需要将其写回设备，因为 read_only 标记让运行时知道它不会被主人。

7.2.2 我们可以用访问器做什么？

使用访问器对象可以完成许多事情。然而，我们能做的最重要的事情是在访问者的名字中拼写出来——访问数据。这通常是通过访问器的 [] 运算符之一完成的。我们在图 7-8 和 7-10 的示例中使用 [] 运算符。该运算符采用可以正确索引多维数据的 id 对象或单个 size_t。当访问器具有多个维度时，可以使用第二种情况。在这种情况下，它返回一个对象，然后用 [] 再次索引该对象，直到我们到达标量值，在二维情况下，该对象的形式为 a[i][j]。

请记住，访问器维度的排序遵循 C++ 的约定，其中最右边的维度是单位步幅维度（迭代“最快”）。

访问器还可以返回指向基础数据的指针。可以按照正常的 C++ 规则直接访问该指针。请注意，该指针的地址空间可能会涉及额外的复杂性。

许多东西也可以从访问器对象中查询。示例包括通过访问器可访问的元素数量、它所覆盖的 Buffer 区域的大小（以字节为单位）或可访问的数据范围。

访问器提供了与 C++ 容器类似的接口，并且可以在许多可以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`）以及多种向前和向后迭代器。

7.3 总结

在本章中，我们学习了 Buffer 和访问器。Buffer 是数据的抽象，它向程序员隐藏了内存管理的底层细节。他们这样做是为了提供更简单、更高层次的抽象。我们通过几个示例向我们展示了构建 Buffer 的不同方法以及可以指定以改变其行为的不同可选属性。我们学习了如何使用主机内存中的数据初始化 Buffer，以及如何在使用完 Buffer 后将数据写回主机内存。

由于我们无法直接访问 Buffer，因此我们学习了如何使用访问器对象来访问 Buffer 中的数据。我们了解了设备访问器和主机访问器之间的区别。我们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将如何以及在何处使用访问器。我们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，并且我们学习了如何区分占位符访问器和非占位符访问器。然后，我们了解了如何通过向访问器声明添加推导标签来为运行时提供有关访问器使用情况的更多信息，从而进一步优化示例程序。最后，我们介绍了在程序中使用访问器的许多不同方式。

在下一章中，我们将更详细地了解运行时如何使用我们通过访问器提供的信息来调度不同 Kernel 的执行。我们还将看到此信息如何通知运行时何时以及如何需要在主机和设备之间复制 Buffer 中的数据。我们将学习如何显式控制涉及 Buffer 的数据移动以及 USM 分配。

8 调度 Kernel 和数据移动

我们需要讨论我们作为并行项目的指挥者的角色。并行程序的正确编排是一件美妙的事情——代码全速运行而无需等待数据，因为我们已经安排了所有数据在正确的时间到达和离开——精心构建的代码可以保持硬件最大程度地忙碌。这是梦想的组成部分！

快车道上的生活——不仅仅是一条车道！——要求我们认真对待我们作为指挥的工作。为了做到这一点，我们可以根据任务图来思考我们的工作。

因此，在本章中，我们将介绍任务图，这是用于正确有效地运行复杂的 Kernel 序列的机制。应用程序中有两件事需要排序：Kernel 执行和数据移动。任务图是我们用来实现正确排序的机制。

首先，我们将快速回顾一下第 3 章中如何使用依赖关系来排序任务。接下来，我们将介绍 SYCL 运行时如何构建图。我们将讨论 SYCL 图的基本构建块，即命令组。然后我们将说明构建常见模式图的不同方法。我们还将讨论如何在 Graph 中表示显式和隐式的数据移动。最后，我们将讨论将 Graph 与主机同步的各种方法。

8.1 什么是图调度？

在第 3 章中，我们讨论了数据管理和数据使用排序。该章描述了 SYCL 中图背后的关键抽象：依赖关系。Kernel 之间的依赖关系基本上基于 Kernel 访问的数据。Kernel 需要确保它读取了正确的数据，然后才能计算其输出。

我们描述了对于确保正确执行非常重要的三种类型的数据依赖性。第一种是写后读 (RAW)，当一个任务需要读取另一个任务生成的数据时发生。这种类型的依赖性描述了两个 Kernel 之间的数据流。当一个任务在另一个任务读取数据后需要更新数据时，就会发生第二种类型的依赖。我们将这种类型的依赖关系称为“读后写”(WAR) 依赖关系。当两个任务尝试写入相同的数据时，会出现最后一种类型的数据依赖。这称为写入后写入 (WAW) 依赖性。

数据依赖性是我们用来构建 Graph 的构建块。这组依赖关系是我们表达简单线性 Kernel 链和包含数百个具有复杂依赖关系的 Kernel 的大型复杂图所需的全部。无论计算需要哪种类型的图，SYCL 图都确保程序将根据所表达的依赖关系正确执行。然而，程序员需要确保 Graph 正确地表达程序中的所有依赖关系。

8.2 SYCL 中的 Graph 如何工作

命令组可以包含三个不同的内容：操作、其依赖项和其他主机代码。在这三件事中，始终需要的是行动，因为没有它，指挥组实际上什么也做不了。大多数命令组也会表达依赖性，但在某些情况下可能不会。一个这样的例子是程序中提交的第一个操作。它不依赖于任何东西来开始执行；因此，我们不会指定任何依赖性。命令组中可能出现的另一件事是在主机上执行的任意 C++ 代码。这是完全合法的，并且对于帮助指定操作或其依赖性很有用，并且此代码在创建命令组时执行（而不是稍后，当基于已满足的依赖性执行操作时）。

命令组通常表示为传递给提交方法的 C++ lambda 表达式。命令组还可以通过队列对象上的快捷方法来表达，该队列对象采用 Kernel 和基于事件的依赖集。

8.2.1 命令组行动

命令组可以执行两种类型的操作：Kernel 执行和显式内存操作。命令组只能执行单个操作。正如我们在前面的章节中所看到的，Kernel 是通过调用 parallel_for 或 single_task 方法来定义的，并表达我们想要在设备上执行的计算。显式数据移动的操作是第二种类型的操作。USM 的示例包括 memcpy、memset 和 fill 操作。Buffer 的示例包括复制、填充和 update_host。

8.2.2 命令组如何声明依赖关系

命令组的另一个主要组成部分是一组依赖关系，在执行该组定义的操作之前必须满足这些依赖关系。SYCL 允许以多种方式指定这些依赖性。

如果程序使用有序 SYCL 队列，则队列的有序语义指定连续排队的命令组之间的隐式依赖关系。在先前提交的任务完成之前，一项任务无法执行。

基于事件的依赖性是指定命令组执行之前必须完成的操作的另一种方法。这些基于事件的依赖性可以用两种样式来指定。当命令组被指定为传递给队列的提交方法的 lambda 时，使用第一种方法。在这种情况下，程序员调用命令组处理程序对象的 dependent_on 方法，传递事件或事件向量作为参数。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。当程序员直接在队列上调用 parallel_for 或 single_task 时，事件或事件向量

可以作为额外参数传递。

指定依赖关系的最后一种方法是通过创建访问器对象。访问器指定如何使用它们来读取或写入 Buffer 对象中的数据，让运行时使用此信息来确定不同 Kernel 之间存在的数据依赖性。

正如我们在本章开头所回顾的，数据依赖的示例包括一个 Kernel 读取另一个 Kernel 生成的数据、两个 Kernel 写入相同的数据，或者一个 Kernel 在另一个 Kernel 读取数据后修改数据。

8.2.3 例子

现在我们将通过几个例子来说明我们刚刚学到的一切。我们将展示如何以多种方式表达两种不同的依赖模式。我们将说明的两种模式是线性依赖链（其中一个任务在另一个任务之后执行）和“Y”模式（其中两个独立任务必须在连续任务之前执行）。

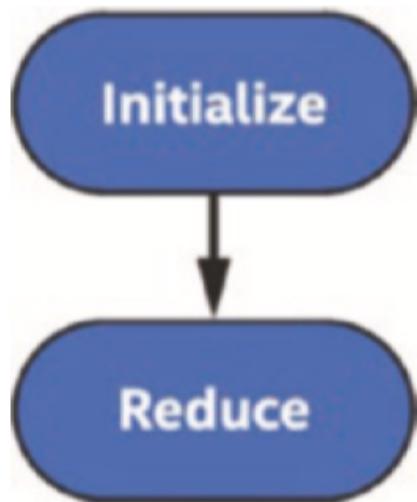


图 8.1: 线性依赖链图

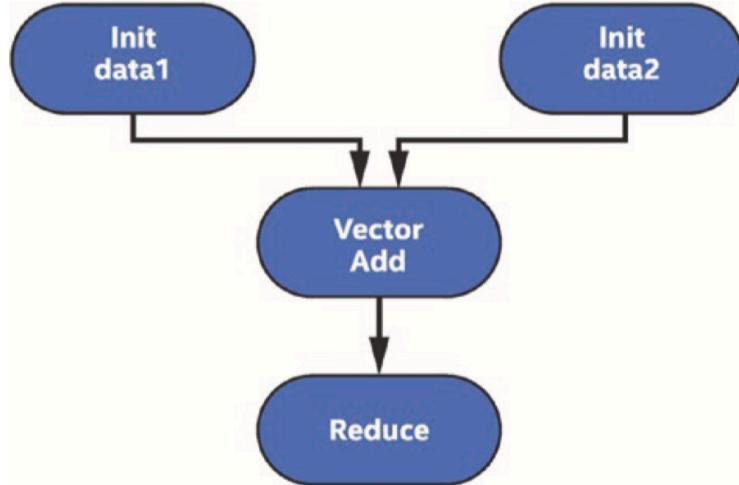


图 8.2: “Y”模式依赖关系图

这些依赖模式的 Graph 如图 8-1 和 8-2 所示。图 8-1 描述了一个线性依赖链。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积为单个结果的归约操作。图 8-2 描绘了一个“Y”模式，其中我们独立地初始化两个不同的数据。数据初始化后，加法 Kernel 会将两个向量相加。最后，图中的最后一个节点将结果累积为单个值。

对于每种模式，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关系。最后一个实现将使用 Buffer 和访问器来表达命令组之间的数据依赖性。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

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

    int *data = malloc_shared<int>(N, q);

    q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

    q.single_task([=]() {
        for (int i = 1; i < N; i++) data[0] += data[i];
    });
    q.wait();

    assert(data[0] == N);
    return 0;
}
```

图 8.3: 具有顺序队列的线性依赖链

图 8-3 显示了如何使用有序队列来表达线性依赖链。这个例子非常简单，因为中序队列的语义已经保证了命令组之间执行的顺序。我们提交的第一个 Kernel 将数组的元素初始化为 1。然后下一个 Kernel 获取这些元素并将它们相加到第一个元素中。由于我们的队列是有序的，因此我们不需要做任何其他事情来表示第二个 Kernel 在第一个 Kernel 完成之前不应执行。最后，我们等待队列完成执行所有任务，并检查是否获得了预期结果。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    int *data = malloc_shared<int>(N, q);

    auto e = q.parallel_for(N, [=](id<1> i) { data[i] = 1; });

    q.submit([&](handler &h) {
        h.depends_on(e);
        h.single_task([=]() {
            for (int i = 1; i < N; i++) data[0] += data[i];
        });
    });
    q.wait();

    assert(data[0] == N);
    return 0;
}
```

图 8.4: 事件的线性依赖链

图 8-4 显示了使用无序队列和基于事件的依赖关系的同一示例。在这里，我们捕获第一次调用 parallel_for 返回的事件。然后，第二个 Kernel 能够通过将其作为参数传递给 depends_on 来指定对该事件及其代表的 Kernel 执行的依赖。我们将在图 8-6 中看到如何使用定义 Kernel 的快捷方法之一来缩短第二个 Kernel 的表达式。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    buffer<int> data{range{N}};

    q.submit([&](handler &h) {
        accessor a{data, h};
        h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
    });

    q.submit([&](handler &h) {
        accessor a{data, h};
        h.single_task(=[]() {
            for (int i = 1; i < N; i++) a[0] += a[i];
        });
    });

    host_accessor h_a{data};
    assert(h_a[0] == N);
    return 0;
}
```

图 8.5: 具有 Buffer 和访问器的线性依赖链

图 8-5 使用 Buffer 和访问器而不是 USM 指针重写了我们的线性依赖链示例。在这里，我们再次使用无序队列，但使用通过访问器指定的数据依赖性而不是基于事件的依赖性来排序命令组的执行。第二个 Kernel 读取第一个 Kernel 生成的数据，运行时可以看到这一点，因为我们基于相同的底层 Buffer 对象声明访问器。与前面的示例不同，我们不会等待队列完成执行所有任务。相反，我们构建一个主机访问器，定义第二个 Kernel 的输出与我们在主机上计算出正确答案的断言之间的数据依赖关系。请注意，虽然主机访问器为我们提供了主机上数据的最新视图，但如果在创建 Buffer 时指定了任何内容，它并不能保证原始主机内存已更新。我们无法安全地访问原始主机内存，除非 Buffer 首先被破坏，或者除非我们使用更高级的机制，如第 7 章中描述的互斥机制。

```
#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

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

    int *data1 = malloc_shared<int>(N, q);
    int *data2 = malloc_shared<int>(N, q);

    q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

    q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

    q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

    q.single_task([=]() {
        for (int i = 1; i < N; i++) data1[0] += data1[i];

        data1[0] /= 3;
    });
    q.wait();

    assert(data1[0] == N);
    return 0;
}
```

图 8.6: 具有顺序队列的“Y”模式

图 8-6 显示了如何使用有序队列表达“Y”模式。在此示例中，我们声明两个数组：data1 和 data2。然后，我们定义两个 Kernel，每个 Kernel 将初始化其中一个数组。这些 Kernel 彼此不依赖，但由于队列是有序的，因此 Kernel 必须一个接一个地执行。请注意，在此示例中交换这两个 Kernel 的顺序是完全合法的。第二个 Kernel 执行后，第三个 Kernel 将第二个数组的元素添加到第一个数组的元素中。最终 Kernel 对第一个数组的元素进行求和，以计算与我们在线性相关链示例中所做的相同结果。这个求和 Kernel 依赖于之前的 Kernel，但是这个线性链也被有序队列捕获。最后，我们等待所有 Kernel 完成并验证我们是否成功计算了幻数。

```

#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    int *data1 = malloc_shared<int>(N, q);
    int *data2 = malloc_shared<int>(N, q);

    auto e1 =
        q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

    auto e2 =
        q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

    auto e3 = q.parallel_for(
        range{N}, {e1, e2},
        [=](id<1> i) { data1[i] += data2[i]; });

    q.single_task(e3, [=]() {
        for (int i = 1; i < N; i++) data1[0] += data1[i];

        data1[0] /= 3;
    });
    q.wait();

    assert(data1[0] == N);
    return 0;
}

```

图 8.7: 带有事件的“Y”模式

图 8-7 显示了我们的“Y”模式示例，其中使用无序队列而不是有序队列。由于队列的顺序导致依赖关系不再是隐式的，因此我们必须使用事件显式指定命令组之间的依赖关系。如图 8-6 所示，我们首先定义两个没有初始依赖性的独立 Kernel。我们用两个事件 e1 和 e2 来表示这些 Kernel。当我们定义第三个 Kernel 时，我们必须指定它依赖于前两个 Kernel。我们通过说它取决于事件 e1 和 e2 在执行之前完成来做到这一点。然而，在这个例子中，我们使用快捷形式来指定这些依赖关系，而不是处理程序的 depends_on 方法。在这里，我们将事件作为额外参数传递给 parallel_for。由于我们想要一次传递多个事件，因此我们使用接受事件 std::vector 的形式，但幸运的是，现代 C++ 通过自动将表达式 e1, e2 转换为适当的向量，为我们简化了这一过程。

```

#include <sycl/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue q;

    buffer<int> data1{range{N}};
    buffer<int> data2{range{N}};

    q.submit([&](handler &h) {
        accessor a{data1, h};
        h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
    });

    q.submit([&](handler &h) {
        accessor b{data2, h};
        h.parallel_for(N, [=](id<1> i) { b[i] = 2; });
    });

    q.submit([&](handler &h) {
        accessor a{data1, h};
        accessor b{data2, h, read_only};
        h.parallel_for(N, [=](id<1> i) { a[i] += b[i]; });
    });

    q.submit([&](handler &h) {
        accessor a{data1, h};
        h.single_task( [=]() {
            for (int i = 1; i < N; i++) a[0] += a[i];

            a[0] /= 3;
        });
    });

    host_accessor h_a{data1};
    assert(h_a[0] == N);
    return 0;
}

```

图 8.8: 带访问器的“Y”型模式

在我们的最后一个示例中，如图 8-8 所示，我们再次用 Buffer 和访问器替换 USM 指针和事件。此示例将两个数组 data1 和 data2 表示为 Buffer 对象。我们的 Kernel 不再使用快捷方法来定义 Kernel，因为我们必须将访问器与命令组处理程序关联起来。再一次，第三个 Kernel 必须捕获对前

两个 Kernel 的依赖。这里这是通过声明 Buffer 的访问器来完成的。由于我们之前已经声明了这些 Buffer 的访问器，因此运行时能够正确排序这些 Kernel 的执行。此外，当我们声明访问器 b 时，我们还向运行时提供额外的信息。我们添加访问标记 `read_only` 让运行时知道我们只会读取这些数据，不会生成新值。正如我们在线性依赖链的 Buffer 和访问器示例中看到的那样，我们的最终 Kernel 通过更新第三个 Kernel 中生成的值来对自身进行排序。我们通过声明一个主机访问器来检索计算的最终值，该主机访问器将等待最终 Kernel 完成执行，然后将数据移回主机，在主机上我们可以读取数据并断言我们计算了正确的结果。

8.2.4 命令组的各个部分何时执行?

由于任务图是异步的，因此有必要了解命令组的确切执行时间。到目前为止，应该清楚的是，一旦满足了 Kernel 的依赖性，就可以执行 Kernel，但是命令组的主机部分会发生什么情况呢？

当命令组提交到队列时，它会立即在主机上执行（在提交调用返回之前）。命令组的该主机部分仅执行一次。命令组中定义的任何 Kernel 或显式数据操作都会排队在设备上执行。

8.3 数据移动

数据移动是 SYCL 中 Graph 的另一个非常重要的方面，这对于理解应用程序性能至关重要。但是，如果数据移动在程序中隐式发生（使用 Buffer 和访问器或使用 USM 共享分配），则通常会意外地忽略这一点。接下来，我们将研究数据移动影响 SYCL 中 Graph 执行的不同方式。

8.3.1 显式数据移动

显式数据移动的优点是它显式地出现在 Graph 中，使程序员可以清楚地了解 Graph 执行过程中发生的情况。我们将显式数据操作分为 USM 和 Buffer 的操作。

正如我们在第 6 章中了解到的，当我们需要在设备分配和主机之间复制数据时，USM 中会发生显式数据移动。这是通过队列和处理程序类中的 `memcpy` 方法完成的。提交操作或命令组会返回一个事件，该事件可用于与其他命令组一起订购副本。

通过调用命令组处理程序对象的 `copy` 或 `update_host` 方法，可以通过 Buffer 进行显式数据移动。

复制方法可用于在主机内存和设备上的访问器对象之间手动交换数据。这样做可以出于多种原因。一个简单的例子是对长时间运行的计算序列设置检查点。使用复制方法，数据可以以单向方式从设备写入任意主机存储器。如果使用 Buffer 完成此操作，则大多数情况（即 Buffer 不是使用 `use_host_ptr` 创建的）将需要首先将数据复制到主机，然后从 Buffer 的内存复制到所需的主机内存。

`update_host` 方法是一种非常特殊的复制形式。如果围绕主机指针创建 Buffer，则此方法会将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使用特殊 `use_mutex` 属性创建的 Buffer 同步，这会很有用。然而，这种用例在大多数程序中不太可能出现。

8.3.2 隐式数据移动

隐式数据移动可能会对 SYCL 中的命令组和任务图产生隐藏的后果。通过隐式数据移动，数据可以通过 SYCL 运行时或通过硬件和软件的某种组合在主机和设备之间复制。无论哪种情况，复制都会在没有用户明确输入的情况下发生。让我们再次分别看看 USM 和缓冲器的情况。

使用 USM，主机和共享分配会发生隐式数据移动。正如我们在第 6 章中了解到的，主机分配实际上并不移动数据，而是远程访问数据，并且共享分配可能会在主机和设备之间迁移。由于此迁移是自动发生的，因此 USM 隐式数据移动和命令组实际上无需考虑。然而，共享分配有一些细微差别值得记住。

预取操作的工作方式与 `memcpy` 类似，以便让运行时在 Kernel 尝试使用共享分配之前开始迁移它们。然而，与必须复制数据以确保正确结果的 `memcpy` 不同，预取通常被视为运行时的提示以提高性能，并且预取不会使内存中的指针值无效（就像复制到新地址范围时那样）。如果在 Kernel 开始执行之前预取尚未完成，程序仍将正确执行，并且许多代码可能选择使图中的命令组不依赖于预取操作，因为它们不是功能要求。

Buffer 也有一些细微差别。使用 Buffer 时，命令组必须为 Buffer 构造访问器，以指定如何使用数据。这些数据依赖性表达了不同命令组之间的顺序，并允许我们构建任务图。然而，带有 Buffer 的命令组有时还具有另一个目的：它们指定数据移动的要求。

访问器指定 Kernel 将读取或写入 Buffer。由此推论，数据也必须在设备上可用，如果不可用，则运行时必须在 Kernel 开始执行之前将其移动到那里。因此，SYCL 运行时必须跟踪 Buffer 当前版本所在的位置，以便可以安排数据移动操作。访问器创建有效地在图中创建了一个额外的隐藏节点。如果需要数据移动，运行时必须首先执行它。只有这样，提交的 Kernel 才能执行。

让我们再看一下图 8-8。在此示例中，我们的前两个 Kernel 将需要将 Buffer data1 和 data2 复制到设备；运行时隐式创建额外的 Graph 节点来执行数据移动。当第三个 Kernel 的命令组被提交时，这些 Buffer 很可能仍然在设备上，因此运行时不需要执行任何额外的数据移动。第四个 Kernel 的数据也可能不需要任何额外的数据移动，但主机访问器的创建需要运行时在访问器可供使用之前安排将 Buffer 数据 1 移动回主机。

8.4 与主机同步

我们要讨论的最后一个主题是如何与主机同步图执行。我们已经在整章中谈到了这一点，但现在我们将研究程序执行此操作的所有不同方式。

主机同步的第一种方法是我们在前面的许多示例中使用的方法：等待队列。队列对象有两个方法：wait 和 wait_and_throw，它们会阻止主机执行，直到提交到队列的每个命令组完成为止。这是一个非常简单的方法，可以处理许多常见情况。然而，值得指出的是，这种方法的粒度非常粗。如果需要更细粒度的同步（例如，可能提高性能），我们将讨论的其他方法之一可能更适合应用程序的需求。

主机同步的下一个方法是同步事件。这为队列同步提供了更大的灵活性，因为它允许应用程序仅在特定操作或命令组上同步。这是通过调用事件的 wait 方法或调用事件类的静态方法 wait 来完成的，该事件类可以接受事件向量。

我们已经看到图 8-5 和 8-8 中使用的下一个方法：主机访问器。主机访问器执行两个功能。首先，顾名思义，它们使数据可在主机上访问。其次，它们通过定义当前访问的图和主机之间的新依赖关系来同步设备和主机。这可确保复制回主机的数据具有 Graph 正在执行的计算的正确值。然而，我们再次注意到，如果 Buffer 是从现有主机内存构造的，则不能保证该原始内存包含更新的值。

请注意，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

会超过主机访问器的创建。同样，当主机访问器存在并保持其数据可用时，Buffer 不能在设备上使用。一种常见的模式是在附加 C++ 作用域内创建主机访问器，以便在不再需要主机访问器时释放数据。这是下一个主机同步方法的示例。

SYCL 中的某些对象在被销毁时具有特殊行为，并调用其析构函数。我们刚刚了解了主机访问器如何使数据保留在主机上直到它们被销毁。当 Buffer 和图像被破坏或离开范围时，它们也有特殊的行为。当 Buffer 被销毁时，它会等待所有使用该 Buffer 的命令组完成执行。一旦 Buffer 不再被任何 Kernel 或内存操作使用，运行时可能必须将数据复制回主机。如果使用主机指针初始化 Buffer 或将主机指针传递给方法 set_final_data，则会发生此复制。然后，运行时将复制回该 Buffer 的数据并在对象被销毁之前更新主机指针。

与主机同步的最后一个选项涉及第 7 章中首先描述的一个不常见的功能。回想一下，Buffer 对象的构造函数可以选择采用属性列表。创建 Buffer 时可以传递的有效属性之一是 use_mutex。当以这种方式创建 Buffer 时，它增加了 Buffer 拥有的内存可以与主机应用程序共享的要求。对该内存的访问由用于初始化 Buffer 的互斥体控制。当可以安全地访问与 Buffer 共享的内存时，主机能够获得互斥体上的锁。如果无法获得锁，用户可能需要排队内存移动操作来与主机同步数据。这种用途非常特殊，在大多数 DPC++ 应用程序中不太可能找到。

8.5 总结

在本章中，我们了解了图以及它们如何在 SYCL 中构建、调度和执行。我们详细介绍了指挥组的含义以及它们的功能。我们讨论了命令组中可以包含的三件事：依赖性、操作和其他主机代码。我们回顾了如何使用事件以及通过访问器描述的数据依赖性来指定任务之间的依赖性。我们了解到命令组中的单个操作可能是 Kernel 或显式内存操作，然后我们查看了几个示例，这些示例展示了构建常见执行图模式的不同方法。接下来，我们回顾了数据移动如何成为 SYCL 图的重要组成部分，并了解了它如何显式或隐式地出现在图中。最后，我们研究了将图的执行与主机同步的所有方法。

了解程序流程可以使我们了解在调试运行时失败时可以打印的调试信息类型。第 13 章“调试运行时故障”部分有一个表格，考虑到我们在本书中这一点所获得的知识，该表格会更有意义。然而，本书并不试图详细讨论这

些高级编译器转储。

希望这让您感觉自己像一位 Graph 专家，可以构建各种复杂程度的 Graph，从线性链到具有数百个节点以及复杂数据和任务依赖性的巨大 Graph！在下一章中，我们将开始深入研究有助于提高特定设备上应用程序性能的底层细节。

9 通讯与同步

在第 4 章中，我们讨论了使用基本数据并行 Kernel 或显式 ND 范围 Kernel 来表达并行性的方法。我们讨论了基本数据并行 Kernel 如何独立地将相同的操作应用于每条数据。我们还讨论了显式 ND 范围 Kernel 如何将执行范围划分为 Work-Items 的 Work-Groups。

在本章中，我们将重新审视如何在不断追求并行思考的过程中将问题分解为小块的问题。本章提供了有关显式 ND 范围 Kernel 的更多详细信息，并描述了如何使用 Work-Items 分组来提高某些类型算法的性能。我们将描述 Work-Items 组如何为并行工作的执行方式提供额外的保证，并且我们将介绍支持 Work-Items 分组的语言功能。在第 15、16 和 17 章中优化特定设备的程序以及在第 14 章中描述常见并行模式时，许多想法和概念非常重要。

9.1 Work-Groups 和 Work-Items

回想一下第 4 章，显式 ND 范围 Kernel 将 Work-Items 组织到 Work-Groups 中，并且同一 Work-Groups 中的所有 Work-Items 都有额外的调度保证。这个属性很重要，因为它意味着 Work-Groups 中的 Work-Items 可以合作解决问题。

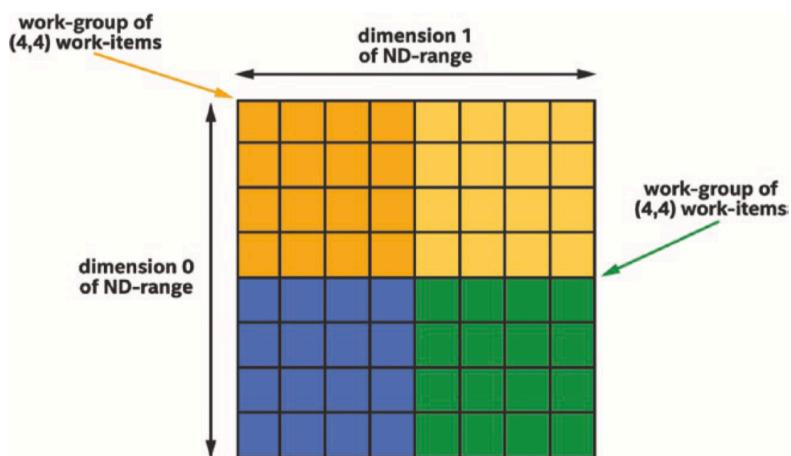


图 9.1: 大小 $(8, 8)$ 的二维 ND 范围分为四个大小为 $(4,4)$ 的 Work-Groups

图 9-1 显示了划分为 Work-Groups 的 ND 范围, 其中每个 Work-Groups 由不同的颜色表示。每个 Work-Groups 中的 Work-Items 可以安全地与共享相同颜色的其他 Work-Items 进行通信。

无法保证不同 Work-Groups 中的 Work-Items 会同时执行, 因此具有一种颜色的 Work-Items 无法与具有不同颜色的 Work-Items 可靠地通信。如果一个 Work-Items 尝试与当前未执行的另一 Work-Items 通信, 则 Kernel 可能会死锁。由于我们希望 Kernel 能够完成执行, 因此我们必须确保当一个 Work-Items 与另一 Work-Items 通信时, 它们位于同一个 Work-Groups 中。

9.2 高效通讯的基石

本节描述支持组中 Work-Items 之间高效通信的构建块。有些是基本构建块, 可以构建自定义算法, 而另一些则是更高级别的, 描述许多 Kernel 使用的常见操作。

9.2.1 通过屏障 (Barriers) 进行同步

通信最基本的组成部分是 Barrier 功能。Barrier 功能有两个主要目的:

首先, Barrier 功能同步组中 Work-Items 的执行。通过同步执行, 一个 Work-Items 可以确保同一组中的另一个 Work-Items 在使用该操作的结果之前已完成该操作。或者, 在另一个 Work-Items 使用操作结果之前, 给一个 Work-Items 时间来完成其操作。

其次, Barrier 函数同步每个 Work-Items 如何查看内存状态。这种类型的同步操作称为强制内存一致性或隔离内存 (更多详细信息请参阅第 19 章)。内存一致性至少与同步执行一样重要, 因为它确保在 Barrier 之前执行的内存操作的结果对于 Barrier 之后的其他 Work-Items 是可见的。如果没有内存一致性, 一个 Work-Items 中的操作就像森林中的一棵树倒下一样, 其他 Work-Items 可能会也可能听不到声音!

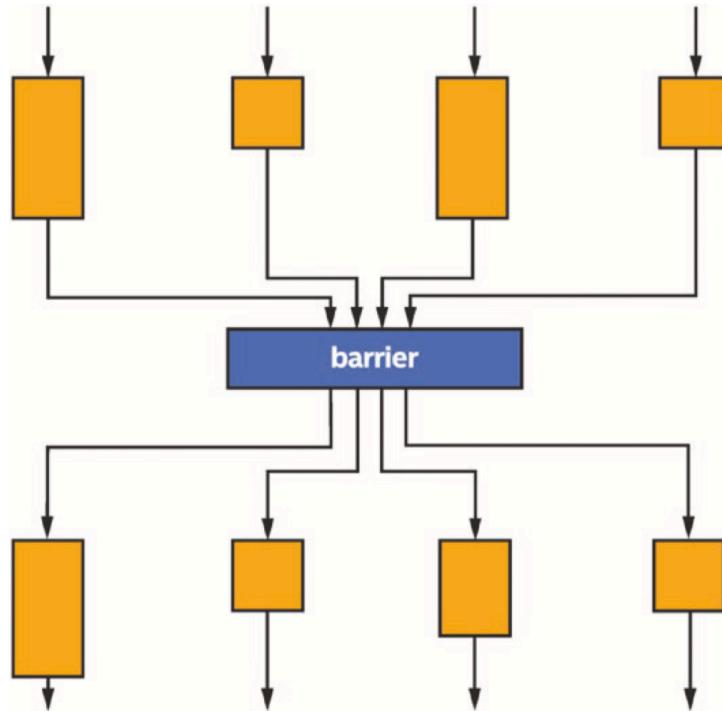


图 9.2: 组中的四个 *Work-Items* 在 *Barrier* 函数处同步

图 9-2 显示了一组在 Barrier 函数处同步的四个 Work-Items。尽管每个 Work-Items 的执行时间可能不同，但在所有 Work-Items 都执行 Barrier 之前，没有任何 Work-Items 可以执行越过 Barrier。执行 Barrier 函数后，所有 Work-Items 都具有一致的内存视图。

注 23 (为什么默认情况下内存不一致?) 对于许多程序员来说，内存一致性的概念（以及不同的 Work-Items 可以具有不同的内存视图）可能会让人感到非常奇怪。如果默认情况下所有 Work-Items 的所有内存都是一致的，那不是更容易吗？简短的回答是它会，但实施起来也会非常昂贵。通过允许 Work-Items 具有不一致的内存视图，并且只要求在程序执行期间定义的点具有内存一致性，加速器硬件可能更便宜，性能更好，或两者兼而有之。

因为 Barrier 函数同步执行，所以组中的所有 Work-Items 都执行 Barrier 或者组中没有 Work-Items 执行 Barrier 是至关重要的。如果组中的某

些 Work-Items 围绕任何 Barrier 函数分支，则组中的其他 Work-Items 可能会永远在 Barrier 处等待，或者至少直到用户放弃并终止程序！

注 24 (集体函数) 当某个函数需要由组中的所有 Work-Items 执行时，它可以称为集体函数，因为该操作是由组执行的，而不是由组中的单个 Work-Items 执行的。Barrier 函数并不是 SYCL 中唯一可用的集体函数。本章稍后将介绍其他集体函数。

9.2.2 Work-Groups 本地内存

Work-Groups Barrier 功能足以协调 Work-Groups 中 Work-Items 之间的通信，但通信本身必须通过内存进行。通信可以通过 USM 或 Buffer 进行，但这可能不方便且效率低下：它需要专门的通信分配，并且需要在 Work-Groups 之间划分分配。

为了简化 Kernel 开发并加速 Work-Groups 中 Work-Items 之间的通信，SYCL 定义了一个特殊的本地内存空间，专门用于 Work-Groups 中 Work-Items 之间的通信。

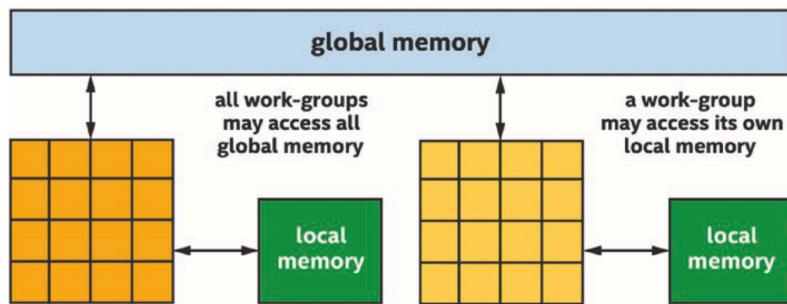


图 9.3: 每个 Work-Groups 可以访问所有全局内存，但只能访问自己的本地内存

图 9-3 显示了两个 Work-Groups。两个 Work-Groups 都可以访问 USM 和全局内存空间中的 Buffer。每个 Work-Groups 可以访问自己的本地内存空间中的变量，但不能访问另一个 Work-Groups 本地内存中的变量。

当 Work-Groups 开始时，其本地内存的内容未初始化，并且在 Work-Groups 执行完成后本地内存不会保留。由于这些属性，本地内存只能在 Work-Groups 执行时用于临时存储。

对于某些设备，例如对于许多 CPU 设备，本地存储器是软件抽象并且使用与全局存储器相同的存储器子系统来实现。在这些设备上，使用本地内存主要是一种方便的通信机制。某些编译器可能会使用内存空间信息进行编译器优化，但在其他情况下，使用本地内存进行通信从根本上来说不会比通过这些设备上的全局内存进行通信更好。

对于其他设备，例如许多 GPU 设备，有专用的本地内存资源。在这些设备上，通过本地内存进行通信比通过全局内存进行通信性能更好。

注 25 使用本地内存时，*Work-Groups* 中 *Work-Items* 之间的通信可以更方便、更快捷！

我们可以使用设备查询 `info::device::local_mem_type` 来确定加速器是否具有用于本地内存的专用资源，或者本地内存是否被实现为全局内存的软件抽象。有关查询设备属性的更多信息，请参阅第 12 章；有关 CPU、GPU 和 FPGA 的本地内存通常如何实现的更多信息，请参阅第 15、16 和 17 章。

9.3 使用 Work-Groups Barrier 和本地内存

现在我们已经确定了 *Work-Items* 之间有效通信的基本构建块，我们可以描述如何在 Kernel 中表达 *Work-Groups Barrier* 和本地内存。请记住，*Work-Items* 之间的通信需要 *Work-Items* 分组的概念，因此这些概念只能针对 ND 范围 Kernel 来表达，并且不包含在基本数据并行 Kernel 的执行模型中。

本章将通过介绍执行矩阵乘法的 *Work-Groups* 中的 *Work-Items* 之间的通信，建立在第 4 章中介绍的朴素矩阵乘法 Kernel 示例的基础上。在许多设备上（但不一定是所有设备！）通过本地内存进行通信将提高矩阵乘法 Kernel 的性能。

注 26（关于矩阵乘法的说明） 在本书中，矩阵乘法 Kernel 用于演示 Kernel 的变化如何影响性能。尽管使用本章中描述的技术可以在许多设备上提高矩阵乘法性能，但矩阵乘法是一项非常重要且常见的操作，因此许多供应商已经实现了矩阵乘法的高度调整版本。供应商投入大量时间和精力为特定设备实现和验证功能，在某些情况下，可能会使用在标准并行 Kernel 中难以或不可能使用的功能或技术。

注 27（使用供应商提供的库！） 当供应商提供函数的库实现时，使用它几乎总是有益的，而不是将函数重新实现为并行 Kernel！对于矩阵乘法，可

以将 *oneMKL* 视为英特尔工具包的一部分，以提供适合 *SYCL* 程序员的 *C++* 解决方案。

```
h.parallel_for(range{M, N}, [=](id<2> id) {
    int m = id[0];
    int n = id[1];

    // Template type T is the type of data stored
    // in the matrix
    T sum = 0;
    for (int k = 0; k < K; k++) {
        sum += matrixA[m][k] * matrixB[k][n];
    }

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

图 9.4: 第 4 章中的朴素矩阵乘法 Kernel

图 9-4 显示了我们将从中开始的朴素矩阵乘法 Kernel，类似于第 4 章中的矩阵乘法 Kernel。对于该 Kernel 以及本章中的所有矩阵乘法 Kernel，*T* 是一个模板类型，指示类型存储在矩阵中的数据的数量，例如 32 位浮点型或 64 位双精度型。

在第 4 章中，我们观察到矩阵乘法算法具有高度的重用性，并且对 Work-Items 进行分组可以提高访问的局部性，因此也可以提高缓存命中率。在本章中，我们修改后的矩阵乘法 Kernel 将不再依赖隐式缓存行为来提高性能，而是使用本地内存作为显式缓存，以保证访问的局部性。

注 28 对于许多算法，将本地内存视为显式缓存会很有帮助。

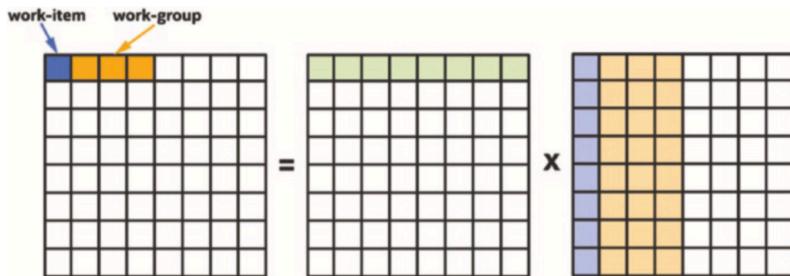
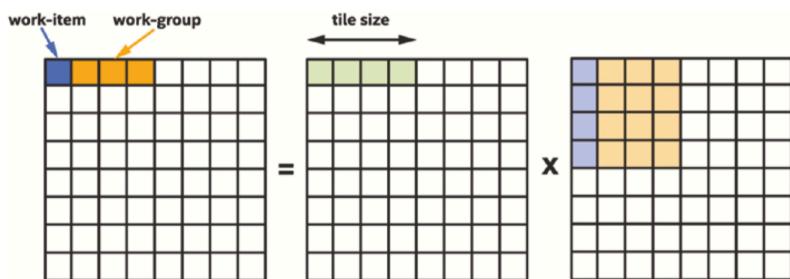


图 9.5: 矩阵乘法到 Work-Groups 和 Work-Items 的映射

图 9-5 是第 4 章的修改图，显示了由单行组成的 Work-Groups，这使得使用本地内存的算法更容易理解。观察到，对于结果矩阵的一行中的元素，每个结果元素都是使用来自输入矩阵之一的唯一数据列计算的（以蓝色和橙色显示）。由于该输入矩阵没有数据共享，因此它不是本地内存的理想候选者。但请注意，该行中的每个结果元素都访问另一个输入矩阵中完全相同的数据（以绿色显示）。由于这些数据被重用，因此它是从 Work-Groups 本地内存中受益的绝佳候选者。

因为我们想要乘以可能非常大的矩阵，并且 Work-Groups 本地内存可能是有限的资源，所以我们修改后的 Kernel 将处理每个矩阵的子部分，我们将其称为矩阵图块。对于每个图块，我们修改后的 Kernel 会将图块的数据加载到本地内存中，同步组中的 Work-Items，然后从本地内存而不是全局内存加载数据。第一个图块访问的数据如图 9-6 所示。

图 9.6: 处理第一个图块：绿色输入数据 (X 的左边) 被重用并从本地内存中读取，蓝色和橙色的输入数据 (X 的右边) 从全局内存中读取

在我们的 Kernel 中，我们选择了与 Work-Groups 大小相等的图块大小。

这不是必需的，但因为它简化了本地内存的传入或传出，所以选择 Work-Groups 大小的倍数的图块大小是常见且方便的。

9.3.1 ND 范围 Kernel 中的 Work-Groups Barrier 和本地内存

本节介绍如何在 ND 范围 Kernel 中表达 Work-Groups Barrier 和本地内存。对于 ND 范围 Kernel，表示是显式的：Kernel 声明本地访问器并对其进行操作，该本地访问器表示本地地址空间中的分配，并调用 Barrier 函数来同步 Work-Groups 中的 Work-Items。

本地访问器 要声明本地内存用于 ND 范围 Kernel，请使用本地访问器。与其他访问器对象一样，本地访问器是在命令组处理程序中构造的，但与第 3 章和第 7 章中讨论的访问器对象不同，本地访问器不是从 Buffer 对象创建的。相反，本地访问器是通过指定类型和描述该类型元素数量的范围来创建的。与其他访问器一样，本地访问器可以是一维、二维或三维的。图 9-7 演示了如何声明本地访问器并在 Kernel 中使用它们。

```

/ This is a typical global accessor.
accessor dataAcc{dataBuf, h};

// This is a 1D local accessor consisting of 16 ints:
auto localIntAcc = local_accessor<int, 1>(16, h);

// This is a 2D local accessor consisting of 4 x 4
// floats:
auto localFloatAcc =
    local_accessor<float, 2>({4, 4}, h);

h.parallel_for(
    nd_range<1>{{size}, {16}}, [=](nd_item<1> item) {
        auto index = item.get_global_id();
        auto local_index = item.get_local_id();

        // Within a kernel, a local accessor may be read
        // from and written to like any other accessor.
        localIntAcc[local_index] = dataAcc[index] + 1;
        dataAcc[index] = localIntAcc[local_index];
    });

```

图 9.7: 声明和使用本地访问器取

请记住，本地内存每个 Work-Groups 开始时未初始化，并且在每个 Work-Groups 完成后不会保留。这意味着本地访问器必须始终是 read_write，否则 Kernel 将无法分配本地内存的内容或查看分配的结果。不过，本地访问器可以选择是原子的，在这种情况下，通过访问器对本地存储器的访问是原子地执行的。第 19 章更详细地讨论了原子访问。

同步功能 要同步 ND 范围 KernelWork-Groups 中的 Work-Items，请使用代表 Work-Groups 的组调用 group_barrier 函数。由于代表 Work-Groups 的组只能从 nd_item 查询，而不能从 item 查询，因此 Work-Groups Barrier 仅适用于 ND 范围 Kernel，不适用于基本数据并行 Kernel。

group_barrier 函数接受一个附加的可选参数来描述 Barrier 执行的任何内存一致性操作的范围。当没有附加参数传递给 group_barrier 函数时，Barrier 函数将根据传入的组确定默认范围。默认范围通常是正确的，因此很少需要显式范围，但如果某些算法需要，可以扩大内存范围。

请注意，显式作用域仅影响由 Barrier 执行的内存操作，并且在 Barrier 处同步执行的 Work-Items 集完全由传递给 Barrier 的组对象确定。我们无法通过将不同的内存范围传递给 Barrier 来同步更多或更少的 Work-Items，但我们可以通过将不同的组对象传递给 Barrier 来同步一组不同的 Work-Items。

完整的 ND 范围 Kernel 示例 现在我们知道如何声明本地内存访问器并使用 Barrier 函数同步对其的访问，我们可以实现矩阵乘法的 ND 范围 Kernel 版本，它协调 Work-Groups 中 Work-Items 之间的通信，以减少全局流量内存。完整的示例如图 9-8 所示。

```

// Traditional accessors, representing matrices in
// global memory:
accessor matrixA{bufA, h};
accessor matrixB{bufB, h};
accessor matrixC{bufC, h};

// Local accessor, for one matrix tile:
constexpr int tile_size = 16;

// Template type T is the type of data stored in the matrix
auto tileA = local_accessor<T, 1>(tile_size, h);

h.parallel_for(
    nd_range<2>{{M, N}, {1, tile_size}},
    [=](nd_item<2> item) {
        // Indices in the global index space:
        int m = item.get_global_id()[0];
        int n = item.get_global_id()[1];

        // Index in the local index space:
        int i = item.get_local_id()[1];

        T sum = 0;
        for (int kk = 0; kk < K; kk += tile_size) {
            // Load the matrix tile from matrix A, and
            // synchronize to ensure all work-items have a
            // consistent view of the matrix tile in local
            // memory.
            tileA[i] = matrixA[m][kk + i];
            group_barrier(item.get_group());

            // Perform computation using the local memory
            // tile, and matrix B in global memory.
            for (int k = 0; k < tile_size; k++) {
                sum += tileA[k] * matrixB[kk + k][n];
            }

            // After computation, synchronize again, to
            // ensure all reads from the local memory tile
            // are complete.
            group_barrier(item.get_group());
        }

        // Write the final result to global memory.
        matrixC[m][n] = sum;
    });

```

图 9.8: 用 *ND* 范围 *parallel_for* 和 *Work-Groups* 本地存储器表达平铺矩阵乘法 Kernel

该 Kernel 中的主循环可以被认为有两个不同的阶段：在第一阶段，Work-Groups 中的 Work-Items 协作将共享数据从 A 矩阵加载到 Work-Groups 本地内存中；在第二个中，Work-Items 使用共享数据执行自己的计算。为了确保所有 Work-Items 在进入第二阶段之前都已完成第一阶段，这两个阶段通过调用 `group_barrier` 来分隔，以同步 Work-Groups 中的所有 Work-Items 并提供内存栅栏。这种模式是一种常见的模式，在 Kernel 中使用 Work-Groups 本地内存几乎总是需要使用 Work-Groups Barrier。

请注意，还必须调用 `group_barrier` 来同步当前图块的计算阶段和下一个矩阵图块的加载阶段之间的执行。如果没有这种同步操作，当前矩阵图块的一部分可能会在另一 Work-Items 完成计算之前被 Work-Groups 中的一个 Work-Items 覆盖。一般来说，每当一个 Work-Items 在本地存储器中读取或写入由另一 Work-Items 读取或写入的数据时，就需要同步。在图 9-8 中，同步是在循环结束时完成的，但在每个循环迭代开始时进行同步也同样正确。

9.4 Sub-Groups

到目前为止，在本章中，Work-Items 已通过 Work-Groups 本地内存交换数据并使用 Work-Groups 上的 `group_barrier` 函数进行同步，从而与 Work-Groups 中的其他 Work-Items 进行通信。

在第 4 章中，我们讨论了另一组 Work-Items。Sub-Groups 是 Work-Groups 中由实现定义的 Work-Items 子集，它们在相同的硬件资源上一起执行或具有额外的调度保证。因为实现决定如何将 Work-Items 分组为 Sub-Groups，所以 Sub-Groups 中的 Work-Items 可能能够比任意 Work-Groups 中的 Work-Items 更有效地通信或同步。

本节描述 Sub-Groups 中 Work-Items 之间通信的构建块。Sub-Groups 还需要 Work-Items 分组的概念，因此 Sub-Groups 也需要 ND 范围 Kernel，并且不包含在基本数据并行 Kernel 的执行模型中。

9.4.1 通过 Sub-Groups Barrier 进行同步

```

h.parallel_for(
    nd_range{{size}, {16}}, [=](nd_item<1> item) {
        auto sg = item.get_sub_group();
        group_barrier(sg);
        // ...
        auto index = item.get_global_id();
        data_acc[index] = data_acc[index] + 1;
    });

```

图 9.9: 查询和使用 *sub_group* 类

正如 Work-Groups 中的 Work-Items 可以如何使用 Work-Groups Barrier 来同步一样, Sub-Groups 中的 Work-Items 可以使用 Sub-Groups Barrier 来同步。要执行 Sub-Groups Barrier, 请调用相同的 `group_barrier` 函数, 但传递代表 Sub-Groups 而不是 Work-Groups 的组对象, 如图 9-9 所示。与 Work-Groups 对象一样, 可以从 ND 范围 Kernel 的 `nd_item` 类查询表示 Sub-Groups 的组对象, 但不能从基本数据并行项查询。

与 Work-Groups Barrier 一样, Sub-Groups Barrier 可以接受可选参数来扩大与 Sub-Groups Barrier 相关的任何内存操作的范围, 但这并不常见, 在大多数情况下我们可以简单地使用默认内存范围。

9.4.2 在 Sub-Groups 内交换数据

与 Work-Groups 不同, Sub-Groups 没有用于交换数据的专用存储空间。相反, Sub-Groups 中的 Work-Items 可以通过 Work-Groups 本地存储器、通过全局存储器或更常见地通过使用 Sub-Groups 集体功能来交换数据。

如前所述, 集体函数是描述由一组 Work-Items 而不是单个 Work-Items 执行的操作的函数。由于 Barrier 同步功能是由一组 Work-Items 执行的操作, 因此它是集体功能的一个示例。

其他集体功能表达了共同的沟通模式。我们将在本章后面详细描述许多集体函数的语义, 但现在我们重点关注 `group_broadcast` 集体函数, 我们将使用它来实现使用 Sub-Groups 的矩阵乘法。

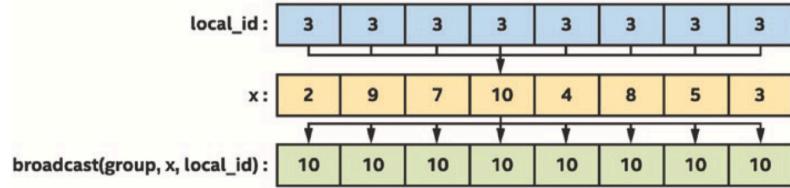


图 9.10: 广播函数

group广播集体函数从组中的一个 Work-Items 获取一个值，并将其传递给组中的所有其他 Work-Items。图 9-10 显示了一个示例。请注意，广播函数的语义要求标识组中要通信的值的 local_id 对于组中的所有 Work-Items 必须相同，以确保广播函数的结果对于所有 Work-Items 也相同在组中。

```

h.parallel_for(
    nd_range<2>{{M, N}, {1, tile_size}},
    [=](nd_item<2> item) {
        // Indices in the global index space:
        int m = item.get_global_id()[0];
        int n = item.get_global_id()[1];

        // Index in the local index space:
        int i = item.get_local_id()[1];

        // Template type T is the type of data stored in
        // the matrix
        T sum = 0;
        for (int kk = 0; kk < K; kk += tile_size) {
            // Load the matrix tile from matrix A, and
            // synchronize to ensure all work-items have a
            // consistent view of the matrix tile in local
            // memory.
            tileA[i] = matrixA[m][kk + i];
            group_barrier(item.get_group());

            // Perform computation using the local memory
            // tile, and matrix B in global memory.
            for (int k = 0; k < tile_size; k++) {
                // Because the value of k is the same for
                // all work-items in the group, these reads
                // from tileA are broadcast operations.
                sum += tileA[k] * matrixB[kk + k][n];
            }

            // After computation, synchronize again, to
            // ensure all reads from the local memory tile
            // are complete.
            group_barrier(item.get_group());
        }

        // Write the final result to global memory.
        matrixC[m][n] = sum;
    });
}

```

图 9.11: 矩阵乘法 Kernel 包括广播操作

如果我们查看本地内存矩阵乘法 Kernel 的最内层循环（如图 9-11 所示），我们可以看到对矩阵图块的访问是广播操作，因为组中的每个 Work-Items 都读取相同的值矩阵瓦片的。

我们将使用带有 Sub-Groups 对象的 group_broadcast 函数来实现不需要 Work-Groups 本地内存或 Barrier 的矩阵乘法 Kernel。在许多设备上，

使用 Work-Groups 本地内存和 Barrier，Sub-Groups 广播比 Work-Groups 广播更快。

9.4.3 完整 Sub-Groups ND 范围 Kernel 示例

```
// Note: This example assumes that the sub-group size
// is greater than or equal to the tile size!
constexpr int tile_size = 4;
h.parallel_for(
    nd_range<2>{{M, N}, {1, tile_size}},
    [=](nd_item<2> item) {
        auto sg = item.get_sub_group();

        // Indices in the global index space:
        int m = item.get_global_id()[0];
        int n = item.get_global_id()[1];

        // Index in the local index space:
        int i = item.get_local_id()[1];

        // Template type T is the type of data stored
        // in the matrix
        T sum = 0;
        for (int kk = 0; kk < K; kk += tile_size) {
            // Load the matrix tile from matrix A.
            T tileA = matrixA[m][kk + i];

            // Perform computation by broadcasting from
            // the matrix tile and loading from matrix B
            // in global memory. The loop variable k
            // describes which work-item in the sub-group
            // to broadcast data from.
            for (int k = 0; k < tile_size; k++) {
                sum += group_broadcast(sg, tileA, k) *
                    matrixB[kk + k][n];
            }
        }

        // Write the final result to global memory.
        matrixC[m][n] = sum;
    });
}
```

图 9.12: 用 ND 范围 `parallel_for` 和 Sub-Groups 集体函数表示的平铺矩阵乘法核

图 9-12 是使用 Sub-Groups 实现矩阵乘法的完整示例。请注意，该 Kernel 不需要 Work-Groups 本地内存或显式同步，而是使用 Sub-Groups 广播集体功能来在 Sub-Groups 中的 Work-Items 之间传达矩阵图块的内容。

9.5 组函数和组算法

在本章的“Sub-Groups”部分中，我们描述了集体功能以及集体功能如何表达常见的沟通模式。我们特别讨论了广播集体功能，它用于将值从组中的一个 Work-Items 传递到组中的其他 Work-Items。本节描述附加的集体功能。

尽管本节中描述的集体功能可以使用原子、Work-Groups 本地内存和 Barrier 等功能直接在我们的程序中实现，但许多设备都包含专用硬件来加速集体功能。即使设备不包含专用硬件，供应商提供的集体函数的实现也可能会针对其运行的设备进行调整，因此调用内置集体函数通常会比我们编写的通用实现执行得更好。

注 29 将集体函数用于常见的通信模式，以简化代码并提高性能！

9.5.1 广播 Broadcast

group_broadcast 函数使组中的一个 Work-Items 能够与组中的所有其他 Work-Items 共享变量的值。广播功能的工作原理如图 9-10 所示。Work-Groups 和 Sub-Groups 都支持 group_broadcast 功能。

9.5.2 投票 Votes

any_of_group、all_of_group 和 none_of_group 函数（以下称为“投票”函数）使 Work-Items 能够比较其组中布尔条件的结果：如果条件对于其中至少一个 Work-Items 为真，则 any_of_group 返回 true。如果组中所有 Work-Items 的条件都为 true，则 all_of_group 返回 true；如果组中的所有 Work-Items 的条件为 false，则 none_of_group 返回 true。图 9-13 显示了示例输入的这两个函数的比较。

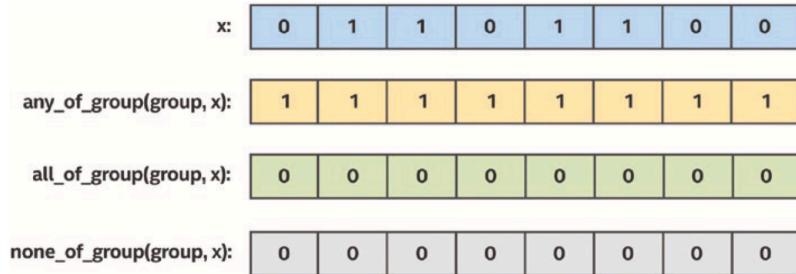


图 9.13: `any_of_group`、`all_of_group` 和 `none_of_group` 功能的比较

SYCL 2020 还支持这些函数的另一种变体，其中组中的 Work-Items 协作评估一系列数据，例如标准 C++ `all_of`、`any_of` 和 `none_of` 算法。这些函数被命名为 `joint_any_of`、`joint_all_of` 和 `joint_none_of`，以区别于组中每个 Work-Items 保存要直接比较的数据的变体。

例如，投票函数对于某些迭代算法非常有用，可以确定组中所有 Work-Items 的解决方案何时收敛。Work-Groups 和 Sub-Groups 支持投票功能。

9.5.3 洗牌 Shuffles

Sub-Groups 最有用的功能之一是能够在各个 Work-Items 之间直接通信，而无需显式内存操作。在许多情况下，例如 Sub-Groups 矩阵乘法 Kernel，这些洗牌操作使我们能够从 Kernel 中删除 Work-Groups 本地内存使用，并避免不必要的重复访问全局内存。这些 shuffle 函数有多种类型可供选择。

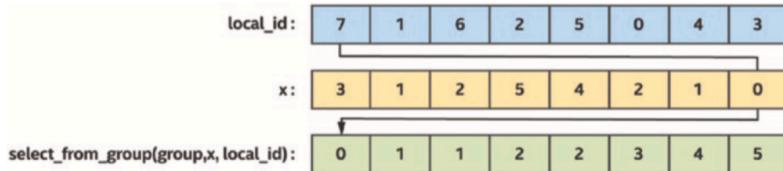


图 9.14: 使用泛型 `select_from_group` 根据预先计算的索引对值进行排序

最通用的 shuffle 函数称为 `select_from_group`，如图 9-14 所示，它允许 Sub-Groups 中任意一对 Work-Items 之间进行任意通信。然而，这种通用性可能会降低性能，我们强烈鼓励尽可能使用更专业的 shuffle 函数。

在图 9-14 中，通用洗牌用于使用预先计算的排列索引对 Sub-Groups 的值进行排序。针对 Sub-Groups 中的一个 Work-Items 显示了箭头，其中洗牌的结果是 local_id 等于 7 的 Work-Items 的 x 值。

请注意，Sub-Groups group_broadcast 函数可以被视为通用 select_from_group 函数的专用版本，其中 Sub-Groups 中所有 Work-Items 的随机索引相同。当已知 Sub-Groups 中所有 Work-Items 的洗牌索引相同时，使用 group_broadcast 而不是 select_from_group 可为编译器提供附加信息，并可能提高某些实现的性能。

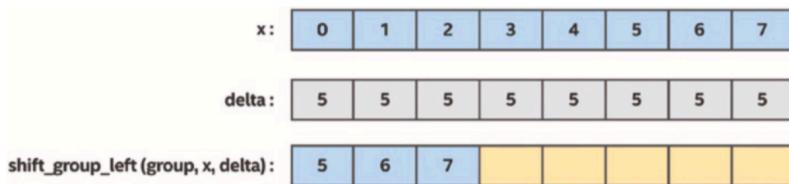


图 9.15: 使用 `shift_group_left` 将 Sub-Groups 的 x 值按 5 个项目移动

`shift_group_right` 和 `shift_group_left` 函数有效地将 Sub-Groups 的内容沿给定方向移动固定数量的元素，如图 9-15 所示。请注意，返回到 Sub-Groups 中最后五个 Work-Items 的值未定义，并在图 9-15 中显示为空白。移位对于并行化具有循环携带依赖性的循环或实现常见算法（例如独占或包含扫描）非常有用。

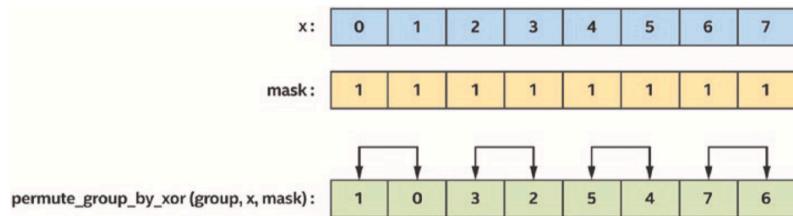


图 9.16: 使用 `permute_group_by_xor` 交换相邻的 x 对

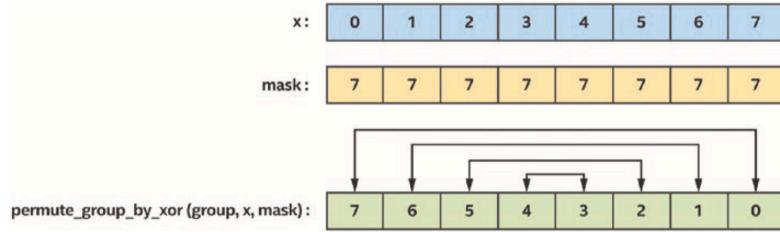


图 9.17: 使用 `permute_group_by_xor` 反转 x 的值

`permute_group_by_xor` 函数交换两个 Work-Items 的值，由应用于 Work-Items 的 Sub-Groups 本地 ID 和固定常量的 XOR 运算的结果指定。如图 9-16 和图 9-17 所示，可以使用 XOR 来表达几种常见的通信模式，例如交换相邻值对或反转 Sub-Groups 值。

由于 `shuffle` 函数非常专业，因此它们仅适用于 Sub-Groups，不适用于 Work-Groups。

注 30 (使用广播、投票和集体进行 Sub-Groups 优化) 应用于 *Sub-Groups* 的 *Broadcast*、*Vote* 和其他集体函数的行为与应用于 *Work-Groups* 时的行为相同，但它们值得额外关注，因为它们可能会在某些编译器中启用积极的优化。例如，编译器可能能够减少广播到 *Sub-Groups* 中所有 *Work-Items* 的变量的寄存器使用量，或者能够根据 *any_of_group* 和 *all_of_group* 函数的使用情况推断控制流分歧。

9.6 总结

本章讨论了组中的 Work-Items 如何进行通信和协作以提高某些类型 Kernel 的性能。

我们首先讨论了 ND 范围 Kernel 如何支持将 Work-Items 分组为 Work-Groups。我们讨论了将 Work-Items 分组到 Work-Groups 中如何改变并行执行模型，从而保证 Work-Groups 中的 Work-Items 以支持通信和同步的方式调度执行。

接下来，我们讨论了 Work-Groups 中的 Work-Items 如何使用 Barrier 进行同步以及 Barrier 如何在 Kernel 中表达。我们还讨论了如何通过 Work-Groups 本地内存执行 Work-Groups 中 Work-Items 之间的通信，以简化

Kernel 并提高性能，并且讨论了如何使用本地访问器表示 Work-Groups 本地内存。

我们讨论了 ND 范围 Kernel 中的 Work-Groups 如何进一步划分为 Work-ItemsSub-Groups，其中 Work-ItemsSub-Groups 可以支持额外的通信模式或调度保证。

对于 Work-Groups 和 Sub-Groups，我们讨论了如何通过使用集体功能来表达和加速常见的沟通模式。

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如何针对第 15、16 和 17 章中的特定设备进行优化的重要基础。

10 定义 Kernel

到目前为止，在本书中，我们的代码示例已经使用 C++ lambda 表达式表示 Kernel。Lambda 表达式是一种在使用时表示 Kernel 的简洁而方便的方法，但它们并不是在 SYCL 中表示 Kernel 的唯一方法。在本章中，我们将详细探讨定义 Kernel 的各种方法，帮助我们选择最适合我们的 C++ 编码需求的 Kernel 形式。

本章解释并比较了表示 Kernel 的三种方法：

- Lambda 表达式。
- 命名函数对象（Functor）。
- 通过与通过其他语言或 API 创建的 Kernel 的互操作性。本章简要介绍了该主题，第 20 章更详细地介绍了该主题。

本章最后讨论了如何显式操作 Kernel 包中的 Kernel 来查询 Kernel 属性并控制何时以及如何编译 Kernel。

10.1 为什么用三种方式来表示 Kernel?

Kernel Representation	Description
Lambda Expression	<p>Pros:</p> <ul style="list-style-type: none"> • Lambda expressions are a concise way to represent a kernel right where it is used. • Lambda expressions are a familiar way to represent kernel-like operations in modern C++ codebases. • Lambda capture rules automatically pass data to kernels. <p>Cons:</p> <ul style="list-style-type: none"> • Kernels represented as lambda expressions cannot be templated, and do not assemble as a library (like regular functions) without extra work. • The lambda syntax may be unfamiliar to some C++ codebases.
Named Function Object (Functor)	<p>Pros:</p> <ul style="list-style-type: none"> • Functors can be templated, reused, and shipped as a part of a library, just like any other C++ class. • Functors provide more control over the data that gets passed into a kernel. <p>Cons:</p> <ul style="list-style-type: none"> • Kernels represented as functors require more code than kernels represented as lambda expressions. • Kernel arguments must be explicitly passed to functors and are not captured automatically.
Interoperability with Other Languages or APIs	<p>Pros:</p> <ul style="list-style-type: none"> • Enables re-use of previously written kernels or libraries. • Enables large application codebases to incrementally add support for SYCL. • Kernel languages from other APIs may support features that have not been added or are difficult to express with SYCL. <p>Cons:</p> <ul style="list-style-type: none"> • Interoperability is an optional feature that may not be supported by all SYCL implementations or by all SYCL devices in an implementation. • Kernels written in other APIs are not compiled by the SYCL device compiler, which may limit compile-time syntax checking, type checking for kernel arguments, and optimization opportunities. • Kernels written in other APIs may not support the latest C++ features.

图 10.1: 表示 Kernel 的三种方式

在深入讨论细节之前，我们首先总结一下为什么有三种定义 Kernel 的方法以及每种方法的优缺点。图 10-1 给出了一个有用的总结。

请记住，Kernel 用于表示计算单元，并且 Kernel 的许多实例通常会在加速器上并行执行。SYCL 支持多种方式来表达 Kernel，以自然、无缝地集成到具有不同编码风格的代码库中，同时还能在各种加速器类型上高效执行。

10.2 作为 Lambda 表达式的 Kernel

C++ lambda 表达式，也称为匿名函数对象、未命名函数对象、闭包或简称 lambda，是在使用 Kernel 时表达 Kernel 的便捷方法。本节介绍如何

将 Kernel 表示为 C++ lambda 表达式。这扩展了第 1 章中 C++ lambda 表达式的介绍性复习，其中包括一些带有输出的基本编码示例。

C++ lambda 表达式非常强大并且具有表达语法，但在 SYCL 中表达 Kernel 时，仅需要（并支持）完整 C++ lambda 表达式语法的特定子集。

10.2.1 Kernel Lambda 表达式的元素

```
h.parallel_for(
    size,
    // This is the start of a kernel lambda expression:
    [=](id<1> i) { data_acc[i] = data_acc[i] + 1; }
    // This is the end of the kernel lambda expression.
);
```

图 10.2: 使用 *lambda* 表达式定义的简单 Kernel

图 10-2 显示了一个用典型 lambda 表达式编写的简单 Kernel——本书到目前为止的代码示例都使用了这种语法。

```
q.submit([&](handler& h) {
    accessor data_acc{data_buf, h};
    h.parallel_for(
        nd_range{{size}, {8}},
        1 2 4 5 [=](id<1> i) noexcept [[sycl::reqd_work_group_size(8)]] ->void {
            data_acc[i] = data_acc[i] + 1; 3
        });
});
```

图 10.3: Kernel lambda 表达式的更多元素，包括可选元素

图 10-3 中的插图显示了可与 Kernel 一起使用的 lambda 表达式的元素，但其中许多元素并不典型。在大多数情况下，lambda 默认值就足够了，因此典型的 Kernel lambda 表达式看起来更像图 10-2 中的 lambda 表达式，而不是图 10-3 中更复杂的 lambda 表达式。

1. lambda 表达式的第一部分描述 lambda 捕获。从周围范围捕获变量使其可以在 lambda 表达式中使用，而无需将其作为参数显式传递给 lambda 表达式。

C++ lambda 表达式支持通过复制或创建对变量的引用来自捕获变量，但对于 Kernel lambda 表达式，只能通过复制来捕获变量。一般做法是简单地使用默认捕获模式 [=]，该模式按值隐式捕获所有变量，尽管也可以在逗号分隔的捕获列表中显式命名每个捕获的变量。Kernel 中使用的任何未按值捕获的变量都将导致编译时错误。请注意，根据 C++ 标准，全局变量不会被 lambda 表达式捕获。

2. lambda 表达式的第二部分描述传递给 lambda 表达式的参数，就像传递给命名函数的参数一样。对于 Kernel lambda 表达式，该参数取决于 Kernel 的调用方式并标识并行执行空间中工作项的索引。有关各种并行执行空间以及如何识别每个执行空间中工作项的索引的更多详细信息，请参阅第 4 章。
3. lambda 表达式的最后一部分定义函数体。对于 Kernel lambda 表达式，函数体描述了应在并行执行空间中的每个索引处执行的操作。
lambda 表达式还有其他部分，但它们要么是可选的、不经常使用的，要么不受 SYCL 2020 支持：
4. SYCL 2020 没有定义任何说明符（例如 mutable），因此示例代码中没有显示任何说明符。
5. 支持异常规范，但如果提供，则必须为 noexcept，因为 Kernel 不支持异常。
6. 支持 Lambda 属性，可用于控制 Kernel 的编译方式。例如，reqd_work_group_size 属性可用于要求 Kernel 的特定工作组大小，而 device_has 属性可用于要求 Kernel 的特定设备方面。第 12 章包含有关使用属性和方面进行 Kernel 专业化的更多信息。
7. 可以指定返回类型，但如果提供则必须为 void，因为 Kernel 不支持非 void 返回类型。

注 31 (LAMBDA 捕获：隐式还是显式？) 由于可能存在的悬空指针问题，某些 C++ 样式指南建议不要对 lambda 表达式进行隐式（或默认）捕获，尤其是当 lambda 表达式跨越范围边界时。当使用 lambda 表示 Kernel 时，可能会出现同样的问题，因为 Kernel lambda 在设备上异步执行，与主机代码分开。

由于隐式捕获有用且简洁，因此它是 SYCL Kernel 的常见做法，也是我们在本书中使用的约定，但最终由我们决定是选择隐式捕获的简洁性还是显式捕获的清晰性。

10.2.2 识别 Kernel Lambda 表达式

```
// In this example, "class Add" names the kernel
// lambda expression.
h.parallel_for<class Add>(size, [=](id<1> i) {
    data_acc[i] = data_acc[i] + 1;
});
```

图 10.4: 识别 Kernel lambda 表达式

当将 Kernel 编写为 lambda 表达式时，在某些情况下还必须提供一个元素：因为 lambda 表达式是匿名的，所以有时 SYCL 需要显式 Kernel 名称模板参数来唯一标识编写为 lambda 表达式的 Kernel。

命名 Kernel lambda 表达式是主机代码编译器在由单独的设备代码编译器编译 Kernel 时识别要调用哪个 Kernel 的一种方式。命名 Kernel lambda 还可以对已编译 Kernel 进行运行时自省或按名称构建 Kernel，如图 10-9 所示。

```
h.parallel_for(size, [=](id<1> i) {
    data_acc[i] = data_acc[i] + 1;
});
```

图 10.5: 使用未命名的 Kernel lambda 表达式

为了在不需要 Kernel 名称模板参数时支持更简洁的代码，对于大多数 SYCL 2020 编译器来说，Kernel 名称模板参数是可选的。当不需要 Kernel 名称模板参数时，我们的代码可以更加紧凑，如图 10-5 所示。

由于大多数情况下不需要 lambda 表达式的 Kernel 名称模板参数，因此我们通常可以从未命名的 lambda 开始，仅在需要 Kernel 名称模板参数的特定情况下添加 Kernel 名称。

注 32 当不需要 *Kernel* 名称模板参数时，最好使用未命名的 *Kernel lambda* 来减少冗长。

10.3 Kernel 作为命名函数对象

命名函数对象，也称为 Functor，是 C++ 中的一种既定模式，允许在维护定义良好的接口的同时对任意数据集合进行操作。当用于表示 Kernel 时，命名函数对象的成员变量定义 Kernel 可以操作的状态，并且为并行执行空间中的每个工作项调用重载函数调用 operator()。

命名函数对象需要比 lambda 表达式更多的代码来表达 Kernel，但额外的冗长提供了更多的控制和附加功能。例如，分析和优化表示为命名函数对象的 Kernel 可能会更容易，因为 Kernel 使用的任何缓冲区和数据值都必须显式传递给 Kernel，而不是由 lambda 表达式自动捕获。

表示为命名函数对象的 Kernel 也可能更容易调试、更容易重用，并且它们可以作为单独的头文件或库的一部分提供。

最后，因为命名函数对象就像任何其他 C++ 类一样，所以可以模板化表示为命名函数对象的 Kernel。C++20 添加了模板化 lambda 表达式，但基于 C++17 的 SYCL 2020 中的 Kernel 不支持模板化 lambda 表达式。

10.3.1 Kernel 命名函数对象的元素

```

class Add {
public:
    Add(accessor<int> acc) : data_acc(acc) {}
    void operator()(id<1> i) const {
        data_acc[i] = data_acc[i] + 1;
    }

private:
    accessor<int> data_acc;
};

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

    for (int i = 0; i < size; i++) {
        data[i] = i;
    }

    {
        buffer data_buf{data};

        queue q;
        std::cout
            << "Running on device: "
            << q.get_device().get_info<info::device::name>()
            << "\n";

        q.submit([&](handler& h) {
            accessor data_acc{data_buf, h};
            h.parallel_for(size, Add(data_acc));
        });
    }
    // ...
}

```

图 10.6: *Kernel* 作为命名函数对象

图 10-6 中的代码演示了表示为命名函数对象的 Kernel 的典型用法。在这个例子中，Kernel 的参数被传递给类构造函数，而 Kernel 本身则在重载函数调用 operator() 中。

当 Kernel 表示为命名函数对象时，命名函数对象类型必须遵循 SYCL 2020 规则才能设备可复制。通俗地说，这意味着命名函数对象可以逐字节安全地复制，从而使命名函数对象的成员变量能够传递到设备上执行的 Kernel

代码并由其访问。任何可普通复制的 C++ 类型都是隐式设备可复制的。

重载函数调用 operator() 的参数取决于 Kernel 的启动方式，就像用 lambda 表达式表示的 Kernel 一样。

```
class AddWithAttribute {
public:
    AddWithAttribute(accessor<int> acc) : data_acc(acc) {}
    [[sycl::reqd_work_group_size(8)]] void operator()(
        id<1> i) const {
        data_acc[i] = data_acc[i] + 1;
    }

private:
    accessor<int> data_acc;
};

class MulWithAttribute {
public:
    MulWithAttribute(accessor<int> acc) : data_acc(acc) {}
    void operator()
        [[sycl::reqd_work_group_size(8)]] (id<1> i) const {
        data_acc[i] = data_acc[i] * 2;
    }

private:
    accessor<int> data_acc;
};
```

图 10.7: 将可选属性与命名函数对象一起使用

图 10-7 中的代码显示了如何在定义为命名函数对象的 Kernel 上使用可选的 Kernel 属性，例如 reqd_work_group_size 属性。当 Kernel 被定义为命名函数对象时，可选 Kernel 属性有两个有效位置。这与编写为 lambda 表达式的 Kernel 不同，其中可选 Kernel 属性只有一个位置有效。

由于所有函数对象都是命名的，因此即使函数对象是模板化的，主机代码编译器也可以使用函数对象类型来识别设备代码编译器生成的 Kernel 代码。不需要额外的 Kernel 名称模板参数来命名 Kernel 函数对象。

10.4 Kernel 包中的 Kernel

我们应该注意的与 SYCL Kernel 相关的最后一个主题涉及 SYCL Kernel 对象和 SYCL Kernel 包。典型的应用程序开发不需要了解 Kernel 对象

和 Kernel 包，但在某些情况下对于调整应用程序性能很有用。了解 Kernel 对象和 Kernel 包还可以帮助理解 SYCL 实现如何组织和管理 Kernel。

SYCL Kernel 包是应用程序使用的 SYCL Kernel 或 SYCL 函数的容器。应用程序中 Kernel 包的数量取决于特定的 SYCL 编译器。一些应用程序可能只有一个 Kernel 包，即使它们有多个 Kernel，而其他应用程序可能有多个 Kernel 包，即使它们只有几个 Kernel。

SYCL Kernel 包及其包含的 Kernel 或函数可以处于以下三种状态之一：

- **输入状态**: 此状态下的 Kernel 包通常采用某种中间表示形式，并且必须先进行即时 (JIT) 编译，然后才能在设备上执行。
- **对象状态**: 此状态下的 Kernel 包通常会被编译但不会链接，就像主机应用程序编译器创建的对象文件一样。
- **可执行状态**: 此状态下的 Kernel 包已完全编译为设备代码，并准备好在设备上执行。在编译主机应用程序时提前 (AOT) 编译的 Kernel 包最初将处于此状态。

虽然规范没有要求，但许多 SYCL 编译器最初将 Kernel 编译为中间表示形式，以便可移植到最大数量的 SYCL 设备。这意味着应用程序 Kernel 包通常最初处于输入状态。然后，许多 SYCL 运行时库根据需要“延迟”地将 Kernel 包从输入状态编译为可执行状态。

这通常是一个很好的策略，因为它可以实现快速应用程序启动，并且如果从未执行 Kernel，则不会不必要地编译 Kernel。然而，这种策略的缺点是，第一次使用 Kernel 比后续使用需要更长的时间，因为它包括编译 Kernel 所需的时间以及提交和执行 Kernel 所需的通常时间。对于复杂的 Kernel，编译 Kernel 的时间可能很长，因此需要在应用程序执行期间将编译转移到不同的点，例如在加载应用程序时，或者转移到单独的后台线程。

```
auto kb = get_kernel_bundle<bundle_state::executable>(
    q.get_context());

std::cout
    << "All kernel compilation should be done now.\n";

q.submit([&](handler& h) {
    // Use the pre-compiled kernel from the kernel bundle.
    h.use_kernel_bundle(kb);

    accessor data_acc{data_buf, h};
    h.parallel_for(range{size}, [=](id<1> i) {
        data_acc[i] = data_acc[i] + 1;
    });
});
```

图 10.8: 使用 *Kernel* 捆绑包显式编译 *Kernel*

为了更好地控制 Kernel 编译的时间和方式，我们可以在将 Kernel 提交到队列之前显式请求编译 Kernel 包。当 Kernel 被提交到队列执行时，可以使用预编译的 Kernel 包。图 10-8 显示了如何在将任何 Kernel 提交到队列之前编译应用程序使用的所有 Kernel，以及如何使用预编译的 Kernel 包。

此示例为与 SYCL 队列关联的 SYCL 上下文中的所有设备请求处于可执行状态的 Kernel 包，这将导致应用程序中的任何 Kernel（如果尚未处于可执行状态）进行即时编译。在这个具体示例中，Kernel 非常短，编译不会花费很长时间，但如果有很多 Kernel，或者它们更复杂，则此步骤可能会花费大量时间。当然，如果所有 Kernel 都被提前编译，或者如果所有 Kernel 都已经被即时编译，则该操作实际上是免费的，因为所有 Kernel 都已经处于可执行状态。

```

auto kid = get_kernel_id<class Add>();
auto kb = get_kernel_bundle<bundle_state::executable>(
    q.get_context(), {q.get_device()}, {kid});

std::cout << "Kernel compilation should be done now.\n";

q.submit([&](handler& h) {
    // Use the pre-compiled kernel from the kernel bundle.
    h.use_kernel_bundle(kb);

    accessor data_acc{data_buf, h};
    h.parallel_for<class Add>(range{size}, [=](id<1> i) {
        data_acc[i] = data_acc[i] + 1;
    });
});

```

图 10.9: 使用 Kernel 捆绑包显式和有选择地编译 Kernel

如果我们想要更多地控制 Kernel 的编译时间和方式，我们可以请求特定设备的 Kernel 包，甚至是程序中的特定 Kernel。这使我们能够有选择地立即编译程序中的某些 Kernel，同时让其他 Kernel 稍后或根据需要进行编译。图 10-9 显示了如何仅编译由类 Add kernel name 标识的 Kernel，并且仅编译与 SYCL 队列关联的 SYCL 设备，而不是程序中的所有 Kernel 和 SYCL 上下文中的所有设备。

这是一种罕见的情况，我们需要命名我们的 Kernel lambda 表达式；否则，我们将无法识别要编译的 Kernel。

注 33 使用 Kernel 捆绑包在应用程序中以可预测的方式编译 Kernel!

Kernel 包中的 Kernel 还可用于查询有关已编译 Kernel 的信息，例如确定特定设备的 Kernel 的最大工作组大小。在某些情况下，可能需要这些类型的 Kernel 查询来选择用于 Kernel 和特定设备的有效值。在其他情况下，Kernel 查询可以提供提示，允许我们的应用程序动态调整并选择 Kernel 和特定设备的最佳值。

```

auto kid = get_kernel_id<class Add>();
auto kb = get_kernel_bundle<bundle_state::executable>(
    q.get_context(), {q.get_device()}, {kid});
auto kernel = kb.get_kernel(kid);

std::cout
    << "The maximum work-group size for the kernel and "
    "this device is: "
    << kernel.get_info<info::kernel_device_specific::
        work_group_size>(
            q.get_device())
    << "\n";

std::cout
    << "The preferred work-group size multiple for the "
    "kernel and this device is: "
    << kernel.get_info<
        info::kernel_device_specific::
        preferred_work_group_size_multiple>(
            q.get_device())
    << "\n";

```

Example Output:

```

Running on device: NVIDIA GeForce RTX 3060
The maximum work-group size for the kernel and this device is: 1024
The preferred work-group size multiple for the kernel and this device is: 32

```

Example Output:

```

Running on device: Intel(R) Data Center GPU Max 1100
The maximum work-group size for the kernel and this device is: 1024
The preferred work-group size multiple for the kernel and this device is: 16

```

Example Output:

```

Running on device: Intel(R) UHD Graphics 770
The maximum work-group size for the kernel and this device is: 512
The preferred work-group size multiple for the kernel and this device is: 64

```

图 10.10: 查询 Kernel 捆绑包中的 Kernel

识别 Kernel、从编译的 Kernel 包中获取 Kernel 对象以及使用 Kernel 对象执行设备特定查询的基本机制如图 10-10 所示。第 12 章描述了更完整的可用 Kernel 查询列表。

这是另一种罕见的情况，我们需要命名我们的 Kernel lambda 表达式；否则，我们将无法识别要查询的 Kernel。

10.5 与其他 API 的互操作性

当 SYCL 实现构建在另一个 API 之上时，该实现可能能够与使用底层 API 机制定义的 Kernel 进行互操作。这允许应用程序轻松地将 SYCL 集成到已经使用底层 API 的现有代码库中。第 20 章详细介绍了这个主题。就本章而言，我们可以简单地认识到与通过其他源语言或 API 创建的 Kernel

或 Kernel 包的互操作性提供了第三种表示 Kernel 的方法。

10.6 总结

在本章中，我们探索了定义 Kernel 的不同方法。我们描述了如何通过将 Kernel 表示为 C++ lambda 表达式或命名函数对象，将 SYCL 无缝集成到现有 C++ 代码库中。对于新的代码库，我们还讨论了不同 Kernel 表示的优缺点，以帮助根据应用程序或库的需求选择定义 Kernel 的最佳方法。

我们描述了 Kernel 通常如何在 SYCL 应用程序中编译，以及如何直接操作 Kernel 包中的 Kernel 来控制编译过程。尽管大多数应用程序不需要这种级别的控制，但在调整应用程序时，这是一种需要注意的有用技术。

11 向量和数学数组

向量是数据的集合。向量很有用，因为计算机中的并行性来自计算机硬件的集合，并且数据通常在相关分组中进行处理（例如，RGB 像素中的颜色通道）。这个概念非常重要，因此我们花了一章的时间讨论不同的 SYCL 向量类型以及如何使用它们。请注意，本章中我们不会深入讨论标量运算的向量化，因为这会根据设备类型和实现而有所不同。第 16 章介绍了标量运算的向量化。

本章旨在解决以下问题：

- 什么是向量类型？
- SYCL 数学数组 (marray) 和向量 (vec) 类型之间有什么区别？
- 我应该何时以及如何使用 marray 和 vec？

我们使用工作代码示例讨论 marray 和 vec，并重点介绍利用这些类型的最重要方面。

11.1 向量类型的歧义

当我们与并行编程专家交谈时，向量是一个令人惊讶的有争议的话题。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向量。

有两种广泛的方式来思考本章所说的向量类型：

1. 作为一种便捷类型，它将我们可能想要引用和操作的数据分组为一组，例如像素的 RGB 或 YUV 颜色通道。我们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，但便利类型可以开箱即用地为我们做到这一点。在许多用于对 GPU 进行编程的着色器语言中都可以找到便利类型，因此这种思维方式在许多 GPU 开发人员中很常见。
2. 作为描述代码如何映射到硬件中的 SIMD（单指令，多数据）指令集的机制。例如，在某些语言和实现中，float8 上的操作可以映射到硬件中的八通道 SIMD 指令。SIMD 向量类型在许多语言中用作 CPU 特定内联函数的高级替代方案，因此这种思维方式在许多 CPU 开发人员中已经很常见。

尽管这两种对向量类型的解释非常不同，但随着 SYCL 和其他语言同时适用于 CPU 和 GPU，它们无意中结合在一起并混在一起。`vec` 类（存在于 SYCL 1.2.1 中，并且仍然存在于 SYCL 2020 中）与任一解释兼容，而 `marray` 类（在 SYCL 2020 中引入）被明确描述为与 SIMD 矢量硬件指令无关的便利类型。

注 34 (变化即将到来：SIMD 类型) *SYCL 2020* 尚未包含与第二种解释（SIMD 映射）明确相关的向量类型。但是，已经有一些扩展允许我们编写显式向量代码，这些代码直接映射到硬件中的 SIMD 指令，专为想要为特定架构调整代码并从编译器矢量器中获取控制权的专家程序员而设计。我们还应该期望另一种向量类型最终出现在 *SYCL* 中，以涵盖第二种解释，可能与建议的 *C++ std::simd* 模板一致。这个新类将非常清楚地说明代码何时以显式向量样式编写，以减少混淆。*SYCL* 中现有的扩展和未来的类似 *std::simd* 的类型都是我们预计很少有开发人员使用的利基功能。

有了 `marray` 和专用的 `SIMD` 类，我们作为程序员的意图将从我们编写的代码中清晰可见。这样就不那么容易出错，不容易混淆，甚至可能减少专家开发人员之间在出现“什么是向量”问题时激烈讨论的次数。

11.2 我们对于 SYCL 向量类型的心智模型

在本书中，我们讨论了如何将 Work-Items 分组在一起以公开强大的通信和同步原语，例如 Sub-Group 屏障和洗牌。为了使这些操作在向量硬件上高效，需要假设 Sub-Group 中的不同 Work-Items 组合并映射到 SIMD 指令。换句话说，多个 Work-Items 由编译器组合在一起，此时它们可以映射到硬件中的 SIMD 指令。请记住第 4 章中的内容，这是在矢量硬件之上运行的 SPMD（单程序、多数据）编程模型的基本前提，其中单个 Work-Items 构成硬件中可能是 SIMD 指令的通道，而不是定义整个操作的 Work-Items，该操作将成为硬件中的 SIMD 指令。当映射到硬件中的 SIMD 指令、以 SPMD 风格编程时，您可以将编译器视为始终跨 Work-Items 进行向量化。

对于来自没有向量类型的语言或来自 GPU 着色语言的开发人员，我们可以将 SYCL 向量类型视为 Work-Items 的本地向量类型，因为如果添加两个四元素向量，则添加可能需要硬件中的四个指令（从 Work-Items 的角度来看它将被标量化）。向量的每个元素将由硬件中的不同指令/时钟周期相加。这与我们对向量类型的解释是一致的——为了方便，我们可以在源代码中的单个操作中添加两个向量，而不是在源代码中执行四个标量操作。

对于具有 CPU 背景的开发人员来说，我们应该知道 SIMD 硬件的隐式向量化在许多编译器中默认发生，与向量类型的使用无关。编译器可以跨 Work-Items 执行这种隐式向量化，从格式良好的循环中提取向量运算，或者在映射到向量指令时尊重向量类型 - 有关更多信息，请参阅第 16 章。

本章的其余部分重点介绍如何使用向量类型（对于 marray 和 vec）的方便解释来教授向量，这是我们在 SYCL 中编程时应该牢记的一点。

注 35 (其他实现的可能！) 从理论上讲，SYCL 的不同编译器和实现可以对代码中的向量数据类型如何映射到 SIMD 向量硬件指令做出不同的决策。我们应该阅读供应商的文档和优化指南，以了解如何编写映射到高效 SIMD 指令的代码，尽管本章中描述的思维和编程模式适用于大多数（理想情况下是所有）SYCL 实现。

11.3 数学数组 (marray)

Type Alias	marray Equivalent
mcharN	marray<int8_t, N>
mucharn	marray<uint8_t, N>
mshortN	marray<int16_t, N>
mushortN	marray<uint16_t, N>
mintN	marray<int32_t, N>
muintN	marray<uint32_t, N>
mlongN	marray<int64_t, N>
mulongN	marray<uint64_t, N>
mhalfN	marray<half, N>
mfloatN	marray<float, N>
mdoubleN	marray<double N>
mboolN	marray<bool, N>

图 11.1: 数学数组的类型别名

SYCL 数学数组类型 (marray)，请参见图 11-1，是 SYCL 2020 中的新增内容，其定义是为了消除对向量类型应如何表现的不同解释的歧义。marray 显式地表示了本章上一节中介绍的向量类型的第一种解释——与向量硬件

指令无关的便利类型。通过从名称中删除“向量”并包含“数组”，可以更轻松地记住和推理类型如何在硬件上逻辑实现。

`marray` 类根据其元素类型和元素数量进行模板化。元素数量参数 `NumElements` 是一个正整数—当 `NumElements` 为 1 时，数组可以隐式转换为等效的标量类型。元素类型参数 `DataT` 必须是 C++ 定义的数值类型。

`Marray` 是一个数组容器，与 `std::array` 类似，还额外支持数组上的数学运算符（例如 `+`、`+=`）和 SYCL 数学函数（例如 `sin`、`cos`）。它旨在为 SYCL 设备上的并行计算提供高效且优化的数组操作。

为了方便起见，SYCL 为数学数组提供了类型别名。对于这些类型别名，元素数量 `N` 必须为 2、3、4、8 或 16。

```
queue q;
marray<float, 4> input{1.0004f, 1e-4f, 1.4f, 14.0f};
marray<float, 4> res[M];
for (int i = 0; i < M; i++)
    res[i] = {-(i + 1), -(i + 1), -(i + 1), -(i + 1)};
{
    buffer in_buf(&input, range{1});
    buffer re_buf(res, range{M});

    q.submit([&](handler &cgh) {
        accessor re_acc{re_buf, cgh, read_write};
        accessor in_acc{in_buf, cgh, read_only};

        cgh.parallel_for(range<1>(M), [=](id<1> idx) {
            int i = idx[0];
            re_acc[i] = cos(in_acc[0]);
        });
    });
}
```

图 11.2: 使用 `marray` 的简单示例

图 11-2 显示了如何将 `cos` 函数应用于由四个浮点数组成的数组中的每个元素的简单示例。此示例强调了使用数组来表达适用于分配给每个 Work-Items 的数据集合的所有元素的操作的便利性。

通过在大范围的数据 `M` 上执行该 Kernel，我们可以在许多不同类型的设备上实现良好的并行性，包括那些比数组的四个元素宽得多的设备，而无需规定我们的代码如何映射到 SIMD 指令集操作关于向量类型。

11.4 矢量 (vec)

SYCL 向量类型 (vec) 存在于 SYCL 1.2.1 中，并且仍包含在 SYCL 2020 中。如前所述，vec 与向量类型的任一解释兼容。在实践中，vec 通常被解释为一种方便类型，因此我们建议使用 marray 来提高代码可读性并减少歧义。但是，此建议有三个例外，我们将在本节中介绍：矢量加载和存储、与后端本机矢量类型的互操作性以及称为“swizzles”的操作。

与 marray 一样，vec 类根据其元素数量和元素类型进行模板化。但是，与 marray 不同的是，NumElements 参数必须为 1、2、3、4、8 或 16，任何其他值都会导致编译失败。这是一个很好的例子，说明了向量类型的混乱影响了 vec 的设计：将向量的大小限制为 2 的小幂对于 SIMD 指令集是有意义的，但从寻求便利类型的程序员的角度来看似乎是任意的。元素类型参数 DataT 可以是设备代码中支持的任何基本标量类型。

此外，与 marray 一样，vec 公开 2、3、4、8 和 16 个元素的简写类型别名。marray 别名以“m”为前缀，而 vec 别名则不然，例如，uint4 是 vec<uint32_t, 4> 的别名，float16 是 vec<float, 16> 的别名。在处理向量类型时，我们必须密切注意这个“m”的存在或不存在，以确保我们知道我们正在处理哪个类，这一点很重要。

11.4.1 加载和存储

vec 类提供用于加载和存储向量元素的成员函数。这些操作作用于存储与向量通道相同类型的对象的连续内存位置。

```
template <access::address_space AddressSpace, access::decorated IsDecorated>
void load(size_t offset, multi_ptr<DataT, AddressSpace, IsDecorated> ptr);

template <access::address_space addressSpace, access::decorated IsDecorated>
void store(size_t offset, multi_ptr<DataT, AddressSpace, IsDecorated> ptr) const;
```

图 11.3: vec 加载和存储功能

加载和存储函数如图 11-3 所示。load 成员函数从 multi_ptr 地址处的内存中读取 DataT 类型的值（偏移量为 NumElements * DataT 的 offset 元素），并将这些值写入 vec 的通道。store 成员函数读取 vec 的通道，并将这些值写入 multi_ptr 地址处的内存，偏移量为 NumElements * DataT 的 offset 元素。

请注意，该参数是 multi_ptr，而不是访问器或原始指针。multi_ptr 的数据类型是 DataT，即 vec 类特化的组件的数据类型。这要求传递给 load 或 store 的指针必须与 vec 实例本身的组件类型匹配。

```

std::array<float, size> fpData;
for (int i = 0; i < size; i++) {
    fpData[i] = 8.0f;
}

buffer fpBuf(fpData);

queue q;
q.submit([&](handler& h) {
    accessor acc{fpBuf, h};

    h.parallel_for(workers, [=](id<1> idx) {
        float16 inpf16;
        inpf16.load(idx, acc.get_multi_ptr<access::decorated::no>());
        float16 result = inpf16 * 2.0f;
        result.store(idx, acc.get_multi_ptr<access::decorated::no>());
    });
});

```

图 11.4: 使用 *load* 和 *store* 成员函数

图 11-4 显示了使用加载和存储函数的简单示例。

SYCL 向量加载和存储函数提供了用于表达向量运算的抽象，但底层硬件架构和编译器优化将决定任何实际的性能优势。我们建议使用分析工具分析性能并尝试不同的策略，以找到特定用例的向量加载和存储操作的最佳利用率。

尽管我们不应该期望向量加载和存储操作映射到 SIMD 指令，但使用向量加载和存储函数仍然有助于提高内存带宽利用率。有效地对向量类型进行操作向编译器暗示每个 Work-Items 正在访问连续的内存块，并且某些设备可能能够利用此信息一次加载或存储多个元素，从而提高效率。

11.4.2 与后端本机向量类型的互操作性

SYCL vec 类模板还可以提供与后端的本机向量类型（如果存在）的互操作性。后端本机向量类型由成员类型 vector_t 定义，并且仅在设备代码中可用。vec 类可以从 vector_t 的实例构造，并且可以隐式转换为 vector_t 的实例。

我们大多数人永远不需要使用 `vector_t`, 因为它的用例非常有限; 它的存在只是为了允许与从 Kernel 函数内调用的后端本机函数进行互操作 (例如, 从 SYCL Kernel 内调用用 OpenCL C 编写的函数)。

11.4.3 Swizzle 操作

在图形应用程序中, 混合意味着重新排列向量的数据元素。例如, 如果向量 `a` 包含元素 1, 2, 3, 4, 并且知道四元素向量的分量可以称为 `x, y, z, w`, 我们可以写成 `b = a.wxyz()`, 向量 `b` 中的值为 4, 1, 2, 3。这种语法在代码紧凑性和具有用于此类操作的高效硬件的应用程序中很常见。

```
template <int... swizzleIndexes>
__swizzled_vec__ swizzle() const;
__swizzled_vec__ XYZW_ACCESS() const;
__swizzled_vec__ RGBA_ACCESS() const;
__swizzled_vec__ INDEX_ACCESS() const;

#ifndef SYCL_SIMPLE_SWIZZLES
// Available only when numElements <= 4
// XYZW_SWIZZLE is all permutations with repetition of:
// x, y, z, w, subject to numElements
__swizzled_vec__ XYZW_SWIZZLE() const;

// Available only when numElements == 4
// RGBA_SWIZZLE is all permutations with repetition of: r,
// g, b, a.
__swizzled_vec__ RGBA_SWIZZLE() const;
#endif
```

图 11.5: `vec swizzle` 成员函数

`vec` 类允许以两种方式之一执行混合, 如图 11-5 所示。

`swizzle` 成员函数 `template` 允许我们通过调用模板成员函数 `swizzle` 来执行 `swizzle` 操作。此成员函数采用可变数量的整数模板参数, 其中每个参数表示向量中相应元素的 `swizzle` 索引。`swizzle` 索引必须是 0 到 `NumElements-1` 之间的整数, 其中 `NumElements` 表示原始 SYCL 向量中的元素数量 (例如, `vec.swizzle<2, 1, 0, 3>()` 表示四个元素的向量)。`swizzle` 成员函数的返回类型始终是 `__swizzled_vec__` 的实例, 它是表示 `swizzle` 向量的实现定义的临时类。请注意, 调用 `swizzle` 时不会立即执行 `swizzle` 操作。相反, 当在表达式中使用返回的 `__swizzled_vec__` 实例时, 会执行

swizzle 操作。

简单的 swizzle 成员函数集(在 SYCL 规范中描述为 XYZW_SWIZZLE 和 RGBA_SWIZZLE) 是作为执行 swizzle 操作的替代方法提供的。这些成员函数仅适用于最多具有四个元素的向量，并且仅当 SYCL_SIMPLE_SWIZZLES 宏在任何 SYCL 头文件之前定义时才可用。简单的 swizzle 成员函数允许我们使用名称 x, y, z, w 或 r, g, b, a 引用向量的元素，并通过使用这些元素名称调用成员函数来执行 swizzle 操作直接地。

例如，简单的 swizzles 启用之前使用的 XYZW swizzle 语法 `a.wxyz()`。通过编写 `a.argb()`，可以使用 RGBA swizzles 等效地执行相同的操作。使用简单的 swizzles 可以生成更紧凑的代码，并且与其他语言（尤其是图形着色语言）更匹配的代码。当向量包含 XYZW 位置数据或 RGBA 颜色数据时，简单的混合也可以更好地表达程序员的意图。简单 swizzle 成员函数的返回类型也是 `__swizzled_vec__`。与 swizzle 成员函数模板一样，当在表达式中使用返回的 `__swizzled_vec__` 实例时，将执行实际的 swizzle 操作。

```

constexpr int size = 16;

std::array<float4, size> input;
for (int i = 0; i < size; i++) {
    input[i] = float4(8.0f, 6.0f, 2.0f, i);
}

buffer b(input);

queue q;
q.submit([&](handler& h) {
    accessor a{b, h};

    // We can access the individual elements of a vector by
    // using the functions x(), y(), z(), w() and so on.
    //
    // "Swizzles" can be used by calling a vector member
    // equivalent to the swizzle order that we need, for
    // example zyx() or any combination of the elements.
    // The swizzle need not be the same size as the
    // original vector.
    h.parallel_for(size, [=](id<1> idx) {
        auto e = a[idx];
        float w = e.w();
        float4 sw = e.xyzw();
        sw = e.xyzw() * sw.wzyx();
        sw = sw + w;
        a[idx] = sw.xyzw();
    });
});

```

图 11.6: 使用 `__swizzled_vec__` 类的示例

图 11-6 演示了简单 swizzles 和 `__swizzled_vec__` 类的用法。虽然 `__swizzled_vec__` 没有直接出现在我们的代码中，但它在 `b.xyzw() * sw.wzyx()` 等表达式中使用 `:b.xyzw()` 和 `sw.wzyx()` 的返回类型是 `__swizzled_vec__` 的实例，并且直到结果被分配回 `float4` 变量 `sw` 后才会计算乘法。

11.5 向量类型如何执行

正如本章所述，向量类型及其如何映射到硬件有两种不同的解释。到目前为止，我们特意只在高层讨论这些映射。在本节中，我们将更深入地研究向量类型的不同解释如何映射到 SIMD 寄存器等低级硬件功能，证明这两

种解释都可以有效利用向量硬件。

11.5.1 向量作为便利类型

关于向量如何从便利类型（例如，`marray` 和通常的 `vec`）映射到硬件实现，我们需要解决三个主要问题：

1. 为了利用 SPMD 编程模型的可移植性和表现力，我们应该考虑组合多个 Work-Items 来创建向量硬件指令。更具体地说，我们不应该认为向量硬件指令是从单个 Work-Items 中孤立创建的。
2. 作为 (1) 的结果，从一个 Work-Items 的角度来看，我们应该将向量上的操作（例如加法）视为按时间执行每个通道或每个元素。在我们的源代码中使用向量通常与利用底层向量硬件指令无关。
3. 如果我们以某些方式编写代码（例如将向量的地址传递给函数），则编译器需要遵守向量和数学数组的内存布局要求，这可能会导致令人惊讶的性能影响。了解这一点可以更轻松地编写编译器可以积极优化的代码。

我们将从进一步描述前两点开始，因为清晰的思维模型可以使编写代码变得更加容易。

如第 4 章和第 9 章所述，Work-Items 是并行层次结构的叶节点，代表 Kernel 函数的单个实例。Work-Items 可以按任何顺序执行，并且不能相互通信或同步，除非通过对本地或全局内存的原子内存操作，或通过组集合函数（例如，`select_from_group`、`group_barrier`）。

便利类型的实例对于单个 Work-Items 来说是本地的，因此可以被认为相当于每个 Work-Items 的私有 `NumElements` 数组。例如，`float4 y4` 声明的存储可以被认为等同于 `float y4[4]`。考虑图 11-7 中所示的示例。

```

h.parallel_for(8, [=](id<1> i) {
    float x = a[i];
    float4 y4 = b[i];
    a[i] = x + sycl::length(y4);
});
```

图 11.7: 向量执行示例

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 Work-Items 执行 Kernel 的结果可能会使用向量寄存器和 SIMD 指令, 但向量化是跨 Work-Items 的, 并且与任何 Work-Items 无关。我们代码中的向量类型。每个 Work-Items 都有自己的标量 x , 可以在编译器生成的隐式 SIMD 硬件指令中形成不同的通道, 如图 11-8 所示。在某些实现和某些硬件上, Work-Items 中的标量数据可以被认为是跨恰好同时执行的 Work-Items 隐式矢量化 (组合成 SIMD 硬件指令), 但是 Work-Items 代码我们编写的代码不会以任何方式对其进行编码——这是 SPMD 编程风格的核心。

以与硬件无关的方式暴露潜在的并行性可确保我们的应用程序可以扩展 (或缩小) 以适应不同平台的功能, 包括具有矢量硬件指令的平台。在应用程序开发过程中, 在 Work-Items 和其他形式的并行性之间取得适当的平衡是我们所有人都必须面对的挑战, 第 15、16 和 17 章对此进行了更详细的介绍。

Work-item ID	w0	w1	w2	w3	w4	w5	w6	w7
SIMD hardware instruction lanes	$x[w0]$	$x[w1]$	$x[w2]$	$x[w3]$	$x[w4]$	$x[w5]$	$x[w6]$	$x[w7]$

图 11.8: 从标量 x 到八位宽的硬件向量指令的可能扩展

通过编译器将标量变量 x 隐式向量扩展为向量硬件指令 (如图 11-8 所示), 编译器根据多个 Work-Items 中发生的标量操作在硬件中创建 SIMD 操作。

回到图 11-7 的代码示例, 对于向量变量 $y4$, 多个 Work-Items (例如

8 个 Work-Items) 的 Kernel 执行结果并没有使用硬件中的向量运算来处理四元素向量。相反，每个 Work-Items 独立地看到自己的向量（在本例中为 float4），并且对该向量元素的操作可能会跨多个时钟周期/指令发生。如图 11-9 所示。我们可以将向量视为已由编译器从 Work-Items 的角度进行标量化。

Scalarized ops	Exec cycle	Work-item ID							
		w0	w1	w2	w3	w4	w5	w6	w7
y4.x	N	y4[w0].x	y4[w1].x	y4[w2].x	y4[w3].x	y4[w4].x	y4[w5].x	y4[w6].x	y4[w7].x
y4.y	N+1	y4[w0].y	y4[w1].y	y4[w2].y	y4[w3].y	y4[w4].y	y4[w5].y	y4[w6].y	y4[w7].y
y4.z	N+2	y4[w0].z	y4[w1].z	y4[w2].z	y4[w3].z	y4[w4].z	y4[w5].z	y4[w6].z	y4[w7].z
y4.w	N+3	y4[w0].w	y4[w1].w	y4[w2].w	y4[w3].w	y4[w4].w	y4[w5].w	y4[w6].w	y4[w7].w

图 11.9: 硬件向量指令访问间隔的 SIMD 内存位置

图 11-9 还演示了本节的第三个关键点，即向量的方便解释可能会产生内存访问的影响，理解这一点很重要。在前面的代码示例中，每个 Work-Items 都会看到 y4 的原始（连续）数据布局，它提供了一个直观的模型来推理和调整。

```
q.submit([&](sycl::handler &h) { // assume sub group size is 8
    // ...
    h.parallel_for(range<1>(8), [=](id<1> i) {
        // ...
        float4 y4 = b[i]; // i=0, 1, 2, ...
        // ...
        float x = dowork(&y4); // the "dowork" expects y4,
                               // i.e., vec_y[8][4] layout
    });
});
```

图 11.10: 具有地址转义的矢量代码示例

从性能角度来看，这种以 Work-Items 为中心的向量数据布局的缺点是，如果编译器跨 Work-Items 进行向量化以创建向量硬件指令，则向量硬件指令的通道不会访问连续的内存位置。取决于矢量数据大小和特定设备的功能；编译器可能需要生成、收集或分散内存指令；如图 11-10 所示。这是必需的，因为向量在内存中是连续的，并且相邻 Work-Items 并行地对不同向量进行操作。有关向量类型如何影响特定设备上的执行的更多讨论，请参阅第 15 章和第 16 章，并务必检查供应商文档、编译器优化报告并使用运行

时分析来了解特定场景的效率。

当编译器可以证明 y_4 的地址不会从当前 Kernel Work-Items 转义时，或者如果所有被调用函数都是内联的，则编译器可能会执行可能提高性能的积极优化。例如，如果 y_4 不可观察，编译器可以合法地转置 y_4 的存储，从而启用连续内存访问，从而避免需要收集或分散指令。编译器优化报告可以提供我们的源代码如何转换为向量硬件指令的信息，并可以提供有关如何调整代码以提高性能的提示。

作为一般准则，只要方便向量（例如，`marray`）具有逻辑意义，我们就应该使用它们，因为使用这些类型的代码更容易编写和维护。只有当我们在应用程序中看到性能热点时，我们才应该调查源代码向量操作是否已降低为次优硬件实现。

11.5.2 作为 SIMD 类型的向量

尽管我们在本章中强调了 `marray` 和 `vec` 不是 SIMD 类型，但为了完整起见，我们在这里简要讨论了 SIMD 类型如何映射到向量硬件。此讨论与我们的 SYCL 源代码中的向量无关，但提供了背景知识，当我们进入本书后面描述特定设备类型（GPU、CPU、FPGA）的章节时，这些背景知识将很有用，并且可能有助于我们为以下内容做好准备：SYCL 的未来版本中可能会引入 SIMD 类型。

SYCL 设备可能包含 SIMD 指令硬件，该硬件对一个向量寄存器或寄存器文件中包含的多个数据值进行操作。

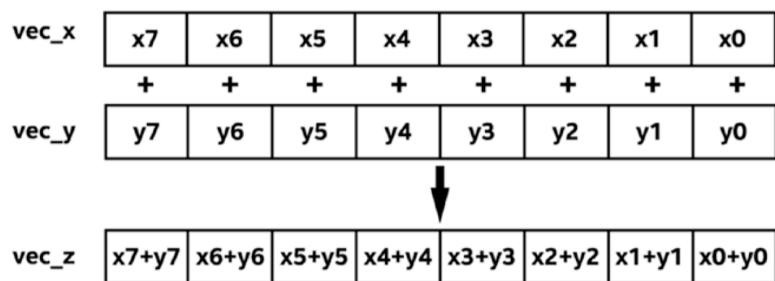


图 11.11: 具有八位数据并行性的 SIMD 添加

在提供 SIMD 硬件的设备上，我们可以考虑进行向量加法运算，例如，对八元素向量进行加法运算，如图 11-11 所示。

此示例中的向量加法可以使用向量硬件在单个指令中执行，与该 SIMD 指令并行添加向量寄存器 `vec_x` 和 `vec_y`。

SIMD 类型到向量硬件的这种映射非常简单且可预测，并且任何编译器都可能以相同的方式执行。这些属性使得 SIMD 类型对于 SIMD 硬件上的低级性能调整非常有吸引力，但也带来了成本——代码的可移植性较差，并且对特定架构的细节变得敏感。SPMD 编程模型的发展是为了应对这些成本。

开发人员期望 SIMD 类型具有可预测的硬件映射属性，这正是通过两种不同的语言功能干净地分离向量的两种解释至关重要的原因：如果开发人员使用一种便利类型，希望其表现得像 SIMD 类型，他们很可能会这样做正在反对编译器优化，并且可能会看到性能低于希望或预期的性能。

11.6 总结

编程语言中的术语向量有多种解释，在编写高性能和可扩展的代码时，理解特定语言或编译器所围绕的解释非常重要。SYCL 的构建理念是，源代码中的向量类型是 Work-Items 本地的便利类型，并且编译器跨 Work-Items 进行的隐式向量化映射到硬件中的 SIMD 指令。当我们（在极少数情况下）想要编写直接映射到矢量硬件的代码时，我们应该查看供应商文档，在某些情况下还应该查看 SYCL 的扩展。大多数应用程序应该假设 Kernel 将跨 Work-Items 进行矢量化，这样做会利用 SPMD 的强大抽象，它提供了易于推理的编程模型，并提供了跨设备和架构的可扩展性能。

本章描述了 `marray` 接口，当我们想要操作类似类型的数据分组（例如，具有多个颜色通道的像素）时，它提供了开箱即用的便利。此外，我们还讨论了遗留的 `vec` 类，它可以方便地表达某些模式（使用 swizzles）或优化（使用加载/存储和后端互操作性）。

12 设备信息和 *Kernel* 特化

在本章中，我们将探讨使我们的程序更加灵活并因此更加可移植的先进概念。这是通过查看与我们的应用程序可能在其上执行的任何系统（和加速器）的功能相匹配的机制以及我们编写的一系列 Kernel 和代码来完成的。这是一个高级主题，因为我们总是可以简单地“使用默认加速器”并运行我们在其上编写的 Kernel，无论它是什么。我们了解到，即使在没有加速器的系统上，这也可能工作，因为 SYCL 保证始终有一个可用的设备可以运行 Kernel，即使它是也在运行我们的主机应用程序的 CPU。

当我们超越“使用默认加速器”和通用 Kernel 时，我们发现可以使用机制来选择要使用的设备，以及创建更特化 Kernel 的机制。我们将在本章中讨论这两种功能。这两种功能共同使我们能够构建高度适应其执行系统的应用程序。

幸运的是，SYCL 规范的创建者考虑到了这些需求，并为我们提供了接口来让我们解决这个问题。SYCL 规范定义了一个设备类，该类封装了可以执行 Kernel 的设备。我们首先涵盖查询设备类别的能力，以便我们的程序能够适应设备的特性和功能。我们偶尔可能会选择为不同的设备编写不同的算法。在本章后面，我们了解到可以将 Aspect 应用到 Kernel 来特化 Kernel 并让编译器利用它。这种特化有助于使 Kernel 更适合某一类设备，同时可能使其不适合其他设备。结合这些概念，我们可以根据自己的意愿或多或少地调整我们的程序。这确保我们可以决定在从广泛的可移植性开始的同时，在挤出性能 Aspect 进行多少投资。

12.1 是否有 GPU?

我们中的许多人都会首先通过逻辑来弄清楚“是否存在 GPU?”告知我们的程序在执行时将做出的选择。这就是本章内容的开始。正如我们将看到的，有更多的信息可以帮助我们使我们的程序变得健壮和高性能。

注 36 对程序进行参数化有助于提高正确性、功能可移植性、性能可移植性和面向未来。

本章深入探讨最重要的查询以及如何在我们的程序中有效地使用它们。实现无疑提供了我们可以查询的更详细的属性。要了解所有可能的查询，我们需要查看最新的 SYCL 规范、特定编译器的文档以及我们可能遇到的任何运行时/驱动程序的文档。

可以使用 `get_info` 函数查询特定于设备的属性，包括访问特定于设备的 Kernel 和 Work-Groups 属性。

12.2 细化 Kernel 代码使其更加规范

考虑到我们的编码（逐个 Kernel）将大致分为以下三类之一是有用的：

- **通用 Kernel 代码**: 在任何地方运行，无需针对特定类别的设备进行调整。
- **设备类型特定的 Kernel 代码**: 在某种类型的设备（例如 GPU、CPU、FPGA）上运行，而不是针对设备类型的特定模型进行调整。这特别有用，因为许多设备类型共享共同的功能，因此可以安全地做出一些不适用于为所有设备编写的完全通用代码的假设。
- **调整的特定于设备的 Kernel 代码**: 在一种设备上运行，并根据设备的特定参数进行调整——这涵盖了从少量调整到非常详细的优化工作的广泛可能性。

注 37 作为程序员，我们的工作是确定何时需要不同的设备类型。我们用第 14、15、16 和 17 章来阐明这一重要思想。

最常见的做法是首先关注如何使用通用 Kernel 的功能正确的实现来工作。第 2 章特化讨论了在开始使用 Kernel 实现时哪些方法最容易调试。一旦 Kernel 开始工作，我们就可以对其进行改进以适应特定设备类型或设备型号的功能。

第 14 章提供了一个在我们深入考虑设备之前首先考虑并行性的思维框架。我们对模式（又名算法）的选择决定了我们的代码，而作为程序员，我们的工作就是确定不同设备何时需要不同的模式。第 15 章 (GPU)、第 16 章 (CPU) 和第 17 章 (FPGA) 更深入地探讨了区分这些设备类型并促使选择使用模式的品质。当最佳方法（模式选择）因不同设备类型而异时，正是这些品质促使我们考虑编写不同版本的 Kernel。

当我们为特定类型的设备（例如特定的 CPU、GPU、FPGA 等）编写 Kernel 时，将其适应特定供应商甚至此类设备的型号是合乎逻辑的。良好的编码风格是根据功能参数化代码（例如，从设备查询中找到的项目大小支持）。

我们应该编写代码来查询描述设备实际功能的参数，而不是其营销信息；查询设备的型号并对其做出反应是不好的编程习惯——这样的代码不太可移植，因为它不适合未来。

通常为我们想要支持的每种设备类型编写不同的 Kernel (GPU 版本的 Kernel 和 FPGA 版本的 Kernel，也许还有通用版本的 Kernel)。当我们变得更具体时，为了支持特定的设备供应商甚至设备模型，当我们可以参数化 Kernel 而不是复制它时，我们可能会受益。我们可以自由地选择我们认为合适的任何一个。因太多参数调整而混乱的代码可能难以阅读或在运行时负担过重。然而，参数可以整齐地适合单个版本的 Kernel 是很常见的。

注 38 当算法大致相同但已针对特定设备的功能进行调整时，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 *Kernel* 要干净得多。

12.3 如何枚举设备和功能

第 2 章列举并解释了选择执行设备的五种方法。本质上，方法 #1 是最不规范的，在某个地方运行它，我们发展到最具规范性的方法 #5，它考虑在一系列设备中的一个相当精确的设备模型上执行。介于两者之间的列举方法兼具灵活性和规范性。图 12-1、图 12-2 和图 12-4 帮助说明我们如何选择设备。

```
queue q;

std::cout << "By default, we are running on "
<< q.get_device().get_info<info::device::name>()
<< "\n";

Example Outputs (one line per run - depends on system):
By default, we are running on NVIDIA GeForce RTX 3060
By default, we are running on AMD Radeon RX 5700 XT
By default, we are running on Intel(R) UHD Graphics 770
By default, we are running on Intel(R) Xeon(R) Gold 6336Y CPU @ 2.40GHz
By default, we are running on Intel(R) Data Center GPU Max 1100
```

图 12.1: 默认情况下已为我们分配的设备

图 12-1 显示，即使我们允许实现为我们选择默认设备（第 2 章中的方法 #1），我们仍然可以查询有关所选设备的信息。

```

auto GPU_is_available = false;

try {
    device testForGPU(gpu_selector_v);
    GPU_is_available = true;
} catch (exception const& ex) {
    std::cout << "Caught this SYCL exception: " << ex.what()
        << std::endl;
}

auto q = GPU_is_available ? queue(gpu_selector_v)
                         : queue(default_selector_v);

std::cout
    << "After checking for a GPU, we are running on:\n "
    << q.get_device().get_info<info::device::name>()
    << "\n";

```

Four Example Outputs (using four different systems, each with a GPU):

After checking for a GPU, we are running on:
AMD Radeon RX 5700 XT

After checking for a GPU, we are running on:
Intel(R) Data Center GPU Max 1100

After checking for a GPU, we are running on:
NVIDIA GeForce RTX 3060

After checking for a GPU, we are running on:
Intel(R) UHD Graphics 770

Example Output (using a system without GPU):

Caught this SYCL exception: No device of requested type 'info::device_type::gpu' available.
...(PI_ERROR_DEVICE_NOT_FOUND)

After checking for a GPU, we are running on:
AMD Ryzen 5 3600 6-Core Processor

图 12.2: 如果可能, 请使用 *try-catch* 选择 GPU 设备, 如果没有, 请使用默认设备

图 12-2 显示了我们如何尝试使用特定设备（在本例中为 GPU）设置队列，但如果没有任何可用的 GPU，则显式回退到默认设备。这让我们能够在一定程度上控制设备的选择，只要有可用的 GPU，我们就会优先选择 GPU。我们知道至少有一个设备始终保证存在，因此我们的 Kernel 始终可以在正确配置的系统中运行。当没有 GPU 时，许多系统会默认使用 CPU 设备，但这并不能保证。同样，如果我们显式请求一个 CPU 设备，则不能保证存

在这样的设备（但我们保证某个设备将存在）。

不建议使用图 12-2 所示的方案。除了看起来有点可怕和容易出错之外，如果运行时可以选择 GPU，图 12-2 并不能让我们控制选择哪个 GPU。尽管既有指导意义又实用，但还有更好的方法。建议我们编写自定义设备选择器，如下一个代码示例（图 12-4）所示。

有关设备的查询依赖于已安装的软件（特殊的用户级驱动程序）来响应有关设备的信息。SYCL 依赖于此，就像操作系统需要驱动程序来访问硬件一样，仅将硬件安装在计算机中是不够的。

12.3.1 Aspect

Standard aspect (all booleans)	The device...
<code>aspect::cpu</code>	executes code on a CPU
<code>aspect::gpu</code>	executes code on a GPU
<code>aspect::accelerator</code>	executes code on an accelerator
<code>aspect::custom</code>	executes fixed functions only, no support for programmable kernels
<code>aspect::emulated</code>	executes code in an emulator, not for performance - typically used for debug profiling, etc.
<code>aspect::host_debuggable</code>	can fully support standard debugging
<code>aspect::fp16</code>	supports the <code>sycl::half</code> data type
<code>aspect::fp64</code>	supports the <code>double</code> data type
<code>aspect::atomic64</code>	supports 64-bit atomic operations
<code>aspect::image</code>	supports images, a topic not covered in this book (we emphasize the more general and portable <code>buffer</code> instead)
<code>aspect::online_compiler</code> <code>aspect::online_linker</code>	supports online compilation and/or linking of device code. Such devices may support the <code>build()</code> , <code>compile()</code> , and <code>link()</code> functions, all very advanced topics not covered in this book
<code>aspect::queue_profiling</code>	supports queue profiling, an advanced topic discussed a bit, along with other practical tips, in Chapter 13
<code>aspect::usm_device_allocations</code> <code>aspect::usm_host_allocations</code> <code>aspect::usm_atomic_host_allocations</code> <code>aspect::usm_shared_allocations</code> <code>aspect::usm_atomic_shared_allocations</code>	supports the corresponding USM capability
<code>aspect::usm_system_allocations</code>	supports sharing data allocated by the system allocators, not just the SYCL USM allocation calls; such usage will impact portability and may impact performance

图 12.3: SYCL 标准定义的 Aspect (实现可以添加更多)

SYCL 标准有一个设备 Aspect 的小列表，可用于了解设备的功能、控制我们选择使用哪些设备以及控制我们向设备提交哪些 Kernel。在本章的最后，我们将讨论“Kernel 特化”和 Kernel 模板化。现在，我们将列举这些 Aspect 以及如何在设备查询和选择中使用它们。图 12-3 列出了 SYCL 标准定义的 Aspect，可用于每个使用 SYCL 的 C++ 程序。Aspect 是布尔值——设备要么有 Aspect，要么没有 Aspect。前四个 (cpu/gpu/加速器/自定

义) 是互斥的, 因为设备类型被 SYCL 2020 定义为枚举。包括 aspect::fp16、aspect::fp64 和 aspect::atomic64 在内的功能是“可选功能”, 因此它们可能不受所有设备的支持 - 对这些设备的测试对于强大的应用程序尤其重要。

12.3.2 自定义设备选择器

```

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

int my_selector(const device& dev) {
    int score = -1;

    // We prefer non-Martian GPUs, especially ACME GPUs
    if (dev.is_gpu()) {
        if (dev.get_info<info::device::vendor>().find("ACME") !=
            std::string::npos)
            score += 25;

        if (dev.get_info<info::device::vendor>().find(
            "Martian") == std::string::npos)
            score += 800;
    }

    // If there is no GPU on the system all devices will be
    // given a negative score and the selector will not select
    // a device. This will cause an exception.
    return score;
}

int main() {
    try {
        auto q = queue{my_selector};
        std::cout
            << "After checking for a GPU, we are running on:\n "
            << q.get_device().get_info<info::device::name>()
            << "\n";
    } catch (exception const& ex) {
        std::cout << "Custom device selector did not select a "
                "device.\n";
        std::cout << "Caught this SYCL exception: " << ex.what()
                << std::endl;
    }

    return 0;
}
Four Example Outputs (using four different
systems, each with a GPU):
After checking for a GPU, we are running on:
Intel(R) Gen9 HD Graphics NEO.
After checking for a GPU, we are running on:
NVIDIA GeForce RTX 3060
After checking for a GPU, we are running on:
Intel(R) Data Center GPU Max 1100
After checking for a GPU, we are running on:
AMD Radeon RX 5700 XT

Example Output (using a system without GPU):
After checking for a GPU, we are running on:
Custom device selector did not select a device.
Caught this SYCL exception: No device of requested
type available. ... (PI_ERROR_DEVICE_NOT_FOUND)

```

图 12.4: 自定义设备选择器—我们的首选解决方案

图 12-4 使用自定义设备选择器。自定义设备选择器首先在第 2 章中作为 Method#5 讨论，用于选择代码运行的位置（图 2-16）。自定义设备选择器评估应用程序可用的每个设备。根据获得的最高分数来选择特定设备（如果最高分数为 -1，则不选择任何设备）。在这个例子中，我们将享受我们的选择器带来的一些乐趣：

- 拒绝非 GPU（返回-1）。
- 优先选择供应商名称中包含“ACME”一词的 GPU（如果是 Martian，则返回 24，否则返回 824）。
- 任何其他非火星 GPU 都是不错的选择（返回 799）。
- 不是 ACME 的 Martian GPU 将被拒绝（返回-1）。

下一节“好奇：get_info<>”将深入探讨 get_devices()、get_platforms() 和 get_info<> 提供的丰富信息。这些接口打开了我们可能想要用来选择设备的任何类型的逻辑，包括图 2-16 和图 12-4 中所示的简单供应商名称检查。

12.3.3 好奇：get_info<>

为了让我们的程序在运行时“知道”哪些设备可用，我们可以让程序从设备类中查询可用设备，然后我们可以使用 get_info<> 查询特定设备来了解更多详细信息。我们提供了一个简单的程序，称为好奇（见图 12-5），它使用这些接口转储信息供我们直接查看。这对于在开发或调试使用这些接口的程序时进行健全性检查特别有用。如果该程序无法按预期工作，通常表明我们需要的软件驱动程序未正确安装。图 12-6 显示了该程序的示例输出，其中包含有关现有设备的高级信息。

注 39 在编写自己的“列出所有可用的 SYCl 设备”程序之前，您可能希望查看您的系统是否支持 *sycl-ls* 等实用程序。

```
// Loop through available platforms
for (auto const& this_platform :
    platform::get_platforms()) {
    std::cout
        << "Found platform: "
        << this_platform.get_info<info::platform::name>()
        << "\n";

    // Loop through available devices in this platform
    for (auto const& this_device :
        this_platform.get_devices()) {
        std::cout
            << " Device: "
            << this_device.get_info<info::device::name>()
            << "\n";
    }
    std::cout << "\n";
}
```

图 12.5: 设备查询机制的简单使用: *curious.cpp*

```
% clang++ -fsycl fig_12_5_curious.cpp -o curious
% ./curious
Found platform: NVIDIA CUDA BACKEND
Device: NVIDIA GeForce RTX 3060

Found platform: AMD HIP BACKEND
Device: AMD Radeon RX 5700 XT

Found platform: Intel(R) OpenCL
Device: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz

Found platform: Intel(R) OpenCL HD Graphics
Device: Intel(R) UHD Graphics P630 [0x3e96]

Found platform: Intel(R) Level-Zero
Device: Intel(R) UHD Graphics P630 [0x3e96]

Found platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
Device: Intel(R) FPGA Emulation Device
```

图 12.6: *curious.cpp* 的输出示例

12.3.4 更好奇：详细的枚举代码

```

template <typename queryT, typename T>
void do_query(const T& obj_to_query,
              const std::string& name, int indent = 4) {
    std::cout << std::string(indent, ' ') << name << " is "
              << obj_to_query.template get_info<queryT>()
              << '\n';
}

int main() {
    // Loop through the available platforms
    for (auto const& this_platform :
        platform::get_platforms()) {
        std::cout << "Found Platform:\n";
        do_query<info::platform::name>(this_platform,
                                         "info::platform::name");
        // query information like these (more in program than
        // shown here in this figure - see book github)

        // Loop through the devices available in this platform
        for (auto& dev : this_platform.get_devices()) {
            std::cout << " Device: "
                  << dev.get_info<info::device::name>()
                  << "\n";
            // is_cpu() == has(aspect::cpu)
            std::cout << " is_cpu(): "
                  << (dev.is_cpu() ? "Yes" : "No") << "\n";
            // is_cpu() == has(aspect::gpu)
            std::cout << " is_gpu(): "
                  << (dev.is_gpu() ? "Yes" : "No") << "\n";
            std::cout << " has(fp16): "
                  << (dev.has(aspect::fp16) ? "Yes" : "No")
                  << "\n";
            // many more queries shown in fig_12_7_very_curious.cpp
            // see book github for source code
        }
        std::cout << "\n";
    }
    return 0;
}

```

图 12.7: 设备查询机制的更详细使用: *verycurious.cpp* (显示的子集)

我们提供了一个程序，我们将其命名为 *verycurious.cpp*（图 12-7），来说明使用 `get_info` 可以获得的一些详细信息。我们再次发现自己编写这样的代码来帮助开发或调试程序。

现在我们已经展示了如何访问信息，我们将讨论在应用程序中查询和

操作最重要的信息字段。

12.3.5 非常好奇：get_info 加上 has()

has() 接口允许程序使用图 12-3 中列出的 Aspect 直接测试某个功能。简单的用法如图 12-7 所示，更多内容请参见 GitHub 书中完整的 verycurious.cpp 源代码。verycurious.cpp 程序有助于查看系统上设备的详细信息。

12.4 设备信息描述符

本章前面使用的“好奇”和“非常好奇”程序示例利用流行的 SYCL 设备类成员函数（即 is_cpu、is_gpu、is_accelerator、get_info、has）。这些成员函数记录在 SYCL 规范中标题为“SYCL 设备类的成员函数”的表中。

“好奇”的程序示例还使用 get_info 成员函数查询信息。所有 SYCL 设备都必须支持一组查询。SYCL 规范中标题为“设备信息描述符”的表中描述了此类项目的完整列表。

12.5 设备特定的 Kernel 信息描述符

与平台和设备一样，我们可以使用 get_info 函数查询有关 Kernel 的信息。此类信息（例如，支持的 Work-Groups 大小、首选 Work-Groups 大小、每个工作项所需的私有内存量）可能是特定于设备的，因此 Kernel 类的 get_info 成员函数接受设备作为参数。

12.6 细节：“正确性”的细节

我们将细节分为有关必要条件（正确性）的信息和对调整有用但对正确性不是必需的信息。

```

queue q;
device dev = q.get_device();

std::cout << "We are running on:\n"
           << dev.get_info<info::device::name>() << "\n";

// Query results like the following can be used to
// calculate how large your kernel invocations can be.
auto maxWG =
    dev.get_info<info::device::max_work_group_size>();
auto maxGmem =
    dev.get_info<info::device::global_mem_size>();
auto maxLmem =
    dev.get_info<info::device::local_mem_size>();

std::cout << "Max WG size is " << maxWG
           << "\nGlobal memory size is " << maxGmem
           << "\nLocal memory size is " << maxLmem << "\n";

```

图 12.8: 获取可用于塑造 Kernel 的参数

注 40 提交违反必需条件（例如 `sub_group_sizes`）的 Kernel 将生成运行时错误。

在第一个正确性类别中，我们将列举 Kernel 正确启动应满足的条件。不遵守这些设备限制将导致程序失败。图 12-8 显示了我们如何获取其中一些参数，使这些值可用于主机代码和 Kernel 代码（通过 lambda 捕获）。我们可以修改我们的代码以利用这些信息；例如，它可以指导我们的代码确定缓冲区大小或 Work-Groups 大小。

12.6.1 设备查询

`device_type`: `cpu`、`gpu`、加速器、自定义^{12.1}、自动、全部。这些最常通过 `is_cpu`、`is_gpu()` 等进行测试（参见图 12-7）：

`max_work_item_sizes`: `nd_range` 的 Work-Groups 的每个维度中允许的最大工作项数。最小值为 `(1, 1, 1)`。

`max_work_group_size`: 在单个计算单元上执行 Kernel 的 Work-Groups 中允许的最大工作项数。最小值为 1。

^{12.1} 本书不讨论自定义设备（不要将“自定义设备”与“自定义设备选择器”混淆）。如果我们发现自己使用自定义类型对标识自己的设备进行编程，我们将需要研究该设备的文档以了解更多信息。说得不那么委婉：定制设备并不常见且很奇怪，所以我们不打算谈论它们——我们故意忽略了它们可能对我们讨论的某些功能施加的限制。

global_mem_size: 全局内存的大小 (以字节为单位)。

local_mem_size: 本地内存的大小 (以字节为单位)。最小大小为 32 K。

max_compute_units: 指示设备上可用的并行量 - 实现定义的, 请小心解释!

sub_group_sizes: 返回设备支持的子组大小集。

请注意, 还有更多特性被编码为 Aspect (参见图 12-3), 例如 USM 功能。

注 41 (我们强烈建议避免在程序逻辑中 MAX_COMPUTE_UNITS)

我们发现, 应该避免查询最大数量的计算单元, 部分原因是定义不够清晰, 无法在代码优化中发挥作用。大多数程序应该表达它们的并行性, 并让运行时将其映射到可用的并行性上, 而不是使用 *max_compute_units*。依靠 *max_compute_units* 的正确性只有在使用特定于实现和设备的信息进行增强时才有意义。专家可能会这样做, 但大多数开发人员不会也不需要这样做! 在这种情况下, 让运行时完成它的工作!

12.6.2 Kernel 查询

执行这些 Kernel 查询需要第 10 章“Kernel 绑定中的 Kernel”下讨论的机制:

work_group_size: 返回可用于在特定设备上执行 Kernel 的最大 Work-Groups 大小

compile_work_group_size: 返回 Kernel 指定的 Work-Groups 大小(如果适用); 否则返回 (0, 0, 0)

compile_sub_group_size: 返回 Kernel 指定的子组大小(如果适用); 否则返回 0

compile_num_sub_groups: 返回 Kernel 指定的子组数量(如果适用); 否则返回 0

max_sub_group_size: 返回使用指定 Work-Groups 大小启动的 Kernel 的最大子组大小

max_num_sub_groups: 返回 Kernel 的最大子组数

12.7 具体内容：“调整/优化”的具体内容

有一些额外的参数可以被视为我们 Kernel 的微调参数。可以忽略这些，而不会危及程序的正确性。这些使我们的 Kernel 能够真正利用硬件的细节来提高性能。

注 42 在优化缓存（如果存在）时，注意这些查询的结果会有所帮助。

12.7.1 设备查询

`global_mem_cache_line_size`: 全局内存缓存行的大小（以字节为单位）。

`global_mem_cache_size`: 全局内存缓存的大小（以字节为单位）。

`local_mem_type`: 支持的本地内存类型。这可以是 `info::local_mem_type::local` 表示特化本地内存存储，例如 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，可能没有性能提升。

12.7.2 Kernel 查询

`Preferred_work_group_size`: 在特定设备上执行 Kernel 的首选 Work-Groups 大小。

`Preferred_work_group_size_multiple`: Work-Groups 大小应是此值 (`preferred_work_group_size_multiple`) 的倍数，以便在特定设备上执行 Kernel 以获得最佳性能。该值不得大于 `work_group_size`。

12.8 运行时与编译时属性

实现可能提供编译时常量/宏或其他功能，但它们不是标准的，因此我们不鼓励使用它们，也不会在本书中讨论它们。本章中描述的查询是通过运行时 API (`get_info`) 执行的，因此直到运行时才知道结果。在下一节中，我们将讨论如何使用属性来控制 Kernel 的编译方式。除了属性之外，SYCL 标准仅提倡使用运行时信息，但有一个相当深奥的例外。SYCL 确实提供了应用程序可用于在编译时查询 Aspect 的两个特征。这些特征特化用于帮助避免为任何设备不支持的设备功能实例化模板化 Kernel。这是一个非常高级且很少使用的功能，我们在本书中不会详细说明。SYCL 标准在“设备 Aspect”部分

末尾有一个示例，该示例展示了为此目的使用 `any_device_has_v<aspect>` 和 `all_devices_have_v<aspect>`。该标准还定义了“特化常量”，我们在本书中不讨论它们，因为它们通常用于非常高级的目标开发，例如在库中。结语中“编译时属性”下讨论了实验性编译时属性扩展。

12.9 Kernel 特化

我们可以通过针对不同用途使用不同的 Kernel 来特化我们的 Kernel，并根据我们目标设备的各个 Aspect（参见图 12-3）选择适当的 Kernel。当然，我们可以显式编写特化的 Kernel 并使用 C++ 模板来提供帮助。我们可以通过使用 SYCL 属性（图 12-9）和 Aspect（图 12-3）来通知编译器我们希望 Kernel 使用特定功能。

Standard attribute	Specifies
<code>device_has(aspect, ...)</code>	This attribute is the only attribute that can be used to decorate a non-kernel function, in addition to the ability (of all attributes) to decorate a kernel function. Requires: that the kernel is only launched with devices meeting the specified aspect(s) from Figure 12-3.
<code>reqd_work_group_size(dim0)</code> <code>reqd_work_group_size(dim0, dim1)</code> <code>reqd_work_group_size(dim0, dim1, dim2)</code>	Requires: that the kernel <i>must</i> be launched with the specified workgroup size.
<code>work_group_size_hint(dim0)</code> <code>work_group_size_hint(dim0, dim1)</code> <code>work_group_size_hint(dim0, dim1, dim2)</code>	Hints: that the kernel <i>will most likely</i> be launched with the specified workgroup size.
<code>reqd_sub_group_size(dim)</code>	Requires: that the kernel must be compiled and executed with the specified sub-group size.

图 12.9: 由 SYCL 标准定义的属性（未弃用）

```

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

int main() {
    queue q;

    constexpr int size = 16;
    std::array<double, size> data;

    // Using "sycl::device_has()" as an attribute does not
    // affect the device we select. Therefore, our host code
    // should check the device's aspects before submitting a
    // kernel which does require that attribute.
    if (q.get_device().has(aspect::fp64)) {
        buffer B{data};
        q.submit([&](handler& h) {
            accessor A{B, h};
            // the attributes here say that the kernel is allowed
            // to require fp64 support any attribute(s) from
            // Figure 12-3 could be specified note that namespace
            // stmt above (for C++) does not affect attributes (a
            // C++ quirk) so sycl:: is needed here
            h.parallel_for(
                size, [=](auto& idx)
                    [[sycl::device_has(aspect::fp64)]]
                {
                    A[idx] = idx * 2.0;
                });
        });
        std::cout << "doubles were used\n";
    } else {
        // here we use an alternate method (not needing double
        // math support on the device) to help our code be
        // flexible and hence more portable
        std::array<float, size> fdata;
        {
            buffer B{fdata};
            q.submit([&](handler& h) {
                accessor A{B, h};
                h.parallel_for(
                    size, [=](auto& idx) { A[idx] = idx * 2.0f; });
            });
        }
        for (int i = 0; i < size; i++) data[i] = fdata[i];
        std::cout << "no doubles used\n";
    }
    for (int i = 0; i < size; i++)
        std::cout << "data[" << i << "] = " << data[i] << "\n";
    return 0;
}

```

图 12.10: 借助属性显式 Kernel 特化

例如，`reqd_work_group_size` 属性（图 12-9）可用于要求 Kernel 的特定 Work-Groups 大小，而 `device_has` 属性可用于要求 Kernel 的特定设备 Aspect。

使用属性有两个作用：

1. 如果 Kernel 提交到不具有所列 Aspect 之一的设备，则 Kernel 将引发异常。
2. 如果 Kernel（或其调用的任何函数）使用与属性中未列出的 Aspect 关联的可选功能（例如，fp16），编译器将发出诊断信息。

第一个有助于防止应用程序在可能失败的情况下继续运行，第二个有助于在编译时捕获错误。由于这些原因，使用属性会很有帮助。

图 12-10 提供了一个示例，该示例使用运行时逻辑在两个代码序列之间进行选择，并使用属性来特化其中一个 Kernel。

12.10 总结

最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其行为。本章打开了获取丰富信息的大门，这些信息可用于允许对我们的代码进行此类定制以适应运行时存在的硬件。我们还讨论了特化 Kernel 的各种方法，以便当我们认为投资值得时，它们可以更紧密地适应特定的设备类型。这些为我们提供了必要的工具来平衡可移植性和性能以满足我们的需求，所有这些都在使用 C++ 和 SYCL 的范围内。

通过对我们的应用程序进行参数化以适应硬件的特性，我们的程序可以在功能上更加便携，在性能上更加便携，并且更加面向未来。我们还可以测试当前的硬件是否在我们在程序设计中所做的任何假设的范围内，并且当发现硬件超出我们的假设范围时发出警告或中止。

13 实用技巧

本章包含许多有用的信息、实用技巧、建议和技术，这些信息在使用 SYCL 进行 C++ 编程时已被证明非常有用。这些主题都没有被详尽地涵盖，因此目的是提高认识并鼓励根据需要学习更多内容。

13.1 获取代码示例和编译器

第 1 章介绍如何获取 SYCL 编译器（例如 oneapi.com/implementations 或 github.com/intel/llvm）以及从何处获取本书中使用的代码示例(github.com/Apress/data-parallel-CPP)。再次提到这一点是为了强调尝试示例（包括进行修改！）以获得实践经验是多么有用。加入那些知道图 1-1 中的代码实际打印出什么内容的人吧！

13.2 在线资源

主要在线资源包括

- sycl.tech/ 上的丰富资源
- 官方 SYCL 主页位于 khronos.org/sycl/, 其中列出了 khronos.org/sycl/resources 上的大量资源
- 帮助使用 SYCL 从 CUDA 迁移到 C++ 的资源, 位于 tinyurl.com/cuda2sycl
- 迁移工具 GitHub 主页 github.com/oneapi-src/SYCLomatic

13.3 平台模型

支持 SYCL 的 C++ 编译器的设计方式和感觉与我们曾经使用过的任何其他 C++ 编译器一样。值得深入了解内部工作原理，使具有 SYCL 支持的编译器能够为主机（例如 CPU）和设备生成代码。

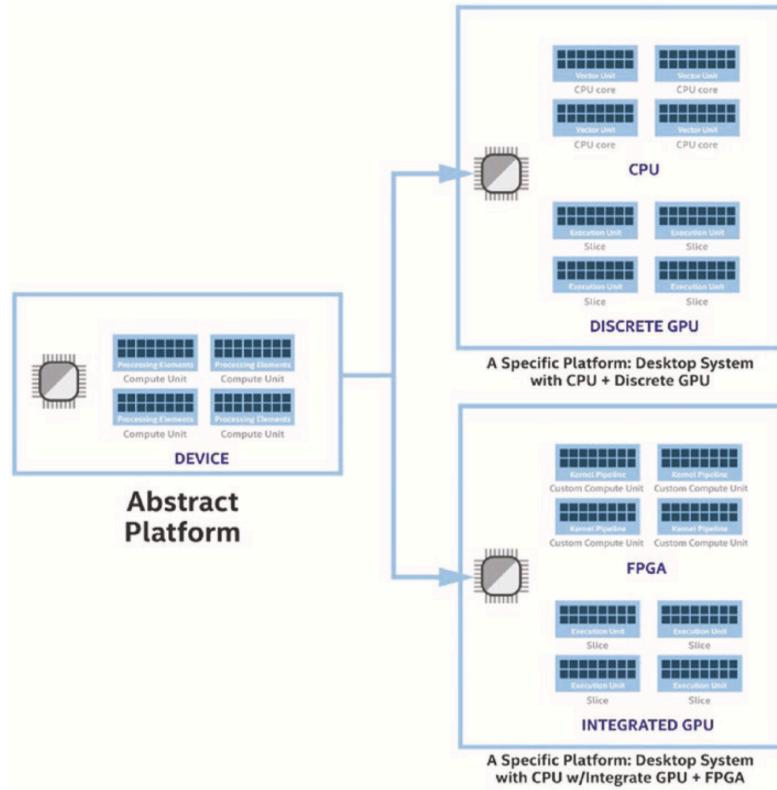


图 13.1: 平台模型：可以抽象使用，也可以具体使用

SYCL 使用的平台模型（图 13-1）指定了一个主机，用于协调和控制在设备上执行的计算工作。第 2 章介绍如何向设备分配工作，第 4 章深入介绍如何对设备进行编程。第 12 章描述了在各个具体级别上使用平台模型。

正如我们在第 2 章中讨论的，使用正确配置的 SYCL 运行时和兼容硬件的系统中应该始终有一个设备可以运行。这允许在假设至少有一个设备可用的情况下编写设备代码。运行设备代码的设备的选择是在程序控制下的——作为程序员，我们是否想要以及如何在特定设备上执行代码完全是我们自己的选择（设备选择选项将在第 12 章中讨论）。

13.3.1 多架构二进制文件

由于我们的目标是拥有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。

多架构二进制文件（又名胖二进制文件）是一个单一的二进制文件，它已扩展为包含我们的异构机器所需的所有已编译代码和中间代码。多架构二进制文件的行为与我们习惯的任何其他 a.out 或 a.exe 类似，但它包含异构计算机所需的所有内容。这有助于自动选择为特定设备运行的正确代码的过程。正如我们接下来讨论的，胖二进制文件中设备代码的一种可能形式是中间格式，它将设备指令的最终创建推迟到运行时。

13.3.2 编译模型

SYCL 的单一源性质允许编译的行为和感觉就像常规 C++ 编译一样。我们不需要为设备调用额外的通道或处理捆绑设备和主机代码。这一切都是由编译器自动为我们处理的。当然，出于多种原因，了解正在发生的事情的细节可能很重要。如果我们想要更有效地针对特定体系结构，这是有用的知识，并且了解我们是否需要调试编译过程中发生的故障也很重要。

我们将审查编译模型，以便我们在需要这些知识时接受教育。由于编译模型支持同时在主机和潜在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（仅针对一种体系结构）更复杂。欢迎来到异构世界！

这种异构的复杂性被编译器故意隐藏起来，并且“正常工作”。

编译器可以生成类似于传统 C++ 编译器的特定于目标的可执行代码（提前（AOT）编译，有时称为离线 Kernel 编译），也可以生成可以即时的中间表示（JIT）在运行时编译为特定目标。

注 43 编译可以是“提前”（AOT）或“即时”（JIT）。

只有提前知道设备目标（在我们编译程序时），编译器才能提前编译。使用 JIT 编译将为我们编译的程序提供更多的可移植性，但需要编译器和运行时在我们的应用程序运行时执行额外的工作。

对于大多数设备，包括 GPU，最常见的做法是依赖 JIT 编译。某些设备（例如 FPGA）的编译过程可能非常慢，因此实践是使用 AOT 编译。

注 44 除非您知道使用 AOT 代码有需要（例如，FPGA）或好处，否则请使用 JIT。

默认情况下，当我们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，系统上的设备驱动程序将及时将中间形式编译为可以在设备上运行，以匹配系统上可用的内容。

注 45 与 AOT 代码不同，JIT 代码的目标是能够在运行时进行编译，以使用系统上的任何设备。这可能包括最初将程序编译为 JIT 代码时不存在的设备。

我们可以要求编译器针对特定设备或设备类别提前进行编译。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使二进制文件变得更胖！提前编译的代码不如即时编译的代码那么可移植，因为它无法在运行时进行调整以匹配可用的硬件。我们可以将两者都包含在我们的二进制文件中，以获得 AOT 和 JIT 的好处。

注 46 为了最大限度地提高可移植性，即使包含一些 AOT 代码，我们也喜欢在我们的二进制文件中加入 JIT 代码。

提前针对特定设备进行编译还可以帮助我们在构建时检查我们的程序是否应该在该设备上运行。使用即时编译，程序可能会在运行时编译失败（可以使用第 5 章中的机制捕获）。在本章即将到来的“调试”部分中有一些调试技巧，第 5 章详细介绍了如何在运行时捕获这些错误，以避免要求我们的应用程序中止。

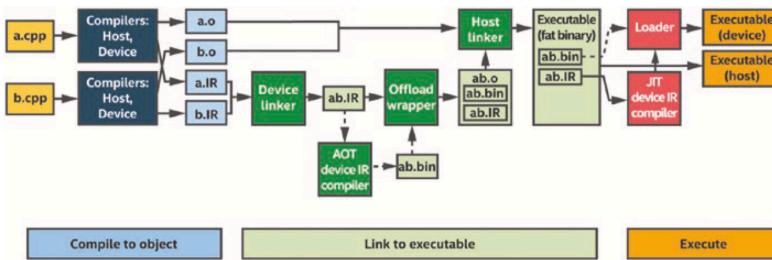


图 13.2: 编译过程：JIT 和 AOT 选项

图 13-2 说明了从源代码到 fat 二进制文件（可执行文件）的编译过程。无论我们选择什么组合，都会组合成一个胖二进制文件。当应用程序执行时，运行时会使用 fat 二进制文件（它是在主机上执行的二进制文件！）。有时，我们可能希望在单独的编译中为特定设备编译设备代码。我们希望这样一个单独编译的结果最终能够合并到我们的胖二进制文件中。当完整编译（进行完整综合布局布线）时间可能非常长时，这对于 FPGA 开发非常有用，而且实际上这是 FPGA 开发的一项要求，以避免需要在运行时系统

上安装综合工具。图 13-3 显示了支持此类需求的捆绑/分拆活动的流程。我们总是可以选择一次编译所有内容，但在开发过程中，中断编译的选项可能非常有用。

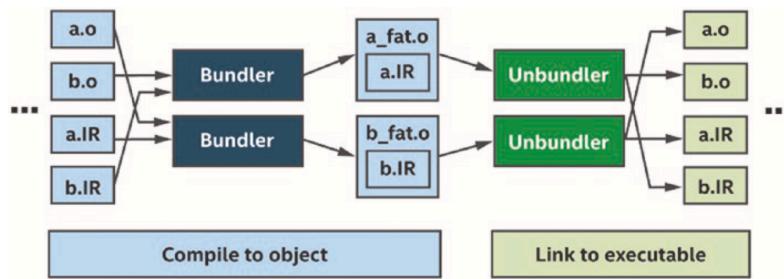


图 13.3: 编译过程: 卸载 *bundler/unbundler*

每个支持 SYCL 的 C++ 编译器都有一个具有相同目标的编译模型，但具体的实现细节会有所不同。此处显示的具体图表由 DPC++ 编译器工具链实现者提供。

13.4 上下文: 需要了解的重要事项

正如第 6 章中提到的，上下文代表我们可以在其上执行 Kernel 的一个设备或一组设备。我们可以将上下文视为运行时存储有关其正在执行的操作的某些状态的方便位置。除了在大多数 SYCL 程序中传递上下文之外，程序员不太可能直接与上下文交互。

设备可以细分为子设备。这对于划分问题很有用。由于子设备被完全视为设备（相同的 C++ 类型），因此我们所说的有关分组设备的所有内容也适用于子设备。

SYCL 抽象地认为设备在平台中分组在一起。在平台内，设备可以通过共享内存等方式进行交互。属于同一上下文的设备必须能够使用某种机制访问彼此的全局内存。仅当设备处于同一上下文中时，才能在设备之间共享 SYCL USM 内存（第 6 章）。USM 内存分配绑定到上下文，而不是设备，因此一个上下文中的 USM 分配无法被其他上下文访问。因此，USM 分配仅限于在单个上下文（可能是设备的子集）内使用。

上下文并不抽象硬件不能支持的内容。例如，我们无法创建一个上下文来包含两个不能彼此共享内存的 GPU。并非所有从同一平台公开的设备都

需要能够在同一上下文中分组在一起。

当我们创建队列时，我们可以指定我们希望将其放置在哪个上下文中。默认情况下，DPC++ 编译器项目为每个平台实现默认上下文，并自动将新队列分配给默认上下文。其他 SYCL 编译器可以自由地执行相同的操作，但标准并不要求这样做。

注 47 上下文的创建成本很高，而创建上下文的成本越低，我们的应用程序就越高效。

将给定平台的所有设备始终放置在同一上下文中具有两个优点：(1) 由于创建上下文的成本很高，因此我们的应用程序更加高效；(2) 允许硬件支持的最大共享（例如 USM）。

13.5 将 SYCL 添加到现有 C++ 程序

将并行性的适当利用添加到现有 C++ 程序中是使用 SYCL 的第一步。如果 C++ 应用程序已经在利用并行执行，这可能是一个好处，也可能是一个令人头疼的问题。这是因为我们将应用程序的工作划分为并行执行的方式极大地影响了我们可以用它做什么。当程序员谈论重构程序并行性时，他们指的是重新安排程序内的执行流和数据流，以使其准备好利用并行性。这是一个复杂的话题，我们只会简单地讨论一下。关于如何准备并行性应用程序，没有一刀切的答案，但有一些值得注意的技巧。

当向 C++ 应用程序添加并行性时，考虑的一个简单方法是在程序中找到并行机会最大的孤立点。我们可以从那里开始修改，然后根据需要继续在其他区域添加并行性。一个复杂的因素是重构（即重新安排程序流程和重新设计数据结构）可能会提高并行性的机会。

一旦我们在程序中找到并行机会最大的孤立点，我们就需要考虑如何在程序中的该点使用 SYCL。这就是本书其余部分所教导的。

从高层次来看，引入并行性的关键步骤包括以下内容：

1. 并发安全（在传统 CPU 编程中通常称为线程安全）：调整所有共享可变数据（可以更改并可能同时执行操作的数据）的使用，以防止数据竞争。参见第 19 章。
2. 引入并发和/或并行性。
3. 并行性调整（最佳扩展、吞吐量或延迟优化）。

首先考虑步骤 #1 很重要。许多应用程序已经针对并发性进行了重构，但许多应用程序还没有。将 SYCL 作为并行性的唯一来源，我们重点关注 Kernel 中使用的数据以及可能与主机共享的数据的安全性。如果我们的程序中有其他引入并行性的技术（OpenMP、MPI、TBB 等），那么这就是我们 SYCL 编程之外的另一个问题。值得注意的是，可以在单个程序中使用多种技术。SYCL 不需要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。

13.6 使用多个编译器时的注意事项

支持 SYCL 的 C++ 编译器还支持与其他 C++ 编译器的目标代码（库、目标文件等）链接。一般来说，使用多个编译器出现的任何问题都与任何 C++ 编译器相同，需要考虑名称修改、针对相同的标准库、对齐调用约定等。这些是我们在混合和使用时必须处理的相同问题。匹配其他语言（例如 Fortran 或 C）的编译器。

此外，应用程序必须使用用于构建程序的编译器附带的 SYCL 运行时。混合和匹配 SYCL 编译器和 SYCL 运行时并不安全 - 不同的运行时对于重要的 SYCL 对象可能有不同的实现和数据布局。

注 48 *SYCL 与非 SYCL 源语言的互操作性是指 SYCL 能够使用用其他编程语言（如 OpenCl、C 或 CUDA）编写的 Kernel 函数或设备函数，或者使用由其他编译器预编译的中间表示形式的代码。有关与非 SYCL 源语言的互操作性的更多信息，请参阅第 20 章。*

最后，还需要使用用于编译 SYCL 设备代码的相同编译器工具链来完成编译的链接阶段。使用来自不同编译器工具链的链接器进行链接不会产生功能性程序，因为不支持 SYCL 的编译器将不知道如何正确集成主机和设备代码。

13.7 调试

本节传达了一些适度的调试建议，以缓解调试并行程序所特有的挑战，尤其是针对异构机器的并行程序。

我们永远不应该忘记，当应用程序在 CPU 设备上运行时，我们可以选择对其进行调试。该调试技巧在第 2 章中被描述为方法 #2。由于设备的体系结构通常比通用 CPU 包含更少的调试挂钩，因此工具通常可以更精确地

探测 CPU 上的代码。在 CPU 上运行所有内容时的一个重要区别是，许多与同步相关的错误将会消失，包括在主机和设备之间来回移动内存。虽然我们最终需要调试所有此类错误，但这可以允许增量调试，因此我们可以在其他错误之前解决一些错误。经验表明，尽可能频繁地在我们的目标设备上运行非常重要，就像在调试过程中利用 CPU（和其他设备）的可移植性一样，运行多个设备将有助于暴露问题，并有助于隔离是否存在问题。我们遇到的错误是特定于设备的。

注 49 (调试提示) 在 CPU 上运行是一个功能强大的调试工具。

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，特别是数据争用和死锁。令我们懊恼的是，当在主机和设备的组合上运行时，我们经常会看到由于此类并行编程错误而导致的程序失败。当出现此类问题时，记住退回到仅 CPU 是一个强大的调试工具，这一点非常有用。值得庆幸的是，SYCL 经过精心设计，让我们可以轻松访问此选项。

注 50 (调试提示) 如果程序处于死锁状态，请检查主机访问器是否被正确销毁，以及 *Kernel* 中的工作项是否遵循 SYCL 规范中的同步规则。

当我们开始调试时，以下编译器选项是一个好主意：

- `-g`: 将调试信息放入输出中
- `-ferror-limit=1`: 在将 C++ 与模板库（例如 SYCL 大量使用的模板库）一起使用时保持理智
- `-Werror -Wall -Wpedantic`: 让编译器强制执行良好的编码，以帮助避免生成错误的代码以在运行时进行调试

我们确实不需要仅仅为了将 C++ 与 SYCL 一起使用而陷入修复迂腐警告的困境，因此选择不使用 `-Wpedantic` 是可以理解的。

当我们让代码在运行时即时编译时，我们可以检查一些代码。这高度依赖于我们的编译器使用的层，因此查看编译器文档以获取建议是一个好主意。

13.7.1 调试死锁和其他同步问题

并行编程依赖于并行发生的工作之间的适当协调。数据使用需要在数据准备好使用时进行控制——这种数据依赖关系需要在我们的程序逻辑中进行编码以获得正确的行为。

当我们的同步/依赖逻辑发生错误时，调试依赖问题（尤其是 USM）可能是一个挑战。我们可能会看到程序挂起（永远不会完成）或间歇性地生成错误信息。在这种情况下，我们可能会看到诸如“它会失败，直到我在调试器中运行它，然后它才能完美运行！”之类的行为。这种间歇性故障通常源于未通过等待、锁定、队列提交之间的显式依赖关系等方式正确同步的依赖关系。

有用的调试技术包括

- 从无序队列切换到有序队列
- 散布 queue.wait() 调用

在调试时使用其中一个或两个可以帮助识别依赖信息可能丢失的位置。如果此类更改使程序故障发生变化或消失，则强烈暗示我们的同步/依赖逻辑中有问题需要纠正。修复后，我们将删除这些临时调试措施。

13.7.2 调试 Kernel 代码

调试 Kernel 代码时，首先在 CPU 设备上运行（如第 2 章所述）。第 2 章中的设备选择器代码可以轻松修改为接受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设备。

```
q.submit([&](handler &h) {
    stream out(1024, 256, h);
    h.parallel_for(range{8}, [=](id<1> idx) {
        out << "Testing my sycl stream (this is work-item ID:"
        << idx << ")\n";
    });
});
```

图 13.4: *sycl::stream*

当调试 Kernel 代码时，SYCL 定义了一个可以在 Kernel 中使用的 C++

风格的流（图 13-4）。DPC++ 编译器还提供了 C 风格 printf 的实验性实现，它具有有用的功能，但有一些限制。

在调试 Kernel 代码时，经验鼓励我们将断点放在 parallel_for 之前或 parallel_for 内部，但实际上不要放在 parallel_for 上。即使在执行下一个操作之后，放置在 parallel_for 上的断点也可以多次触发断点。此 C++ 调试建议适用于许多模板扩展，例如 SYCL 中的模板扩展，其中模板调用上的断点在编译器扩展时将转换为一组复杂的断点。实现可能有一些方法可以缓解这个问题，但这里的关键点是，我们可以通过不在 parallel_for 本身上精确设置断点来避免所有实现上的一些混乱。

13.7.3 调试运行时故障

当即时编译时发生运行时错误时，我们要么正在处理明确使用可用硬件无法支持的功能（例如，fp16 或 simd8）的情况，要么是编译器/运行时错误，要么是我们意外地使用了可用硬件无法支持的功能（例如，fp16 或 simd8）。编写的无意义内容直到运行时出错并创建难以理解的运行时错误消息后才被检测到。在这三种情况下，深入研究这些错误可能有点令人生畏。值得庆幸的是，即使是粗略的观察也可以让我们更好地了解导致特定问题的原因。它可能会产生一些额外的知识来指导我们避免该问题，或者它可能只是帮助我们向编译器团队提交一份简短的错误报告。无论哪种方式，了解一些可以提供帮助的工具都很重要。

我们的程序的指示运行时失败的输出可能类似于以下示例：

```
terminate called after throwing an instance of 'sycl::__V1::runtime_error'  
what(): Native API failed. Native API returns: ...
```

or

```
terminate called after throwing an instance of 'sycl::__V1::compile_program_error'  
what(): The program was built for 1 devices  
...
```

```
error: Kernel compiled with required subgroup size 8, which is unsupported  
on this platform in kernel: 'typeinfo name for main::`lambda'(sycl::__V1::nd_item<2>)'  
error: backend compiler failed build.
```

-11 (PI_ERROR_BUILD_PROGRAM_FAILURE)

在这里看到此类异常让我们知道我们的主机程序可以被构造为捕获此错误。第一个显示了访问本机不支持的任何 API 时出现的一些包罗万象的

错误（在这种情况下，它使用了平台不支持的主机端内存分配）；第二个更容易认识到 SIMD8 是为不支持它的设备指定的（在本例中它支持 SIMD16）。运行时编译器失败不需要中止我们的应用程序；我们可以抓住它们，或者编写代码来避免它们，或者两者兼而有之。第 5 章深入探讨这个主题。

当我们看到运行时故障并且在快速调试时遇到困难时，值得尝试使用提前编译进行重建。如果我们的目标设备具有提前编译选项，那么这可能是一件容易尝试的事情，可能会产生更容易理解的诊断。如果我们的错误可以在编译时而不是 JIT 或运行时看到，那么通常会在编译器的错误消息中找到更有用的信息，而不是我们通常从 JIT 或运行时看到的少量错误信息。

Environment variables	Value	description
ONEAPI_DEVICE_SELECTOR	See online documentation for examples of the numerous options in the documents at intel.github.io .	Can be used to limit the choice of devices available when a SYCL-using application is run. Useful for limiting devices to a certain type (like GPUs or accelerators) or backends (like Level Zero or OpenCL).
SYCL_PI_TRACE	1 (basic), 2 (advanced), -1 (all)	Runtime: Value of 1 enables tracing of Runtime Plugin Interface (PI) for plugin and device discovery; Value of 2 enables tracing of all PI calls. Value of -1 unleashes all levels of tracing.
SYCL_PRINT_EXECUTION_GRAPH	always (or ask to dump only select files by specifying: before_addCG, after_addCG, before_addCopyBack, after_addCopyBack, before_addHostAcc, or after_addHostAcc)	Runtime: create text files (with DOT extension) tracing the execution graph. Relatively easy to browse traces of what is happening during runtime.
IGC_ShaderDumpEnable	0 or 1	Linux only. Runtime: ask the Intel Graphics Compiler (JIT) to dump some information.
IGC_ShaderDumpEnableAll	0 or 1	Linux only. Runtime: ask the Intel Graphics Compiler (JIT) to dump lots of information.

图 13.5: DPC++ 高级调试选项

图 13-5 列出了编译器或运行时支持的两个标志和附加环境变量（在 Windows 和 Linux 上受支持），以帮助进行高级调试。这些是 DPC++ 编译器特定的高级调试选项，用于检查和控制编译模型。本书中没有讨论或使用它们；GitHub 项目在 intel.github.io/llvm-docs/EnvironmentVariables.html 和 tinyurl.com/IGCOptions 上对它们进行了在线详细解释。

本书中没有对这些选项进行更多描述，但这里提到它们是为了根据需要开辟这种高级调试的途径。这些选项可以让我们深入了解如何解决问题或错误。我们的源代码有可能无意中触发了一个问题，可以通过更正源代码来解决。否则，使用这些选项是为了对编译器本身进行非常高级的调试。因

此，它们与编译器开发人员的关系比与编译器用户的关系更多。一些高级用户发现这些选项很有用；因此，它们在这里被提及，并且在本书中不再被提及。要深入了解，请参阅 DPC++ 编译器 GitHub 项目 intel.github.io/llvm-docs/EnvironmentVariables.html。

注 51 (调试提示) 当其他选项用尽并且我们需要调试运行时问题时，我们会寻找可能为我们提供原因提示的转储工具。

13.7.4 队列分析和由此产生的计时功能

许多设备支持队列分析 (`device::has(aspect::queue_profiling)`) ——有关一般方面的更多信息，请参阅第 12 章。一个简单而强大的接口可以轻松访问有关队列提交、实际执行开始的详细计时信息设备上的完成、设备上的完成以及命令的完成。与使用主机计时机制（例如 `chrono`）相比，此分析对于设备计时更加精确，因为它们通常不包括主机到设备的数据传输时间。请参阅图 13-6 和图 13-7 中所示的示例，以及图 13-8 中所示的示例输出。图 13-8 中所示的示例输出说明了此技术的可能性，但尚未优化，不应使用以任何方式代表任何特定系统选择的优点。

`aspect::queue_profiling` 方面指示设备通过 `property::queue::enable_profiling` 支持队列分析。

对于此类设备，我们可以在构造队列时指定 `property::queue::enable_profiling` — 属性列表是队列构造函数的可选最终参数。这样做会激活 SYCL 运行时捕获提交到该队列的命令组的分析信息。然后通过 SYCL 事件类 `get_profiling_info` 成员函数提供捕获的信息。如果队列的关联设备没有 `aspect::queue_profiling`，这将导致构造函数抛出带有 `errc::feature_not_supported` 错误代码的同步异常。

可以使用事件类的 `get_profiling_info` 成员函数来查询事件的分析信息，并指定 `info::event_profiling` 中枚举的分析信息参数之一。每个信息参数的可能值和任何限制在与事件关联的 SYCL 后端的规范中定义。`info::event_profiling` 中的所有信息参数均在 SYCL 规范标题为“SYCL 事件类的分析信息描述符”的表中指定，并且 `info::event_profiling` 的概要在规范附录“事件信息描述符”下描述。

每个分析描述符返回一个时间戳，表示自某些实现定义的时间基准以来已经过去的纳秒数。共享同一后端的所有事件都保证共享相同的时基；因

此，来自同一后端的两个时间戳之间的差异产生了这些事件之间经过的纳秒数。

最后一点，我们提醒您，启用事件分析确实会增加开销，因此最佳实践是在开发或调整期间启用它，然后在生产中禁用它。

注 52 由于开销很小，请仅在开发或优化期间启用队列分析，而对生产环境禁用。

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

// Array type and data size for this example.
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;
// Define VectorAdd (see Figure 13-7)

void InitializeArray(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++) a[i] = i;
}

int main() {
    IntArray a, b, sum;
    InitializeArray(a);
    InitializeArray(b);

    queue q(property::queue::enable_profiling{});

    std::cout << "Vector size: " << a.size()
        << "\nRunning on device: "
        << q.get_device().get_info<info::device::name>()
        << "\n";

    VectorAdd(q, a, b, sum);

    return 0;
}
```

图 13.6：设置为使用队列分析

```
void VectorAdd(queue &q, const IntArray &a,
               const IntArray &b, IntArray &sum) {
    range<1> num_items{a.size()};
    buffer a_buf(a), b_buf(b);
    buffer sum_buf(sum.data(), num_items);
    auto t1 =
        std::chrono::steady_clock::now(); // Start timing

    event e = q.submit([&](handler &h) {
        auto a_acc = a_buf.get_access<access::mode::read>(h);
        auto b_acc = b_buf.get_access<access::mode::read>(h);
        auto sum_acc =
            sum_buf.get_access<access::mode::write>(h);

        h.parallel_for(num_items, [=](id<1> i) {
            sum_acc[i] = a_acc[i] + b_acc[i];
        });
    });
    q.wait();

    double timeA =
        (e.template get_profiling_info<
            info::event_profiling::command_end>() -
         e.template get_profiling_info<
            info::event_profiling::command_start>());

    auto t2 =
        std::chrono::steady_clock::now(); // Stop timing

    double timeB = (std::chrono::duration_cast<
                    std::chrono::microseconds>(t2 - t1)
                    .count());

    std::cout
        << "profiling: Vector add completed on device in "
        << timeA << " nanoseconds\n";
    std::cout << "chrono: Vector add completed on device in "
        << timeB * 1000 << " nanoseconds\n";
    std::cout << "chrono more than profiling by "
        << (timeB * 1000 - timeA) << " nanoseconds\n";
}
```

图 13.7: 使用队列分析

```
Vector size: 65536
Running on device: Intel(R) UHD Graphics P630 [0x3e96]
profiling: Vector add completed on device in 57602 nanoseconds
chrono: Vector add completed on device in 2.85489e+08 nanoseconds
chrono more than profiling by 2.85431e+08 nanoseconds

Vector size: 65536
Running on device: NVIDIA GeForce RTX 3060
profiling: Vector add completed on device in 17410 nanoseconds
chrono: Vector add completed on device in 3.6071e+07 nanoseconds
chrono more than profiling by 3.60536e+07 nanoseconds

Vector size: 65536
Running on device: Intel(R) Data Center GPU Max 1100
profiling: Vector add completed on device in 9440 nanoseconds
chrono: Vector add completed on device in 5.6976e+07 nanoseconds
chrono more than profiling by 5.69666e+07 nanoseconds
```

图 13.8: 队列分析示例的三个示例输出

13.7.5 跟踪和分析工具接口

跟踪和分析工具可以帮助我们了解应用程序中的运行时行为，并且通常可以揭示改进算法的机会。见解通常是可移植的，因为它们可以推广到各种设备，因此我们建议在您喜欢的任何平台上使用您认为最有价值的任何跟踪和分析工具。当然，微调任何平台可能需要在相关平台上进行。对于最大程度的便携应用，我们鼓励首先寻找机会进行调整，并着眼于使任何调整尽可能便携。

当我们的 SYCL 程序在 OpenCL 运行时之上运行并使用 OpenCL 后端时，我们可以使用 OpenCL 拦截层运行我们的程序：github.com/intel/opencl-intercept-layer。这是一个可以检查、记录和修改应用程序（或更高级运行时）生成的 OpenCL 命令的工具。它支持很多控件，但最初设置的好控件是 ErrorLogging、BuildLogging，也许还有 CallLogging（尽管它会生成大量输出）。使用 DumpProgramSPIRV 可以进行有用的转储。OpenCL 拦截层是一个单独的实用程序，不属于任何特定 OpenCL 实现的一部分，因此它可以与许多 SYCL 编译器配合使用。

还有许多其他优秀工具可用于收集 SYCL 开发人员常用的性能数据。它们是开源的 (github.com/intel/pti-gpu) 以及帮助我们入门的示例。

两个最流行的工具如下：

- onetrace：用于 OpenCL 和零级后端的主机和设备跟踪工具，支持 DPC++（适用于 CPU 和 GPU）和 OpenMP GPU 卸载

- oneprof: 适用于 OpenCL 和零级后端的 GPU 硬件指标收集工具，支持 DPC++ 和 OpenMP* GPU 卸载

这两种工具都使用来自检测运行时的信息，因此它们适用于 GPU 和 CPU。使用这些运行时的编译器中的 SYCL、ISPC 和 OpenMP 支持都可以从这些工具中受益。请查阅网站上的工具，了解它们对您的使用的适用性。一般来说，我们可以找到一个受支持的平台，并使用这些工具来了解有关您的程序的有用信息，即使我们目标的每个平台都不支持。我们学到的关于程序的大部分知识在任何地方都是有用的。

13.8 初始化数据并访问 Kernel 输出

在本节中，我们将深入探讨一个导致 SYCL 新用户感到困惑的主题，该主题会导致我们作为新 SYCL 开发人员遇到的最常见（根据我们的经验）的第一个错误。

简而言之，当我们从主机内存分配（例如数组或向量）创建 Buffer 时，在 Buffer 被销毁之前我们无法直接访问主机分配。在 Buffer 的整个生命周期内，Buffer 拥有在构造时传递给它的任何主机分配。很少使用允许我们在 Buffer 仍处于活动状态时访问主机分配的机制（例如 Buffer 互斥体），但这些高级功能无助于解决此处描述的早期错误。

注 53 (每个人都会犯这个错误——知道这一点可以帮助我们快速调试它，而不是长时间纠结它!!)
如果我们从主机内存分配构造 Buffer，则在 Buffer 被销毁之前，我们不能直接访问主机分配！当它处于活动状态时，Buffer 拥有分配。了解 Buffer 范围和范围内的规则！

当主机程序访问主机分配而 Buffer 仍然拥有该分配时，会出现一个常见错误。一旦发生这种情况，所有的赌注都会消失，因为我们不知道 Buffer 使用分配的目的是什么。如果数据不正确，请不要感到惊讶 - 我们尝试从中读取输出的 Kernel 可能还没有开始运行！如第 3 章和第 8 章所述，SYCL 是围绕异步任务图机制构建的。在尝试使用任务图操作的输出数据之前，我们需要确保已达到代码中执行图的同步点并使数据可供主机使用。Buffer 破坏和主机访问器的创建都是导致此同步的操作。

```

constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create host containers to initialize on the host
std::vector<int> in_vec(N), out_vec(N);

// Initialize input and output vectors
for (int i = 0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// Nuance: Create new scope so that we can easily cause
// buffers to go out of scope and be destroyed
{
    // Create buffers using host allocations (vector in this
    // case)
    buffer in_buf{in_vec}, out_buf{out_vec};

    // Submit the kernel to the queue
    q.submit([&](handler& h) {
        accessor in{in_buf, h};
        accessor out{out_buf, h};

        h.parallel_for(
            range{N}, [=](id<1> idx) { out[idx] = in[idx]; });
    });

    // Close the scope that buffer is alive within! Causes
    // buffer destruction which will wait until the kernels
    // writing to buffers have completed, and will copy the
    // data from written buffers back to host allocations
    // (our std::vectors in this case). After the buffer
    // destructor runs, caused by this closing of scope,
    // then it is safe to access the original in_vec and
    // out_vec again!
}

// Check that all outputs match expected value
// WARNING: The buffer destructor must have run for us to
// safely use in_vec and out_vec again in our host code.
// While the buffer is alive it owns those allocations,
// and they are not safe for us to use! At the least they
// will contain values that are not up to date. This code
// is safe and correct because the closing of scope above
// has caused the buffer to be destroyed before this point
// where we use the vectors again.
for (int i = 0; i < N; i++)
    std::cout << "out_vec[" << i << "]=" << out_vec[i]
        << "\n";

```

图 13.9: 常见模式: 从主机分配创建 Buffer

图 13-9 显示了我们经常编写的代码的常见模式，其中我们通过关闭定义 Buffer 的块作用域来销毁 Buffer。通过使 Buffer 超出范围并被销毁，我们可以通过传递给 Buffer 构造函数的原始主机分配安全地读取 Kernel 结果。

将 Buffer 与现有主机内存关联有两个常见原因，如图 13-9 所示：

1. 简化 Buffer 中数据的初始化。我们可以从我们（或应用程序的其他部分）已经初始化的主机内存构造 Buffer。
2. 减少键入的字符，因为使用“}”关闭作用域比创建 Buffer 的 host_accessor 稍微简洁一些（尽管更容易出错）。

如果我们使用主机分配来转储或验证 Kernel 的输出值，则需要将 Buffer 分配放入块作用域（或其他作用域）中，以便我们可以控制它何时被破坏。然后，我们必须确保在访问主机分配以获取 Kernel 输出之前 Buffer 已被销毁。图 13-9 显示了正确完成的操作，而图 13-10 显示了一个常见错误，即在 Buffer 仍处于活动状态时访问输出。

注 54 高级用户可能更喜欢使用 *Buffer* 销毁将结果数据从 *Kernel* 返回到主机内存分配中。但对于大多数用户，尤其是新开发人员，建议使用作用域内的主机访问器。

```

constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create host containers to initialize on the host
std::vector<int> in_vec(N), out_vec(N);

// Initialize input and output vectors
for (int i = 0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// Create buffers using host allocations (vector in this
// case)
buffer in_buf{in_vec}, out_buf{out_vec};

// Submit the kernel to the queue
q.submit([&](handler& h) {
    accessor in{in_buf, h};
    accessor out{out_buf, h};

    h.parallel_for(range{N},
                  [=](id<1> idx) { out[idx] = in[idx]; });
});

// BUG!!! We're using the host allocation out_vec, but the
// buffer out_buf is still alive and owns that allocation!
// We will probably see the initialization value (zeros)
// printed out, since the kernel probably hasn't even run
// yet, and the buffer has no reason to have copied any
// output back to the host even if the kernel has run.
for (int i = 0; i < N; i++)
    std::cout << "out_vec[" << i << "]=" << out_vec[i]
    << "\n";

```

图 13.10: 常见 bug: 在 Buffer 生存期内直接从主机分配中读取数据

为了避免这些错误，我们建议在开始使用带有 SYCL 的 C++ 时使用主机访问器而不是 Buffer 范围。主机访问器提供对主机 Buffer 的访问，一旦它们的构造函数完成运行，我们就可以保证之前对 Buffer 的任何写入（例如，在创建 host_accessor 之前提交的 Kernel）都已执行并且可见。本书混合使用了两种风格（即主机访问器和传递给 Buffer 构造函数的主机分配）来熟悉这两种风格。开始时使用主机访问器往往不太容易出错。图 13-11 显示了如何使用主机访问器从 Kernel 读取输出，而无需先破坏 Buffer。

```
constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create host containers to initialize on the host
std::vector<int> in_vec(N), out_vec;

// Initialize input and output vectors
for (int i = 0; i < N; i++) in_vec[i] = i;
std::fill(out_vec.begin(), out_vec.end(), 0);

// Create buffers using host allocations (vector in this
// case)
buffer in_buf{in_vec}, out_buf{out_vec};

// Submit the kernel to the queue
q.submit([&](handler& h) {
    accessor in{in_buf, h};
    accessor out{out_buf, h};

    h.parallel_for(range{N},
                  [=](id<1> idx) { out[idx] = in[idx]; });
});

// Check that all outputs match expected value
// Use host accessor! Buffer is still in scope / alive
host_accessor A{out_buf};

for (int i = 0; i < N; i++)
    std::cout << "A[" << i << "]=" << A[i] << "\n";
```

图 13.11: 建议: 使用主机访问器读取 *Kernel* 结果

只要 Buffer 处于活动状态, 例如在典型 Buffer 生命周期的两端, 就可以使用主机访问器 - 用于初始化 Buffer 内容并从 Kernel 读取结果。图 13-12 显示了此模式的示例。

```

constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create buffers of size N
buffer<int> in_buf{N}, out_buf{N};

// Use host accessors to initialize the data
{ // CRITICAL: Begin scope for host_accessor lifetime!
    host_accessor in_acc{in_buf}, out_acc{out_buf};
    for (int i = 0; i < N; i++) {
        in_acc[i] = i;
        out_acc[i] = 0;
    }
} // CRITICAL: Close scope to make host accessors go out
   // of scope!

// Submit the kernel to the queue
q.submit([&](handler& h) {
    accessor in{in_buf, h};
    accessor out{out_buf, h};

    h.parallel_for(range{N},
                  [=](id<1> idx) { out[idx] = in[idx]; });
});

// Check that all outputs match expected value
// Use host accessor! Buffer is still in scope / alive
host_accessor A{out_buf};

for (int i = 0; i < N; i++)
    std::cout << "A[" << i << "]=" << A[i] << "\n";

```

图 13.12: 建议: 使用主机访问器进行 *Buffer* 初始化和结果读取

最后要提到的一个细节是, 主机访问器有时会在应用程序中引起相反的错误, 因为它们也有生命周期。当 Buffer 的主机访问器处于活动状态时, 运行时将不允许任何设备使用该 Buffer! 运行时不会分析我们的主机程序来确定它们何时可以访问主机访问器, 因此它知道主机程序已完成访问 Buffer 的唯一方法是运行 host_accessor 析构函数。如图 13-13 所示, 如果我们的主机程序正在等待某些 Kernel 运行 (例如, queue::wait() 或获取另一个主机访问器) 并且 SYCL 运行时正在等待某些 Kernel 运行, 则这可能会导致应用程序挂起。我们早期的主机访问器在运行使用 Buffer 的 Kernel 之前会

被销毁。

```

constexpr size_t N = 1024;

// Set up queue on any available device
queue q;

// Create buffers using host allocations (vector in this
// case)
buffer<int> in_buf{N}, out_buf{N};

// Use host accessors to initialize the data
host_accessor in_acc{in_buf}, out_acc{out_buf};
for (int i = 0; i < N; i++) {
    in_acc[i] = i;
    out_acc[i] = 0;
}

// BUG: Host accessors in_acc and out_acc are still alive!
// Later q.submit will never start on a device, because
// the runtime doesn't know that we've finished accessing
// the buffers via the host accessors. The device kernels
// can't launch until the host finishes updating the
// buffers, since the host gained access first (before the
// queue submissions). This program will appear to hang!
// Use a debugger in that case.

// Submit the kernel to the queue
q.submit([&](handler& h) {
    accessor in{in_buf, h};
    accessor out{out_buf, h};

    h.parallel_for(range{N},
                  [=](id<1> idx) { out[idx] = in[idx]; });
});

std::cout << "This program will deadlock here!!! Our "
          "host_accessors used\n"
          << " for data initialization are still in "
          "scope, so the runtime won't\n"
          << " allow our kernel to start executing on "
          "the device (the host could\n"
          << " still be initializing the data that is "
          "used by the kernel). The next line\n"
          << " of code is acquiring a host accessor for "
          "the output, which will wait for\n"
          << " the kernel to run first. Since in_acc "
          "and out_acc have not been\n"
          << " destructed, the kernel is not safe for "
          "the runtime to run, and we deadlock.\n";

// Check that all outputs match expected value
// Use host accessor! Buffer is still in scope / alive
host_accessor A{out_buf};

for (int i = 0; i < N; i++)
    std::cout << "A[" << i << "]=" << A[i] << "\n";

```

图 13.13: 由于不当使用 `host_accessor` 而导致的死锁（错误 - 挂起!）

注 55 使用主机访问器时，请确保在不再需要 *Kernel* 或其他主机访问器解锁 *Buffer* 使用时销毁它们。

13.9 多个翻译单元

当我们想要调用 *Kernel* 内部不同翻译单元中定义的函数时，这些函数需要使用 `SYCL_EXTERNAL` 进行标记。如果没有这种修饰，编译器将仅编译在设备代码外部使用的函数（使得从设备代码内调用该外部函数是非法的）。

如果我们在同一翻译单元中定义函数，则 `SYCL_EXTERNAL` 函数有一些限制不适用：

- `SYCL_EXTERNAL` 只能用于函数。
- `SYCL_EXTERNAL` 函数不能使用原始指针作为参数或返回类型。必须改用显式指针类。
- `SYCL_EXTERNAL` 函数无法调用 `parallel_for_work_item` 方法。
- 不能从 `parallel_for_work_group` 范围内调用 `SYCL_EXTERNAL` 函数。

如果我们尝试编译一个调用不在同一翻译单元内且未使用 `SYCL_EXTERNAL` 声明的函数的 *Kernel*，那么我们可以预期会出现类似于以下内容的编译错误：

```
error: SYCL kernel cannot call an undefined function without SYCL_EXTERNAL attribute
```

如果函数本身是在没有 `SYCL_EXTERNAL` 属性的情况下编译的，我们可以预期会看到链接或运行时失败，例如

```
terminate called after throwing an instance of '...compile_program_error'...
error: undefined reference to ...
```

`SYCL` 不要求编译器支持 `SYCL_EXTERNAL`；一般来说，它是一个可选功能。`DPC++` 支持 `SYCL_EXTERNAL`。

13.9.1 多个翻译单元的性能影响

编译模型的含义（请参阅本章前面的内容）是，如果我们将设备代码分散到多个翻译单元中，则与设备代码共置相比，可能会触发更多的即时编译

调用。这高度依赖于实现，并且随着实现的成熟，可能会随着时间的推移而发生变化。

这种对性能的影响很小，足以在我们的大部分开发工作中忽略，但是当我们进行微调以最大限度地提高代码性能时，我们可以考虑以下两件事来减轻这些影响：(1) 将设备代码组合在一起相同的翻译单元，以及 (2) 使用提前编译来完全避免即时编译效应。由于这两者都需要我们付出一些努力，因此我们只有在完成开发并试图从应用程序中榨取每一盎司性能时才这样做。当我们确实采取这种详细的调整时，值得测试更改以观察它们对我们正在使用的确切 SYCL 实现的影响。

13.10 当匿名 Lambda 需要名称时

SYCL 允许为 lambda 分配名称，以防工具需要它并用于调试目的（例如，以用户定义的名称启用显示）。根据 SYCL 2020 规范，命名 lambda 是可选的。在本书的大部分内容中，匿名 lambda 都用于 Kernel，因为在使用 C++ 和 SYCL 时不需要名称（除了传递编译选项，如第 10 章中 lambda 命名讨论所述）。

当我们需要在代码库中混合来自多个供应商的 SYCL 工具时，该工具可能要求我们命名 lambda。这是通过将 `<class unquename>` 添加到使用 lambda 的 SYCL 操作构造（例如，`parallel_for`）来完成的。这种命名允许来自多个供应商的工具在单个编译中以定义的方式进行交互，并且还可以通过显示我们在调试工具和层中定义的 Kernel 名称来提供帮助。

如果我们想使用 Kernel 查询，我们还需要命名 Kernel。SYCL 标准委员会无法在 SYCL 2020 标准中找到解决方案来满足这一要求。例如，查询 Kernel 的 `preferred_work_group_size_multiple` 需要我们调用 Kernel 类的 `get_info()` 成员函数，这需要 Kernel 类的实例，这最终需要我们知道 Kernel 的名称（和 `kernel_id`）才能提取来自相关 `kernel_bundle` 的句柄。

13.11 总结

当今的流行文化经常将技巧称为生活窍门。不幸的是，编程文化常常给 hack 赋予负面含义，因此作者没有将本章命名为“SYCL Hacks”。毫无疑问，本章只是触及了使用 C++ 和 SYCL 的实用技巧的表面。当我们一起学习如何通过 SYCL 充分利用 C++ 时，我们所有人都可以分享更多技巧。

14 常见的并行模式

当我们作为程序员处于最佳状态时，我们会认识到工作中的模式并应用经过时间考验的最佳解决方案技术。并行编程也不例外，如果不研究已被证明在该领域有用的模式，那将是一个严重的错误。考虑大数据应用程序采用的 MapReduce 框架；他们的成功很大程度上源于基于两种简单而有效的并行模式——映射和归约。

并行编程中有许多常见的模式，它们会一次又一次地出现，与我们使用的编程语言无关。这些模式用途广泛，可以在任何并行级别（例如 Sub-Groups、Work-Groups、完整设备）和任何设备（例如 CPU、GPU、FPGA）上使用。然而，模式的某些属性（例如它们的可扩展性）可能会影响它们对不同设备的适用性。在某些情况下，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；在其他情况下，我们也许能够通过选择完全不同的模式来提高性能。

了解如何、何时、何地使用这些常见并行模式是提高 SYCL（以及一般并行编程）熟练程度的关键部分。对于那些具有现有并行编程经验的人来说，了解这些模式在 SYCL 中的表达方式可以是快速启动并熟悉该语言功能的方法。

本章旨在回答以下问题：

- 我们应该了解哪些常见模式？
- 这些模式与不同设备的功能有何关系？
- 哪些模式已作为 SYCL 函数和库提供？
- 如何使用直接编程来实现这些模式？

14.1 理解模式

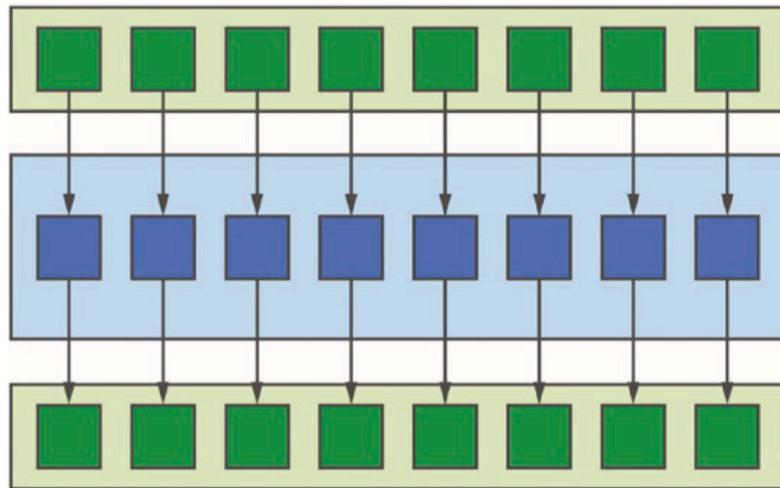
这里讨论的模式是 McCool 等人的《结构化并行编程》一书中描述的并行模式的子集。我们不讨论与并行类型相关的模式（例如，fork-join、branch-and-bound），而是关注一些对于编写数据并行 Kernel 最有用的算法模式。

Pattern	Useful For	Key Attributes	Device Affinity
Map	Simple parallel kernels	No data dependences and high scalability	All
Stencil	Structured data dependences	Data dependences and data re-use	Depends on stencil size
Reduction	Combining partial results	Data dependences	All
Scan Pack/Unpack	Filtering and restructuring data	Limited scalability	Depends on problem size

图 14.1: 并行模式及其对不同设备类型的亲和力

我们全心全意地相信，理解并行模式的这个子集对于成为一名有效的 SYCL 程序员至关重要。图 14-1 中的表提供了不同模式的高级概述，包括它们的主要用例、关键属性以及它们的属性如何影响它们对不同硬件设备的亲和力。

14.1.1 映射 Map

图 14.2: *Map* 模式

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会立即熟悉。如图 14-2 所示，范围的每个输入元素通过应用某种函数独立地映射到输出。许多数据并行操作可以表示为映射模式的实例（例如，向量加法）。

由于函数的每个应用程序都是完全独立的，因此映射的表达式通常非常简单，依赖于编译器和/或运行时来完成大部分艰苦的工作。我们应该期望写入映射模式的 Kernel 适用于任何设备，并且这些 Kernel 的性能能够随着可用硬件并行性的数量很好地扩展。

然而，在决定将整个应用程序重写为一系列映射 Kernel 之前，我们应该仔细考虑！这种开发方法效率很高，并保证应用程序可以移植到各种设备类型，但鼓励我们忽略可能显着提高性能的优化（例如，提高数据重用、融合 Kernel）。

14.1.2 模版 Stencil

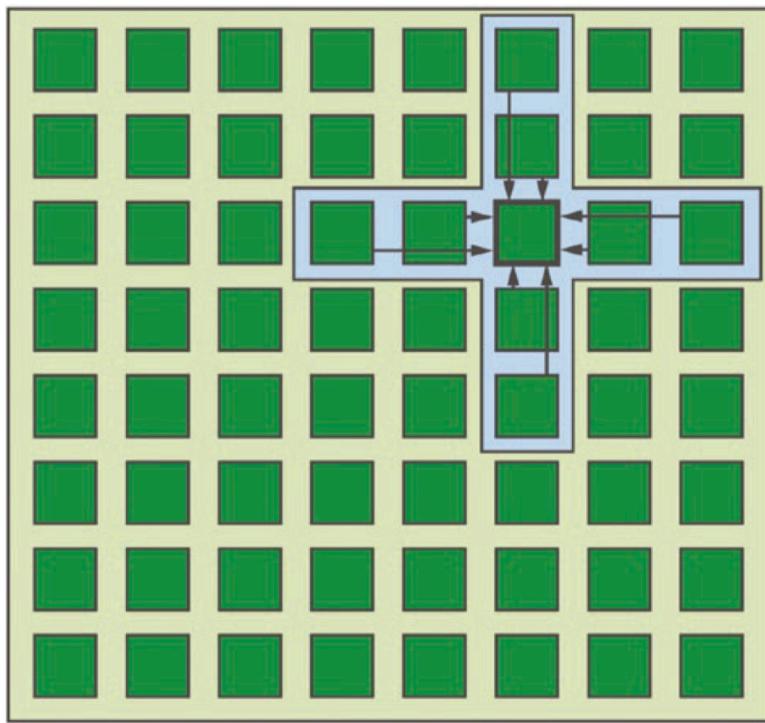


图 14.3: *Stencil* 模式

模板模式与映射模式密切相关。如图 14-3 所示，将一个函数应用于一个输入和由模板描述的一组相邻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（例如，有限差分代码）和计算机视觉/机器学习应用（例如，图像卷积）。

当模板模式异位执行时（即，将输出写入单独的存储位置），该函数可以独立应用于每个输入。在现实世界中调度模板通常比这更复杂：计算相邻输出需要相同的数据，并且多次从内存加载该数据会降低性能；我们可能希望就地应用模板（即覆盖原始输入值），以减少应用程序的内存占用。

因此，模板 Kernel 对不同设备的适用性很大程度上取决于模板的属性和输入问题。一般来说，

- 小型模板可以受益于 GPU 的暂存器存储。

- 大型模板可以受益于（相对）较大的 CPU 缓存。
- 在小输入上运行的小型模板可以通过在 FPGA 上作为脉动阵列实现来实现显着的性能增益。

由于模板很容易描述，但有效实现却很复杂，因此许多模板应用程序都使用特定于域的语言 (DSL)。已经有一些嵌入式 DSL 利用 C++ 的模板元编程功能在编译时生成高性能模板 Kernel。

14.1.3 归约 Reduction

归约是一种常见的并行模式，它使用通常是关联和交换的运算符（例如加法）来组合部分结果。最常见的归约示例是计算总和（例如，在计算点积时）或计算最小值/最大值（例如，使用最快速度来设置时间步长）。

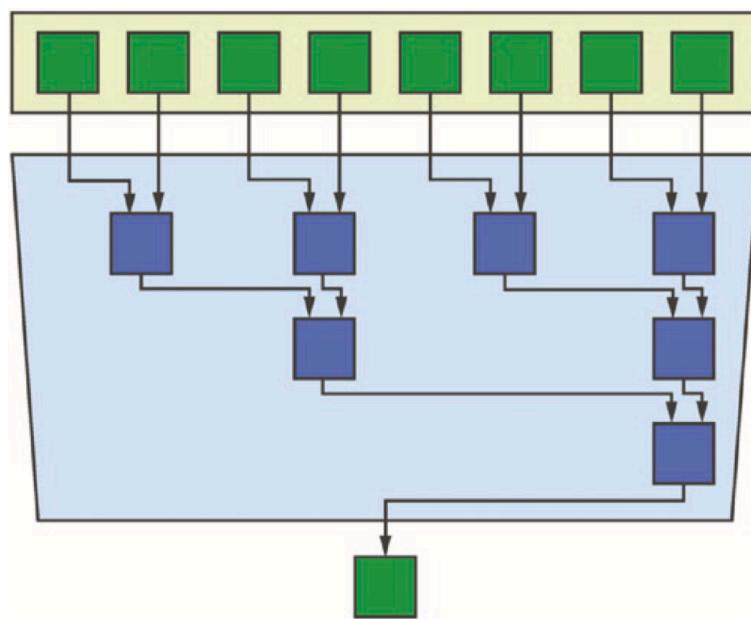


图 14.4: Reduction 模式

图 14-4 显示了通过树归约实现的归约模式，这是一种流行的实现，需要对一系列 N 个输入元素进行 $\log_2(N)$ 组合操作。尽管树归约很常见，但其他实现也是可能的 - 一般来说，我们不应该假设归约按特定顺序组合值。

在现实生活中, Kernel 很少是令人尴尬的并行, 即使是这样, 它们也经常与归约 (如在 MapReduce 框架中) 配对来总结其结果。这使得归约成为需要理解的最重要的并行模式之一, 并且我们必须能够在任何设备上高效执行该模式。

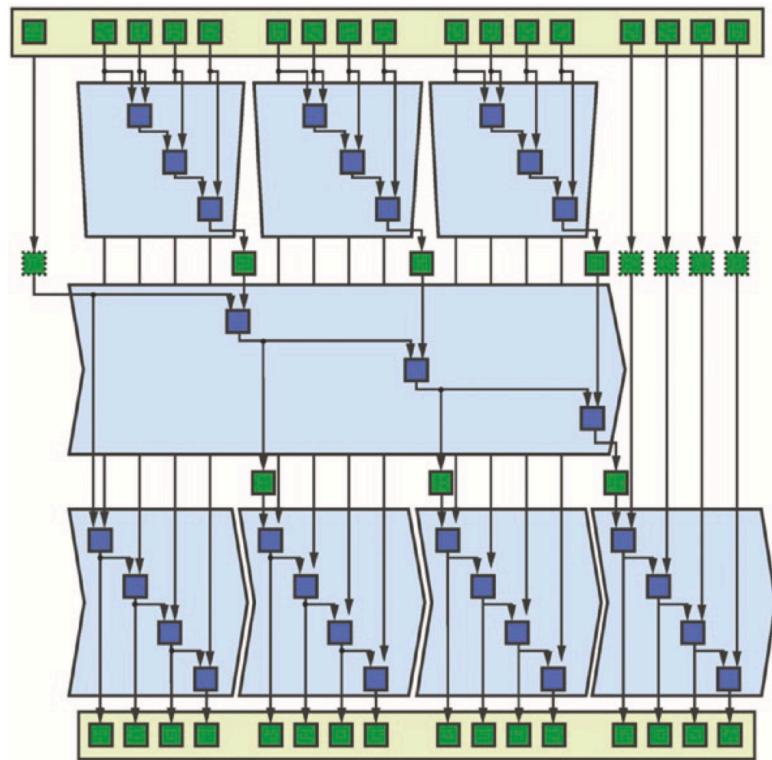
调优不同设备的归约是计算部分结果所花费的时间和组合它们所花费的时间之间的微妙平衡行为; 使用太少的并行性会增加计算时间, 而使用太多的并行性会增加组合时间。

通过使用不同的设备来执行计算和组合步骤来提高整体系统利用率可能很诱人, 但这种调整工作必须仔细注意在设备之间移动数据的成本。在实践中, 我们发现在同一设备上直接对生成的数据进行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并行性, 而是依赖于另一个级别的数据并行性 (即, 每个设备对部分输入数据执行归约)。

14.1.4 扫描 Scan

扫描模式使用二元关联运算符计算广义前缀和, 并且输出的每个元素代表部分结果。如果元素 i 的部分和是范围 $[0, i]$ 中所有元素的总和 (即包括 i 的总和), 则称扫描是包含的。如果元素 i 的部分和是 $[0, i)$ 范围内所有元素的和 (即不包括 i 的和), 则称扫描是排他的。

乍一看, 扫描似乎是一种本质上的串行操作——每个输出的值取决于前一个输出的值! 虽然扫描确实比其他模式具有更少的并行机会 (因此可扩展性可能较差), 但图 14-5 表明可以使用对相同数据的多次扫描来实现并行扫描。

图 14.5: *Scan* 模式

由于扫描操作中并行性的机会有限，因此执行扫描的最佳设备高度依赖于问题大小：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来饱和图形处理器。对于 FPGA 和其他空间架构来说，问题大小不太重要，因为扫描自然适合管道并行性。与归约的情况一样，在生成数据的同一设备上执行扫描操作通常是一个好主意，考虑到优化期间扫描操作在何处以及如何适应应用程序，通常会比专注于优化数据产生更好的结果。隔离扫描操作。

14.1.5 打包和拆包

打包和解包模式与扫描密切相关，并且通常在扫描功能之上实现。我们在这里单独介绍它们，因为它们可以实现常见操作（例如附加到列表）的高性能实现，而这些操作可能与前缀和没有明显的联系。

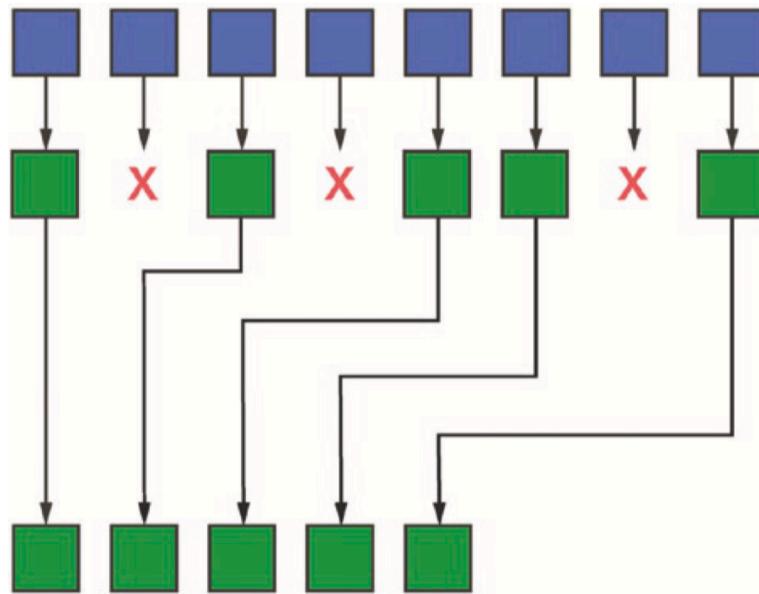
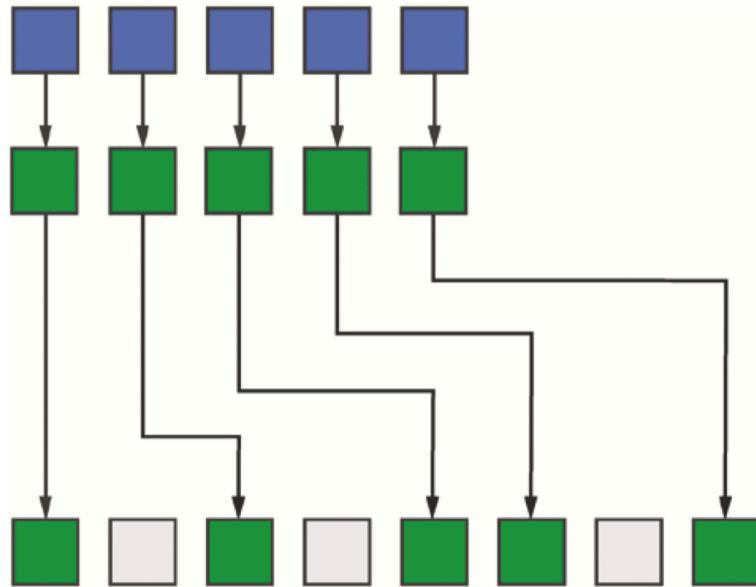


图 14.6: Pack 模式

打包 Pack 如图 14-6 所示，打包模式根据布尔条件丢弃输入范围的元素，将未丢弃的元素打包到输出范围的连续位置。该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些函数来在线计算。

与扫描一样，打包操作具有固有的串行性质。给定要打包/复制的输入元素，计算其在输出范围中的位置需要有关有多少先前元素也被打包/复制到输出中的信息。该信息相当于对驱动包的布尔条件进行独占扫描。

图 14.7: *Unpack* 模式

拆包 Unpack 如图 14-7 所示（顾名思义），解包模式与打包模式相反。输入范围的连续元素被解包为输出范围的不连续元素，而其他元素保持不变。此模式最明显的用例是解压缩先前打包的数据，但它也可用于填充先前计算产生的数据中的“间隙”。

14.2 使用内置函数和库

其中许多模式可以使用 SYCL 的内置功能或供应商提供的用 SYCL 编写的库直接表达。利用这些函数和库是在真正的大型软件工程项目中平衡性能、可移植性和生产力的最佳方式。

14.2.1 SYCL 归约库

SYCL 不需要我们每个人都维护自己的可移植且高性能的归约 Kernel 库，而是提供了一种方便的抽象，用于使用归约语义来描述变量。这种抽象简化了约简 Kernel 的表达，并使约简的执行变得明确，允许实现针对设备、数据类型和约简操作的不同组合在不同的约简算法之间进行选择。

```
h.parallel_for(
    range<1>{N}, reduction(sum, plus<>()),
    [=](id<1> i, auto& sum) { sum += data[i]; });
```

图 14.8: 使用 *reduction* 库表示为数据并行 Kernel 的 Reduction

图 14-8 中的 Kernel 显示了使用归约库的示例。请注意，Kernel 主体不包含任何对归约的引用 - 我们必须指定的是 Kernel 包含使用加法 Sub-Groups 合 sum 变量实例的归约。这为实现自动生成优化的归约序列提供了足够的信息。

在 Kernel 完成之前，不能保证归约结果被写回原始变量。除了此限制之外，访问归约结果的行为与访问 SYCL 中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，并且访问存储在 USM 分配中的归约结果可能需要显式同步和/或内存移动。

SYCL 归约库与其他语言中的归约抽象不同的一个重要方式是，它限制我们在 Kernel 执行期间对归约变量的访问 - 我们无法检查归约变量的中间值，并且禁止更新归约变量使用指定组合函数以外的任何变量。这些限制可以防止我们犯下难以调试的错误（例如，在尝试计算最大值时添加归约变量），并确保可以在各种不同的设备上有效地实现归约。

```

template <typename BufferT, typename BinaryOperation>
unspecified reduction(BufferT variable, handler& h,
                      BinaryOperation combiner,
                      const property_list& properties = {});

template <typename BufferT, typename BinaryOperation>
unspecified reduction(BufferT variable, handler& h,
                      const BufferT::value_type& identity,
                      BinaryOperation combiner,
                      const property_list& properties = {});

template <typename T, typename BinaryOperation>
unspecified reduction(T* variable, BinaryOperation combiner,
                      const property_list& properties = {});

template <typename T, typename BinaryOperation>
unspecified reduction(T* variable, const T& identity,
                      BinaryOperation combiner,
                      const property_list& properties = {});

template <typename T, typename Extent,
          typename BinaryOperation>
unspecified reduction(span<T, Extent> variables,
                      BinaryOperation combiner,
                      const property_list& properties = {});

template <typename T, typename Extent,
          typename BinaryOperation>
unspecified reduction(span<T, Extent> variables,
                      const T& identity,
                      BinaryOperation combiner,
                      const property_list& properties = {});

```

图 14.9: a

reduction 类 归约类是我们用来描述 Kernel 中存在的归约的接口。构造归约对象的唯一方法是使用图 14-9 中所示的函数之一。请注意，共有三个归约函数系列（用于缓冲区、USM 指针和跨度），每个系列都有两个重载（带和不带恒等变量）。

如果使用缓冲区或 USM 指针初始化归约，则归约是标量归约，对数组中的第一个对象进行操作。如果使用跨度初始化归约，则归约是数组归约。数组归约的每个组成部分都是独立的——我们可以认为对大小为 N 的数组进行数组归约操作相当于具有相同数据类型和运算符的 N 标量归约。

该函数最简单的重载允许我们指定归约变量和用于组合每个 Work-Items 的贡献的运算符。第二组重载允许我们提供与归约运算符关联的可

选标识值 - 这是对用户定义归约的优化，我们稍后将重新讨论。

请注意，归约函数的返回类型是未指定的，并且归约类本身完全是实现定义的。尽管这对于 C++ 类来说可能有点不寻常，但它允许实现使用不同的类（或具有任意数量的模板参数的单个类）来表示不同的归约算法。SYCL 的未来版本可能会决定重新审视此设计，以便使我们能够在特定执行上下文中显式请求特定的归约算法（最有可能通过 `property_list` 参数）。

```
template <typename T, typename BinaryOperation,
          /* implementation-defined */>
class reducer {
    // Combine partial result with reducer's value
    void combine(const T& partial);
};

// Other operators are available for standard binary
// operations
template <typename T>
auto& operator+=(reducer<T, plus::<T>>&, const T&);
```

图 14.10: `reducer` 类的简化定义

reducer 类 `reducer` 类的实例封装了一个归约变量，公开了一个有限的接口，确保我们无法以实现可能认为不安全的任何方式更新归约变量。`reducer` 类的简化定义如图 14-10 所示。

与归约类一样，归约器类的精确定义是实现定义的 - 归约器的类型将取决于归约的执行方式，为了最大限度地提高性能，在编译时了解这一点非常重要。然而，允许我们更新归约变量的函数和运算符已明确定义，并且保证受到任何 SYCL 实现的支持。

具体来说，每个 `reducer` 都提供一个 `combine()` 函数，它将部分结果（来自单个 Work-Items）与 `reduction` 变量的值组合起来。这个组合函数的行为方式是实现定义的，但在编写 Kernel 时我们不需要担心。还需要一个 `reducer` 来根据 `reducer` 操作符来提供其他操作符；例如，`+=` 运算符被定义为加减法。提供这些附加运算符只是为了方便程序员并提高可读性；在可用的情况下，这些运算符与直接调用 `combine()` 具有相同的行为。

当处理数组约简时，`reducer` 提供了一个额外的下标运算符（即，`operator[]`），允许访问数组的各个元素。该运算符不是直接返回对数组元素的引

用，而是返回另一个化简器对象，该对象公开与标量约简关联的约简器相同的 `merge()` 函数和简写运算符。图 14-11 显示了一个使用数组归约来计算直方图的 Kernel 的简单示例，其中下标运算符用于仅访问由 Work-Items 更新的直方图箱。

```
h.parallel_for(
    range{N},
    reduction(span<int, 16>(histogram, 16), plus<>()),
    [=](id<1> i, auto& histogram) {
        histogram[i % B]++;
    });
}
```

图 14.11：使用数组约简计算直方图的示例 Kernel

用户定义的归约 几种常见的归约算法（例如，树归约）不会看到每个 Work-Items 直接更新单个共享变量，而是将一些部分结果累积在私有变量中，该变量将在将来的某个时刻进行组合。这样的私有变量引入了一个问题：实现应该如何初始化它们？将变量初始化为每个 Work-Items 的第一个贡献具有潜在的性能影响，因为需要额外的逻辑来检测和处理未初始化的变量。相反，将变量初始化为归约运算符的身份可以避免性能损失，但只有在身份已知的情况下才可能实现。

仅当对简单算术类型进行归约操作并且归约运算符是几个标准函数对象（例如，加号）之一时，SYCL 实现才能自动确定要使用的正确标识值。对于用户定义的归约（即，对用户定义类型和/或使用用户定义函数对象进行操作的归约），我们可以通过直接指定标识值来提高性能。

对用户定义归约的支持仅限于可简单复制的类型和组合函数，没有副作用，但这足以支持许多现实生活中的用例。例如，图 14-12 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。

```

template <typename T, typename I>
using minloc = minimum<std::pair<T, I>>;
```

```

int main() {
    constexpr size_t N = 16;

    queue q;
    float* data = malloc_shared<float>(N, q);
    std::pair<float, int>* res =
        malloc_shared<std::pair<float, int>>(1, q);
    std::generate(data, data + N, std::mt19937{});

    std::pair<float, int> identity = {
        std::numeric_limits<float>::max(),
        std::numeric_limits<int>::min()};
    *res = identity;

    auto red =
        sycl::reduction(res, identity, minloc<float, int>());

    q.submit([&](handler& h) {
        h.parallel_for(
            range<1>{N}, red, [=](id<1> i, auto& res) {
                std::pair<float, int> partial = {data[i], i};
                res.combine(partial);
            });
    }).wait();

    std::cout << "minimum value = " << res->first << " at "
          << res->second << "\n";
    ...
}

```

图 14.12: 使用用户定义的约简来查找最小值的位置

14.2.2 集体算法

SYCL 设备代码中对并行模式的支持由单独的组算法库提供。这些函数利用特定 Work-Items 组（即 Work-Groups 或 Sub-Groups）的并行性在有限范围内实现常见并行算法，并且可以用作构建其他更复杂算法的构建块。

SYCL 中的组算法的语法基于 C++ 中的算法库的语法，并且适用 C++ 算法的任何限制。然而，有一个关键的区别：STL 的算法是从顺序（主机）代码调用的，并表明库有机会采用并行性，而 SYCL 的组算法则设计为在已经并行执行的（设备）代码内调用。为了确保这种差异不会被忽视，组算法的语法和语义与其 C++ 对应算法略有不同。

SYCL 区分两种不同类型的并行算法。如果一个算法由一个组中的所

有 Work-Items 协作执行，但在其他方面与 STL 中的算法行为相同，则该算法以“joint”前缀命名（因为该组的成员“joint”在一起执行该算法）。此类算法从内存中读取输入并将结果写入内存，并且只能对给定组中的所有 Work-Items 可见的内存位置中的数据进行操作。如果算法在反映组本身的隐式范围内运行，并且输入和输出存储在 Work-Items 私有内存中，则算法名称将被修改为包含单词“组”（因为该算法直接对属于该组的数据执行）团体）。

```

// std::reduce
// Each work-item reduces over a given input range
q.parallel_for(number_of_reductions, [=](size_t i) {
    output1[i] = std::reduce(
        input + i * elements_per_reduction,
        input + (i + 1) * elements_per_reduction);
}).wait();

// sycl::joint_reduce
// Each work-group reduces over a given input range
// The elements are automatically distributed over
// work-items in the group
q.parallel_for(nd_range<1>{number_of_reductions *
                           elements_per_reduction,
                           elements_per_reduction},
                           [=](nd_item<1> it) {
        auto g = it.get_group();
        int sum = joint_reduce(
            g,
            input + g.get_group_id() *
                elements_per_reduction,
            input + (g.get_group_id() + 1) *
                elements_per_reduction,
            plus<>());
        if (g.leader()) {
            output2[g.get_group_id()] = sum;
        }
    })
    .wait();

// sycl::reduce_over_group
// Each work-group reduces over data held in work-item
// private memory. Each work-item is responsible for
// loading and contributing one value
q.parallel_for(
    nd_range<1>{
        number_of_reductions * elements_per_reduction,
        elements_per_reduction},
        [=](nd_item<1> it) {
            auto g = it.get_group();
            int x = input[g.get_group_id() *
                          elements_per_reduction +
                          g.get_local_id()];
            int sum = reduce_over_group(g, x, plus<>());
            if (g.leader()) {
                output3[g.get_group_id()] = sum;
            }
        })
    .wait();

```

图 14.13: `std::reduce`、`sycl::joint_reduce` 和 `sycl::reduce_over_group` 的比较

图 14-13 中的代码示例演示了这两种不同类型的算法，将 `std::reduce` 的行为与 `sycl::joint_reduce` 和 `sycl::reduce_over_group` 的行为进行了比较。

请注意，在这两种情况下，每个组算法的第一个参数接受 `group` 或 `sub_group` 对象来代替执行策略，以描述应用于执行算法的 Work-Items 集。由于算法是由指定组中的所有 Work-Items 协同执行的，因此它们也必须像组屏障一样对待——组中的所有 Work-Items 必须在聚合控制流中遇到相同的算法（即，组中的所有 Work-Items）组必须类似地遇到或不遇到算法调用），并且所有 Work-Items 提供的参数必须使得所有 Work-Items 都同意正在执行的操作。例如，`sycl::joint_reduce` 要求所有 Work-Items 的所有参数都相同，以确保组中的所有 Work-Items 对相同的数据进行操作并使用相同的运算符来累积结果。

C++ Algorithm	SYCL "Joint" Algorithm	SYCL "Group" Algorithm	Group Types
<code>std::any_of</code>	<code>sycl::joint_any_of</code>	<code>sycl::any_of_group</code>	All
<code>std::all_of</code>	<code>sycl::joint_all_of</code>	<code>sycl::all_of_group</code>	All
<code>std::none_of</code>	<code>sycl::joint_none_of</code>	<code>sycl::none_of_group</code>	All
<code>std::shift_left</code>	N/A	<code>sycl::shift_group_left</code>	sub group
<code>std::shift_right</code>	N/A	<code>sycl::shift_group_right</code>	sub group
N/A	N/A	<code>sycl::permute_group_by_xor</code>	sub group
N/A	N/A	<code>sycl::select_from_group</code>	sub group
<code>std::reduce</code>	<code>sycl::joint_reduce</code>	<code>sycl::reduce_over_group</code>	All
<code>std::exclusive_scan</code>	<code>sycl::joint_exclusive_scan</code>	<code>sycl::exclusive_scan_over_group</code>	All
<code>std::inclusive_scan</code>	<code>sycl::joint_inclusive_scan</code>	<code>sycl::inclusive_scan_over_group</code>	All

图 14.14: C++ 算法和 SYCL 组算法之间的映射

图 14-14 中的表显示了 STL 中可用的并行算法与组算法的关系，以及对可以使用的组类型是否有任何限制。请注意，在某些情况下，组算法只能与 Sub-Groups 一起使用；这些情况对应于前面章节中介绍的“shuffle”操作。

在撰写本文时，组算法仅限于支持基本数据类型和 SYCL 识别的一组内置运算符（即 `plus`, `multiplies`, `bit_and`, `bit_or`, `bit_xor`, `logical_and`, `logical_or`, `minimum`, 和 `maximum`）。这足以涵盖最常见的用例，但 SYCL 的未来版本预计将集体支持扩展到用户定义的类型和运算符。

14.3 直接编程

尽管我们建议尽可能利用库，但通过了解如何使用“本机”SYCL Kernel 实现每种模式，我们可以学到很多东西。

本章剩余部分中的 Kernel 不应期望达到与高度调优的库相同的性能水

平，但有助于更好地理解 SYCL 的功能，甚至可以作为新库功能原型设计的起点。

注 56 (使用供应商提供的库！) 当供应商提供函数的库实现时，使用它几乎总是有益的，而不是将函数重新实现为 *Kernel*!

14.3.1 映射 Map

由于其简单性，映射模式可以直接实现为基本并行 *Kernel*。图 14-15 中所示的代码显示了这样的实现，使用映射模式来计算范围内每个输入元素的平方根。

```
// Compute the square root of each input value
q.parallel_for(N, [=](id<1> i) {
    output[i] = sqrt(input[i]);
}).wait();
```

图 14.15: 在数据并行 *Kernel* 中实现 *Map* 模式

14.3.2 模版 Stencil

直接将模板实现为具有多维缓冲区的多维基本数据并行 *Kernel*，如图 14-16 所示，非常简单且易于理解。

```
q.submit([&](handler& h) {
    accessor input{input_buf, h};
    accessor output{output_buf, h};

    // Compute the average of each cell and its immediate
    // neighbors
    h.parallel_for(stencil_range, [=](id<2> idx) {
        int i = idx[0] + 1;
        int j = idx[1] + 1;

        float self = input[i][j];
        float north = input[i - 1][j];
        float east = input[i][j + 1];
        float south = input[i + 1][j];
        float west = input[i][j - 1];
        output[i][j] =
            (self + north + east + south + west) / 5.0f;
    });
});
```

图 14.16: 在数据并行 *Kernel* 中实现 *Stencil* 模式

然而，这种模板模式的表达方式非常幼稚，不应期望表现得很好。正如本章前面提到的，众所周知，需要利用局部性（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。图 14-17 显示了使用 Work-Groups 本地内存的空间阻塞的简单示例。

```

q.submit([&](handler& h) {
    accessor input{input_buf, h};
    accessor output{output_buf, h};

    constexpr size_t B = 4;
    range<2> local_range(B, B);
    range<2> tile_size =
        local_range +
        range<2>(2, 2); // Includes boundary cells
    auto tile = local_accessor<float, 2>(tile_size, h);

    // Compute the average of each cell and its immediate
    // neighbors
    h.parallel_for(
        nd_range<2>(stencil_range, local_range),
        [=](nd_item<2> it) {
            // Load this tile into work-group local memory
            id<2> lid = it.get_local_id();
            range<2> lrange = it.get_local_range();
            for (int ti = lid[0]; ti < B + 2;
                ti += lrange[0]) {
                int gi = ti + B * it.get_group(0);
                for (int tj = lid[1]; tj < B + 2;
                    tj += lrange[1]) {
                    int gj = tj + B * it.get_group(1);
                    tile[ti][tj] = input[gi][gj];
                }
            }
            group_barrier(it.get_group());
        }
    );
    // Compute the stencil using values from local
    // memory
    int gi = it.get_global_id(0) + 1;
    int gj = it.get_global_id(1) + 1;

    int ti = it.get_local_id(0) + 1;
    int tj = it.get_local_id(1) + 1;

    float self = tile[ti][tj];
    float north = tile[ti - 1][tj];
    float east = tile[ti][tj + 1];
    float south = tile[ti + 1][tj];
    float west = tile[ti][tj - 1];
    output[gi][gj] =
        (self + north + east + south + west) / 5.0f;
});
});

```

图 14.17: 使用 *Work-Groups* 本地内存实现 *ND* 范围 *Kernel* 中实现 *Stencil* 模式

为给定模板选择最佳优化需要在编译时对块大小、邻域和模板函数本身进行自省，这需要比此处讨论的更复杂的方法。

14.3.3 归约 Reduction

```
q.parallel_for(N, [=](id<1> i) {
    atomic_ref<int, memory_order::relaxed,
        memory_scope::system,
        access::address_space::global_space>(
            *sum) += data[i];
}).wait();
```

图 14.18: 实现以数据并行 *Kernel* 表示的朴素归约

```
q.parallel_for(nd_range<1>{N, B}, [=](nd_item<1> it) {
    int i = it.get_global_id(0);
    auto grp = it.get_group();
    int group_sum =
        reduce_over_group(grp, data[i], plus<>());
    if (grp.leader()) {
        atomic_ref<int, memory_order::relaxed,
            memory_scope::system,
            access::address_space::global_space>(
                *sum) += group_sum;
    }
}).wait();
```

图 14.19: 实现以 *ND* 范围 *Kernel* 表示的朴素归约

通过利用提供 Work-Items 之间的同步和通信功能的语言功能（例如原子操作、Work-Groups 和 Sub-Groups 功能、Sub-Groups“洗牌”），可以在 SYCL 中实现归约 Kernel。图 14-18 和图 14-19 中的 Kernel 显示了两种可能的归约实现：使用基本的 parallel_for 和每个 Work-Items 的原子操作的简单归约，以及使用 ND 范围的 parallel_for 和利用局部性的稍微聪明的归约。分别是 Work-Groups 归约功能。我们将在第 19 章中更详细地回顾这些原子操作。

还有许多其他方法可以编写归约 Kernel，并且由于原子操作的硬件支持、Work-Groups 本地内存大小、全局内存大小、快速设备范围屏障的可用

性或甚至可以使用专用的归约指令。在某些体系结构上，使用 $\log_2(N)$ 个单独的 Kernel 调用执行树归约甚至可能更快（或必要!）。

我们强烈建议仅在 SYCL 归约库不支持的情况下或在针对特定设备的功能微调 Kernel 时才应考虑手动实现归约，即使如此，也只有在 100% 确定 SYCL 的内置归约表现不佳！

14.3.4 扫描 Scan

正如我们在本章前面所看到的，实现并行扫描需要对数据进行多次扫描，并且每次扫描之间发生同步。由于 SYCL 不提供同步 ND 范围内所有 Work-Items 的机制，因此设备范围扫描的直接实现必须使用多个 Kernel，这些 Kernel 通过全局内存传达部分结果。

```
// Phase 1: Compute local scans over input blocks
q.submit([&](handler& h) {
    auto local = local_accessor<int32_t, 1>(L, h);
    h.parallel_for(nd_range<1>(N, L), [=](nd_item<1> it) {
        int i = it.get_global_id(0);
        int li = it.get_local_id(0);

        // Copy input to local memory
        local[li] = input[i];
        group_barrier(it.get_group());

        // Perform inclusive scan in local memory
        for (int32_t d = 0; d <= log2((float)L) - 1; ++d) {
            uint32_t stride = (1 << d);
            int32_t update =
                (li >= stride) ? local[li - stride] : 0;
            group_barrier(it.get_group());
            local[li] += update;
            group_barrier(it.get_group());
        }

        // Write the result for each item to the output
        // buffer Write the last result from this block to
        // the temporary buffer
        output[i] = local[li];
        if (li == it.get_local_range()[0] - 1) {
            tmp[it.get_group(0)] = local[li];
        }
    });
}).wait();
```

图 14.20: 在 *ND* 范围 *Kernel* 中实现全局包容性扫描的第 1 阶段: 跨每个 *Work-Groups* 进行计算

```

// Phase 2: Compute scan over partial results
q.submit([&](handler& h) {
    auto local = local_accessor<int32_t, 1>(G, h);
    h.parallel_for(nd_range<1>(G, G), [=](nd_item<1> it) {
        int i = it.get_global_id(0);
        int li = it.get_local_id(0);

        // Copy input to local memory
        local[li] = tmp[i];
        group_barrier(it.get_group());

        // Perform inclusive scan in local memory
        for (int32_t d = 0; d <= log2((float)G) - 1; ++d) {
            uint32_t stride = (1 << d);
            int32_t update =
                (li >= stride) ? local[li - stride] : 0;
            group_barrier(it.get_group());
            local[li] += update;
            group_barrier(it.get_group());
        }

        // Overwrite result from each work-item in the
        // temporary buffer
        tmp[i] = local[li];
    });
}).wait();

```

图 14.21: 在 ND 范围 Kernel 中实现全局包容性扫描的第 2 阶段: 扫描每个 Work-Groups 的结果

```

// Phase 3: Update local scans using partial results
q.parallel_for(nd_range<1>(N, L), [=](nd_item<1> it) {
    int g = it.get_group();
    if (g > 0) {
        int i = it.get_global_id(0);
        output[i] += tmp[g - 1];
    }
}).wait();

```

图 14.22: 第 3 阶段 (最终阶段), 用于在 ND 范围 Kernel 中实现全局包容性扫描

如图 14-20、14-21 和 14-22 所示的代码演示了使用多个 Kernel 实现的

包含扫描。第一个 Kernel 将输入值分布在 Work-Groups 之间，在 Work-Groups 本地内存中计算 Work-Groups 本地扫描（请注意，我们可以使用 Work-Groups `Include_scan` 函数来代替）。第二个 Kernel 使用单个 Work-Groups 计算本地扫描，这次是针对每个块的最终值。第三个 Kernel 结合这些中间结果来最终确定前缀和。这三个 Kernel 对应于图 14-5 中的三层。

图 14-20 和图 14-21 非常相似；唯一的区别是范围的大小以及输入和输出值的处理方式。此模式的实际实现可以使用采用不同参数的单个函数来实现这两个阶段，并且出于教学原因，它们仅在此处呈现为不同的代码。

14.3.5 打包和拆包

打包和解包也称为聚集和分散操作。这些操作处理数据在内存中的排列方式以及我们希望如何将其呈现给计算资源的差异。

打包 Pack 由于 pack 依赖于独占扫描，因此实现适用于 ND 范围的所有元素的 pack 也必须通过全局内存并在多个 Kernel 队列的过程中进行。然而，pack 有一个常见的用例，不需要将操作应用于 ND 范围的所有元素，即仅跨特定 Work-Groups 或 Sub-Groups 中的项目应用包。

```
uint32_t index =  
    exclusive_scan(g, (uint32_t)predicate, plus<>());  
if (predicate) dst[index] = value;
```

图 14.23: 在独占扫描之上实现组包操作

```

range<2> global(N, 8);
range<2> local(1, 8);
q.parallel_for(nd_range<2>(global, local), [=](nd_item<2>
    it) {
    int i = it.get_global_id(0);
    sub_group sg = it.get_sub_group();
    int sglid = sg.get_local_id()[0];
    int sgrange = sg.get_local_range()[0];

    uint32_t k = 0;
    for (int j = sglid; j < N; j += sgrange) {
        // Compute distance between i and neighbor j
        float r = distance(position[i], position[j]);

        // Pack neighbors that require
        // post-processing into a list
        uint32_t pack = (i != j) and (r <= CUTOFF);
        uint32_t offset =
            exclusive_scan_over_group(sg, pack, plus<>());
        if (pack) {
            neighbors[i * MAX_K + k + offset] = j;
        }

        // Keep track of how many neighbors have been
        // packed so far
        k += reduce_over_group(sg, pack, plus<>());
    }
    num_neighbors[i] =
        reduce_over_group(sg, k, maximum<>());
}).wait();

```

图 14.24: 使用 *Sub-Groups* 打包操作构建需要额外后处理的元素列表

图 14-23 中的代码片段展示了如何在独占扫描之上实现组包操作。

图 14-24 中的代码演示了如何在 Kernel 中使用此类打包操作来构建需要一些额外后处理（在未来的 Kernel 中）的元素列表。所示示例基于分子动力学模拟的真实 Kernel：分配给粒子 i 的 Sub-Groups 中的 Work-Items 合作识别 i 固定距离内的所有其他粒子，并且仅识别此“邻居列表”中的粒子将用于计算作用在每个粒子上的力。

请注意，打包模式永远不会对元素重新排序 - 打包到输出数组中的元素的显示顺序与输入中的顺序相同。pack 的这个属性很重要，它使我们能够使用 pack 功能来实现其他更抽象的并行算法（例如 std::copy_if 和 std::stable_partition）。然而，还有其他并行算法可以在不需要维护顺序的

包功能之上实现（例如 std::partition）。

```
uint32_t index =
    exclusive_scan(sg, (uint32_t)predicate, plus<>());
return (predicate) ? new_value[index] : original_value;
```

图 14.25: 在独占扫描之上实现 Sub-Groups 解压缩操作

```
// Keep iterating as long as one work-item has work to do
while (any_of_group(sg, i < Nx)) {
    uint32_t converged = next_iteration(
        params, i, j, count, cr, ci, zr, zi, mandelbrot);
    if (any_of_group(sg, converged)) {
        // Replace pixels that have converged using an
        // unpack. Pixels that haven't converged are not
        // replaced.
        uint32_t index = exclusive_scan_over_group(
            sg, converged, plus<>());
        i = (converged) ? iq + index : i;
        iq += reduce_over_group(sg, converged, plus<>());

        // Reset the iterator variables for the new i
        if (converged) {
            reset(params, i, j, count, cr, ci, zr, zi);
        }
    }
}
```

图 14.26: 使用 Sub-Groups 解包操作来改进具有不同控制流的 Kernel 的负载平衡

拆包 Unpack 与 pack 一样, 我们可以使用 scan 来实现 unpack。图 14-25 显示了如何在独占扫描之上实现 Sub-Groups 解包操作。

图 14-26 中的代码演示了如何使用这样的 Sub-Groups 解包操作来改善具有发散控制流的 Kernel 中的负载平衡 (在本例中, 计算 Mandelbrot 集)。每个 Work-Items 都分配有一个单独的像素来计算和迭代, 直到收敛或达到最大迭代次数。然后使用解包操作来用新像素替换完成的像素。

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, 因为检查完成情况和执行解包操作都会带来一些开销! 因此, 在实际应

用中成功使用此模式将需要根据存在的发散量和正在执行的计算进行一些微调（例如，仅当活动 Work-Items 的数量低于某个阈值时才引入启发式方法来执行解包操作）。

14.4 总结

本章演示了如何使用 SYCL 功能（包括内置函数和库）实现一些最常见的并行模式。

SYCL 生态系统仍在发展中，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，我们期望为这些模式发现新的最佳实践。

14.4.1 了解更多信息

结构化并行编程：高效计算模式，作者：Michael McCool、Arch Robison 和 James Reinders，2012，Morgan Kaufmann 出版，ISBN 978-0-124-15993-8。

算法库，C++ 参考，<https://en.cppreference.com/w/cpp/algorithms>。

15 GPU 编程

在过去的几十年里，图形处理单元 (GPU) 已经从能够在屏幕上绘制图像的专用硬件设备发展为能够执行复杂并行 Kernel 的通用设备。如今，几乎每台计算机都在传统 CPU 旁边配备了 GPU，并且可以通过将部分并行算法从 CPU 卸载到 GPU 来加速许多程序。

在本章中，我们将描述典型 GPU 的工作原理、GPU 软件和硬件如何执行 SYCL 应用程序，以及在为 GPU 编写和优化并行 Kernel 时要记住的提示和技术。

15.1 性能注意事项

与任何处理器类型一样，GPU 因供应商而异，甚至因产品一代而异；因此，一种设备的最佳实践可能并不适用于其他设备的最佳实践。本章中的建议可能会让许多 GPU 现在和将来受益，但是……

注 57 要实现特定 GPU 的最佳性能，请务必查阅 GPU 供应商的文档！

本章末尾提供了来自许多 GPU 供应商的文档链接。

15.2 GPU 的工作原理

本节介绍典型 GPU 的工作原理以及 GPU 与其他加速器类型的不同之处。

15.2.1 GPU 构建模块

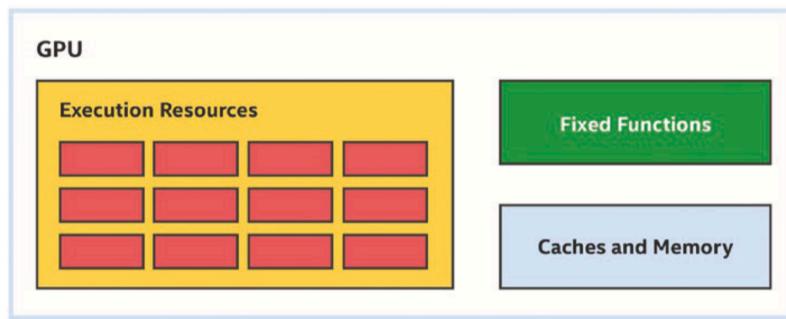


图 15.1: 典型的 GPU 构建块—无法扩展!

图 15-1 显示了一个非常简化的 GPU，由三个高级构建块组成：

1. **执行资源:** GPU 的执行资源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，但所有现代 GPU 都由多个可编程处理器组成。处理器可以是异构的并且专门用于特定任务，例如变换顶点和着色像素，或者它们可以是同构的并且可互换。大多数现代 GPU 的处理器都是同质且可互换的。
2. **固定函数:** GPU 固定函数是比执行资源可编程性更差的硬件单元，专门用于单个任务。当 GPU 用于图形时，图形管道的许多部分（例如光栅化或光线追踪）都是使用固定函数执行的，以提高功效和性能。当 GPU 用于数据并行计算时，固定函数可以用于诸如工作负载调度、纹理采样和依赖性跟踪等任务。
3. **高速缓存和内存:** 与其他处理器类型一样，GPU 经常具有高速缓存来存储执行资源访问的数据。GPU 缓存可以是隐式的，在这种情况下，它们不需要程序员执行任何操作，或者可以是显式暂存器存储器，在这种情况下，程序员必须在使用数据之前有目的地将数据移动到缓存中。许多 GPU 还拥有大量内存，可以快速访问执行资源所使用的数据。

15.2.2 更简单的处理器（但数量更多）

传统上，在执行图形操作时，GPU 会处理大量数据。例如，典型的游戏帧或渲染工作负载涉及数千个顶点，每帧产生数百万个像素。为了保持交互式帧速率，必须尽快处理这些大批量数据。

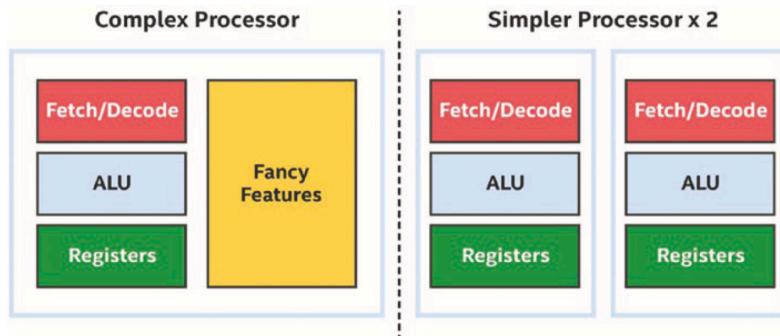


图 15.2: GPU 处理器更简单，但数量更多

典型的 GPU 设计权衡是消除形成加速单线程性能的执行资源的处理器的功能，并利用这些节省来构建额外的处理器，如图 15-2 所示。例如，GPU 处理器可能不包括其他类型处理器使用的复杂的乱序执行功能或分支预测逻辑。由于这些权衡，单个数据元素在 GPU 上的处理速度可能比在另一个处理器上的速度慢，但处理器数量较多使 GPU 能够快速高效地处理许多数据元素。

为了在执行 Kernel 时利用这种权衡，为 GPU 提供足够大范围的数据元素进行处理非常重要。为了证明卸载大量数据的重要性，请考虑我们在整本书中开发和修改的矩阵乘法 Kernel。

注 58 (关于矩阵乘法的提醒) 在本书中，矩阵乘法 Kernel 用于演示 Kernel 的变化或其调度方式如何影响性能。尽管使用本章中描述的技术显著提高了矩阵乘法性能，但矩阵乘法是一项非常重要且常见的操作，以至于许多硬件 (GPU、CPU、FPGA、DSP 等) 供应商已经实现了许多例程（包括矩阵乘法）的高度调整版本。这些供应商投入了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 Kernel 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，使用它几乎总是有益的，而不是将函数重新实现为 Kernel! *oneMKL* 项目 (*oneAPI* 的一部分) 提出了一些接口，这些接口将调用 *intel* 的 *MKL* 用于 *intel*, *cUBLAS* 用于 *NVIDIA*, *hipBlas* 用于 *AMD*。如果此类接口可用，它们可能会使事情变得更容易。否则，我们需要做自己的工作，以确保我们为目标硬件使用最好的库。

通过将矩阵乘法 Kernel 作为单个任务提交到队列中，可以在 GPU 上轻松执行。该矩阵乘法 Kernel 的主体看起来与主机 CPU 上执行的函数完全相同，如图 15-3 所示。

```
h.single_task( [=] () {
    for (int m = 0; m < M; m++) {
        for (int n = 0; n < N; n++) {
            T sum = 0;
            for (int k = 0; k < K; k++) {
                sum += matrixA[m * K + k] * matrixB[k * N + n];
            }
            matrixC[m * N + n] = sum;
        }
    }
});
```

图 15.3: 单任务矩阵乘法看起来很像 CPU 主机代码

如果我们尝试在 CPU 上执行该 Kernel，它的性能可能还不错，但不是很好，因为预计它不会利用 CPU 的任何并行功能，但对于小矩阵大小来说可能足够好。然而，如图 15-4 所示，如果我们尝试在 GPU 上执行该 Kernel，它的性能可能会很差，因为单个任务只会使用单个 GPU 处理器。

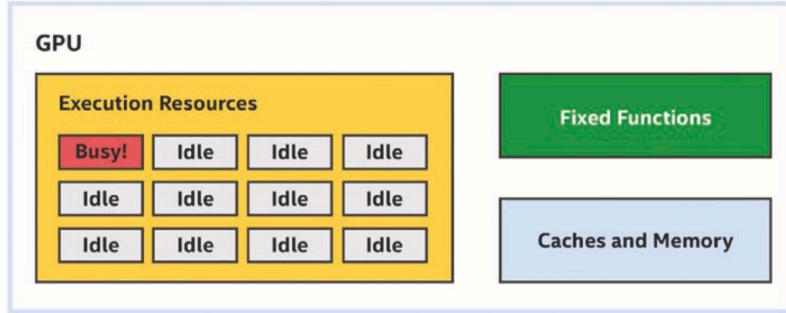


图 15.4: GPU 上的单任务 Kernel 会使许多执行资源处于空闲状态

表达并行性 为了提高该 Kernel 对于 CPU 和 GPU 的性能, 我们可以通过将其中一个循环转换为 parallel_for 来提交一系列数据元素进行并行处理。对于矩阵乘法 Kernel, 我们可以选择提交代表两个最外层循环之一的一系列数据元素。在图 15-5 中, 我们选择并行处理结果矩阵的行。

```
h.parallel_for(range{M}, [=](id<1> idx) {
    int m = idx[0];

    for (int n = 0; n < N; n++) {
        T sum = 0;
        for (int k = 0; k < K; k++) {
            sum += matrixA[m * K + k] * matrixB[k * N + n];
        }
        matrixC[m * N + n] = sum;
    }
});
```

图 15.5: 某种平行矩阵乘法

注 59 (选择如何并行化) 选择要并行化的维度是针对 GPU 和其他设备类型调整应用程序的一种非常重要的方法。本章的后续部分将介绍为什么在一个维度中并行化可能比在另一个维度中并行化效果更好的一些原因。

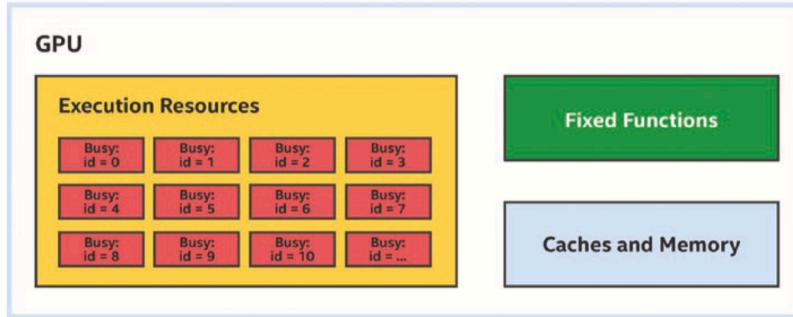


图 15.6: 某种程度上并行的 Kernel 使更多的处理器资源保持忙碌

尽管有点并行的 Kernel 与单任务 Kernel 非常相似，但它在 CPU 上运行得更好，在 GPU 上运行得更好。如图 15-6 所示，parallel_for 使代表结果矩阵行的 Work-Items 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状态。

请注意，未指定行分区和分配给不同处理器资源的确切方式，从而使实现能够灵活地选择如何最好地在设备上执行 Kernel。例如，实现可以选择在同一处理器上执行连续的行以获得局部性优势，而不是在处理器上执行单独的行。

```

h.parallel_for(range{M, N}, [=](id<2> idx) {
    int m = idx[0];
    int n = idx[1];

    T sum = 0;
    for (int k = 0; k < K; k++) {
        sum += matrixA[m * K + k] * matrixB[k * N + n];
    }

    matrixC[m * N + n] = sum;
});
```

图 15.7: 更多并行矩阵乘法

表达更多的并行性 通过选择并行处理两个外部循环，我们可以进一步并行化矩阵乘法 Kernel。因为 parallel_for 可以在最多三个维度上表达并行循

环, 所以这很简单, 如图 15-7 所示。在图 15-7 中, 请注意传递给 parallel_for 的范围和表示并行执行空间中索引的项现在都是二维的。

当在 GPU 上运行时, 暴露额外的并行性可能会提高矩阵乘法 Kernel 的性能。即使矩阵行数超过 GPU 处理器的数量, 情况也可能如此。接下来的几节将描述出现这种情况的可能原因。

15.2.3 简化的控制逻辑 (SIMD 指令)

许多 GPU 处理器利用大多数数据元素倾向于通过 Kernel 采用相同的控制流路径这一事实来优化控制逻辑。例如, 在矩阵乘法 Kernel 中, 每个数据元素执行最内层循环的次数相同, 因为循环边界是不变的。

当数据元素通过 Kernel 采用相同的控制流路径时, 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组执行来降低管理指令流的成本。实现此目的的一种方法是实现单指令、多数据或 SIMD 指令集, 其中单指令同时处理多个数据元素。

注 60 (线程 VS. 指令流) 在许多并行编程上下文和 GPU 文献中, 术语“线程”用于表示“指令流”。在这些上下文中, “线程”不同于传统的操作系统线程, 通常更轻量级。然而, 情况并非总是如此, 在某些情况下, “线程”被用来描述完全不同的东西。

由于术语“线程”过载且容易被误解, 即使在不同的 GPU 供应商之间也是如此, 因此本章改用术语“指令流”。

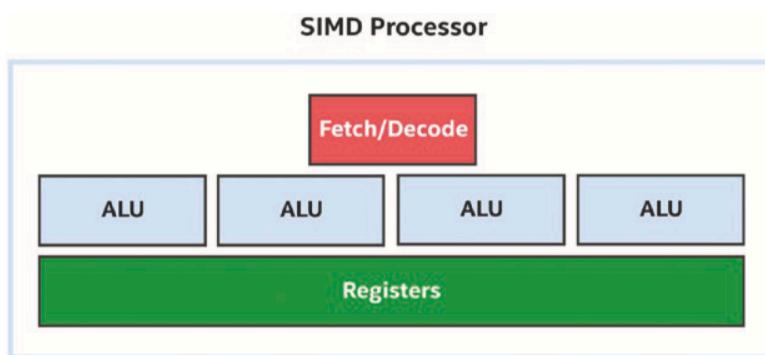


图 15.8: 四宽 SIMD 处理器: 四个 ALU 共享获取/解码逻辑

由单个指令同时处理的数据元素的数量有时被称为指令或执行指令的处理器的 SIMD 宽度。在图 15-8 中，四个 ALU 共享相同的控制逻辑，因此可以将其描述为四宽 SIMD 处理器。

GPU 处理器并不是唯一实现 SIMD 指令集的处理器。其他处理器类型也实现 SIMD 指令集，以提高处理大型数据集时的效率。GPU 处理器与其他处理器类型之间的主要区别在于，GPU 处理器依靠并行执行多个数据元素来实现良好的性能，并且 GPU 处理器可以支持比其他处理器类型更宽的 SIMD 宽度。例如，GPU 处理器支持 16、32 或更多数据元素的 SIMD 宽度并不罕见。

注 61 (编程模型：SPMD 和 SIMD) 尽管 GPU 处理器实现具有不同宽度的 SIMD 指令集，但这通常是一个实现细节，并且对于在 GPU 处理器上执行数据并行 Kernel 的应用程序是透明的。这是因为许多 GPU 编译器和运行时 APIs 实现单个程序、多个数据或 SPMD 编程模型，其中 GPU 编译器和运行时 APIs 确定要使用 SIMD 指令流处理的最有效的数据元素组，而不是显式表示 SIMD 指令。第 9 章的“Sub-Groups”部分探讨了数据元素分组对应用程序可见的情况。

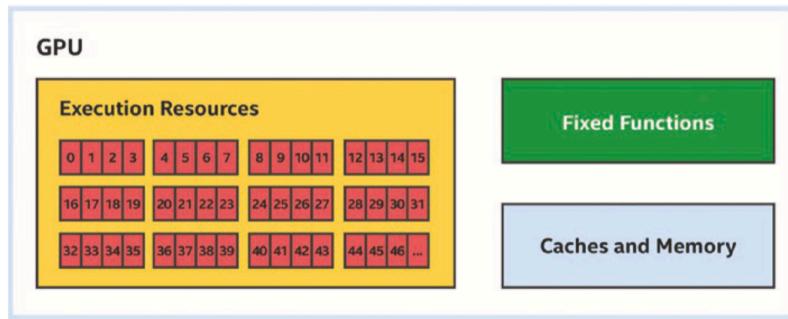


图 15.9: 在 SIMD 处理器上执行某种并行 Kernel

在图 15-9 中，我们扩展了每个执行资源以支持四宽 SIMD，从而允许我们并行处理四倍数量的矩阵行。

使用并行处理多个数据元素的 SIMD 指令是图 15-5 和 15-7 中并行矩阵乘法 Kernel 的性能能够扩展到超出处理器数量的方法之一。在许多情况下，SIMD 指令的使用还提供了自然的局部性优势，包括通过在同一处理器上执行连续数据元素来进行矩阵乘法。

注 62 *Kernel* 受益于处理器之间的并行性和处理器内的并行性！

预测和掩蔽 只要所有数据元素通过 Kernel 中的条件代码采用相同的路径，在多个数据元素之间共享指令流就可以很好地工作。当数据元素通过条件代码采取不同的路径时，控制流被称为发散。当控制流在 SIMD 指令流中出现分歧时，通常会执行两个控制流路径，并屏蔽或预测某些通道。这确保了正确的行为，但正确性是以性能为代价的，因为被屏蔽的通道不会执行有用的工作。

```

h.parallel_for(array_size, [=](id<1> i) {
    auto condition = i[0] & 1;
    if (condition) {
        dataAcc[i] = dataAcc[i] * 2; // odd
    } else {
        dataAcc[i] = dataAcc[i] + 1; // even
    }
});
```

图 15.10: 具有发散控制流的 *Kernel*

为了展示预测和屏蔽的工作原理，请考虑图 15-10 中的 Kernel，它将每个具有“奇数”索引的数据元素乘以 2，并将每个具有“偶数”索引的数据元素递增 1。

假设我们在图 15-8 所示的四宽 SIMD 处理器上执行此 Kernel，并且我们在一个 SIMD 指令流中执行前四个数据元素，在不同的 SIMD 指令流中执行接下来的四个数据元素，等等。图 15-11 显示了可以屏蔽通道和预测执行以使用发散控制流正确执行该 Kernel 的方法之一。

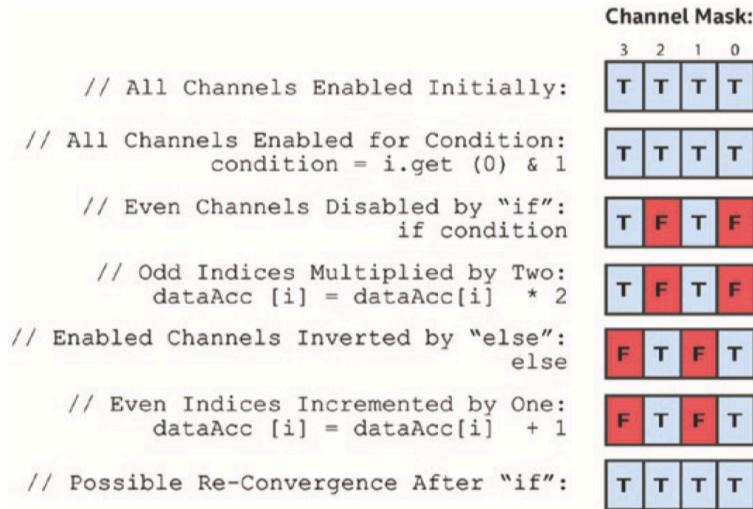


图 15.11: 发散 Kernel 的可能通道掩码

SIMD 效率 SIMD 效率衡量 SIMD 指令流与等效标量指令流相比的性能。在图 15-11 中, 由于控制流将通道分为两个相等的组, 因此发散控制流中的每条指令的执行效率只有一半。

在最坏的情况下, 对于高度发散的 Kernel, 效率可能会降低处理器 SIMD 宽度的一个因子。

所有实现 SIMD 指令集的处理器都将遭受影响 SIMD 效率的发散性惩罚, 但由于 GPU 处理器通常支持比其他处理器类型更宽的 SIMD 宽度, 因此在优化时重构算法以最小化发散控制流并最大化收敛执行可能特别有益 GPU 的 Kernel。这并不总是可能的, 但作为示例, 选择沿具有更收敛执行的维度进行并行化可能比沿具有高度发散的执行的不同维度进行并行化表现得更好。

SIMD 效率和项目组 到目前为止, 本章中的所有 Kernel 都是基本数据并行 Kernel, 它们没有指定执行范围内的任何项目分组, 这使得实现可以自由地选择设备的最佳分组。例如, 具有较宽 SIMD 宽度的设备可能更喜欢较大的分组, 但具有较窄 SIMD 宽度的设备可能适合较小的分组。

当 Kernel 是具有显式 Work-Items 分组的 ND 范围 Kernel 时, 应注意选择能够最大化 SIMD 效率的 ND 范围 Work-Groups 大小。当 Work-Groups

大小不能被处理器的 SIMD 宽度整除时，部分 Work-Groups 可能会在 Kernel 的整个持续时间内禁用通道来执行。针对 preferred_work_group_size_multiple 的设备特定 Kernel 查询可用于选择有效的 Work-Groups 大小。有关如何查询设备属性的详细信息，请参阅第 12 章。

```

h.parallel_for(
    nd_range<1>{M, 1}, [=](nd_item<1> idx) {
        int m = idx.get_global_id(0);

        for (int n = 0; n < N; n++) {
            T sum = 0;
            for (int k = 0; k < K; k++) {
                sum += matrixA[m * K + k] * matrixB[k * N + n];
            }
            matrixC[m * N + n] = sum;
        }
    });

```

图 15.12: 低效的单项、有点平行的矩阵乘法

选择由单个 Work-Items 组成的 Work-Groups 大小可能会表现得很差，因为许多 GPU 将通过屏蔽除一个通道之外的所有 SIMD 通道来实现单 Work-Items 的 Work-Groups。例如，图 15-12 中的 Kernel 的性能可能会比图 15-5 中非常相似的 Kernel 差很多，尽管两者之间唯一的显着区别是从基本数据并行 Kernel 到低效的单并行 Kernel 的变化。Work-Items ND 范围 Kernel (`nd_range<1>{M, 1}`)。

15.2.4 切换工作以隐藏延迟

许多 GPU 使用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处理器上，而不是在处理器上执行单个指令流。

在处理器上驻留多个指令流是有益的，因为它使每个处理器都可以选择要执行的工作。如果一个指令流正在执行长延迟操作，例如从内存读取，则处理器可以切换到准备运行的不同指令流，而不是等待操作完成。有了足够的指令流，当处理器切换回原始指令流时，长延迟操作可能已经完成，而无需处理器等待。

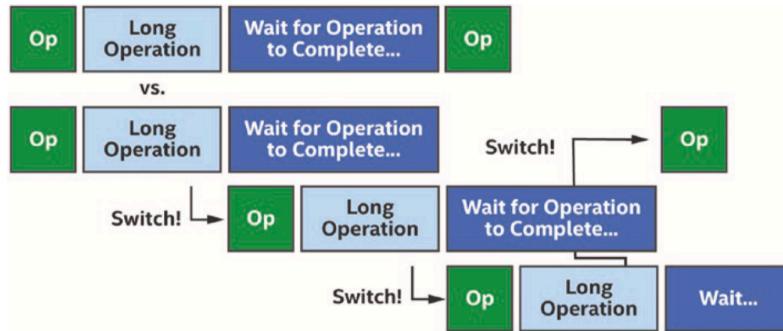


图 15.13: 切换指令流以隐藏延迟

图 15-13 显示了处理器如何使用多个同时指令流来隐藏延迟并提高性能。尽管第一个指令流在多个流中执行的时间稍长，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，而无需空闲地等待长时间操作完成。

GPU 分析工具可以使用诸如占用率之类的术语来描述 GPU 处理器目前正在执行的指令流数量与理论指令流总数的关系。

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保持忙碌。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都执行低效、长延迟的操作，GPU 处理器可能仍然需要等待。不过，在其他条件相同的情况下，增加占用率可以最大限度地提高 GPU 处理器隐藏延迟的能力，并且通常会提高性能。增加占用率是使用图 15-7 中更加并行的 Kernel 可以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 GPU 和数据并行处理。回想一下图 15-2，GPU 处理器通常比其他处理器类型更简单，因此缺乏复杂的延迟隐藏功能。这使得 GPU 处理器更容易受到延迟问题的影响，但由于数据并行编程涉及处理大量数据，GPU 处理器通常有大量指令流要执行！

15.3 将 Kernel 卸载到 GPU

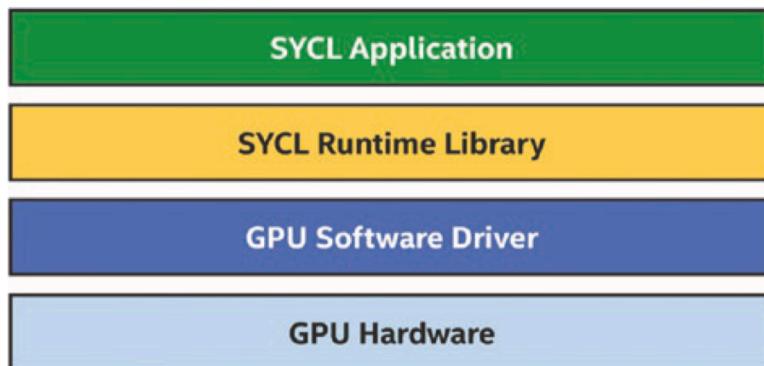


图 15.14: 将并行 *Kernel* 卸载到 *GPU* (简化)

本节介绍应用程序、SYCL 运行时库和 GPU 软件驱动程序如何协同工作以在 GPU 硬件上卸载 Kernel。图 15-14 中的图表显示了具有这些抽象层的典型软件堆栈。在许多情况下，这些层的存在对于应用程序来说是透明的，但在调试或分析应用程序时理解并考虑它们非常重要。

15.3.1 SYCL 运行时库

SYCL 运行时库是 SYCL 应用程序与之交互的主要软件库。运行时库负责实现队列、Buffer 和访问器等类以及这些类的成员函数。运行时库的某些部分可能位于头文件中，因此可以直接编译到应用程序可执行文件中。运行时库的其他部分作为库函数实现，作为应用程序构建过程的一部分与应用程序可执行文件链接。运行时库通常不是特定于设备的，并且同一运行时库可以协调卸载到 CPU、GPU、FPGA 或其他设备。

15.3.2 GPU 软件驱动程序

尽管理论上 SYCL 运行时库可以直接卸载到 GPU，但实际上，大多数 SYCL 运行时库与 GPU 软件驱动程序连接以将工作提交到 GPU。

GPU 软件驱动程序通常是 API 的实现，例如 OpenCL、零级或 CUDA。大多数 GPU 软件驱动程序都是在 SYCL 运行时调用的用户模式驱动程序库中实现的，并且用户模式驱动程序可以调用操作系统或 Kernel 模式驱动

程序来执行系统级任务，例如分配内存或向设备提交工作。用户模式驱动程序还可以调用其他用户模式库；例如，GPU 驱动程序可以调用 GPU 编译器来将 Kernel 从中间表示及时编译为 GPU ISA（指令集架构）。这些软件模块以及它们之间的交互如图 15-15 所示。

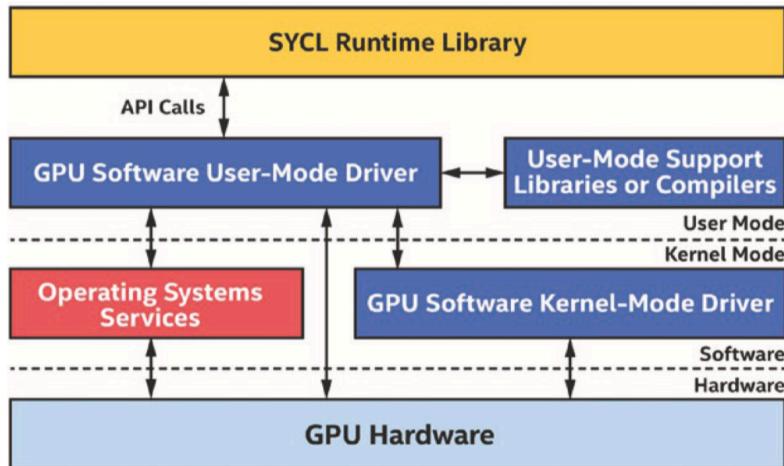


图 15.15: 典型的 GPU 软件驱动程序模块

15.3.3 GPU 硬件

当运行时库或 GPU 软件用户模式驱动程序被明确请求提交工作时，或者当 GPU 软件试探性地确定工作应该开始时，它通常会通过操作系统或 Kernel 模式驱动程序调用来开始执行工作 GPU。在某些情况下，GPU 软件用户模式驱动程序可能会直接向 GPU 提交工作，但这种情况不太常见，并且可能并非所有设备或操作系统都支持。

当 GPU 上执行的工作结果被主机处理器或另一个加速器消耗时，GPU 必须发出信号以指示工作已完成。工作完成涉及的步骤与工作提交的步骤非常相似，执行方向相反：GPU 可能会向操作系统或 Kernel 模式驱动程序发出信号，表明其已完成执行，然后通知用户模式驱动程序，最后运行时库将通过 GPU 软件 API 调用观察到工作已完成。

每个步骤都会引入延迟，并且在许多情况下，运行时库和 GPU 软件会在较低延迟和较高吞吐量之间进行权衡。例如，更频繁地向 GPU 提交工作可能会减少延迟，但频繁提交也可能会因每次提交的开销而降低吞吐量。收

集大批量的工作会增加延迟，但可以分摊更多工作的提交开销，并引入更多并行执行的机会。运行时和驱动程序经过调整以做出正确的权衡，并且通常会做得很好，但是如果我们怀疑驱动程序启发式提交工作效率低下，我们应该查阅文档以查看是否有方法使用 API 覆盖默认驱动程序行为 - 特定的甚至特定于实现的机制。第 20 章中描述的直接与 API 后端交互的技术对于调整 GPU 提交策略非常有用。

15.3.4 当心卸载成本！

尽管 SYCL 实现和 GPU 供应商不断创新和优化，以降低将工作卸载到 GPU 的成本，但在 GPU 上开始工作以及在主机或其他设备上观察结果时总会产生开销。选择在何处执行算法时，请考虑在设备上执行算法的好处以及将算法及其所需的任何数据移动到设备的成本。在某些情况下，使用主机处理器执行并行操作可能是最有效的，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。

注 63 考虑我们算法的整体性能 - 在一台设备上低效地执行部分算法可能比将执行转移到另一台设备更有效！

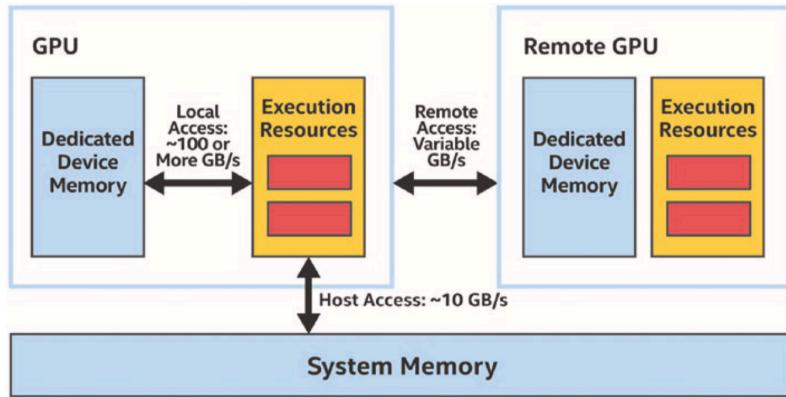


图 15.16: 设备内存、远程内存和主机内存之间的典型差异

与设备内存之间的传输 在具有专用内存的 GPU 上，请特别注意专用 GPU 内存与主机或其他设备上的内存之间的传输成本。图 15-16 显示了系统中不同内存类型之间的典型内存带宽差异。

回想一下第 3 章, GPU 更喜欢在专用设备内存上运行, 这样速度可以快一个数量级或更多, 而不是在主机内存或其他设备的内存上运行。尽管对专用设备内存的访问比对远程内存或系统内存的访问要快得多, 但如果数据尚未位于专用设备内存中, 则必须对其进行复制或迁移。

只要数据会被频繁访问, 将其移至专用设备内存中就是有益的, 特别是当 GPU 执行资源忙于处理另一项任务时可以异步执行传输时。然而, 当数据访问不频繁或不可预测时, 即使每次访问成本较高, 最好还是节省传输成本并远程或在系统内存中对数据进行操作。第 6 章介绍了控制内存分配位置的方法以及将数据复制和预取到专用设备内存的不同技术。这些技术在优化 GPU 程序执行时非常重要。

15.4 GPU Kernel 最佳实践

前面的部分描述了传递给 parallel_for 的调度参数如何影响 Kernel 分配给 GPU 处理器资源的方式以及在 GPU 上执行 Kernel 所涉及的软件层和开销。本节介绍 Kernel 在 GPU 上执行时的最佳实践。

从广义上讲, Kernel 要么是内存限制的, 这意味着它们的性能受到 GPU 上执行资源的数据读写操作的限制, 要么是计算限制, 这意味着它们的性能受到 GPU 上执行资源的限制。为 GPU (以及许多其他处理器) 优化 Kernel 时, 第一步是确定我们的 Kernel 是内存限制型还是计算限制型, 因为经常改进内存限制型 Kernel 的技术不会使计算限制型 Kernel 受益反之亦然。GPU 供应商通常会提供分析工具来帮助做出这一决定。

注 64 需要不同的优化技术, 这取决于我们的 *Kernel* 是内存受限还是计算受限!

由于 GPU 往往具有许多处理器和较宽的 SIMD 宽度, 因此 Kernel 往往更多地受到内存限制而不是计算限制。如果我们不确定从哪里开始, 检查 Kernel 如何访问内存是一个很好的第一步。

15.4.1 访问全局内存

有效访问全局内存对于优化应用程序性能至关重要, 因为 Work-Items 或 Work-Groups 操作的几乎所有数据都源自全局内存。如果 Kernel 对全局内存的操作效率低下, 它几乎总是会表现得很差。尽管 GPU 通常包含用于读取和写入内存中任意位置的专用硬件收集和分散单元, 但对全局内

存的访问性能通常由数据访问的局部性驱动。如果 Work-Groups 中的一个 Work-Items 正在访问存储器中与 Work-Groups 中的另一 Work-Items 所访问的元素相邻的元素，则全局存储器访问性能可能良好。如果 Work-Groups 中的 Work-Items 改为跨步或随机访问内存，则全局内存访问性能可能会更差。一些 GPU 文档将附近内存访问的操作描述为合并内存访问。

回想一下，对于图 15-5 中的稍微并行的矩阵乘法 Kernel，我们可以选择是否并行处理结果矩阵的行或列，并且我们选择并行操作结果矩阵的行。事实证明这是一个糟糕的选择：如果 id 等于 m 的一个 Work-Items 与 id 等于 $m-1$ 或 $m+1$ 的相邻 Work-Items 分组，则用于访问矩阵 B 的索引对于每个 Work-Items 都是相同的 Work-Items，但用于访问矩阵 A 的索引相差 K ，这意味着访问是高度跨步的。矩阵 A 的访问模式如图 15-17 所示。

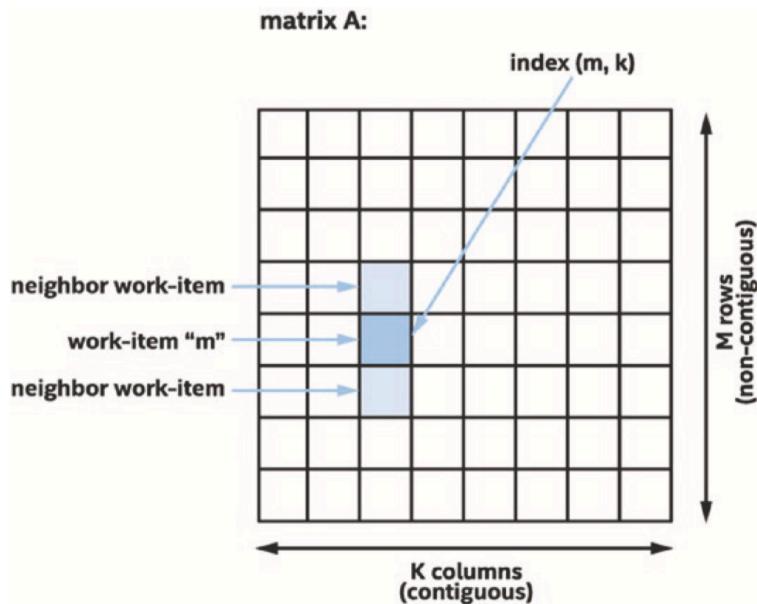


图 15.17: 对矩阵 A 的访问非常缓慢且效率低下

相反，如果我们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 Kernel 在结构上与图 15-5 中的 Kernel 非常相似，唯一的区别是图 15-18 中的每个 Work-Items 对结果矩阵的列进行操作，而不是对结果矩阵的行进行操作。

```
h.parallel_for(N, [=](item<1> idx) {
    int n = idx[0];

    for (int m = 0; m < M; m++) {
        T sum = 0;
        for (int k = 0; k < K; k++) {
            sum += matrixA[m * K + k] * matrixB[k * N + n];
        }
        matrixC[m * N + n] = sum;
    }
});
```

图 15.18: 并行计算结果矩阵的列, 而不是行

尽管这两个 Kernel 在结构上非常相似, 但在许多 GPU 上, 对数据列进行操作的 Kernel 将显着优于对数据行进行操作的 Kernel, 这纯粹是由于更高效的内存访问: 如果一个 Work-Items 的 id 相等 to n 与 id 等于 n-1 或 n+1 的相邻 Work-Items 分组, 用于访问矩阵 A 的索引现在对于每个 Work-Items 都是相同的, 并且用于访问矩阵 B 的索引是连续的。矩阵 B 的访问模式如图 15-19 所示。

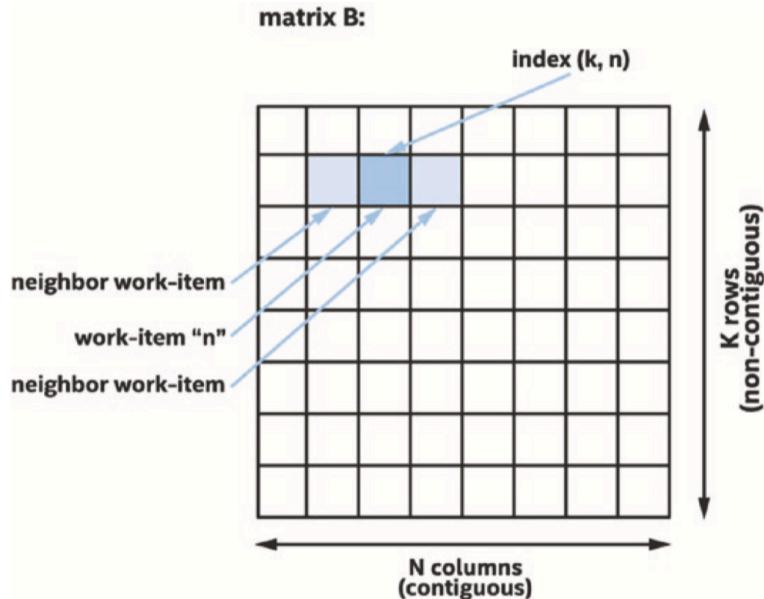


图 15.19: 对 *matrixB* 的访问是连续且高效的

对连续数据的访问通常非常有效。一个好的经验法则是，一组 Work-Items 对全局内存的访问性能是所访问的 GPU 缓存行数量的函数。如果所有访问都在单个高速缓存行内，则访问将以峰值性能执行。如果访问需要两个高速缓存行，例如通过访问每个其他元素或从高速缓存未对齐的地址开始，则访问可能会以一半的性能运行。当组中的每个 Work-Items 访问唯一的高速缓存行时，例如跨步或随机访问，访问可能会以最低性能运行。

注 65 (分析 Kernel 变体) 对于矩阵乘法，选择沿一个维度并行化显然会导致更有效的内存访问，但对于其他 Kernel，选择可能并不那么明显。对于实现最佳性能很重要的 Kernel，如果不清楚要并行化哪个维度，有时值得开发和分析沿每个维度并行化的不同 Kernel 变体，以了解什么更适合设备和数据集。

15.4.2 访问 Work-Groups 本地内存

在上一节中，我们描述了对全局内存的访问如何受益于局部性，以最大限度地提高缓存性能。正如我们所看到的，在某些情况下，我们可以设计算

法来有效地访问内存，例如选择在一个维度而不是另一个维度进行并行化。然而，这种技术并非在所有情况下都可行。本节介绍如何使用 Work-Groups 本地内存来有效支持更多内存访问模式。

回想一下第 9 章，Work-Groups 中的 Work-Items 可以通过 Work-Groups 本地内存进行通信并使用 Work-Groups 障碍进行同步来合作解决问题。该技术对于 GPU 特别有利，因为典型的 GPU 具有专门的硬件来实现屏障和 Work-Groups 本地内存。不同的 GPU 供应商和不同的产品可能会以不同的方式实现 Work-Groups 本地内存，但与全局内存相比，Work-Groups 本地内存通常有两个好处：本地内存可以支持比全局内存访问更高的带宽和更低的延迟，即使全局内存访问会命中缓存，本地内存通常分为不同的内存区域，称为存储体。只要组中的每个 Work-Items 访问不同的存储体，本地内存访问就会以最佳性能执行。与全局内存相比，分组访问允许本地内存以峰值性能支持更多的访问模式。

许多 GPU 供应商会将连续的本地内存地址分配给不同的 bank。这确保了连续的内存访问始终以最佳性能运行，无论起始地址如何。然而，当内存访问跨步时，组中的某些 Work-Items 可能会访问分配给同一存储体的内存地址。发生这种情况时，会被视为存储体冲突并导致串行访问和性能降低。

注 66 为了实现最大的全局内存性能，请最大程度地减少访问的缓存行数。

为了获得最大的本地内存性能，请最大限度地减少 bank 冲突的数量！

	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!

图 15.20: 不同访问模式、全局和本地内存的可能性能

全局内存和本地内存的访问模式和预期性能摘要如图 15-20 所示。假设当 ptr 指向全局内存时，指针与 GPU 缓存行的大小对齐。通过从缓存对齐地址开始连续访问内存，可以实现访问全局内存时的最佳性能。访问未对齐的地址可能会降低全局内存性能，因为该访问可能需要访问额外的缓存行。由于访问未对齐的本地地址不会导致额外的存储体冲突，因此本地内存性能保持不变。

跨步案例值得更详细地描述。访问全局内存中的每个其他元素需要访问更多的缓存行，并且可能会导致性能降低。访问本地内存中的所有其他元素可能会导致存储体冲突和性能降低，但前提是存储体的数量可以被 2 整除。如果银行的数量是奇数，这种情况也将在全性能下运行。

当访问之间的步幅非常大时，每个 Work-Items 都会访问唯一的缓存行，从而导致性能最差。但对于本地内存，性能取决于步幅和存储体数量。当步长 N 等于 Bank 数时，每次访问都会导致 Bank 冲突，并且所有访问都是串行的，导致性能最差。然而，如果步长 M 和存储体数量没有共同因素，则访问将以全部性能运行。因此，许多优化的 GPU Kernel 会在本地内存中填充数据结构，以选择减少或消除存储体冲突的步幅。

15.4.3 通过 Sub-Groups 完全避免本地内存

正如第 9 章所讨论的，Sub-Groups 集体功能是在组中的 Work-Items 之间交换数据的另一种方法。对于许多 GPU 来说，Sub-Groups 代表由单个指令流处理的 Work-Items 的集合。在这些情况下，Sub-Groups 中的 Work-Items 可以廉价地交换数据并同步，而无需使用 Work-Groups 本地存储器。许多性能最佳的 GPU Kernel 都使用 Sub-Groups，因此对于昂贵的 Kernel，非常值得检查我们的算法是否可以重新表述为使用 Sub-Groups 集体函数。

15.4.4 使用小数据类型优化计算

本节介绍在消除或减少内存访问瓶颈后优化 Kernel 的技术。需要牢记的一个非常重要的观点是，GPU 传统上被设计用于在屏幕上绘制图片。尽管 GPU 的纯计算能力随着时间的推移不断发展和提高，但在某些领域，其图形传统仍然很明显。

例如，考虑对 Kernel 数据类型的支持。许多 GPU 针对 32 位浮点运算进行了高度优化，因为这些运算在图形和游戏中很常见。对于可以处理较低精度的算法，许多 GPU 还支持较低精度的 16 位浮点类型，以牺牲精度来

换取更快的处理速度。相反，虽然许多 GPU 确实支持 64 位双精度浮点运算，但额外的精度是有代价的，而且 32 位运算的性能通常比 64 位等效运算要好得多。

对于整数数据类型也是如此，其中 32 位整数数据类型的性能通常优于 64 位整数数据类型，而 16 位整数的性能可能甚至更好。如果我们可以构建我们的计算以使用较小的整数，我们的 Kernel 可能会执行得更快。需要特别注意的一个领域是寻址操作，它通常对 64 位 `size_t` 数据类型进行操作，但有时可以重新排列以使用 32 位数据类型执行大部分计算。在某些本地内存情况下，16 位索引就足够了，因为大多数本地内存分配都很小。

15.4.5 优化数学函数

Kernel 可能会为了性能而牺牲准确性的另一个领域涉及 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明确定义的精度。大多数 GPU 本身并不支持这些功能，而是使用一长串其他指令来实现它们。尽管数学函数实现通常针对 GPU 进行了很好的优化，但如果我们的应用程序可以容忍较低的精度，我们应该考虑采用精度较低但性能较高的不同实现。有关 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 库包括具有降低的或实现定义的精度要求的快速或本机函数变体。对于某些 GPU，这些函数可能比其精确的等效函数快一个数量级，因此如果它们对算法具有足够的精度，则非常值得考虑。例如，许多图像后处理算法具有明确定义的输入，并且可以容忍较低的精度，因此是使用快速或本机数学函数的良好候选者。

15.4.6 特化功能和扩展

优化 GPU Kernel 时的最后一个考虑因素是许多 GPU 中常见的专用指令。举一个例子，几乎所有 GPU 都支持 `mad` 或 `fma` 乘加指令，该指令在单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个乘法和加法以使用单个指令，但 SYCL 还包括可以显式调用的 `mad` 和 `fma` 函数。当然，如果我们希望 GPU 编译器为我们优化乘法和加法，我们应该确保我们不会通过禁用浮点收缩来阻止优化！

其他专用 GPU 指令可能只能通过编译器优化、SYCL 语言扩展或直接与低级 GPU 后端交互来获得。例如，某些 GPU 支持专门的点积累加指令，编译器将尝试识别并优化该指令，或者可以直接调用该指令。有关如何查询

GPU 实现支持的扩展的更多信息，请参阅第 12 章，有关后端互操作性的信息，请参阅第 20 章。

注 67 如果算法可以容忍较低的精度，我们可以使用更小的数据类型或更低精度的数学函数来提高性能！

15.5 总结

在本章中，我们首先描述典型 GPU 的工作原理以及 GPU 与传统 CPU 的不同之处。我们描述了如何通过将加速单个指令流的处理器功能换成额外的处理器来针对大量数据进行优化。

我们描述了 GPU 如何使用宽 SIMD 指令并行处理多个数据元素，以及 GPU 如何使用预测和掩码来使用 SIMD 指令执行具有复杂流程控制的 Kernel。我们讨论了预测和掩码如何降低 SIMD 效率并降低高度发散的 Kernel 的性能，以及选择沿一个维度与另一维度并行化如何减少 SIMD 发散。

由于 GPU 拥有如此多的处理资源，我们讨论了为 GPU 提供足够的工作以保持高占用率的重要性。我们还描述了 GPU 如何使用指令流来隐藏延迟，这使得为 GPU 提供大量执行工作变得更加重要。

接下来，我们讨论了将 Kernel 卸载到 GPU 所涉及的软件和硬件层以及卸载的成本。我们讨论了在单个设备上执行算法如何比将执行从一个设备转移到另一个设备更有效。

最后，我们描述了 Kernel 在 GPU 上执行时的最佳实践。我们描述了有多少 Kernel 从内存限制开始，以及如何有效地访问全局内存和本地内存，或者如何通过使用 Sub-Groups 操作完全避免本地内存。当 Kernel 受计算限制时，我们描述了如何通过以较低精度换取更高性能或使用自定义 GPU 扩展来访问专用指令来优化计算。

15.5.1 了解更多信息

关于 GPU 编程还有很多东西需要学习，本章只是触及了表面！

GPU 规范和白皮书是了解有关特定 GPU 和 GPU 架构的更多信息的好方法。许多 GPU 供应商提供了有关其 GPU 以及如何对其进行编程的非常详细的信息。

在撰写本文时，有关主要 GPU 的相关阅读可在 software.intel.com、devblogs.nvidia.com 和 amd.com 上找到。

一些 GPU 供应商拥有开源驱动程序或驱动程序组件。如果可用，检查或单步执行驱动程序代码可能会很有帮助，以了解哪些操作成本较高或应用程序中哪些地方可能存在开销。

本章完全关注通过 Buffer 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 还包含一个固定函数纹理采样器，可以加速图像操作。有关图像和采样器的更多信息，请参阅 SYCL 规范。

16 CPU 编程

Kernel 编程最初作为一种 GPU 编程方式而流行。随着 Kernel 编程的普遍化，了解 Kernel 编程风格如何影响代码到 CPU 的映射非常重要。

CPU 已经发展了很多年。2005 年左右发生了重大转变，当时时钟速度提高所带来的性能提升逐渐减弱。并行性成为受欢迎的解决方案——CPU 生产商没有提高时钟速度，而是引入了多核芯片。计算机在同时执行多项任务时变得更加有效！

虽然多核作为提高硬件性能的途径盛行，但要实现软件性能的提升需要付出不小的努力。多核处理器要求开发人员提出不同的算法，这样硬件的改进才会引人注目，但这并不总是那么容易。我们拥有的核心越多，让它们高效地忙碌就越困难。SYCL 是应对这些挑战的编程语言之一，它具有许多有助于利用 CPU（和其他体系结构）上各种形式的并行性的结构。

本章讨论 CPU 架构的一些细节、CPU 硬件通常如何执行 SYCL 应用程序，并提供为 CPU 平台编写 SYCL 代码时的最佳实践。

16.1 性能注意事项

SYCL 为并行化我们的应用程序或从头开始开发并行应用程序铺平了一条可移植的路径。应用程序在 CPU 上运行时的性能很大程度上取决于以下因素：

- Kernel 代码启动和执行的底层性能
- 在并行 Kernel 中运行的程序的百分比及其可扩展性
- CPU 利用率、有效的数据共享、数据局部性和负载平衡
- 工作项之间的同步和通信量
- 创建、恢复、管理、挂起、销毁和同步工作项执行的任何线程所引入的开销，该开销受串行到并行或并行到串行转换数量的影响
- 共享内存引起的内存冲突（包括错误共享内存）
- 共享资源（例如内存、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 可能因供应商而异，甚至因产品一代而异。适用于一种 CPU 的最佳实践可能并非适用于其他 CPU 和配置的最佳实践。

注 68 要在 CPU 上实现最佳性能，请尽可能多地了解 CPU 架构的特征！

16.2 多核 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广泛接受。CPU 在笔记本电脑、台式机和服务器级别提供并行计算平台，使其无处不在，几乎无处不在。CPU 架构最常见的形式是高速缓存一致性非均匀内存访问 (cc-NUMA)，其特点是内存访问时间不完全均匀。许多小型双路通用 CPU 系统都有这种内存系统。由于处理器中的核心数量以及插槽数量不断增加，这种架构已成为主导。

在 cc-NUMA CPU 系统中，每个插槽连接到系统中总内存的一个子集。高速缓存一致性互连将所有套接字粘合在一起，并为程序员提供单一系统内存视图。这样的内存系统是可扩展的，因为总内存带宽随着系统中套接字的数量而扩展。互连的好处是应用程序可以透明地访问系统中的所有内存，无论数据驻留在何处。然而，这是有代价的：从内存访问数据的延迟不再一致（即，我们不再有固定的访问延迟）。相反，延迟取决于数据在系统中的存储位置。在良好的情况下，数据来自直接连接到代码运行的套接字的内存。在糟糕的情况下，数据必须来自连接到系统中较远的套接字的内存，并且由于 cc-NUMA CPU 系统上套接字之间互连的跳数，内存访问的成本可能会增加。

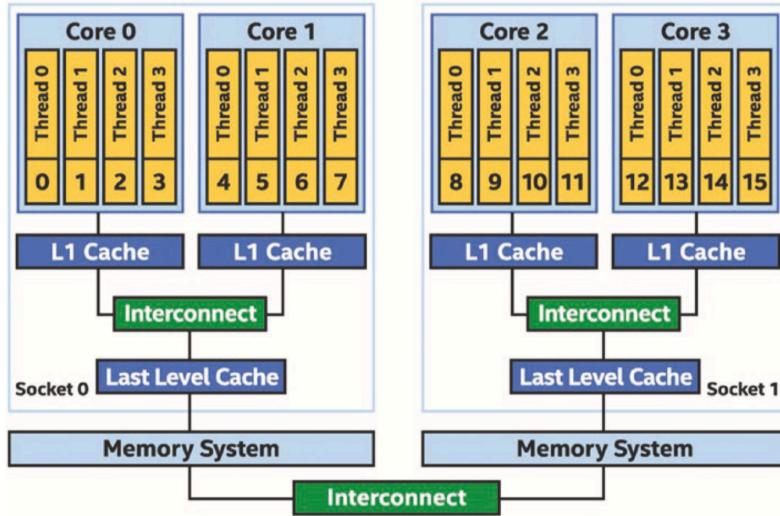


图 16.1: 通用多核 CPU 系统

图 16-1 显示了具有 cc-NUMA 内存的通用 CPU 架构。这是一种简化的系统架构，包含当今通用多插槽系统中的核心和内存组件。在本章的其余部分中，该图将用于说明相应代码示例的映射。

为了实现最佳性能，我们需要确保了解特定系统的 cc-NUMA 配置的特征。例如，英特尔最新的服务器使用了网状互连架构。在此配置中，核心、高速缓存和内存控制器被组织成行和列。在努力实现系统的最佳性能时，了解处理器与内存的连接性至关重要。

图 16-1 中的系统有两个插槽，每个插槽有两个 Kernel，每个 Kernel 有四个硬件线程。每个核心都有自己的 1 级 (L1) 缓存。L1 缓存连接到共享的最后一级缓存，该缓存连接到套接字上的内存系统。套接字内的内存访问延迟是统一的，这意味着它是一致的并且可以准确预测。

这两个套接字通过缓存一致性互连进行连接。内存分布在整个系统中，但所有内存都可以从系统中的任何位置透明地访问。当访问不在运行访问代码的套接字中的内存时，内存读写延迟是不均匀的，这意味着从远程套接字访问数据时，它可能会带来更长且不一致的延迟。然而，互连的一个关键方面是一致性。我们不需要担心整个系统内存中数据的不一致视图，而是可以关注我们访问分布式内存系统的方式对性能的影响。更高级的优化（例如，具有宽松内存顺序的原子操作）可以实现不再需要那么多硬件内存一致

性的操作，但是当我们需要一致性时，硬件会为我们提供一致性。

CPU 中的硬件线程是执行工具。这些是执行指令流的单元。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讨论的符号。除非另有说明，本章中对 CPU 系统的所有引用均指图 16-1 中所示的参考 cc-NUMA 系统。

16.3 SIMD 硬件基础知识

1996 年，广泛部署的 SIMD 指令集是 x86 架构之上的 MMX 扩展。此后，许多 SIMD 指令集扩展在英特尔架构以及整个行业得到广泛应用。CPU Kernel 通过执行指令来执行其工作，Kernel 知道如何执行的具体指令由指令集（例如 x86、x86_64、AltiVec、NEON）和指令集扩展（例如 SSE、AVX、AVX-512）它实现的。指令集扩展添加的许多操作都集中在 SIMD 上。

SIMD 指令允许通过使用大于正在处理的数据的基本单位的寄存器和硬件，在单个 Kernel 上同时执行多个计算。例如，使用 512 位寄存器，我们可以使用一条机器指令执行 8 个 64 位计算。

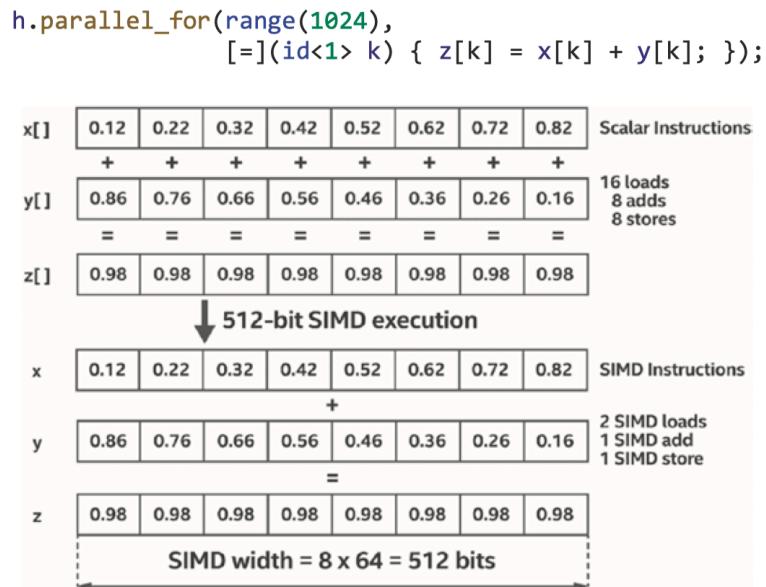


图 16.2: 在 CPU 硬件线程中执行 SIMD

理论上, 图 16-2 中所示的示例可以使我们的速度提高八倍。实际上, 它可能会有所缩减, 因为八倍加速的一部分用于消除一个瓶颈并暴露下一个瓶颈, 例如内存吞吐量。一般来说, 使用 SIMD 的性能优势因具体场景而异, 在少数情况下, 例如广泛的分支发散、非单位步长内存访问的聚集/分散以及 SIMD 加载和存储的缓存行分割, 它可以甚至比更简单的非 SIMD 等效代码执行得更差。也就是说, 当我们知道何时以及如何应用(或让编译器应用) SIMD 时, 当今的处理器可以实现相当大的收益。与所有性能优化一样, 程序员应该在将典型目标机器投入生产之前测量其增益。本章以下各节提供了有关预期性能提升的更多详细信息。

带有 SIMD 单元的 cc-NUMA CPU 架构构成了多核处理器的基础, 它可以以至少五种不同的方式利用从指令级并行开始的广泛并行性, 如图 16-3 所示。

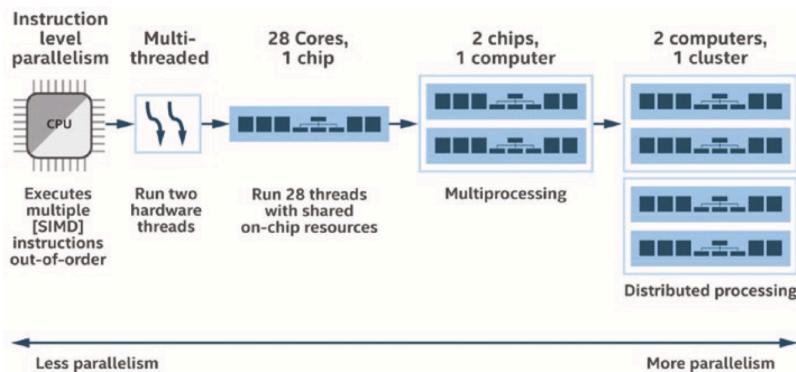


图 16.3: 并行执行指令的五种方式

在图 16-3 中, 指令级并行可以通过单线程内标量指令的乱序执行和 SIMD 并行来实现。线程级并行可以通过在同一核心或不同规模的多个核心上执行多个线程来实现。更具体地说, 线程级并行性可以从以下方面体现出来:

- 现代 CPU 架构允许一个核心同时执行两个或多个线程的指令。
- 每个处理器内包含两个或多个 Kernel 的多核架构。操作系统将其每个执行核心视为具有所有关联执行资源的离散处理器。
- 处理器(芯片)级别的多处理, 可以通过执行单独的代码线程来完成。

因此，处理器可以让一个线程从应用程序运行，另一个线程从操作系统运行，或者可以让并行线程从单个应用程序内运行。

- 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。

随着多处理器计算机和多核技术变得越来越普遍，使用并行处理技术作为标准实践来提高性能非常重要。本章后面的部分将介绍 SYCL 中的编码方法和性能调优技术，使我们能够在多核 CPU 上实现峰值性能。

与其他并行处理硬件（例如 GPU）一样，为 CPU 提供足够大的数据元素集进行处理非常重要。为了演示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。

```
// C++ STREAM Triad workload
// __restrict is used to denote no memory aliasing among
// arguments
template <typename T>
double triad(T* __restrict VA, T* __restrict VB,
             T* __restrict VC, size_t array_size,
             const T scalar) {
    double ts = timer_start();
    for (size_t id = 0; id < array_size; id++) {
        VC[id] = VA[id] + scalar * VB[id];
    }
    double te = timer_end();
    return (te - ts);
}
```

图 16.4: STREAM Triad C++ 循环

注 69 (STREAM TRIAD WORKLOAD 的说明) *Stream triad workload* (www.cs.virginia.edu/stream) 是 CPU 供应商用来演示内存带宽功能的重要且流行的基准测试工作负载。我们使用 *Stream triad Kernel* 来演示并行 *Kernel* 的代码生成，以及通过本章中描述的技术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，但足以以易于理解的方式显示许多优化。布里斯托大学有一个流实现，称为 *Babelstream*，其中包括带有 *sYCL* 版本的 *C++*。

STREAM Triad 循环可以在使用单个 CPU 核心进行串行执行的 CPU 上简单地执行。优秀的 C++ 编译器将执行循环向量化，为具有利用指令级

SIMD 并行性的硬件的 CPU 生成 SIMD 代码。例如，对于支持 AVX-512 的 Intel Xeon 处理器，Intel C++ 编译器会生成如图 16-5 所示的 SIMD 代码。至关重要的是，编译器对代码的转换通过在每次循环迭代中执行更多工作（使用 SIMD 指令和循环展开）减少了循环迭代次数。

```
# %bb.0:
vbroadcastsd    %xmm0, %zmm0      # broadcast "scalar" to SIMD reg zmm0
movq    $32, %rax
.p2align 4, 0x90
.LBB0_1:          # %loop.19
                  # =>This Loop Header: Depth=1
vmovupd 256(%rdx,%rax,8), %zmm1 # load 8 elements from memory to zmm1
vfmadd213pd   256(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
# perform SIMD FMA for 8 data elements
# VC[id:8] = scalar*VB[id:8]+VA[id:8]
vmovupd %zmm1, 256(%rdi,%rax,8) # store 8-element result to mem from zmm1
# This SIMD loop body is unrolled by 4
vmovupd 320(%rdx,%rax,8), %zmm1
vfmadd213pd   320(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
vmovupd %zmm1, 320(%rdi,%rax,8)

vmovupd 384(%rdx,%rax,8), %zmm1
vfmadd213pd   384(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
vmovupd %zmm1, 384(%rdi,%rax,8)

vmovupd 448(%rdx,%rax,8), %zmm1
vfmadd213pd   448(%rsi,%rax,8), %zmm0, %zmm1 # zmm1=(zmm0*zmm1)+mem
vmovupd %zmm1, 448(%rdi,%rax,8)
addq    $32, %rax
cmpq    $134217696, %rax      # imm = 0x7FFFFFFE0
.jb     .LBB0_1
```

图 16.5: AVX-512

如果我们尝试在 CPU 上执行此函数，它可能会在较小的数组大小下运

行良好，但效果并不好，因为它不利用 CPU 的任何多核或线程功能。然而，如果我们尝试在 CPU 上使用较大的数组大小执行此函数，它的性能可能会非常差，因为单个线程仅利用单个 CPU 核心，并且当该核心的内存带宽饱和时，就会出现瓶颈。

16.4 利用线程级并行性

为了提高 STREAM Triad Kernel 的性能，我们可以通过将循环转换为 parallel_for Kernel 来计算一系列可以并行处理的数据元素。

```

constexpr int num_runs = 10;
constexpr size_t scalar = 3;

double triad(const std::vector<float>& vecA,
             const std::vector<float>& vecB,
             std::vector<float>& vecC) {
    assert(vecA.size() == vecB.size() &&
           vecB.size() == vecC.size());
    const size_t array_size = vecA.size();
    double min_time_ns = std::numeric_limits<double>::max();

    queue q{property::queue::enable_profiling{}};
    std::cout << "Running on device: "
    << q.get_device().get_info<info::device::name>()
    << "\n";

    buffer<float> bufA(vecA);
    buffer<float> bufB(vecB);
    buffer<float> bufC(vecC);

    for (int i = 0; i < num_runs; i++) {
        auto Q_event = q.submit([&](handler& h) {
            accessor A{bufA, h};
            accessor B{bufB, h};
            accessor C{bufC, h};

            h.parallel_for(array_size, [=](id<1> idx) {
                C[idx] = A[idx] + B[idx] * scalar;
            });
        });

        double exec_time_ns =
            Q_event.get_profiling_info<
                info::event_profiling::command_end>() -
            Q_event.get_profiling_info<
                info::event_profiling::command_start>();

        std::cout << "Execution time (iteration " << i
        << ") [sec]: "
        << (double)exec_time_ns * 1.0E-9 << "\n";
        min_time_ns = std::min(min_time_ns, exec_time_ns);
    }

    return min_time_ns;
}

```

图 16.6: SYCL STREAM Triad parallel_for Kernel 代码

该 STREAM Triad SYCL 并行 Kernel 的主体与在 CPU 上以串行 C++ 执行的 STREAM Triad 循环的主体完全相同，如图 16-6 所示。

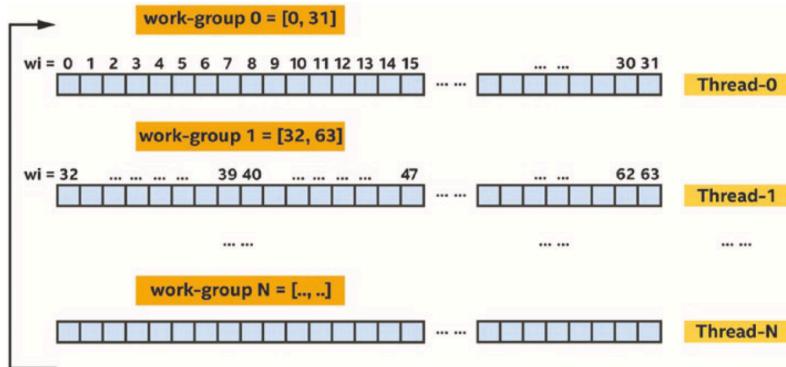


图 16.7: STREAM Triad 并行 Kernel 的映射

尽管并行 Kernel 与使用循环的串行 C++ 编写的 STREAM Triad 函数非常相似，但它的运行速度要快得多，因为 `parallel_for` 允许在多个 Kernel 上并行处理数组的不同元素。图 16-7 显示了如何将该 Kernel 映射到 CPU。假设我们的系统有一个套接字、四个核心，每个核心有两个硬件线程（总共八个线程），并且该实现在每个包含 32 个工作项的工作组中处理数据。如果我们有 1024 个双精度数据元素需要处理，我们将有 32 个工作组。工作组调度可以按循环顺序完成，即 $\text{thread-id} = \text{work-group-id} \bmod 8$ 。本质上，每个线程将执行四个工作组。每轮可以并行执行八个工作组。请注意，在这种情况下，工作组是由 SYCL 编译器和运行时隐式形成的一组工作项。

请注意，在 SYCL 程序中，未指定数据元素被分区并分配给不同处理器核心（或线程）的确切方式。这为 SYCL 实现提供了灵活性，可以选择如何最好地在特定 CPU 上执行并行 Kernel。话虽如此，实现可以为程序员提供某种程度的控制，以实现性能调整（例如，通过编译器选项或环境变量）。

虽然 CPU 可能会施加相对昂贵的线程上下文切换和同步开销，但在处理器核心上驻留更多软件线程可能是有益的，因为它为每个处理器核心提供了要执行的工作的选择。如果一个软件线程正在等待另一线程产生数据，则处理器核心可以切换到准备运行的不同软件线程，而不会使处理器核心空闲。

注 70 (选择如何绑定和调度线程) 选择一种有效的方案来在线程之间分区和调度工作对于在 CPU 和其他设备类型上调整应用程序非常重要。后续部分将介绍一些技术。

16.4.1 线程亲和力洞察

线程亲和性指定执行特定线程的 CPU 核心。如果线程在核心之间移动，性能可能会受到影响，例如，如果线程不在同一核心上执行，并且数据在不同核心之间来回移动，则缓存局部性可能会变得效率低下。

DPC++ 编译器的运行时库支持多种通过环境变量 DPCPP_CPU CU_AFFINITY、DPCPP_CPU_PLACES、DPCPP_CPU_NUM_CUS 和 DPCPP_CPU_SCHEDULE 将线程绑定到 Kernel 的方案，这些方案不是由 SYCL 定义的。其他实现可能会公开类似的环境变量。

第一个是环境变量 DPCPP_CPU CU_AFFINITY。使用这些环境变量控件进行调整既简单又成本低，但会对许多应用程序产生巨大影响。该环境变量的说明如图 16-8 所示。

DPCPP_CPU CU_AFFINITY	Description
spread	Bind successive threads to distinct sockets starting with socket 0 in a round-robin order
close	Bind successive threads to distinct hardware threads starting with thread 0 in a round-robin order

图 16.8: *DPCPP_CPU CU_AFFINITY* 环境变量

当指定环境变量 DPCPP_CPU CU_AFFINITY 时，软件线程通过以下公式绑定到硬件线程：

spread: $\text{boundHT} = (\text{tid} \bmod \text{numHT}) + (\text{tid} \bmod \text{numSocket}) \times \text{numHT}$
 close: $\text{boundHT} = \text{tid} \bmod (\text{numSocket} \times \text{numHT})$

其中

- tid 表示软件线程标识符
- boundHT 表示线程 tid 绑定到的硬件线程（逻辑核心）
- numHT 表示每个套接字的硬件线程数
- numSocket 表示系统中的套接字数量

假设我们在双核双插槽系统上运行一个具有八个线程的程序，换句话说，我们有四个核心，总共有八个线程要编程。图 16-9 显示了线程如何映射到不同 DPCPP_CPU CU_AFFINITY 设置的硬件线程和 Kernel 的示例。

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>

图 16.9: 使用硬件线程将线程映射到 *Kernel*

与环境变量 DPCPP_CPU CU_AFFINITY 结合使用，还有其他支持 CPU 性能调优的环境变量：

- DPCPP_CPU_NUM_CUS = [n]，设置用于 Kernel 执行的线程数。它的默认值是系统中硬件线程的数量。
- DPCPP_CPU_PLACES = [sockets| numa_domains | numa_domains | cores| threads]，它指定将设置关联性的位置，类似于 OpenMP 5.1 中的 OMP_PLACES。默认设置是核心。
- DPCPP_CPU_SCHEDULE = [dynamic| affinity| static]，指定调度工作组的算法。它的默认设置是动态的。
 - dynamic：启用 auto_partitioner，它通常会执行足够的分割以平衡工作线程之间的负载。
 - affinity：启用 affinity_partitioner，它可以提高缓存亲和力，并在将子范围映射到工作线程时使用比例分割。
 - static：启用 static_partitioner，它尽可能均匀地在工作线程之间分配迭代。

当使用 Intel 的 OpenCL CPU 运行时在 CPU 上运行时，工作组调度由线程构建块 (TBB) 库处理。使用 DPCPP_CPU_SCHEDULE 确定使用哪个 TBB 分区程序。请注意，TBB 分区程序还使用粒度大小来控制工作拆分，默认粒度大小为 1，这表示所有工作组都可以独立执行。更多信息请访问 tinyurl.com/oneTBBpart。

缺乏线程亲和性调整并不一定意味着性能较低。性能通常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。使用基准测试应用程序是确定线程关联是否对性能产生影响的一种方法。STREAM

Triad 代码如图 16-1 所示，在没有线程关联设置的情况下，一开始性能较低。通过控制亲和力设置并通过环境变量使用软件线程的静态调度（Linux 的导出如下所示），性能得到了提高：

```
export DPCPP_CPU_PLACES=numa_domains  
export DPCPP_CPU CU_AFFINITY=close
```

通过使用 numa_domains 作为亲和性的场所设置，TBB 任务区域绑定到 NUMA 节点或套接字，并且工作均匀分布在任务区域之间。一般来说，环境变量 DPCPP_CPU_PLACES 建议与 DPCPP_CPU CU_AFFINITY 一起使用。这些环境变量设置帮助我们在具有 2 个插槽、每个插槽 28 个 Kernel、每个 Kernel 2 个硬件线程、运行频率为 2.5 GHz 的 Intel Xeon 服务器系统上实现约 30% 的性能增益。不过，我们仍然可以做得更好，进一步提高这款 CPU 的性能。

16.4.2 注意第一次接触内存

内存存储在第一次接触（使用）的地方。由于我们示例中的初始化循环是由主机线程串行执行的，因此所有内存都与主机线程正在其上运行的套接字相关联。其他套接字的后续访问将从附加到初始套接字（用于初始化）的内存中访问数据，这对于性能来说显然是不受欢迎的。我们可以通过并行化初始化循环来控制跨套接字的首次触摸效果，从而在 STREAM Triad Kernel 上实现更高的性能，如图 16-10 所示。

```

template <typename T>
void init(queue& deviceQueue, T* VA, T* VB, T* VC,
          size_t array_size) {
    range<1> numOfItems{array_size};

    buffer<T, 1> bufferA(VA, numOfItems);
    buffer<T, 1> bufferB(VB, numOfItems);
    buffer<T, 1> bufferC(VC, numOfItems);

    auto queue_event = deviceQueue.submit([&](handler& cgh) {
        auto aA = bufA.template get_access<sycl_write>(cgh);
        auto aB = bufB.template get_access<sycl_write>(cgh);
        auto aC = bufC.template get_access<sycl_write>(cgh);

        cgh.parallel_for<class Init<T>>(numOfItems, [=](id<1> wi) {
            aA[wi] = 2.0;
            aB[wi] = 1.0;
            aC[wi] = 0.0;
        });
    });

    queue_event.wait();
}

```

图 16.10: STREAM Triad 并行初始化 Kernel, 用于控制首次触摸效果

在初始化代码中利用并行性可以提高 Kernel 在 CPU 上运行时的性能。在本例中，我们在 Intel Xeon 处理器系统上实现了约 2 倍的性能提升。

本章最近的部分表明，通过利用线程级并行性，我们可以有效地利用 CPU Kernel 和线程。然而，我们还需要利用 CPU 核心硬件中的 SIMD 矢量级并行性来实现峰值性能。

注 71 SYCL 并行 Kernel 受益于跨 Kernel 和硬件线程的线程级并行性！

16.5 CPU 上的 SIMD 矢量化

虽然编写良好且没有跨工作项依赖性的 SYCL Kernel 可以在 CPU 上有效地并行运行，但实现也可以将矢量化应用于 SYCL Kernel，以利用类似于第 15 章中描述的 GPU 支持的 SIMD 硬件。本质上，CPU 处理器可以利用大多数数据元素通常位于连续内存中并通过数据并行 Kernel 采用相同控制流路径的事实，使用 SIMD 指令优化内存加载、存储和操作。例如，在具有语句 $a[i] = a[i] + b[i]$ 的 Kernel 中，每个数据元素通过在多个数据元素之间共享硬件逻辑来执行相同的指令流加载、加载、添加和存储，并且

将它们作为一个组执行，这可以自然地映射到硬件的 SIMD 指令集上。具体地，可以通过单个指令同时处理多个数据元素。

<i>Serial execution</i>				<i>SIMD execution</i>
<i>work-0</i>	<i>work-1</i>	<i>work-2</i>	<i>work 3</i>	<i>vector sub-group</i>
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	simdadd vr0, vr1
store a[0], r0	store a[1], r0	store a[2], r0	store a[3], r0	simdstore a[0..3], vr0

图 16.11: SIMD 执行指令流

单个指令同时处理的数据元素的数量有时称为指令或执行该指令的处理器的向量长度（或 SIMD 宽度）。在图 16-11 中，我们的指令流以四路 SIMD 执行方式运行。

CPU 处理器并不是唯一实现 SIMD 指令集的处理器。GPU 等其他处理器实现 SIMD 指令以提高处理大量数据时的效率。与其他处理器类型相比，Intel Xeon CPU 处理器的一个关键区别在于具有三个固定大小的 SIMD 寄存器宽度（128 位 XMM、256 位 YMM 和 512 位 ZMM），而不是可变长度的 SIMD 宽度。当我们使用 Sub-Groups 或向量类型编写具有 SIMD 并行性的 SYCL 代码时（请参阅第 11 章），我们需要注意硬件中 SIMD 宽度和 SIMD 向量寄存器的数量。

16.5.1 确保 SIMD 执行合法性

从语义上讲，SYCL 执行模型确保 SIMD 执行可以应用于任何 Kernel，并且每个工作组（即 Sub-Groups）中的一组工作项可以使用 SIMD 指令同时执行。某些实现可能会选择使用 SIMD 指令在 Kernel 中执行循环，但当且仅当保留所有原始数据依赖性，或者编译器基于私有化和归约语义解决数据依赖性时，这才是可能的。这种实现可能会报告 Sub-Groups 大小为 1。

单个 SYCL Kernel 执行可以从处理单个工作项转换为使用工作组内的 SIMD 指令处理一组工作项。在 ND 范围模型下，编译器矢量化器会选择增长最快（单位步长）的维度来生成 SIMD 代码。本质上，要在给定 ND 范围的情况下启用矢量化，同一 Sub-Groups 中的任何两个工作项之间不应存在跨工作项依赖关系，或者编译器需要在同一 Sub-Groups 中保留跨工作项前向依赖关系。Sub-Groups。

```

const int n = 16, w = 16;

queue q;
range<2> G = {n, w};
range<2> L = {1, w};

int *a = malloc_shared<int>(n * (n + 1), q);

for (int i = 0; i < n; i++)
    for (int j = 0; j < n + 1; j++) a[i * n + j] = i + j;

q.parallel_for(
    nd_range<2>{G, L},
    [=](nd_item<2> it) [[sycl::reqd_sub_group_size(w)]] {
        // distribute uniform "i" over the sub-group with
        // 16-way redundant computation
        const int i = it.get_global_id(0);
        sub_group sg = it.get_sub_group();

        for (int j = sg.get_local_id()[0]; j < n; j += w) {
            // load a[i*n+j+1:16] before updating a[i*n+j:16]
            // to preserve loop-carried forward dependency
            auto va = a[i * n + j + 1];
            group_barrier(sg);
            a[i * n + j] = va + i + 2;
        }
        group_barrier(sg);
    })
.wait();

```

图 16.12: 使用 Sub-Groups 对具有前向依赖关系的循环进行矢量化

当工作项的 Kernel 执行映射到 CPU 上的线程时，细粒度同步的成本很高，而且线程上下文切换开销也很高。因此，在为 CPU 编写 SYCLKernel 时，消除工作组内工作项之间的依赖性是一项重要的性能优化。另一种有效的方法是将这种依赖性限制在 Sub-Groups 内的工作项，如图 16-12 中的先读后写依赖性所示。如果 Sub-Groups 在 SIMD 执行模型下执行，则 Kernel 中的 Sub-Groups 屏障可以被编译器视为 noop，并且在运行时不会产生真正的同步成本。

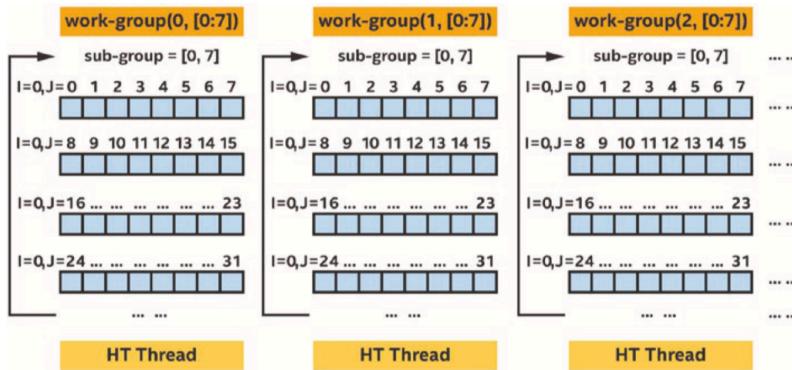


图 16.13: 具有前向依赖关系的循环的 SIMD 矢量化

Kernel 被向量化(以向量长度为 8 为例),其 SIMD 执行如图 16-13 所示。工作组的组大小为 (1, 8), Kernel 内部的循环迭代分布在这些 Sub-Groups 工作项上,并以八路 SIMD 并行方式执行。

在此示例中,如果 Kernel 中的循环主导性能,则允许跨 Sub-Groups 进行 SIMD 矢量化将带来显着的性能改进。

使用并行处理数据元素的 SIMD 指令是让 Kernel 性能超越 CPU 核心和硬件线程数量的一种方法。

16.5.2 SIMD 掩蔽和成本

在实际应用中,我们可以期待条件语句,例如 if 语句,条件表达式,例如 $a = b > a? a: b$ 、具有可变迭代次数的循环、switch 语句等。任何有条件的事情都可能导致标量控制流不执行相同的代码路径,就像在 GPU 上一样(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 Kernel 中的条件语句生成。考虑一个示例,其中 $A=1, 2, 3, 4, B=3, 7, 8, 1$ 以及比较表达式 $a < b$ 。比较返回一个具有四个值 1, 1, 1, 0 的掩码,这些值可以存储在硬件掩码寄存器中,以指示后续 SIMD 指令的哪些通道应执行由比较保护(启用)的代码。

如果 Kernel 包含条件代码,则会使用基于与每个数据元素(SIMD 指令中的通道)关联的掩码位执行的掩码指令对其进行矢量化。每个数据元素的掩码位是掩码寄存器中的相应位。

使用屏蔽可能会导致性能低于相应的非屏蔽代码。这可能是由于

- 每个负载上的附加蒙版混合操作
- 对目的的依赖

屏蔽是有成本的，因此仅在必要时使用。当 Kernel 是 ND 范围 Kernel 且在执行范围内具有显式工作项分组时，在选择 ND 范围工作组大小时应小心，以通过最小化屏蔽成本来最大化 SIMD 效率。当工作组大小不能被处理器的 SIMD 宽度整除时，部分工作组可能会在 Kernel 屏蔽的情况下执行。

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: 用于 *Kernel* 中屏蔽的三个屏蔽代码生成

图 16-14 显示了使用合并屏蔽如何创建对目标寄存器的依赖：

- 在没有屏蔽的情况下，处理器每个周期执行两次乘法 (vmulps)。
- 通过合并掩码，处理器每四个周期执行两次乘法，因为乘法指令 (vmulps) 将结果保存在目标寄存器中，如图 16-17 所示。
- 零掩码不依赖于目标寄存器，因此每个周期可以执行两次乘法 (vmulps)。

访问缓存对齐的数据比访问未对齐的数据具有更好的性能。在许多情况下，地址在编译时未知，或者已知但未对齐。当使用循环时，可以实现存储器访问的剥离，以使用掩码访问处理前几个元素，直到第一个对齐地址，然后通过多版本技术处理未掩码访问，然后处理掩码剩余部分。此方法增加了代码大小，但总体上改进了数据处理。当使用并行 Kernel 时，我们作为程序员可以通过手动采用类似的技术或通过确保分配适当对齐来提高性能。

16.5.3 避免结构数组以提高 SIMD 效率

AOS (结构数组) 结构会导致聚集和分散，这既会影响 SIMD 效率，也会为内存访问带来额外的带宽和延迟。硬件收集-分散机制的存在并不能消除这种转换的需要——收集-分散访问通常需要比连续负载高得多的带宽和

延迟。给定 struct {float x; float y; float z; float w;} a[4] 考虑一个对其进行操作的 Kernel，如图 16-15 所示。

```
cgh.parallel_for<class aos<T>>(numOfItems,[=](id<1> wi) {
    x[wi] = a[wi].x; // lead to gather x0, x1, x2, x3
    y[wi] = a[wi].y; // lead to gather y0, y1, y2, y3
    z[wi] = a[wi].z; // lead to gather z0, z1, z2, z3
    w[wi] = a[wi].w; // lead to gather w0, w1, w2, w3
});
```

图 16.15: Kernel 中的 SIMD gather

当编译器沿着一组工作项对 Kernel 进行矢量化时，由于需要非单位跨度内存访问，它会导致 SIMD 收集指令生成。例如， $a[0].x$ 、 $a[1].x$ 、 $a[2].x$ 和 $a[3].x$ 的步长是 4，而不是更有效的单位步长 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
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

在 Kernel 中，我们通常可以通过消除内存聚集-分散操作来实现更高的 SIMD 效率。某些代码受益于数据布局更改，该更改将以结构数组 (AOS) 表示形式编写的数据结构转换为数组结构 (SOA) 表示形式，即为每个结构字段使用单独的数组以保留内存访问执行 SIMD 矢量化时是连续的。例如，考虑 struct {float x[4]; float y[4]; float z[4]; float w[4];} a; 的 SOA 数据布局，如图所示：

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
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

Kernel 可以使用单位步长（连续）向量加载和存储来操作数据，如图 16-16 所示，即使是向量化时也是如此！

```
cgh.parallel_for<class aos<T>>(numOfItems,[=](id<1> wi) {
    x[wi] = a.x[wi]; // lead to unit-stride vector load x[0:4]
    y[wi] = a.y[wi]; // lead to unit-stride vector load y[0:4]
    z[wi] = a.z[wi]; // lead to unit-stride vector load z[0:4]
    w[wi] = a.w[wi]; // lead to unit-stride vector load w[0:4]
});
```

图 16.16: Kernel 中的 SIMD 单位步幅向量负载

SOA 数据布局有助于防止跨数组元素访问结构的一个字段时发生聚集，并帮助编译器在与工作项关联的连续数组元素上对 Kernel 进行矢量化。请注意，考虑到使用这些数据结构的所有位置，此类 AOS 到 SOA 或 AOSOA 数据布局转换预计将在程序级别（由我们）完成。仅在循环级别执行此操作将涉及循环之前和之后格式之间的昂贵转换。然而，我们也可能依赖编译器对 AOS 数据布局执行向量加载和洗牌优化，但需要付出一些代价。当 SOA（或 AOS）数据布局的成员具有向量类型时，编译器向量化可以根据底层硬件进行水平扩展或垂直扩展以生成最优代码。

16.5.4 数据类型对 SIMD 效率的影响

当 C++ 程序员知道数据适合 32 位有符号类型时，他们通常会使用整数数据类型，这通常会导致如下代码

```
int id = get_global_id(0); a[id] = b[id] + c[id];
```

但是，鉴于 `get_global_id(0)` 的返回类型是 `size_t`（无符号整数，通常是 64 位），转换可能会减少编译器可以合法执行的优化。当编译器对 Kernel 中的代码进行矢量化时，这可能会导致 SIMD 收集/分散指令，例如：

- 读取 `[get_global_id(0)]` 可能会导致 SIMD 单位步长向量加载。
- 读取 `[(int)get_global_id(0)]` 可能会导致非单位步长收集指令。

这种微妙的情况是由从 `size_t` 到 `int`（或 `uint`）的数据类型转换的环绕行为（C++ 标准中未指定的行为和/或明确定义的环绕行为）引入的，这主要是基于 C 语言的演变的历史产物。具体来说，某些转换的溢出是未定义的行为，这允许编译器假设此类情况永远不会发生并更积极地进行优化。图 16-17 为那些想要了解细节的人展示了一些示例。

get_global_id(0)	a[(int) get_global_id(0)]	get_global_id(0)	a[(uint) get_global_id(0)]
0x7FFFFFFE	a[MAX_INT-1]	0xFFFFFFFFFE	a[MAX_UINT-1]
0x7FFFFFFF	a[MAX_INT (big positive)]	0xFFFFFFFFFF	a[MAX_UINT]
0x80000000	a[MIN_INT (big negative)]	0x100000000	a[0]
0x80000001	a[MIN_INT+1]	0x1000000001	a[1]

图 16.17: 整数类型的 *value wrapper around*

SIMD 聚集/分散指令比 SIMD 单位步长向量加载/存储操作慢。为了实现最佳的 SIMD 效率，无论使用哪种编程语言，避免聚集/分散对于应用程序都是至关重要的。

大多数 SYCL `get_*_id()` 系列函数具有相同的细节，尽管许多情况适合 `MAX_INT`，因为可能的返回值是有限的（例如，工作组内的最大 id）。因此，只要合法，SYCL 编译器就可以假定跨相邻工作项块的单位步长内存地址，以避免聚集/分散。如果由于全局 ID 的值和/或全局 ID 的导数值可能溢出而导致编译器无法安全地生成线性单位步长向量内存加载/存储，则编译器将生成聚集/分散。

在为用户提供最佳性能的理念下，DPC++ 编译器假设没有溢出，并且在实践中几乎所有时间都捕捉到了现实，因此编译器可以生成最佳的 SIMD 代码以实现良好的性能。然而，DPC++ 编译器为我们提供了一个编译器选项 `-fno-sycl-id-queries-fit-in-int`，告诉编译器将会出现溢出，并且从 id 查询派生的向量化访问可能不安全。这可能会对性能产生很大的影响，并且应该在不安全的情况下使用，以假设没有溢出。关键要点是程序员应确保全局 ID 的值适合 32 位 int。否则，应使用编译器选项 `-fno-sycl-idqueries-fit-in-int` 来保证程序的正确性，这可能会导致性能降低。

16.5.5 使用 `single_task` 执行 SIMD

在单任务执行模型下，没有要矢量化的工作项。与向量类型和函数相关的优化是可能的，但这取决于编译器。编译器和运行时可以自由地启用显式 SIMD 执行或在 `single_task` Kernel 中选择标量执行，结果将取决于编译器的实现。

当编译到 CPU 时，C++ 编译器可能会将 `single_task` 内部出现的向量类型映射到 SIMD 指令。`vec load`、`store` 和 `swizzle` 函数直接对向量变量执

行操作，通知编译器数据元素正在从内存中的同一（统一）位置开始访问连续数据，并使我们能够请求连续数据的优化加载/存储。正如第 11 章中所讨论的，这种对 `vec` 的解释是有效的，但是，我们应该预期该功能最终会被弃用，而支持更明确的向量类型（例如，`std::simd`）。

```
queue q;

bool *resArray = malloc_shared<bool>(1, q);
resArray[0] = true;

q.single_task([=]() {
    sycl::vec<int, 4> old_v =
        sycl::vec<int, 4>(0, 100, 200, 300);
    sycl::vec<int, 4> new_v = sycl::vec<int, 4>();

    new_v.rgb() = old_v.abgr();
    int vals[] = {300, 200, 100, 0};

    if (new_v.r() != vals[0] || new_v.g() != vals[1] ||
        new_v.b() != vals[2] || new_v.a() != vals[3]) {
        resArray[0] = false;
    }
}).wait();
```

图 16.18: 在 `single_taskKernel` 中使用向量类型和 `swizzle` 操作

在图 16-18 所示的示例中，在单任务执行下，声明了一个具有三个数据元素的向量。使用 `old_v.abgr()` 执行 `swizzle` 操作。如果 CPU 为某些 `swizzle` 操作提供 SIMD 硬件指令，我们可以通过在应用程序中使用 `swizzle` 操作来实现一些性能优势。

注 72 (SIMD 矢量化指南) *CPU* 处理器实现具有不同 SIMD 宽度的 SIMD 指令集。在许多情况下，这是一个实现细节，对于在 *CPU* 上执行 *Kernel* 的应用程序是透明的，因为编译器可以确定一组有效的数据元素，以特定的 SIMD 大小进行处理，而不是要求我们显式使用 SIMD 指令。*Sub-Groups* 可用于更直接地表示数据元素分组应在 *Kernel* 中执行 SIMD 的情况。

考虑到计算的复杂性，选择最适合矢量化的代码和数据布局最终可能会带来更高的性能。在选择数据结构时，请尝试选择数据布局、对齐方式和数据宽度，以便最常执行的计算能够以具有最大并行度的 SIMD 友好方式

访问内存，如本章所述。

16.6 总结

为了充分利用 CPU 上的线程级并行性和 SIMD 矢量级并行性，我们需要牢记以下目标：

- 熟悉所有类型的 SYCL 并行性以及我们希望针对的底层 CPU 架构。
- 在最匹配硬件资源的线程级别利用适量的并行性（不多也不少）。使用分析器和分析器等供应商工具来帮助指导我们的调优工作以实现这一目标。
- 注意线程亲和性和内存首次接触对程序性能的影响。
- 通过数据布局、对齐方式和数据宽度来设计数据结构，以便最常执行的计算能够以 SIMD 友好的方式访问内存，并具有最大的 SIMD 并行性。
- 注意平衡屏蔽与代码分支的成本。
- 使用清晰的编程风格，最大限度地减少潜在的内存别名和副作用。
- 请注意使用向量类型和接口的可扩展性限制。如果编译器实现将它们映射到硬件 SIMD 指令，则固定向量大小可能无法在多代 CPU 和来自不同供应商的 CPU 中很好地匹配 SIMD 寄存器的 SIMD 宽度。

17 FPGA 编程

基于 Kernel 的编程最初作为一种访问 GPU 的方式而流行。由于它现已推广到多种类型的加速器，因此了解我们的编程风格如何影响代码到 FPGA 的映射也很重要。

大多数软件开发人员都不熟悉现场可编程门阵列 (FPGA)，部分原因是大多数台式计算机除了典型的 CPU 和 GPU 之外不包含 FPGA。但 FPGA 值得了解，因为它们在许多应用中具有优势。我们需要问与其他加速器相同的问题，例如“我什么时候应该使用 FPGA?”、“我的应用程序的哪些部分应该卸载到 FPGA?”以及“如何编写性能良好的代码”在 FPGA 上?”

本章为我们提供了开始回答这些问题的知识，至少让我们能够确定 FPGA 是否适合我们的应用，并了解通常使用哪些结构来实现性能。本章是我们可以阅读供应商文档以填写特定产品和工具链的详细信息的起点。我们首先概述程序如何映射到 FPGA 等空间架构，然后讨论使 FPGA 成为加速器的良好选择的一些属性，最后介绍用于实现性能的编程结构。

本章中的“如何思考 FPGA”部分适用于思考任何 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。本章中描述的 FPGA 特定供应商支持目前是 DPC++ 独有的，即 FPGA 选择器和管道。FPGA 选择器和管道是本章中使用的唯一 DPC++ 扩展。希望供应商能够采用类似或兼容的方式来支持 FPGA，DPC++ 作为开源项目也鼓励这样做。

17.1 性能注意事项

与任何处理器或加速器一样，FPGA 器件因供应商不同、甚至不同代产品也不同；因此，一种设备的最佳实践可能并不适用于其他设备的最佳实践。本章中的建议可能会使许多 FPGA 设备受益，无论是现在还是将来，但是……

17.2 如何看待 FPGA

17.2.1 管道并行性

17.2.2 Kernel 消耗芯片“区域”

17.3 何时使用 FPGA

17.3.1 很多很多的工作

17.3.2 自定义操作或操作宽度

17.3.3 标量数据流

17.3.4 低延迟和丰富的连接性

17.3.5 定制内存系统

17.4 在 FPGA 上运行

17.4.1 编译时间

17.4.2 FPGA 仿真器

17.4.3 FPGA 硬件编译“提前”进行

17.5 为 FPGA 编写 Kernel

17.5.1 暴露并行性

17.5.2 使用 ND 范围保持管道繁忙

17.5.3 管道不介意数据依赖性！

17.5.4 循环的空间管道实现

17.5.5 循环启动间隔

17.5.6 管道

17.5.7 定制内存系统

17.6 一些结束语

17.6.1 FPGA 构建模块

17.6.2 时钟频率

17.7 概括

在本章中，我们介绍了编译器如何将算法映射到 FPGA 的空间架构。我们还介绍了一些概念，这些概念可以帮助我们确定 FPGA 是否对我们的

应用有用，并且可以帮助我们更快地启动和运行代码开发。从这个起点开始，我们应该能够很好地浏览供应商编程和优化手册并开始编写 FPGA 代码！FPGA 提供的性能并支持无法很好地映射到其他加速器的应用程序，因此我们应该将它们放在我们心理工具箱的前面！

18 标准库

我们用整本书来宣传编写我们自己的代码的艺术。现在我们终于承认一些伟大的程序员已经编写了我们可以使用的代码。库是完成我们工作的最佳方式。这并不是因为懒惰，而是因为有更好的事情要做，而不是重新发明别人的工作。

本章涵盖三组不同的库函数：

1. SYCL 规范定义的内置函数
2. C++ 标准库
3. C++17 并行算法，由 oneAPI DPC++ 库 (oneDPL) 支持

SYCL 定义了一组丰富的内置函数，提供主机和设备代码共享的通用函数。所有 SYCL 实现都支持这些函数，因此我们可以依赖所有 SYCL 设备上可用的关键数学库。

不保证所有 SYCL 实现在设备代码中都支持 C++ 标准库。然而，DPC++ 编译器（和其他编译器）支持将此作为 SYCL 的扩展，因此我们在这里简要讨论该扩展的局限性。

最后，oneAPI DPC++ 库 (oneDPL) 提供了一组基于 C++17 算法的算法，并在 SYCL 中实现，为 SYCL 程序员提供高生产力的解决方案。这可以最大限度地减少跨 CPU、GPU 和 FPGA 的编程工作。尽管 oneDPL 不是 SYCL 2020 的一部分，但由于它是在 SYCL 之上实现的，因此它应该与任何 SYCL 2020 编译器兼容。

18.1 内置函数

SYCL 提供了一组丰富的内置函数，支持各种数据类型。这些内置函数在主机和设备上的 `sycl` 命名空间中可用，可分为以下几类：

- 浮点数学函数：`asin`、`acos`、`log`、`sqrt`、`floor` 等。
- 整数函数：`abs`、`max`、`min` 等。
- 常用函数：`clamp`、`smoothstep` 等。
- 几何函数：`cross`、`dot`、`distance` 等。

- 关系函数：`isequal`、`isless`、`isfinite` 等。

有关此广泛函数集合的文档可以在 SYCL 2020 规范中找到，在线文档位于 registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html 的第 4.17.5 至 4.17.9 节中。

一些编译器可能提供选项来控制这些函数的精度。例如，DPC++ 编译器提供了多个此类选项，包括 `-mfma`、`-ffast-math` 和 `-ffp-contract=fast`。检查 SYCL 实现的文档以了解类似选项（及其默认值）的可用性非常重要。

一些 SYCL 内置函数在 C++ 标准库中具有等效函数（例如 `sycl::log` 和 `std::log`）。SYCL 实现不需要支持在设备代码中调用 C++ 标准库函数，但某些实现（例如 DPC++）可以。

```
constexpr int size = 9;
std::array<float, size> a;
std::array<float, size> b;

bool pass = true;

for (int i = 0; i < size; ++i) {
    a[i] = i;
    b[i] = i;
}

queue q;

range sz{size};

buffer<float> bufA(a);
buffer<float> bufB(b);
buffer<bool> bufP(&pass, 1);

q.submit([&](handler &h) {
    accessor accA{bufA, h};
    accessor accB{bufB, h};
    accessor accP{bufP, h};

    h.parallel_for(size, [=](id<1> idx) {
        accA[idx] = std::log(accA[idx]);
        accB[idx] = sycl::log(accB[idx]);
        if (!sycl::isequal(accA[idx], accB[idx])) {
            accP[0] = false;
        }
    });
});
});
```

图 18.1: 使用 `std::log` 和 `sycl::log`

图 18-1 演示了 C++ `std::log` 函数和 SYCL 内置 `sycl::log` 函数在设备代码中的用法。使用 DPC++ 编译器实现，两个函数产生相同的数值结果。在示例中，内置关系函数 `sycl::isequal` 用于比较 `std::log` 和 `sycl::log` 的结果。

请注意，SYCL 2020 规范并不要求 SYCL 数学函数实现必须针对给定硬件目标生成与其对应的 C 和 C++ 标准数学函数完全相同的数值结果。

该规范允许在实现中进行某些变化，以考虑不同硬件平台的特性和限制。因此，SYCL 实现在实践中可能会产生匹配结果，如图 18-1 中的代码示例所示。

18.1.1 使用带有内置函数的 `sycl::` 前缀

我们强烈建议调用 SYCL 内置函数，并在名称前添加显式 `sycl::`。仅调用 `sqrt()` 并不能保证调用所有实现上内置的 SYCL，即使“`using namespace sycl;`”已经用过。

注 73 SYCL 内置函数应始终在内置名称前面使用显式 `sycl::` 进行调用。不遵循此建议可能会导致奇怪且不可移植的结果。

在编写可移植代码时，我们建议避免使用命名空间 `sycl`；完全赞成显式使用 `std::` 和 `sycl::` 命名空间。通过明确，我们消除了在某些 SYCL 实现中遇到无法解决的冲突的可能性。这也可能使代码将来更容易调试（例如，如果实现为 `std::` 和 `sycl::` 命名空间中的数学函数提供不同的精度保证）。

18.2 C++ 标准库

如前所述，SYCL 规范不保证设备代码支持 C++ 标准库中的函数。然而，有几个编译器确实支持这些函数：这简化了现有 C++ 代码到 SYCL 设备的卸载，并使编写使用 SYCL 作为实现细节的库变得更容易（例如，将函数传递到库中的用户可以编写该函数无需使用任何 SYCL 特定函数）。

注 74 由于设备代码中对 `std::` 命名空间函数的支持因 SYCL 实现而异，因此我们无法确定采用 C++ 标准库的内核是否可以跨多个 SYCL 编译器和实现移植。

DPC++ 编译器与一组经过测试的 C++ 标准 API 兼容——我们只需包含相应的 C++ 头文件并使用 `std` 命名空间即可。所有这些 API 都可以在设备 Kernel 中使用，就像在典型的 C++ 主机应用程序中使用一样。图 18-2 显示了如何在设备代码中使用 `std::swap` 的示例。

```
int main() {
    std::array<int, 2> arr{8, 9};
    buffer<int> buf{arr};

    {
        host_accessor host_A(buf);
        std::cout << "Before: " << host_A[0] << ", "
                  << host_A[1] << "\n";
    } // End scope of host_A so that upcoming kernel can
      // operate on buf

    queue q;
    q.submit([&](handler &h) {
        accessor a{buf, h};
        h.single_task([=]() {
            // Call std::swap!
            std::swap(a[0], a[1]);
        });
    });

    host_accessor host_B(buf);
    std::cout << "After: " << host_B[0] << ", " << host_B[1]
              << "\n";
    return 0;
}

Sample output:
8, 9
9, 8
```

图 18.2: 在设备代码中使用 `std::swap`

图 18-3 列出了带有“Y”的 C++ 标准 API，表示在撰写本文时，这些 API 已在 CPU、GPU 和 FPGA 设备的 SYCL Kernel 中进行了测试。空白表示本书出版时覆盖不完整（并非所有三种设备类型）。

图 18.3: 支持 CPU/GPU/FPGA 的库 (在图书出版时)

经测试的标准 C++ API 在带有 gcc 7.5.0+ 的 libstdc++ (GNU) 和带有 clang 11.0+ 的 libc++ (LLVM) 以及带有用于主机 CPU 的 Microsoft Visual Studio 2019+ 的 MSVC 标准 C++ 库中受支持。

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。

如果我们想使用 libc++, 请使用编译选项 -stdlib=libc++ -nostdinc++ 来利用 libc++ 并且不包含系统中的 C++ std 标头。DPC++ 编译器已在 Linux 上的 SYCL Kernel 中使用 libc++ 进行了验证，但运行时需要使用 libc++ 而不是 libstdc++ 重新构建。详细信息请参见 <https://intel.github.io/llvm-docs/GetStartedGuide.html#build-dpc-toolchain-with-libc-library>。由于这些额外的步骤，如果没有特定的原因，libc++ 并不是推荐我们一般使用的 C++ 标准库。

注 75 为了实现跨架构的可移植性，如果图 18-3 中 `std::` 函数未标记为“Y”，

我们需要注意不要为我们的应用程序创建函数不正确（或构建失败），因为它运行在我们尚未测试的目标设备上！

18.3 oneAPI DPC++ 库 (oneDPL)

C++17 引入了 C++ 标准库中定义的算法的并行版本。与串行算法不同，每个并行算法都接受执行策略作为其第一个参数 - 该执行策略表示算法如何执行。

宽松地说，执行策略与实现进行通信，以确定它是否可以使用线程、SIMD 指令或两者来并行化算法。我们可以传递 seq、unseq、par 或 par_unseq 之一作为执行策略，其含义如图 18-4 所示。

Execution Policy	Meaning
seq	Sequential execution.
unseq	Unsequenced SIMD execution. This policy requires that all functions provided are safe to execute in SIMD.
par	Parallel execution by multiple threads.
par_unseq	Combined effect of unseq and par.

图 18.4: 执行策略

oneDPL 扩展了标准执行策略以提供对 SYCL 设备的支持。这些 SYCL 感知执行策略不仅指定算法应如何执行，还指定算法应在何处执行。SYCLaware 策略继承了标准 C++ 执行策略，封装了 SYCL 设备或队列，并允许我们设置可选的 Kernel 名称。SYCLaware 执行策略可与所有支持符合 C++17 标准的执行策略的标准 C++ 算法一起使用。

oneDPL 不依赖于任何单个 SYCL 编译器，它旨在支持所有 SYCL 编译器。

在我们可以使用 oneDPL 及其 SYCL 感知执行策略之前，我们需要添加一些额外的头文件。我们包含哪些标头取决于我们打算使用的算法，一些常见的示例包括：

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/numeric>
#include <oneapi/dpl/memory>
```

18.3.1 SYCL 执行政策

目前，只有具有并行未排序策略 (par_unseq) 的算法才能安全卸载到 SYCL 设备。这种限制源于 SYCL 中工作项提供的向前进度保证，这与其他执行策略（例如 par）的要求不兼容。

使用 SYCL 执行策略分为三个步骤：

1. 将 #include <oneapi/dpl/execution> 添加到我们的代码中。
2. 通过提供标准策略类型、作为模板参数（可选）的唯一 Kernel 名称的类类型以及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 Kernel 名称的现有策略对象
3. 将创建的策略对象传递给算法。

oneapi::dpl::execution::dpcpp_default 对象是使用默认 Kernel 名称和默认队列创建的预定义 device_policy。这可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）。

```
auto policy_b = device_policy<parallel_unsequenced_policy,
                           class PolicyB>{
    sycl::device{sycl::gpu_selector{}};
    std::for_each(policy_b, ...);
}
auto policy_c =
    device_policy<parallel_unsequenced_policy,
                  class PolicyC>{sycl::default_selector{}};
std::for_each(policy_c, ...);
auto policy_d =
    make_device_policy<class PolicyD>(default_policy);
std::for_each(policy_d, ...);
auto policy_e =
    make_device_policy<class PolicyE>(sycl::queue{});
std::for_each(policy_e, ...);
```

图 18.5: 创建执行策略

图 18-5 显示了假设使用 using 命名空间 oneapi::dpl::execution 的示例；引用策略类和函数时的指令。

18.3.2 将 oneDPL 与 Buffer 结合使用

C++ 标准库中的算法都是基于迭代器的。为了支持将 SYCL Buffer 传递到这些算法中，oneDPL 定义了两个特殊的辅助函数：oneapi::dpl::begin 和 oneapi::dpl::end。

这些函数接受 SYCL Buffer 并返回满足以下要求的未指定类型的对象：

- 可复制构造、可复制分配，并可与运算符 == 和 != 进行比较。
- 以下表达式有效： $a + n$ 、 $a - n$ 和 $a - b$ ，其中 a 和 b 是该类型的对象， n 是整数值。
- 具有不带参数的 get_buffer 方法。

该方法返回传递给 oneapi::dpl::begin 和 oneapi::dpl::end 函数的 SYCL Buffer。

请注意，使用这些辅助函数需要我们将 #include <oneapi/dpl/iterator> 添加到我们的代码中。默认情况下不包含此函数，因为使用 USM 时不需要这些迭代器（我们将很快重新讨论）。

```
#include <oneapi/dpl/algorithms>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>

int main() {
    sycl::queue q;
    sycl::buffer<int> buf{1000};

    auto buf_begin = oneapi::dpl::begin(buf);
    auto buf_end = oneapi::dpl::end(buf);

    auto policy = oneapi::dpl::execution::make_device_policy<
        class fill>(q);
    std::fill(policy, buf_begin, buf_end, 42);

    return 0;
}
```

图 18.6：使用 `std::fill`

图 18.6 中的代码显示了如何将 `std::fill` 函数与开始/结束帮助程序结合使用来填充 SYCL Buffer。请注意，算法位于 `std::` 命名空间中，只有执行

策略位于非标准命名空间中——这不是拼写错误！C++ 标准库明确允许实现定义自己的执行策略来支持这样的编码模式。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>

int main() {
    std::vector<int> v(100000);
    std::fill(oneapi::dpl::execution::dpcpp_default,
              v.begin(), v.end(), 42);

    if (v[788] == 42)
        std::cout << "passed" << std::endl;
    else
        std::cout << "failed" << std::endl;

    return 0;
}
```

图 18.7: 将 `std::fill` 与默认策略和主机端迭代器结合使用

图 18-7 中的代码显示了该代码的更简单版本，使用默认策略和普通（主机端）迭代器。在这种情况下，会创建一个临时 SYCL Buffer，并将数据复制到该 Buffer。设备上的临时 Buffer 处理完成后，数据将复制回主机。建议直接使用现有的 SYCL Buffer（如果可能），以减少主机和设备之间的数据移动以及 Buffer 创建和销毁的任何不必要的开销。

```
#include <oneapi/dpl/algorithm>
#include <iostream>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>

using namespace sycl;

int main() {
    buffer<uint64_t, 1> kB{range<1>(10)};
    buffer<uint64_t, 1> vB{range<1>(5)};
    buffer<uint64_t, 1> rB{range<1>(5)};
{
    host_accessor k{kB};
    host_accessor v{vB};

    // Initialize data, sorted
    k[0] = 0;
    k[1] = 5;
    k[2] = 6;
    k[3] = 6;
    k[4] = 7;
    k[5] = 7;
    k[6] = 8;
    k[7] = 8;
    k[8] = 9;
    k[9] = 9;

    v[0] = 1;
    v[1] = 6;
    v[2] = 3;
    v[3] = 7;
    v[4] = 8;
}
// create dpc++ iterators
auto k_beg = oneapi::dpl::begin(kB);
auto k_end = oneapi::dpl::end(kB);
auto v_beg = oneapi::dpl::begin(vB);
auto v_end = oneapi::dpl::end(vB);
auto r_beg = oneapi::dpl::begin(rB);

// create named policy from existing one
auto policy = oneapi::dpl::execution::make_device_policy<
    class bSearch>(oneapi::dpl::execution::dpcpp_default);

// call algorithm
oneapi::dpl::binary_search(policy, k_beg, k_end, v_beg,
                           v_end, r_beg);

// check data
host_accessor r{rB};
if ((r[0] == false) && (r[1] == true) &&
    (r[2] == false) && (r[3] == true) && (r[4] == true)) {
    std::cout << "Passed. \nRun on "
        << policy.queue()
            .get_device()
            .get_info<info::device::name>()
        << "\n";
} else
    std::cout << "Failed: values do not match.\n";
}

return 0;
}
```

图 18-8 显示了一个示例，该示例对输入序列执行二分搜索，以查找所提供的搜索序列中的每个值。作为搜索序列的第 i 个元素的搜索结果，指示是否在输入序列中找到搜索值的布尔值被分配给结果序列的第 i 个元素。该算法返回一个迭代器，该迭代器指向分配了结果的结果序列的最后一个元素。该算法假设输入序列已由提供的比较器排序。如果未提供比较器，则将使用使用运算符 $<$ 来比较元素的函数对象。

前面描述的复杂性强调我们应该尽可能利用库函数，而不是编写我们自己的类似算法的实现，这可能需要大量的调试和调整时间。我们可以利用的库的作者通常是我们所针对的设备架构内部的专家，并且可能有权访问我们无法访问的信息，因此我们应该始终在可用时利用优化的库。

图 18-8 中所示的代码示例演示了将 oneDPL 与 SYCL Buffer 结合使用时的三个典型步骤：

1. 从我们的 Buffer 创建 SYCL 迭代器。
2. 根据现有策略创建命名策略。
3. 调用并行算法。

18.3.3 将 oneDPL 与 USM 结合使用

在本节中，我们将探讨将 oneDPL 与 USM 结合使用的两种方法：

- 通过 USM 指针
- 通过 USM 分配器

与 Buffer 不同，我们可以直接使用 USM 指针作为传递给算法的迭代器。具体来说，我们可以将指向分配开始和（过去）结束的指针传递给并行算法。重要的是要确保执行策略和分配本身是为同一队列或上下文创建的，以避免运行时未定义的行为。（请记住，这不是特定于 oneDPL 的，我们在使用 USM 时必须始终密切注意上下文！）

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <sycl/sycl.hpp>

int main() {
    sycl::queue q;
    const int n = 10;
    int* h_head = sycl::malloc_host<int>(n, q);
    int* d_head = sycl::malloc_device<int>(n, q);
    std::fill(oneapi::dpl::execution::make_device_policy(q),
              d_head, d_head + n, 78);
    q.wait();

    q.memcpy(h_head, d_head, n * sizeof(int));
    q.wait();

    if (h_head[8] == 78)
        std::cout << "passed" << std::endl;
    else
        std::cout << "failed" << std::endl;

    sycl::free(h_head, q);
    sycl::free(d_head, q);
    return 0;
}
```

图 18.9: 将 oneDPL 与 USM 指针配合使用

如果相同的 USM 分配要由多个算法处理，我们可以使用有序队列或显式等待每个算法完成，然后再在下一个算法中使用相同的分配（这是使用 USM 时的典型操作排序）。我们还应该小心确保在访问主机上的数据之前等待完成，如图 18-9 所示。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <sycl/sycl.hpp>

int main() {
    sycl::queue q;
    const int n = 10;
    sycl::usm_allocator<int, sycl::usm::alloc::shared> alloc(
        q);
    std::vector<int, decltype(alloc)> vec(n, alloc);

    std::fill(oneapi::dpl::execution::make_device_policy(q),
              vec.begin(), vec.end(), 78);
    q.wait();

    return 0;
}
```

Figure 18-10. Using oneDPL with a USM allocator

图 18.10: 将 oneDPL 与 USM 分配器配合使用

或者，我们可以将 `std::vector` 与 USM 分配器一起使用，如图 18-10 所示。通过这种方法，`std::vector` 管理自己的内存（正常情况下），但通过对 `sycl::malloc_shared` 的内部调用来分配它需要的任何内存。然后，`begin()` 和 `end()` 成员函数返回逐步执行 USM 分配的迭代器。这种编程风格非常方便，尤其是在迁移已经使用容器和算法的现有 C++ 代码时。

18.3.4 使用 SYCL 执行策略进行错误处理

正如第 5 章所详述的，SYCL 错误处理模型支持两种类型的错误。对于同步错误，运行时会引发异常，而异步错误仅在程序执行期间的指定时间由异步错误处理程序处理。

对于使用 SYCL 感知执行策略执行的算法，所有错误（同步或异步）的处理是调用者的责任。具体来说，

- 算法不会显式抛出异常。
- 主机 CPU 上的运行时抛出的异常（包括 SYCL 同步异常）将传递给调用者。

- SYCL 异步错误不由 oneDPL 处理，因此调用者必须使用通常的 SYCL 异步异常机制来处理（如果需要任何处理）。

18.4 总结

我们应该在异构应用程序中尽可能使用库，以避免浪费时间重写和测试通用函数和并行模式。我们应该利用他人的工作，而不是自己编写所有内容，并且我们应该在可行的情况下使用这种方法来简化应用程序开发并（通常）实现卓越的性能。

本章简要介绍了我们认为每个 SYCL 开发人员都应该熟悉的三组库函数：

1. SYCL 内置函数，用于常见的数学运算
2. 标准 C++ 库，用于其他常用操作
3. C++17 并行算法（由 oneDPL 支持），用于完整 Kernel

对于任何库，在生产中依赖它们之前，了解哪些设备、编译器和实现经过测试和支持非常重要。这不是 SYCL 特定的建议，但值得记住 - 像 SYCL 这样的可移植编程解决方案的潜在目标数量巨大，作为程序员，我们有责任确定哪些库与我们的目标一致。

19 内存模型和原子

如果我们想成为并行程序员，内存一致性并不是一个深奥的概念。它帮助我们确保数据在我们需要的时候出现在我们需要的地方，并且它的值是我们所期望的。本章揭示了我们需要掌握的关键知识，以确保我们的程序正确运行。这个主题并不是 SYCL 独有的。

对于任何想要允许并发更新内存的程序员来说，对编程语言的内存（一致性）模型有基本的了解是必要的（无论这些更新是否来自同一 Kernel、多个设备或两者中的多个 Work-Items）。无论内存如何分配都是如此，无论我们选择使用 Buffer 还是 USM 分配，本章的内容对我们来说都同样重要。

在前面的章节中，我们重点关注简单 Kernel 的开发，其中 Work-Items 要么对完全独立的数据进行操作，要么使用可以使用语言和/或库功能直接表达的结构化通信模式共享数据。当我们转向编写更复杂和更现实的 Kernel 时，我们可能会遇到 Work-Items 可能需要以不太结构化的方式进行通信的情况 - 了解内存模型如何与 SYCL 语言功能以及我们目标硬件的功能相关。设计正确、可移植、高效的程序的必要前提。

注 76 (执行线程) C++17 引入了“执行线程”（通常简称为“线程”）的概念，以帮助描述与并行性和并发性相关的库功能（例如，并行算法）的行为。C++ 内存一致性模型和执行模型完全根据这些“线程”之间的交互来定义。

为了简化 SYCL 和 C++ 之间的比较，本章通常使用术语“线程”来表示“执行线程”。SYCL 工作项等效于具有弱并行前向进度保证的 C++ 执行线程，因此可以安全地互换使用这些术语偶尔，在讨论特定于 SYCL 的概念时，我们可能仍会使用“工作项”来突出显示。

C++ 的内存一致性模型足以编写完全在主机上执行的应用程序，但它被 SYCL 修改，以解决编程异构系统时可能出现的复杂性。具体来说，我们需要能够

- 系统中哪些设备可以访问哪些类型的内存分配（Buffer 和 USM）的原因
- 通过使用 Barrier 和原子来防止 Kernel 执行期间不安全的并发内存访问（数据竞争）
- 使用 Barrier、栅栏、原子、内存顺序和内存范围实现 Work-Items 之间的安全通信

- 使用 Barrier、栅栏、原子、内存顺序和内存范围，防止可能意外改变并行应用程序行为的优化，同时仍允许其他优化

内存模型是一个复杂的主题，但有一个很好的理由——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我们的代码！我们在本章中努力分解这种复杂性并突出最关键的概念和语言特征。本章使我们不仅了解内存模型的内部和外部，而且还了解并行编程的一个重要方面，而许多人不知道它的存在。如果阅读此处的描述和示例代码后仍有问题，我们强烈建议访问本章末尾列出的网站或参考 C++ 和 SYCL 规范。

19.1 内存模型中有什么？

本节扩展了编程语言包含内存模型的动机，并介绍了并行程序员应该熟悉的一些核心概念：

- 数据竞争和同步
- Barrier 和栅栏
- 原子操作
- 内存排序

为了理解它们在 C++ 和 SYCL 中的表达和用法，需要从高层次上理解这些概念。在并行编程（尤其是使用 C++）方面具有丰富经验的读者可能希望跳过。

19.1.1 数据竞争和同步

我们在程序中编写的操作通常不会直接映射到单个硬件指令或微操作。一个简单的加法运算（例如 `data[i] += x`）可以分解为一系列指令或微操作：

- 将 `data[i]` 从内存加载到临时（寄存器）中。
- 计算将 `x` 添加到 `data[i]` 的结果。
- 将结果存储回 `data[i]`。

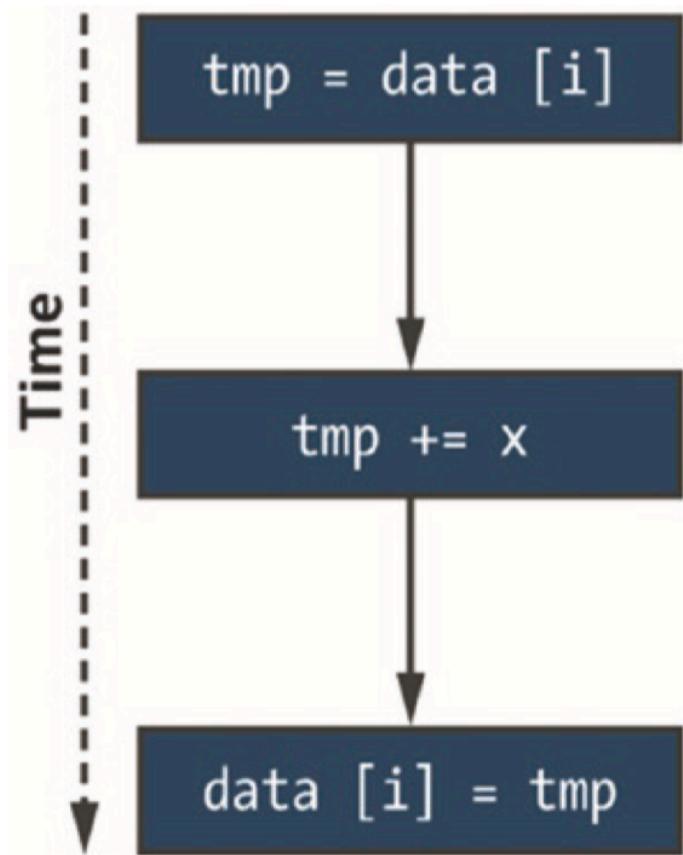


图 19.1: 数据的顺序执行 $[i] += x$ 分为三个单独的操作

这不是我们在开发顺序应用程序时需要担心的事情——加法的三个阶段将按照我们期望的顺序执行, 如图 19-1 所示。

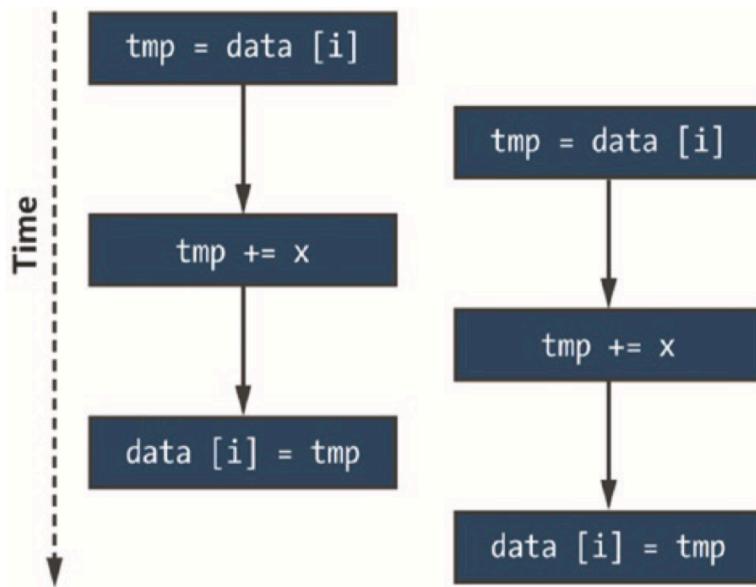


图 19.2: 一种可能的数据交错 $[i] += x$ 同时执行

切换到并行应用程序开发会带来额外的复杂性：如果我们同时将多个操作应用于同一数据，我们如何确定他们对该数据的看法是一致的？考虑图 19-2 中所示的情况，其中 $\text{data}[i] += x$ 的两次执行在两个线程上交错执行。如果两个线程使用不同的 i 值，应用程序将正确执行。如果它们使用相同的 i 值，则两者都会从内存中加载相同的值，并且其中一个结果会被另一个结果覆盖！这只是调度操作的多种方式之一，我们的应用程序的行为取决于哪个线程首先获取哪个数据——我们的应用程序包含数据竞争。

```

int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

q.parallel_for(N, [=](id<1> i) {
    int j = i % M;
    data[j] += 1;
}).wait();

for (int i = 0; i < N; ++i) {
    std::cout << "data [" << i << "] = " << data[i] << "\n";
}

```

图 19.3: 包含数据争用的内核

N = 2, M = 2:
 data [0] = 1
 data [1] = 1

N = 2, M = 1:
 data [0] = 1
 data [1] = 0

图 19.4: 图 19-3 中 N 和 M 小值的代码输出示例

图 19-3 中的代码及其图 19-4 中的输出显示了这种情况在实践中是多么容易发生。如果 M 大于等于 N，则每个线程使用的 j 的值是唯一的；如果不是，j 的值将发生冲突，并且更新可能会丢失。我们说可能会丢失，因为包含数据竞争的程序仍然可以在某些或一直产生正确的答案（取决于实现和硬件如何安排工作）。编译器和硬件都不可能知道这个程序打算做什么，或者运行时 N 和 M 的值可能是什么——作为程序员，我们有责任了解我们的程序是否可能包含数据竞争以及它们是否对数据竞争敏感。执行顺序。

一般来说，在开发大规模并行 SYCL 应用程序时，我们不应该关心各

个 Work-Items 执行的确切顺序 - 每个 Kernel 中希望有数百（或数千！）Work-Items，并试图强加对它们的特定排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正确执行的可移植应用程序，我们可以通过向编译器（和硬件）提供有关 Work-Items 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合法的信息来实现这一点。

注 77 大规模并行应用程序不应关注单个工作项执行的确切顺序！

19.1.2 Barrier 和栅栏

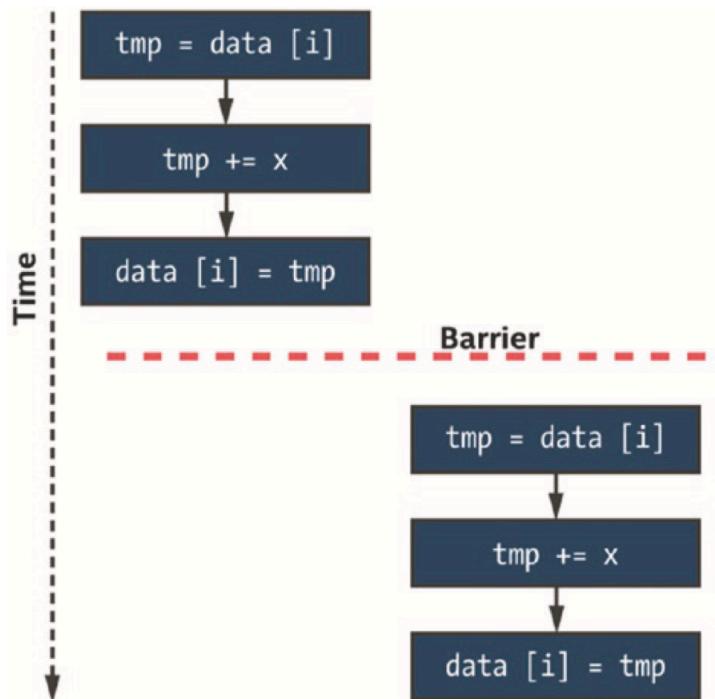


图 19.5: 两次数据执行 $[i] += x$ 由 Barrier 隔开

```

int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

// Launch exactly one work-group
// Number of work-groups = global / local
range<1> global{N};
range<1> local{N};

q.parallel_for(nd_range<1>{global, local},
    [=](nd_item<1> it) {
        int i = it.get_global_id(0);
        int j = i % M;
        for (int round = 0; round < N; ++round) {
            // Allow exactly one work-item update
            // per round
            if (i == round) {
                data[j] += 1;
            }
            group_barrier(it.get_group());
        }
    })
.wait();

for (int i = 0; i < N; ++i) {
    std::cout << "data [" << i << "] = " << data[i] << "\n";
}

```

图 19.6: 使用 *Barrier* 避免进行数据争用

防止同一组中的 Work-Items 之间发生数据争用的一种方法是使用 Work-Groups Barrier 和适当的内存栅栏在不同线程之间引入同步。我们可以使用 Work-Groups Barrier 来排序 data[i] 的更新，如图 19-5 所示，并且图 19-6 中给出了示例 Kernel 的更新版本。请注意，由于 Work-Groups Barrier 不会同步不同组中的 Work-Items，因此只有当我们自己限制为单个 Work-Groups 时，我们的简单示例才能保证正确执行！

尽管使用 Barrier 来实现此模式是可能的，但通常不鼓励这样做 - 它强制组中的 Work-Items 按特定顺序顺序执行，这可能会导致在存在负载不平衡的情况下长时间处于不活动状态。它还可能引入比严格必要的同步更多的同步——如果不同的 Work-Items 碰巧使用不同的 i 值，它们仍将被迫在 Barrier 处同步。

Barrier 同步是一个有用的工具，可确保 Work-Groups 或 Sub-Groups 中的所有 Work-Items 在进入下一阶段之前完成 Kernel 的某个阶段，但对

于细粒度（以及潜在的数据同步）来说过于严厉。对于更通用的同步模式，我们必须关注原子操作。

19.1.3 原子操作

原子操作支持对内存位置的并发访问，而不会引入数据竞争。当多个原子操作访问同一内存时，保证它们不会重叠。请注意，如果只有部分访问使用原子性，则此保证不适用，并且作为程序员，我们有责任确保我们不会使用具有不同原子性保证的操作同时访问相同的数据。

注 78 在同一内存位置上同时混合原子和非原子操作会导致未定义的行为！

```
int* data = malloc_shared<int>(N, q);
std::fill(data, data + N, 0);

q.parallel_for(N, [=](id<1> i) {
    int j = i % M;
    atomic_ref<int, memory_order::relaxed,
               memory_scope::system,
               access::address_space::global_space>
        atomic_data(data[j]);
    atomic_data += 1;
}).wait();

for (int i = 0; i < N; ++i) {
    std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

图 19.7: 使用原子操作避免进行数据竞争

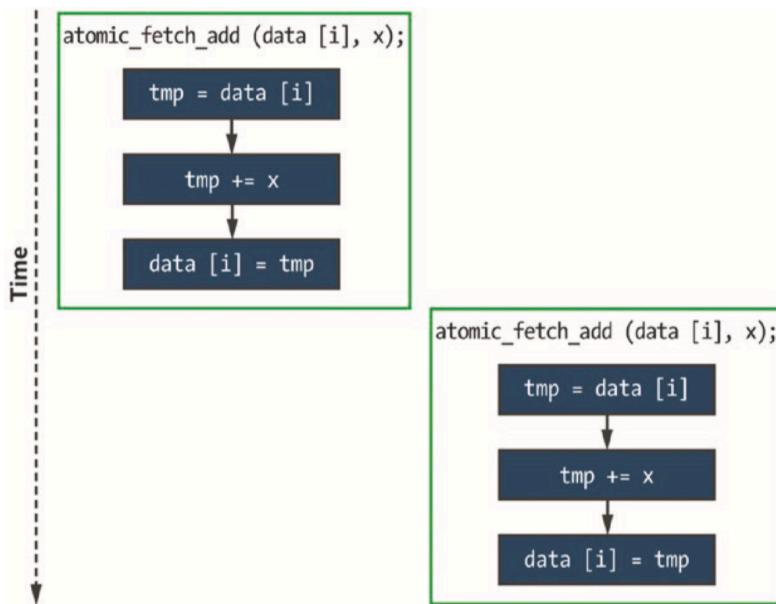


图 19.8: 与原子操作同时执行的数据交错 $[i] += x$

如果我们的简单加法使用原子操作来表达，结果可能如图 19-8 所示——每次更新现在都是一个不可分割的工作块，我们的应用程序将始终产生正确的结果。相应的代码如图 19-7 所示——我们将在本章后面回顾 `atomic_ref` 类及其模板参数的含义。

然而，值得注意的是，这仍然只是一种可能的执行顺序。使用原子操作可以保证两个更新不会重叠（如果两个线程使用相同的 i 值），但仍然无法保证两个线程中的哪一个将首先执行。更重要的是，无法保证这些原子操作相对于不同线程中的任何非原子操作的排序方式。

19.1.4 内存序

即使在顺序应用程序中，优化编译器和硬件也可以自由地重新排序操作，如果它们不改变应用程序的可观测行为。换句话说，应用程序的行为必须完全按照程序员编写的方式运行。

不幸的是，这种假设的保证不足以帮助我们推理并行程序的执行。我们现在需要担心两个重新排序的来源：编译器和硬件可能会对每个顺序线程内的语句执行重新排序，并且线程本身可以以任何（可能是交错的）顺序执

行。为了设计和实现线程之间的安全通信协议，我们需要能够限制这种重新排序。通过向编译器提供有关我们所需的内存顺序的信息，我们可以防止与应用程序的预期行为不兼容的重新排序优化。

三种常用的内存顺序是：

1. 宽松的内存排序
2. 获取-释放或释放-获取内存顺序
3. 顺序一致的内存排序

在宽松的内存排序下，内存操作可以不受任何限制地重新排序。宽松内存模型最常见的用法是递增共享变量（例如，单个计数器、直方图计算期间的值数组）。

在获取-释放内存顺序下，一个线程释放一个原子变量，另一个线程获取相同的原子变量，充当这两个线程之间的同步点，并保证释放线程发出的任何先前对内存的写入对于获取线程都是可见的。通俗地说，我们可以认为原子操作将其他内存操作的副作用释放到其他线程，或者获取其他线程上的内存操作的副作用。如果我们想通过内存线程对之间传递值，则需要这样的内存模型，这可能比我们想象的更常见。当程序获取锁时，它通常会继续执行一些额外的计算并在最终释放锁之前修改一些内存 - 只有锁变量会被原子更新，但我们希望受锁保护的内存更新能够免受数据竞争的影响。此行为依赖于获取-释放内存顺序的正确性，并且尝试使用宽松的内存顺序来实现锁将不起作用。

在顺序一致的内存排序下，获取-释放顺序的保证仍然有效，但还存在所有原子操作的单个全局顺序。这种内存排序的行为是三者中最直观的，也是最接近我们在开发顺序应用程序时习惯依赖的原始假设保证的。通过顺序一致性，推理线程组（而不是线程对）之间的通信变得更加容易，因为所有线程必须就所有原子操作的全局顺序达成一致。

了解编程模型和设备的组合支持哪些内存顺序是设计可移植并行应用程序的必要部分。明确地描述应用程序所需的内存顺序可确保当我们所需的行为不受支持时，应用程序可以预见地失败（例如，在编译时），并防止我们做出不安全的假设。

19.2 内存模型

到目前为止，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

- 如何表达 Kernel 的内存排序要求
- 如何查询特定设备支持的内存顺序
- 内存模型在不相交的地址空间和多个设备方面的行为方式
- 内存模型如何与 Barrier、栅栏和原子交互
- Buffer 和 USM 之间使用原子操作有何不同

该内存模型基于 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 SYCL 应该帮助告知 C++ 的未来：类的默认行为和命名与 C++ 标准库紧密一致，旨在扩展 C++ 功能而不是限制它。

图 19-9 中的表总结了 C++ (C++11、C++14、C++17、C++20) 与 SYCL 中如何将不同的内存模型概念公开为语言功能。C++14、C++17 和 C++20 标准还包含一些影响 C++ 实现的说明。这些说明不应影响我们编写的应用程序代码，因此我们在此不讨论它们。

Feature	C++	SYCL
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>acq_rel</code> <code>seq_cst</code>	<code>relaxed</code> <code>acquire</code> <code>release</code> <code>acq_rel</code> <code>seq_cst</code>
Memory Scopes	Not available. Behavior of atomics and fences matches SYCL 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>sycl::group_barrier</code>
Address Spaces	All memory is in a single (host) address space.	Host Device (Global) Device (Local) Device (Private) Shared (USM)

图 19.9: 比较 C++ 和 SYCL 内存模型

19.2.1 memory_order 枚举类

内存模型通过 memory_order 枚举类的五个值公开不同的内存顺序（注意：C++“consume”不是 SYCL 的一部分），这些值可以作为参数提供给栅栏和原子操作。向操作提供内存顺序参数告诉编译器相对于该操作的所有其他内存操作（到任何地址）需要什么内存顺序保证，如下所述：

- memory_order::relaxed 读写操作可以在操作之前或之后重新排序，没有任何限制，没有序保证。
- memory_order::acquire 程序中在该操作之后出现的读和写操作必须在该操作之后出现（即，它们不能在该操作之前重新排序）。
- memory_order::release 程序中出现在该操作之前的读写操作必须发生在该操作之前（即操作之后不能重新排序），并且保证前面的写操作对已同步的其他 Work-Items 可见通过相应的获取操作（即使用相同变量和 memory_order::acquire 或 Barrier 函数的原子操作）。
- memory_order::acq_rel 该操作既充当获取又充当释放。读取和写入操作不能围绕该操作重新排序，并且前面的写入必须可见，如之前针对 memory_order::release 所描述的。
- memory_order::seq_cst 该操作分别充当获取、释放或两者，具体取决于它是读、写还是读-修改-写操作。具有此内存顺序的所有操作都以连续一致的顺序观察。

Functions	Supported <code>memory_order</code> Values				
	relaxed	acquire	release	acq_rel	seq_cst
load	✓	✓	✗	✗	✓
store	✓	✗	✓	✗	✓
exchange					
compare_exchange_*	✓	✓	✓	✓	✓
fetch_*					
fence	✓	✓	✓	✓	✓

图 19.10: 支持原子操作 `memory_order`

每个操作支持哪些内存顺序有几个限制。图 19-10 中的表总结了哪些组合是有效的。

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，存储操作不会从内存中读取值，因此与获取语义不兼容。其余的读-修改-写原子操作和栅栏与所有内存顺序兼容。

注 79 (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。但是, C++17 不鼓励使用它，并指出其定义正在修订中。因此，将其纳入 SYCL 有待考虑用于未来的规范。

19.2.2 `memory_scope` 枚举类

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。这些假设均不适用于 SYCL 应用程序：应用程序的各个部分在不同的设备（即主机和一个或多个加速器设备）上执行；每个设备都有多个地址空间（即私有、本地和全局）；每个设备的全局地址空间可能不相交，也可能不相交（取决于 USM 支持）。

为了解决这个问题，SYCL 扩展了内存顺序的 C++ 概念，以包含原子操作的范围，表示给定内存顺序约束适用的最小 Work-Items 集。这组范围是通过 `memory_scope` 枚举类来定义的：

- `memory_scope::work_item`

内存排序约束仅适用于调用 Work-Items。此范围仅对图像操作有用，因为 Work-Items 内的所有其他操作都已保证按程序顺序执行。

- `memory_scope::sub_group, memory_scope::work_group`

内存排序约束仅适用于与调用 Work-Items 位于同一 Sub-Groups 或 Work-Groups 中的 Work-Items。

- `memory_scope::device`

内存排序约束仅适用于与调用 Work-Items 在同一设备上执行的 Work-Items。

- `memory_scope::system`

内存排序约束适用于系统中的所有 Work-Items。

除非设备功能施加限制，否则所有内存范围都是所有原子和栅栏操作的有效参数。但是，在以下三种情况之一中，范围参数可能会自动降级为更窄的范围：

1. 如果原子操作更新 Work-Groups 本地内存中的值，则任何比 `memory_scope::work_group` 更宽的范围都会缩小（因为本地内存仅对同一 Work-Groups 中的 Work-Items 可见）。
2. 如果设备不支持 USM，则指定 `memory_scope::system` 始终等同于 `memory_scope::device`（因为多个设备不能同时访问 Buffer）。
3. 如果原子操作使用 `memory_order::relaxed`，则没有顺序保证，并且内存范围参数实际上被忽略。

19.2.3 查询设备能力

为了确保与早期版本 SYCL 支持的设备兼容并最大限度地提高可移植性，SYCL 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（例如某些类别的嵌入式设备）。SYCL 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顺序和内存范围：

- `atomic_memory_order_capabilities`

返回特定设备上原子操作支持的所有内存排序的列表。

所有设备都必须至少支持 `memory_order::relaxed`。

- `atomic_fence_order_capabilities`

返回特定设备上的栅栏操作支持的所有内存排序的列表。

所有设备都必须至少支持 `memory_order::relaxed`、`memory_order::acquire`、`memory_order::release` 和 `memory_order::acq_rel`。请注意，栅栏的最低要求强于原子操作的最低要求，因为此类栅栏对于在存在 Barrier 的情况下推理内存顺序至关重要。

- `atomic_memory_scope_capabilities`

`atomic_fence_scope_capabilities`

返回特定设备上原子和栅栏操作支持的所有内存范围的列表。所有设备都必须至少支持 `memory_order::work_group`。

一开始可能很难记住哪些功能和设备功能的组合支持哪些内存顺序和范围。在实践中，我们可以通过遵循下面概述的两种开发方法之一来避免这种复杂性：

1. 开发具有顺序一致性和系统围栏的应用程序。

在性能调优期间仅考虑采用不太严格的内存顺序。

2. 开发具有宽松一致性和 Work-Groups 界限的应用程序。

仅在需要正确性时才考虑采用更严格的内存顺序和更广泛的内存范围。

第一种方法确保所有原子操作和栅栏的语义与 C++ 的默认行为相匹配。这是最简单且最不易出错的选项，但性能和可移植性特征最差。

第二种方法更符合以前版本的 SYCL 和 OpenCL 等语言的默认行为。尽管更复杂（因为它要求我们更加熟悉不同的内存顺序和范围），但它确保我们编写的大部分 SYCL 代码都可以在任何设备上运行，而不会造成性能损失。

19.2.4 Barrier 和栅栏

到目前为止，本书中所有之前对 Barrier 和栅栏的使用都依赖于默认行为，忽略了内存顺序和范围的问题。

默认情况下，SYCL 中的每个组 Barrier 充当调用 Work-Items 可访问的所有地址空间的获取-释放栅栏，并使先前的写入至少对同一组中的所有

其他 Work-Items 可见（由组的 fence_scope 定义）成员变量）。这确保了 Barrier 后一组 Work-Items 内的内存一致性，符合我们对同步含义的直觉（以及 C++ 中同步关系的定义）。可以通过将显式的 Memory_scope 参数传递给 group_Barrier 函数来覆盖此默认行为。

atomic_fence 函数为我们提供了比这更细粒度的控制，允许 Work-Items 执行指定内存顺序和范围的栅栏。

19.2.5 SYCL 中的原子操作

SYCL 提供对多种数据类型的多种原子操作的支持。所有设备都保证支持常见操作的原子版本（例如加载、存储、算术运算符），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 位类型支持是可选的。

atomic 类 C++11 中的 std::atomic 类提供了用于创建和操作原子变量的接口。原子类的实例拥有自己的数据，不能移动或复制，只能使用原子操作进行更新。这些限制显着减少了错误使用类和引入未定义行为的机会。不幸的是，它们还阻止该类在 SYCL Kernel 中使用——不可能在主机上创建原子对象并将它们传输到设备！我们可以继续在主机代码中使用 std::atomic，但尝试在设备 Kernel 中使用它会导致编译器错误。

注 80 (SYCL 2020 中不推荐使用的原子类) SYCL 1.2.1 规范包括一个 *cl::sycl::atomic* 类，该类松散地基于 C++ 11 中的 std::atomic 类。我们之所以这么说，是因为这两个类的接口之间存在一些差异，最明显的是 SYCL 1.2.1 版本不拥有其数据，并且默认为宽松的内存排序。

cl::sycl::atomic 类在 SYCL 2020 中已弃用。应使用 atomic_ref 类（在下一节中介绍）来代替它。

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 的实例不拥有其数据，而是从现有的非原子变量构造而成。创建原子引用有效地充当了一个承诺，即仅在引用的生命周期内以原子方式访问引用的变量。这些正是 SYCL 所需的语义，因为它们允许我们在

主机上创建非原子数据，将该数据传输到设备，并仅在传输后将其视为原子数据。因此，SYCL Kernel 中使用的 atomic_ref 类基于 std::atomic_ref。

```
template <typename T, memory_order DefaultOrder,
          memory_scope DefaultScope,
          access::address_space AddressSpace>
class atomic_ref {
public:
    using value_type = T;
    static constexpr size_t required_alignment =
        /* implementation-defined */;
    static constexpr bool is_always_lock_free =
        /* implementation-defined */;
    static constexpr memory_order default_read_order =
        memory_order_traits<DefaultOrder>::read_order;
    static constexpr memory_order default_write_order =
        memory_order_traits<DefaultOrder>::write_order;
    static constexpr memory_order
        default_read_modify_write_order = DefaultOrder;
    static constexpr memory_scope default_scope =
        DefaultScope;

    explicit atomic_ref(T& obj);
    atomic_ref(const atomic_ref& ref) noexcept;
};
```

图 19.11: atomic_ref 类的构造函数和静态成员

我们说基于是因为该类的 SYCL 版本包括三个附加模板参数，如图 19-11 所示。

如前所述，不同 SYCL 设备的功能各不相同。为 SYCL 的原子类选择默认行为是一个困难的提议：默认为 C++ 行为（即 memory_order::seq_cst、memory_scope::system）会限制代码仅在功能最强大的设备上执行；另一方面，在迁移现有 C++ 代码时，打破 C++ 约定并默认使用最低公分母（即 memory_order::relaxed、memory_scope::work_group）可能会导致意外行为。SYCL 采用的设计提供了一种折衷方案，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）。其他顺序和范围可以作为我们认为合适的特定原子操作的运行时参数提供 - DefaultOrder 和 DefaultScope 仅影响我们不或无法覆盖默认行为的操作（例如，当使用像 += 这样的速记运算符时）。最后一个（可选）模板参数表

示分配引用对象的地址空间。请注意，如果未指定最终模板参数，则引用的变量可以分配在任何地址空间中 - 尽管此处指定地址空间是可选的，但我们建议提供显式地址空间（如果可能），以便为编译器提供更多信息并避免不需要的信息性能开销。

原子引用根据其引用的对象类型提供对不同操作的支持。所有类型都支持的基本操作如图 19-12 所示，提供了原子地将数据移入和移出内存的能力。

```
void store(
    T operand, memory_order order = default_write_order,
    memory_scope scope = default_scope) const noexcept;
T operator=(
    T desired) const noexcept; // equivalent to store

T load(memory_order order = default_read_order,
       memory_scope scope = default_scope) const noexcept;
operator T() const noexcept; // equivalent to load

T exchange(
    T operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_weak(
    T &expected, T desired, memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_weak(
    T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(
    T &expected, T desired, memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(
    T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;
```

图 19.12: 所有类型的 *atomic_ref* 的基本操作

```
Integral fetch_add(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_sub(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_and(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_or(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_min(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_max(
    Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral operator++(int) const noexcept;
Integral operator--(int) const noexcept;
Integral operator++() const noexcept;
Integral operator--() const noexcept;
Integral operator+=(Integral) const noexcept;
Integral operator-=(Integral) const noexcept;
Integral operator&=(Integral) const noexcept;
Integral operator|=(Integral) const noexcept;
```

图 19.13: 仅针对整型的 *atomic_ref* 附加操作

```
Floating fetch_add(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_sub(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_min(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_max(
    Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating operator+=(Floating) const noexcept;
Floating operator-=(Floating) const noexcept;
```

图 19.14: 仅对浮点类型进行 *atomic_ref* 的附加操作

对整型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 和图 19-14 所示。设备需要支持原子浮点类型，无论它们是否在硬件中具有对浮点原子的本机支持，并且许多设备期望使用原子比较交换来模拟原子浮点加法。这种模拟是在 SYCL 中提供性能和可移植性的重要部分，我们应该在算法需要的任何地方随意使用浮点原子——生成的代码将在任何地方正确工作，并将受益于浮点原子的未来改进硬件无需任何修改！

19.2.6 将原子与 Buffer 一起使用

正如上一节所讨论的，SYCL 中无法分配原子数据并在主机和设备之间移动它。要将原子操作与 Buffer 结合使用，我们必须创建一个要传输到设备的非原子数据 Buffer，然后通过原子引用访问该数据。

```
q.submit([&](handler& h) {
    accessor acc{buf, h};
    h.parallel_for(N, [=](id<1> i) {
        int j = i % M;
        atomic_ref<int, memory_order::relaxed,
                    memory_scope::system,
                    access::address_space::global_space>
            atomic_acc(acc[j]);
        atomic_acc += 1;
    });
});
```

图 19.15: 通过显式创建的 `atomic_ref` 访问 `Buffer`

图 19-15 中的代码是使用显式创建的原子引用对象在 SYCL 中表达原子性的示例。Buffer 存储普通整数，我们需要一个具有读写权限的访问器。然后，我们可以使用 `+=` 运算符作为 `fetch_add` 成员函数的简写替代，为每个数据访问创建一个 `atomic_ref` 实例。

如果我们想要在同一 Kernel 中混合对 Buffer 的原子和非原子访问，以避免在不需要时支付原子操作的性能开销，则此模式非常有用。如果我们知道 Buffer 中的内存位置的子集将被多个 Work-Items 同时访问，则在访问该子集时只需要使用原子引用。或者，如果我们知道同一 Work-Groups 中的 Work-Items 仅在 Kernel 的一个阶段（即两个 Work-Groups Barrier 之间）同时访问本地内存，则我们只需要在该阶段使用原子引用。当像这样混合原子和非原子访问时，重要的是要注意对象的生命周期——虽然存在引用特定对象的任何 `atomic_ref`，但对该对象的所有访问都必须通过 `atomic_ref` 的实例（原子地）发生。

19.2.7 将原子与统一共享内存结合使用

```
q.parallel_for(N, [=](id<1> i) {
    int j = i % M;
    atomic_ref<int, memory_order::relaxed,
               memory_scope::system,
               access::address_space::global_space>
        atomic_data(data[j]);
    atomic_data += 1;
}).wait();
```

图 19.16: 通过显式创建的 *atomic_ref* 访问 USM 分配

如图 19-16 所示（从图 19-7 复制），我们可以按照与 Buffer 完全相同的方式从 USM 中存储的数据构造原子引用。实际上，此代码与图 19-15 中所示的代码之间的唯一区别是 USM 代码不需要 Buffer 或访问器。

19.3 在现实生活中使用原子

原子的潜在用途是如此广泛和多样，我们不可能在本书中提供每种用途的示例。我们提供了两个具有跨领域广泛适用性的代表性示例：

1. 计算直方图
2. 实现全设备同步

19.3.1 计算直方图

```

q.submit([&](handler& h) {
    auto local = local_accessor<uint32_t, 1>{B, h};
    h.parallel_for(
        nd_range<1>{num_groups * num_items, num_items},
        [=](nd_item<1> it) {
            auto grp = it.get_group();

            // Phase 1: Work-items co-operate to zero local
            // memory
            for (int32_t b = it.get_local_id(0); b < B;
                b += it.get_local_range(0)) {
                local[b] = 0;
            }
            group_barrier(grp); // Wait for all to be zeroed

            // Phase 2: Work-groups each compute a chunk of
            // the input. Work-items co-operate to compute
            // histogram in local memory
            const auto [group_start, group_end] =
                distribute_range(grp, N);
            for (int i = group_start + it.get_local_id(0);
                i < group_end; i += it.get_local_range(0)) {
                int32_t b = input[i] % B;
                atomic_ref<uint32_t, memory_order::relaxed,
                    memory_scope::work_group,
                    access::address_space::local_space>(local[b])++;
            }
            group_barrier(
                grp); // Wait for all local histogram
            // updates to complete

            // Phase 3: Work-items co-operate to update
            // global memory
            for (int32_t b = it.get_local_id(0); b < B;
                b += it.get_local_range(0)) {
                atomic_ref<uint32_t, memory_order::relaxed, memory_scope::system,
                    access::address_space::global_space>(histogram[b]) +=
                    local[b];
            }
        });
    }).wait();
}

```

图 19.17: 使用不同内存空间中的原子引用计算直方图

图 19.17 中的代码演示了如何将宽松原子与 Work-Groups Barrier 结合使用来计算直方图。Kernel 被 Barrier 分为三个阶段，每个阶段都有自己的原子性要求。请记住，Barrier 既充当同步点又充当获取-释放栅栏，这确保了一个阶段中的任何读取和写入对于后续阶段中 Work-Groups 中的所有 Work-Items 都是可见的。

第一阶段将某些 Work-Groups 本地内存的内容设置为零。每个 Work-Groups 中的 Work-Items 按照设计更新 Work-Groups 本地内存中的独立位置——不会发生竞争条件，并且不需要原子性。

第二阶段将部分直方图结果累积在本地存储器中。同一 Work-Groups 中的 Work-Items 可以更新 Work-Groups 本地内存中的相同位置，但同步可以推迟到阶段结束——我们可以使用 `memory_order::relaxed` 和 `memory_scope::work_group` 来满足原子性要求。

第三阶段将部分直方图结果贡献到存储在全局存储器中的总数中。同一 Work-Groups 中的 Work-Items 保证从 Work-Groups 本地内存中的独立位置读取，但可能会更新全局内存中的相同位置——我们不再要求 Work-Groups 本地内存的原子性，并且可以满足原子性像以前一样使用 `memory_order::relaxed` 和 `memory_scope::system` 对全局内存的要求。

19.3.2 实现设备范围的同步

回到第 4 章，我们警告不要编写试图跨 Work-Groups 同步 Work-Items 的 Kernel。然而，我们完全预计本章的一些读者会渴望在原子操作之上实现他们自己的设备范围同步例程，并且我们的警告将被忽略。

注 81 设备范围的同步目前是不可移植的，最好留给专业程序员。*SYCL* 的未来版本将解决这个问题。

本节中讨论的代码很危险，不应期望在所有设备上都能工作，因为设备硬件功能和 SYCL 实现之间存在潜在差异。原子提供的内存排序保证与转发进度保证正交，并且在撰写本文时，SYCL 中的 Work-Groups 调度完全是实现定义的。正式化描述 SYCL ND 范围执行模型所需的概念和术语以及与 Work-Items、Sub-Groups 和 Work-Groups 相关的前进进度保证是目前活跃的学术研究领域 - 预计将构建 SYCL 的未来版本在此工作上提供额外的调度查询和控制。目前，这些主题应被视为仅供专家讨论。

```
struct device_latch {
    explicit device_latch(size_t num_groups)
        : counter(0), expected(num_groups) {}

    template <int Dimensions>
    void arrive_and_wait(nd_item<Dimensions>& it) {
        auto grp = it.get_group();
        group_barrier(grp);
        // Elect one work-item per work-group to be involved in
        // the synchronization. All other work-items wait at the
        // barrier after the branch.
        if (grp.leader()) {
            atomic_ref<size_t, memory_order::acq_rel,
                memory_scope::device,
                access::address_space::global_space>
            atomic_counter(counter);

            // Signal arrival at the barrier.
            // Previous writes should be visible to all work-items
            // on the device.
            atomic_counter++;

            // Wait for all work-groups to arrive.
            // Synchronize with previous releases by all
            // work-items on the device.
            while (atomic_counter.load() != expected) {
            }
        }
        group_barrier(grp);
    }

    size_t counter;
    size_t expected;
};
```

图 19.18: 在原子引用之上构建一个简单的器件范围锁存器

```

// Allocate a one-time-use device_latch in USM
void* ptr = sycl::malloc_shared(sizeof(device_latch), q);
device_latch* latch = new (ptr) device_latch(num_groups);
q.submit([&](handler& h) {
    h.parallel_for(R, [=](nd_item<1> it) {
        // Every work-item writes a 1 to its location
        data[it.get_global_linear_id()] = 1;

        // Every work-item waits for all writes
        latch->arrive_and_wait(it);

        // Every work-item sums the values it can see
        size_t sum = 0;
        for (int i = 0; i < num_groups * items_per_group;
             ++i) {
            sum += data[i];
        }
        sums[it.get_global_linear_id()] = sum;
    });
}).wait();
free(ptr, q);

```

图 19.19: 使用图 19-18 中的器件范围锁存器

图 19-18 显示了设备范围锁存器(一次性 Barrier)的简单实现, 图 19-19 显示了其用法的简单示例。每个 Work-Groups 选择一个 Work-Items 来发出该组到达锁存器的信号, 并使用朴素的自旋循环等待其他组的到达, 而其他 Work-Items 则使用 Work-Groups Barrier 等待选定的 Work-Items。正是这种自旋循环使得设备范围的同步不安全; 如果任何 Work-Groups 尚未开始执行或当前正在执行的 Work-Groups 未公平调度, 则代码可能会死锁。

注 82 在没有足够强的前向进度保证的情况下, 仅依靠内存顺序来实现同步原语可能会导致死锁!

为了使代码正确运行, 必须满足以下三个条件:

1. 原子操作必须使用至少与所示一样严格的内存顺序, 以保证生成正确的栅栏。
2. NRange 中每个 Work-Groups 的当选领导者必须独立于其他 Work-Groups 中的领导者取得进展, 以避免单个 Work-Items 在循环中旋转, 导致其他尚未增加计数器的 Work-Items 挨饿。

3. 设备必须能够同时执行 ND 范围内的所有 Work-Groups，并具有强大的前进进度保证，以确保 ND 范围内每个 Work-Groups 的当选领导者最终到达闩锁。

虽然此代码不保证可移植，但我们将其包含在这里是为了强调两个关键点：(1) SYCL 具有足够的表达能力，可以实现特定于设备的调整，有时会牺牲可移植性；(2) SYCL 已经包含实现高级同步例程所需的构建块，这些例程可能会包含在该语言的未来版本中。

19.4 概括

本章对内存模型和原子类进行了高级介绍。了解如何使用（以及如何不使用！）这些类是开发正确、可移植且高效的并行程序的关键。

内存模型是一个极其复杂的主题，我们这里的重点是建立编写实际应用程序的基础。如果需要更多信息，可以参考下面引用的一些专门针对内存模型的网站、书籍和讲座。

19.4.1 了解更多信息

A. Williams, 《C++ 并发实践：实用多线程》，Manning, 2012 年，
978-1933988771

H. Sutter, “原子 <> 武器：C++ 内存模型和现代硬件”，herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modernhardware/

H-J. Boehm, “Temporarily discourage memory_order_consume”，wg21.link/p0371

C++ 参考，“std::atomic”，en.cppreference.com/w/cpp/atomic/atomic

C++ 参考，“std::atomic_ref”，en.cppreference.com/w/cpp/atomic/atomic_ref

20 后端互操作性

在本章中，我们将了解后端互操作性，这是一种 SYCL 功能，可以将 SYCL 增量添加到已经使用其他数据并行技术或 API 的应用程序。

我们还将了解熟悉低级 API 的专家程序员如何使用后端互操作性来“窥视幕后”并直接使用 SYCL 程序中的底层数据并行 API。这可以在必要时直接访问特定于 API 的功能，同时保留 SYCL 的可移植性和易用性优势。

20.1 什么是后端互操作性？

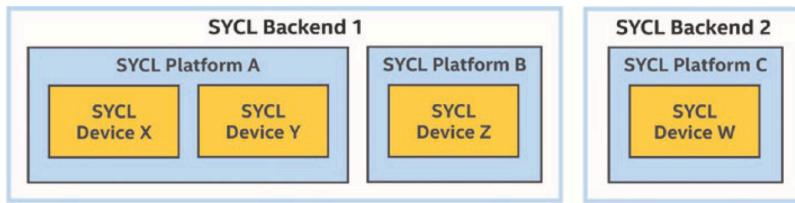


图 20.1: SYCL 后端、平台和设备之间的关系

到目前为止，在本书中，我们提到了在 SYCL 设备上运行的 SYCL 程序，但实际上，许多 SYCL 实现都基于较低级别的 API（例如 OpenCL、零级、CUDA 或其他 API）来访问系统中的并行硬件。当 SYCL 实现基于较低级别的 API 构建时，我们将目标 API 称为 SYCL 后端。图 20-1 显示了 SYCL 后端、平台和设备之间的关系。大多数 SYCL 实现可以同时在多个 SYCL 后端上运行 SYCL 程序，以利用系统中的所有并行硬件。

```

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

int main() {
    for (auto& p : platform::get_platforms()) {
        std::cout << "SYCL Platform: "
        << p.get_info<info::platform::name>()
        << " is associated with SYCL Backend: "
        << p.get_backend() << std::endl;
    }
    return 0;
}

Example Output:
SYCL Platform: Portable Computing Language is associated with SYCL Backend: opencl
SYCL Platform: Intel(R) OpenCL HD Graphics is associated with SYCL Backend: opencl
SYCL Platform: Intel(R) OpenCL is associated with SYCL Backend: opencl
SYCL Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM) is associated with SYCL
Backend: opencl
SYCL Platform: Intel(R) Level-Zero is associated with SYCL Backend:
ext_oneapi_level_zero
SYCL Platform: NVIDIA CUDA BACKEND is associated with SYCL Backend: ext_oneapi_cuda
SYCL Platform: AMD HIP BACKEND is associated with SYCL Backend: ext_oneapi_hip

```

图 20.2: 查询 SYCL 平台的 SYCL 后端

我们可以通过先查询 SYCL 平台，然后查询与每个平台关联的 SYCL 后端来查询系统中的 SYCL 后端，如图 20-2 所示。该程序的输出将取决于系统中 SYCL 设备的数量和类型。如果不同的 SYCL 后端支持同一设备，则它可能会为每个后端枚举为 SYCL 设备。

可以查询大多数 SYCL 对象的关联后端，而不仅仅是 SYCL 平台。例如，我们还可以查询关联后端的 SYCL 设备、SYCL 上下文或 SYCL 队列。

后端互操作性使我们能够使用关联后端的知识来与代表关联后端的 SYCL 对象的底层本机后端对象进行交互并进行操作。

20.2 后端互操作性何时有用？

许多 SYCL 程序员永远不需要使用后端互操作性。事实上，使用后端互操作性可能是不可取的；后端互操作性通常会导致程序变得更加复杂，因为它需要多个 SYCL 后端的多个代码路径，或者会降低程序的可移植性，因为它将限制执行到具有单个关联后端的设备。

尽管如此，后端互操作性仍然是我们工具箱中解决某些特定问题的有用工具。在本节中，我们将探讨后端互操作性有用的几个常见用例。

注 83 (后端互操作性就像一个内联汇编程序) 后端互操作性的一个有用心智模型是，后端互操作性之于 SYCL，就像内联汇编程序之于 C++ 主机代

码一样：后端互操作性对于学习 SYCL 或使用 SYCL 提高工作效率不是必需的，并且后端互操作性通常是不可取的，因为它会增加复杂性或降低可移植性。尽管如此，它是我们工具箱中解决特定问题的有用工具。

20.2.1 将 SYCL 添加到现有代码库

本书中的 SYCL 程序旨在教授特定的 SYCL 概念，因此它们故意简单明了且简短。相比之下，大多数现实世界的软件都是庞大而复杂的，由数千或数百万行代码组成，可能是由许多人多年来开发的。即使我们想这样做，完全重写大型应用程序以使用 SYCL 也可能不可行。

后端互操作性提供的主要优势之一是能够通过从该 API 的本机后端对象创建 SYCL 对象，将 SYCL 增量添加到已使用低级 API 的现有代码库。例如，假设我们有一个大型 OpenCL 应用程序，它创建 OpenCL 上下文和 OpenCL 内存对象。后端互操作性具有 make_context 和 make_buffer 等模板化函数，使我们可以从这些 OpenCL 对象无缝创建 SYCL 对象。从 OpenCL 对象创建 SYCL 对象后，它们可以被 SYCL 队列和 SYCL Kernel 使用，就像任何其他 SYCL 对象一样，如图 20-3 所示。

```
// Create SYCL objects from the native backend objects.
context c =
    make_context<backend::opencl>(openclContext);
device d = make_device<backend::opencl>(openclDevice);
buffer data_buf =
    make_buffer<backend::opencl, int>(openclBuffer, c);

// Now use the SYCL objects to create a queue and submit
// a kernel.
queue q{c, d};

q.submit([&](handler& h) {
    accessor data_acc{data_buf, h};
    h.parallel_for(size, [=](id<1> i) {
        data_acc[i] = data_acc[i] + 1;
    });
}).wait();
```

图 20.3: 从 OpenCL 对象创建 SYCL 对象

SYCL 2020 规范仅定义了与 OpenCL 后端的互操作性，但 SYCL 实

现可以通过扩展提供与其他后端的互操作性。图 20-4 显示了如何使用 `sycl_ext_oneyapi_backend_level_zero` 扩展从零级对象创建 SYCL 对象。

```
// Create SYCL objects from the native backend objects.
device d = make_device<backend::ext_oneyapi_level_zero>(
    level0Device);
context c =
    make_context<backend::ext_oneyapi_level_zero>(
        {level0Context,
         {d},
         ext::oneyapi::level_zero::ownership::keep});
buffer data_buf =
    make_buffer<backend::ext_oneyapi_level_zero, int>(
        {level0Ptr,
         ext::oneyapi::level_zero::ownership::keep},
        c);

// Now use the SYCL objects to create a queue and submit
// a kernel.
queue q{c, d};

q.submit([&](handler& h) {
    accessor data_acc{data_buf, h};
    h.parallel_for(size, [=](id<1> i) {
        data_acc[i] = data_acc[i] + 1;
    });
}).wait();
```

图 20.4: 从 *Level Zero* 对象创建 *SYCL* 对象

请注意，对于零级后端，创建 SYCL 对象时传递的参数略有不同。对于任何受支持的后端互操作性来说，这通常都是正确的，因为每个后端可能需要不同的信息来正确创建 SYCL 对象。否则，相同的 `make_device`、`make_context` 和 `make_buffer` 函数将用于 OpenCL 和零级后端互操作性。

另请注意，每个后端对所有权的处理方式不同。对于 OpenCL 后端，SYCL 实现使用 OpenCL 提供的引用计数来管理本机后端对象的生命周期。对于零级后端，必须明确告知 SYCL 实现是否应该获取本机后端对象的所有权，或者我们的应用程序是否将保留所有权。如果 SYCL 实现拥有本机后端对象的所有权，则当 SYCL 对象被销毁时，本机后端对象也将被销毁；否则，我们的应用程序负责直接释放本机后端对象。

20.2.2 将现有库与 SYCL 结合使用

后端互操作性还可用于从 SYCL 对象中提取本机后端对象。这对于在我们的 SYCL 应用程序中使用现有的低级库或其他辅助函数非常有用。有两种方法可以执行此操作：第一种方法使用 `get_native` 自由函数从 SYCL 对象获取本机后端对象。第二个使用 `host_task` 和 `interop_handle` 从 SYCL 运行时调度的代码中的 SYCL 对象获取本机后端对象。

```
cl_device_id openclDevice =
    get_native<backend::opencl>(d);
cl_context openclContext = get_native<backend::opencl>(c);

// Query the device name from OpenCL:
size_t sz = 0;
clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, 0, nullptr,
    &sz);
std::string openclDeviceName(sz, ' ');
clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, sz,
    &openclDeviceName[0], nullptr);
std::cout << "Device name from OpenCL is: "
    << openclDeviceName << "\n";

// Allocate some memory from OpenCL:
cl_mem openclBuffer = clCreateBuffer(
    openclContext, 0, sizeof(int), nullptr, nullptr);

// Clean up OpenCL objects when done:
clReleaseDevice(openclDevice);
clReleaseContext(openclContext);
clReleaseMemObject(openclBuffer);
```

图 20.5: 使用 `get_native` 自由函数从 SYCL 对象中提取 OpenCL 对象

使用自由函数获取后端对象 例如，假设我们有一个优化的 OpenCL 库，我们希望将其与 SYCL 应用程序一起使用。我们可以调用后端互操作性 `get_native` 函数从 SYCL 对象中获取本机 OpenCL 对象，然后可以将其与 OpenCL 库一起使用。为简单起见，图 20-5 中的代码仅执行查询并使用本机 OpenCL 对象分配一些内存，但它们也可以用于执行更复杂的操作，例如创建命令队列、编译程序和执行 Kernel。

```
ze_device_handle_t level0Device =
    get_native<backend::ext_oneapi_level_zero>(d);
ze_context_handle_t level0Context =
    get_native<backend::ext_oneapi_level_zero>(c);

// Query the device name from Level Zero:
ze_device_properties_t level0DeviceProps = {};
level0DeviceProps.sotype =
    ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;

zeDeviceGetProperties(level0Device, &level0DeviceProps);

std::cout << "Device name from SYCL is: "
    << d.get_info<info::device::name>() << "\n";
std::cout << "Device name from Level Zero is: "
    << level0DeviceProps.name << "\n";

// Allocate some memory from Level Zero:
void* level0Ptr = nullptr;
ze_host_mem_alloc_desc_t level0HostAllocDesc = {};
level0HostAllocDesc.sotype =
    ZE_STRUCTURE_TYPE_HOST_MEM_ALLOC_DESC;
zeMemAllocHost(level0Context, &level0HostAllocDesc,
    sizeof(int), 0, &level0Ptr);

// Clean up Level Zero objects when done:
zeMemFree(level0Context, level0Ptr);
```

图 20.6: 使用 `get_native` 自由函数从 SYCL 对象中提取 Level Zero 对象

作为 `sycl_ext_oneapi_backend_level_zero` 扩展的一部分，还为零级后端添加了相同的 `get_native` 函数，如图 20-6 所示。

通过互操作句柄获取后端对象 使用 `get_native` 自由函数是获取直接使用后端 API 的大段代码的后端特定对象的有效方法。但在许多情况下，我们只想使用后端 API 在 SYCL 任务图中执行特定操作。在这些情况下，我们可以使用带有特殊 `interop_handle` 参数的 SYCL `host_task` 来执行特定于后端的操作。`interop_handle` 表示调用主机任务时 SYCL 运行时的状态，并提供对表示 SYCL 队列、设备、上下文以及为主机任务捕获的任何缓冲区的本机后端对象的访问。

```
q.submit([&](handler& h) {
    accessor a{b, h};
    h.host_task([=](interop_handle ih) {
        // Get the OpenCL device from the interop handle:
        auto openclDevice =
            ih.get_native_device<backend::opencl>();

        // Query the device name from the OpenCL device:
        size_t sz = 0;
        clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, 0,
                        nullptr, &sz);
        std::string openclDeviceName(sz, ' ');
        clGetDeviceInfo(openclDevice, CL_DEVICE_NAME, sz,
                        &openclDeviceName[0], nullptr);
        std::cout << "Device name from OpenCL is: "
                << openclDeviceName << "\n";

        // Get the OpenCL buffer from the interop handle:
        auto openclMem =
            ih.get_native_mem<backend::opencl>(a)[0];

        // Query the size of the OpenCL buffer:
        clGetMemObjectInfo(openclMem, CL_MEM_SIZE, sizeof(sz),
                            &sz, nullptr);
        std::cout << "Buffer size from OpenCL is: " << sz
                << " bytes\n";
    });
});
```

图 20.7: 使用 `interop_handle` 从 SYCL 对象中提取 OpenCL 对象

图 20-7 显示了如何使用 `interop_handle` 从 SYCL 运行时调度的 `host_task` 获取本机 OpenCL 对象。为简单起见，此示例也仅使用本机 OpenCL 对象执行一些查询，但实际应用程序代码通常会使用本机 OpenCL 对象对 Kernel 进行排队或调用库。由于这些操作是从主机任务执行的，因此它们将与 SYCL 队列中的任何其他操作一起正确调度。

请注意，当我们的访问器获取本机 OpenCL 对象时，`interop_handle` 的 `get_native_mem` 成员函数返回 `cl_mem` 内存对象的向量。这是 SYCL 2020 规范中的要求，其中 `interop_handle` 的成员函数的返回类型必须与 `get_native` 自由函数匹配，但对于 `interop_handle` 用法，我们可以简单地使用向量的第一个元素。

```
q.submit([&](handler& h) {
    accessor a{b, h};
    h.host_task([=](interop_handle ih) {
        // Get the Level Zero device from the interop handle:
        auto level0Device = ih.get_native_device<
            backend::ext_oneapi_level_zero>();

        // Query the device name from Level Zero:
        ze_device_properties_t level0DeviceProps = {};
        level0DeviceProps.stype =
            ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES;
        zeDeviceGetProperties(level0Device,
            &level0DeviceProps);
        std::cout << "Device name from Level Zero is: "
            << level0DeviceProps.name << "\n";

        // Get the Level Zero context and memory allocation
        // from the interop handle:
        auto level0Context = ih.get_native_context<
            backend::ext_oneapi_level_zero>();
        auto ptr =
            ih.get_native_mem<backend::ext_oneapi_level_zero>(
                a);

        // Query the size of the memory allocation:
        size_t sz = 0;
        zeMemGetAddressRange(level0Context, ptr, nullptr,
            &sz);
        std::cout << "Buffer size from Level Zero is: " << sz
            << " bytes\n";
    });
});
```

图 20.8: 使用 `interop_handle` 从 SYCL 对象中提取 OpenCL 对象

与 `get_native` 免费函数一样，也可以通过扩展为其他 SYCL 后端提供类似的功能。图 20.8 显示了如何使用 `sycl_ext_oneapi_backend_level_zero` 扩展对零级后端执行类似的操作。

20.3 使用 Kernel 的后端互操作性

本节介绍如何使用后端互操作性来编译 Kernel 和操作 Kernel 包。这是 SYCL 2020 中经过重大重新设计的区域，以提高稳健性并增加支持不同 SYCL 后端所需的灵活性。

早期版本的 SYCL 支持两种 Kernel 互操作机制。第一种机制允许从 API 定义的句柄创建 Kernel。第二个支持从 API 定义的源或中间表示（例如 OpenCL C 源或 SPIR-V 中间表示）创建 Kernel。这两种机制在 SYCL 2020 中仍然存在，尽管这两种机制的语法已更新并且现在使用后端互操作性。

20.3.1 与 API 定义的 Kernel 对象的互操作性

通过这种形式的互操作性，Kernel 对象本身是使用低级 API 创建的，然后使用后端互操作性导入到 SYCL 中。图 20-9 中的代码显示了如何从 SYCL 上下文获取 OpenCL 上下文，如何使用此 OpenCL 上下文创建 OpenCL Kernel，以及如何从 OpenCL Kernel 对象创建和使用 SYCL Kernel。

```

// Get the native OpenCL context from the SYCL context:
auto openclContext = get_native<backend::opencl>(c);
const char* kernelSource =
R"CLC(
    kernel void add(global int* data) {
        int index = get_global_id(0);
        data[index] = data[index] + 1;
    }
)CLC";
// Create an OpenCL kernel using this context:
cl_program p = clCreateProgramWithSource(
    openclContext, 1, &kernelSource, nullptr, nullptr);
clBuildProgram(p, 0, nullptr, nullptr, nullptr,
    nullptr);
cl_kernel k = clCreateKernel(p, "add", nullptr);

// Create a SYCL kernel from the OpenCL kernel:
auto sk = make_kernel<backend::opencl>(k, c);

// Use the OpenCL kernel with a SYCL queue:
q.submit([&](handler& h) {
    accessor data_acc{data_buf, h};

    h.set_arg(data_acc);
    h.parallel_for(size, sk);
});

// Clean up OpenCL objects when done:
clReleaseContext(openclContext);
clReleaseProgram(p);
clReleaseKernel(k);

```

图 20.9: 从 *OpenCL Kernel* 对象创建的 *Kernel*

由于 SYCL 编译器无法直接查看使用低级 API 创建的 SYCL Kernel，因此必须使用 `set_arg()` 或 `set_args()` 接口显式传递任何 Kernel 参数。此外，SYCL 运行时和低级 API Kernel 必须就将对象作为 Kernel 参数传递的约定达成一致。该约定应被描述为后端互操作性规范的一部分。在此示例中，访问器 `data_acc` 作为全局指针 Kernel 参数数据传递。

注 84 *SYCL 2020* 标准将 `set_arg()` 和 `set_args()` 接口的精确语义留给每个 *SYCL* 后端规范定义。这允许灵活性，但这是我们编写的使用后端互操作性的代码可能特定于我们目标的后端的另一种方式。

20.3.2 与非 SYCL 源语言的互操作性

通过这种形式的互操作性, Kernel 的内容被描述为源代码或未由 SYCL 定义的中间表示形式。这种形式的互操作性允许重用以其他源语言编写的 Kernel 库或使用以中间表示形式生成代码的特定于域的语言 (DSL)。

SYCL 的早期版本包含诸如 `build_with_source` 之类的函数, 可直接从 API 定义的源语言创建 SYCL 程序, 但此功能在 SYCL 2020 中被删除。当后端直接支持 API 定义的源语言时, 例如使用的 OpenCL C Kernel 对于图 20-9 中的 OpenCL 后端, 这种删除不是问题, 但是如果后端不直接支持特定的源语言, 我们该怎么办?

```

// Compile OpenCL C kernel source to SPIR-V intermediate
// representation using the online compiler:
const char* kernelSource =
R"CLC(
    kernel void add(global int* data) {
        int index = get_global_id(0);
        data[index] = data[index] + 1;
    }
)CLC";
online_compiler<source_language::opencl_c> compiler(d);
std::vector<byte> spirv =
    compiler.compile(kernelSource);

// Get the native Level Zero context and device:
auto level0Context =
    get_native<backend::ext_oneapi_level_zero>(c);
auto level0Device =
    get_native<backend::ext_oneapi_level_zero>(d);

// Create a Level Zero kernel using this context:
ze_module_handle_t level0Module = nullptr;
ze_module_desc_t moduleDesc = {};
moduleDesc.stype = ZE_STRUCTURE_TYPE_MODULE_DESC;
moduleDesc.format = ZE_MODULE_FORMAT_IL_SPIRV;
moduleDesc.inputSize = spirv.size();
moduleDesc.pInputModule = spirv.data();
zeModuleCreate(level0Context, level0Device, &moduleDesc,
    &level0Module, nullptr);

ze_kernel_handle_t level0Kernel = nullptr;
ze_kernel_desc_t kernelDesc = {};
kernelDesc.stype = ZE_STRUCTURE_TYPE_KERNEL_DESC;
kernelDesc.pKernelName = "add";
zeKernelCreate(level0Module, &kernelDesc,
    &level0Kernel);

// Create a SYCL kernel from the Level Zero kernel:
auto skb =
    make_kernel_bundle<backend::ext_oneapi_level_zero,
        bundle_state::executable>(
        {level0Module}, c);
auto sk = make_kernel<backend::ext_oneapi_level_zero>(
    {skb, level0Kernel}, c);

// Use the Level Zero kernel with a SYCL queue:
q.submit([&](handler& h) {
    accessor data_acc{data_buf, h};

    h.set_args(data_acc);
    h.parallel_for(size, sk);
});
```

图 20.10: 使用 *SPIR-V* 和在线编译器创建的 *Kernel*

一些 SYCL 实现可以提供显式在线编译器来从后端不能直接使用的源语言编译为后端支持的不同格式。图 20-10 显示了如何使用实验性 `sycl_ext_intel_online_compiler` 扩展从零级后端不支持的 OpenCL C 源代码编译为零级后端支持的 SPIR-V 中间表示形式。使用这种方法，Kernel 可以被任何后端使用，只要它可以被在线编译器编译成后端支持的格式。

注 85 (注意，实验性扩展！) `sycl_ext_intel_online_compiler` 扩展是一个实验性扩展，因此可能会更改或删除！我们之所以将它包含在本书中，是因为它提供了一种实现与以前的 `SYCL build_with_source` 函数类似的功能的方法，并且因为它是演示特定领域语言如何与 `SYCL` 后端交互以执行 `Kernel` 的便捷方法。

在此示例中，`Kernel` 源字符串在与 `SYCL` 主机 API 调用相同的文件中表示为 C++ 原始字符串文字，但不要求是这种情况，并且某些应用程序可能会从文件中读取 `Kernel` 源字符串甚至及时生成它。

和以前一样，由于 `SYCL` 编译器无法查看以 API 定义的源语言编写的 `SYCL Kernel`，因此必须使用 `set_arg()` 或 `set_args()` 接口显式传递任何 `Kernel` 参数。

20.4 后端互操作性提示和技巧

本节介绍有效使用后端互操作性的实用提示和技巧。

20.4.1 为特定后端选择设备

正确使用后端互操作性的第一个要求是选择与所需 `SYCL` 后端关联的 `SYCL` 设备。有几种方法可以实现这一点。

第一个是通过在对每个设备评分时查询关联的后端，将所需的 `SYCL` 后端集成到现有的自定义设备选择逻辑中。如果我们的应用程序已经在使用自定义设备选择逻辑，那么这应该是一个简单的添加。该机制也是可移植的，因为它仅使用标准 `SYCL` 查询。

```

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

int main() {
    auto find_device = [] (backend b,
                           info::device_type t =
                           info::device_type::all) {
        for (auto d : device::get_devices(t)) {
            if (d.get_backend() == b) {
                return d;
            }
        }
        throw sycl::exception(errc::runtime,
                            "Could not find a device with "
                            "the requested backend!");
    };

    try {
        device d{find_device(backend::opencl)};
        std::cout << "Found an OpenCL SYCL device: "
                  << d.get_info<info::device::name>() << "\n";
    } catch (const sycl::exception &e) {
        std::cout << "No OpenCL SYCL devices were found.\n";
    }

    try {
        device d{find_device(backend::ext_oneapi_level_zero)};
        std::cout << "Found a Level Zero SYCL device: "
                  << d.get_info<info::device::name>() << "\n";
    } catch (const sycl::exception &e) {
        std::cout << "No Level Zero SYCL devices were found.\n";
    }

    return 0;
}

Example Output:
Found an OpenCL SYCL device: pthread-12th Gen Intel(R) Core(TM) i9-12900K
Found a Level Zero SYCL device: Intel(R) UHD Graphics 770 [0x4680]

```

图 20.11: 查找具有特定后端的 SYCL 设备

对于尚未使用自定义设备选择逻辑的应用程序，我们可以编写一个简短的 C++ lambda 表达式来迭代所有设备，以查找具有所请求后端的设备，如图 20-11 所示。由于此版本的 `find_device` 不请求特定的设备类型，因此它实际上是标准 `default_selector_v` 的替代品。

最后，为了快速原型化，一些 SYCL 实现可以使用外部机制（例如环境变量）来影响它们枚举的 SYCL 设备。例如，DPC++ SYCL 运行时可以使用 `ONEAPI_DEVICE_SELECTOR` 环境变量将枚举设备限制为特定设备类型或关联的设备后端（请参阅第 13 章）。对于生产代码来说，这不是一个理想的解决方案，因为它需要外部配置，但对于原型代码来说，它是一种有用的机制，可以确保应用程序使用来自特定后端的特定设备。

20.4.2 小心上下文!

回想一下第 6 章和第 13 章，许多 SYCL 对象（例如 Kernel 和 USM 分配）如果是在不同的 SYCL 上下文中创建的，通常无法通过 SYCL 上下文访问。使用后端互操作性时仍然如此；因此，使用后端 API 创建的特定于后端的上下文通常无法访问在不同 SYCL 上下文中创建的对象（反之亦然），即使 SYCL 上下文与同一后端关联也是如此。

为了在 SYCL 和后端之间安全地共享对象，我们应该始终使用 `make_context` 从本机后端上下文创建 SYCL 上下文，或者应该使用 `get_native` 从 SYCL 上下文获取本机后端上下文。

注 86 始终从本机后端上下文创建 SYCL 上下文或从 SYCL 上下文获取本机后端上下文，以便在 SYCL 和后端之间安全地共享对象！

20.4.3 访问低级 API 特性

有时，尖端功能会先在低级 API 中提供，然后再在 SYCL 中提供，甚至作为 SYCL 扩展。有些功能甚至可能是特定于后端或特定于设备的，以至于它们永远不会通过 SYCL 公开。例如，某些本机后端 API 可以提供对具有特定属性的队列或特定加速器硬件的独特 Kernel 指令的访问。尽管我们希望并期望这些情况很少见，但当这些类型的功能存在时，我们仍然可以使用后端互操作性来访问它们。

20.4.4 对其他后端的支持

本章中的示例演示了后端与 OpenCL 和零级后端的互操作性，但 SYCL 是一个不断发展的生态系统，SYCL 实现定期添加对其他后端和设备的支持。例如，支持 CUDA 和 HIP 后端的多个 SYCL 实现已经对与这些后端的互操作性提供了一些支持。检查 SYCL 实现的文档，以确定支持哪些 SYCL 后端以及它们是否支持后端互操作性！

20.5 总结

在本章中，我们了解了每个 SYCL 对象如何与底层 SYCL 后端关联以及如何查询系统中的 SYCL 后端。我们描述了后端互操作性如何为我们的 SYCL 应用程序提供直接与底层后端 API 交互的机制。我们讨论了这如何

使我们能够将 SYCL 增量添加到直接使用后端 API 的应用程序，或重用专门为后端 API 编写的库或实用程序函数。我们还讨论了后端互操作性如何通过限制应用程序将在哪些 SYCL 设备上运行来降低应用程序的可移植性。

我们专门探讨了 Kernel 的后端互操作性如何在 SYCL 2020 中提供早期版本 SYCL 中存在的类似功能。我们研究了在线编译器扩展如何启用 Kernel 的某些源语言，即使某些 SYCL 后端不能直接理解它们。

最后，我们回顾了在程序中有效使用后端互操作性的实用提示和技巧，例如如何为特定 SYCL 后端选择 SYCL 设备、如何为后端互操作性设置 SYCL 上下文以及后端互操作性如何提供对功能，即使它们尚未添加到 SYCL 中。

21 迁移 CUDA 代码

本书的许多读者可能遇到过用 CUDA 编写的数据并行代码。有些读者甚至可能是 CUDA 专家！在本章中，我们将描述 CUDA 和 SYCL 之间的一些相似之处、一些差异，以及帮助使用 SYCL 将 CUDA 代码有效且高效地迁移到 C++ 的有用工具和技术。

21.1 CUDA 和 SYCL 之间的设计差异

在我们深入了解细节之前，首先确定 CUDA 和 SYCL 之间的关键设计差异具有指导意义。这可以提供有用的背景来说明为什么存在一些差异，了解哪些差异可能会随着时间的推移而消失以及哪些差异可能会保留。

21.1.1 多个目标与单个设备目标

CUDA 和 SYCL 之间最大的设计差异之一是它们设计支持的设备范围。CUDA 旨在支持来自单一设备供应商的 GPU 设备，因此大多数 CUDA 设备看起来相对相似。例如，所有 CUDA 设备当前都包含纹理采样硬件，并且所有 CUDA 设备当前都支持相同的最大 Work-Groups 大小。这降低了复杂性，但也减少了 CUDA 应用程序可以运行的位置。

相比之下，SYCL 旨在支持多种异构加速器，包括来自不同设备供应商的不同设备。这种灵活性使 SYCL 程序可以自由地利用现代异构系统中的计算资源；然而，这种灵活性的代价并不大。例如，作为 SYCL 程序员，我们可能需要枚举系统中的设备，检查它们的属性，并选择最适合运行程序的不同部分的设备。

当然，如果我们的 SYCL 程序不打算利用系统中的所有计算资源，则可以使用各种快捷方式来减少代码冗长，例如标准设备选择器。图 21-1 显示了一个基本 SYCL 示例，该示例使用由 SYCL 实现选择的默认设备队列。

```
// Declare an in-order SYCL queue for the default device
queue q{property::queue::in_order()};
std::cout << "Running on device: "
<< q.get_device().get_info<info::device::name>()
<< "\n";

int* buffer = malloc_host<int>(count, q);
q.fill(buffer, 0, count);

q.parallel_for(count, [=](auto id) {
    buffer[id] = id;
}).wait();
```

图 21.1: 在默认 SYCL 设备上运行 Kernel

此 SYCL 代码与等效的 CUDA 代码非常相似，如图 21-2 所示。

```
// The CUDA kernel is a separate function
__global__ void TestKernel(int* dst) {
    auto id = blockIdx.x * blockDim.x + threadIdx.x;
    dst[id] = id;
}

int main() {
    // CUDA uses device zero by default
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);
    std::cout << "Running on device: " << deviceProp.name << "\n";

    int* buffer = nullptr;
    cudaMallocHost(&buffer, count * sizeof(int));
    cudaMemset(buffer, 0, count * sizeof(int));

    TestKernel<<<count / 256, 256>>>(buffer);
    cudaDeviceSynchronize();
    // ...
```

图 21.2: 在默认 CUDA 设备上运行 Kernel

现实世界的 SYCL 代码通常更复杂。例如，许多 SYCL 应用程序将通过搜索特定设备特征（请参阅第 12 章）来枚举并选择要运行的特定设备或设备组合（请参阅第 2 章）。然而，当不需要或不需要这种复杂性时，存在简洁的选项，并且 SYCL 经过精心设计，可以在需要时支持额外的复杂性。

21.1.2 对齐 C++ 与扩展 C++

CUDA 和 SYCL 之间的另一个重要设计差异是它们如何与其他编程语言（尤其是 C++）交互。SYCL 代码是标准 C++ 代码，没有任何语言扩展。通过学习阅读、理解和编写 C++ 代码，我们也能阅读和理解 SYCL 代码。同样，如果编译器可以解析 C++ 代码，它也可以解析 SYCL 代码。

CUDA 做出了不同的决定。相反，CUDA 通过添加新关键字和特殊语法来执行 Kernel 来扩展 C++。有时，语言扩展可以更简洁，但它们也是一种需要学习和记住的语法，并且语言扩展意味着 CUDA 代码只能由支持 CUDA 的编译器编译。

要在实践中看到这种设计差异，请注意图 21-1 中的 SYCL 示例如何使用标准 C++ lambda 表达式来表示 Kernel 代码，并使用标准 C++ 函数调用提交 Kernel 执行。图 21-2 中的 CUDA 示例使用特殊的 `__global__` 关键字来标识 Kernel 代码，并使用特殊的 `<<<>>` 语法来提交 Kernel 执行。

21.2 CUDA 和 SYCL 之间的术语差异

现在我们了解了 SYCL 和 CUDA 之间的一些关键设计差异，我们几乎准备好开始检查具体的相似点和差异。不过，我们首先需要了解一点背景知识：因为 CUDA 和 SYCL 经常对相似的概念使用不同的术语，所以我们需要一个解码器，以便我们可以有意义地比较这两个 API，如图 21-3 中的摘要所示。

Concept	SYCL Term	CUDA Term
A function that is executed in parallel on a device.	Kernel	Kernel
The N-dimensional parallel index space.	Range (generally), or ND-Range (with grouping)	Grid (always has grouping)
A kernel instance executing at a point in the parallel index space.	Work-Item	Thread
An application-defined group of kernel instances in the parallel index space that can communicate and synchronize.	Work-Group	Block
An implementation-defined group of kernel instances with additional communication and synchronization capabilities.	Sub-Group	Warp
Memory used to exchange data among instances in a group.	Local Memory	Shared Memory
Function used to synchronize instances in a group.	group_barrier()	<code>__syncthreads()</code> , <code>__syncwarp()</code> , <code>coop_group.sync()</code>
Object used to execute kernels or other work on a device.	Queue	Stream

图 21.3: CUDA 和 SYCL 解码器环

与本书其余部分一致使用 SYCL 术语不同，本章可以互换使用 CUDA 术语和 SYCL 术语。

21.3 共同点和不同点

本节介绍 SYCL 和 CUDA 之间的一些语法和行为相似之处以及 SYCL 和 CUDA 的不同之处。

21.3.1 执行模型

从根本上来说，SYCL 和 CUDA 都使用第 4 章中介绍并在本书中描述的相同数据并行 Kernel 执行模型。术语可能略有不同，例如，SYCL 指的是 ND 范围，CUDA 指的是网格，但我们可以使用图 21-3 中的解码器环将关键概念从 SYCL 转换为 CUDA，反之亦然。

有序队列与无序队列 尽管执行模型有许多相似之处，但确实存在一些差异。一个区别是 CUDA 流是无条件有序的。这意味着提交到 CUDA 流的任何 Kernel 或内存操作必须在下一个提交的 Kernel 或内存复制操作开始之前完成。相反，SYCL 队列默认是无序的，但可以选择在创建 SYCL 队列时通过传递 `in_order` 队列属性来按顺序排列（请参阅第 8 章）。

有序 CUDA 流更简单，因为它不需要显式调度或依赖性管理。这种简单性意味着 CUDA 应用程序通常不使用访问器或 `dependent_on` 等机制来对流中的操作进行排序。不过，有序语义也限制了执行，并且不提供任何在单个流中重叠执行两个命令的机会。由于 CUDA 应用程序不能重叠执行单个流中的两个命令，因此当 CUDA 应用程序想要（可能）同时执行命令时，它将把命令提交到不同的 CUDA 流，因为不同 CUDA 流中的命令可能会同时执行。

这种提交到多个有序队列以同时执行 Kernel 或内存操作的相同模式也适用于 SYCL，并且许多 SYCL 实现和 SYCL 设备都经过优化来处理这种情况。不过，无序 SYCL 队列提供了一种替代机制，可以仅与单个队列重叠执行，并且许多 SYCL 实现和 SYCL 设备也经过优化以处理这种情况。

最终，是否使用多个有序 SYCL 队列或更少的无序 SYCL 队列取决于个人喜好和编程风格，我们可以选择对我们的 SYCL 程序最有意义的选项。本章中的 SYCL 示例创建有序 SYCL 队列，以尽可能接近等效的 CUDA 示例。

连续维度 可能让新手和专家 CUDA 程序员感到困惑的另一个区别涉及多维 SYCL 范围或 CUDA 网格：SYCL 将其约定与标准 C++ 中的多维数组对齐，因此最后一个维度是连续维度，也称为单位跨度维度或移动最快的维度。相反，CUDA 与图形约定保持一致，因此第一个维度是连续维度。由于这种差异，与等效的 CUDA 代码相比，多维 SYCL 范围将出现转置，并且 SYCL `id` 的最高维度将对应于可比较 CUDA 内置变量的 `x` 分量，而不是最低维度。

```

__global__ void ExchangeKernel(int* dst) {
    auto index = get_global_linear_id(); // helper function
    auto fastest = threadIdx.x;
    auto neighbor = __shfl_xor_sync(0xFFFFFFFF, fastest, 1);
    dst[index] = neighbor;
}
...
dim3 threadsPerBlock(16, 2);
ExchangeKernel<<<1, threadsPerBlock>>>(buffer);
cudaDeviceSynchronize();

```

图 21.4: *x-component* 是 CUDA 中的连续维度

为了演示这种差异, 请考虑图 21-4 中的 CUDA 示例。在此示例中, 每个 CUDA 线程与其邻居交换其 threadIdx.x 值。由于 x 组件是 CUDA 中移动最快的组件, 因此我们不期望 CUDA 线程的值与其相邻线程的值匹配。

```

q.parallel_for(nd_range<2>{{2, 16}, {2, 16}},
              [=](auto item) {
                  auto index = item.get_global_linear_id();
                  auto fastest = item.get_local_id(1);
                  auto sg = item.get_sub_group();
                  auto neighbor =
                      permute_group_by_xor(sg, fastest, 1);
                  buffer[index] = neighbor;
              })
.wait();

```

图 21.5: 最后一个维度是 SYCL 中的连续维度

等效的 SYCL 示例如图 21-5 所示。请注意, 在 SYCL 示例中, ND 范围是 {2, 16}, 而不是 CUDA 示例中的 (16, 2), 因此并行索引空间似乎已转置。SYCL 示例还将 ND 范围描述为 {2, 16} 全局范围, 分为大小为 {2, 16} 的 Work-Groups, 而 CUDA 示例描述了一个块的网格, 每个块有 (16, 2) 个 CUDA 线程。

此外, 请注意, 每个 SYCL Work-Items 都会与其邻居交换其 item.get_local_id(1) (不是 item.get_local_id(0)!) 的值, 因为最后一个维度是 SYCL 中移动最快的组件。在此 SYCL 示例中, 我们也不期望 SYCL Work-Items 的值与其相邻 Work-Items 的值相匹配。

Sub-Groups 尺寸 (变形尺寸) 如果仔细查看这些示例，我们还可以发现更多差异，特别是与用于与邻居交换数据的函数相关的差异。

CUDA 示例使用函数 `__shfl_xor_sync(0xFFFFFFFF,fastest,1)` 与邻居交换数据。对于此函数，第一个参数 `0xFFFFFFFF` 是一个位域掩码，指示参与调用的 CUDA 线程集。对于 CUDA 设备，32 位掩码就足够了，因为当前所有 CUDA 设备的扭曲大小均为 32。

SYCL 示例使用函数 `permute_group_by_xor(sg,fastest,1)` 与其邻居交换数据。对于此函数，第一个参数描述参与调用的 Work-Items 集。在这种情况下，`sg` 代表整个 Sub-Groups。由于 Work-Items 集是由组对象而不是位域掩码指定的，因此它可以表示任意大小的集。这种灵活性是可取的，因为对于某些 SYCL 设备，Sub-Groups 大小可能小于或大于 32。

```
__global__ void ExchangeKernelCoopGroups(int* dst) {
    namespace cg = cooperative_groups;
    auto index = cg::this_grid().thread_rank();
    auto fastest = threadIdx.x;
    auto warp = cg::tiled_partition<32>(cg::this_thread_block());
    auto neighbor = warp.shfl_xor(fastest, 1);
    dst[index] = neighbor;
}
```

图 21.6: 与 CUDA 合作小组交换数据

在这种特定情况下，可以重写 CUDA 示例以使用更现代的 CUDA 协作组语法，而不是旧的 `__shfl_xor_sync` 语法。等效的 CUDA 协作组如图 21-6 所示。该版本看起来更像 SYCL Kernel，是 CUDA 和 SYCL 2020 的后续版本如何变得更加紧密的一个很好的例子。

前进进度保证 如果我们仔细观察图 21-4 和 21-5 中的示例，我们会发现另一个差异，尽管这种差异更为微妙。同样，差异与用于与邻居交换数据的 `__shfl_xor_sync` 函数有关，在这种情况下，差异由函数上的 `_sync` 后缀暗示。`_sync` 后缀表示该函数正在同步 CUDA 线程，尽管这自然会导致我们问，为什么在调用该函数之前 CUDA 线程可能首先不同步？

在第 15 章和第 16 章中，我们为在 CPU 或 GPU 上执行的数据并行 Kernel 开发了一个心智模型，其中使用 SIMD 指令同步处理一组 Work-Items。虽然这对于许多供应商的 CPU 和 GPU 来说是一个有用的心智模型，但它并不是使用 SYCL 或 CUDA 执行数据并行 Kernel 的唯一方法，并

且这种心智模型崩溃的情况之一是对于支持的较新 CUDA 设备称为独立线程调度的功能。

对于具有独立线程调度的 CUDA 设备，各个 CUDA 线程独立进行，而不是作为一个组进行。这些额外的前向进度保证使代码模式能够在 CUDA 设备上安全执行，而如果没有更强的前向进度保证，这些代码模式可能无法在 SYCL 设备上正确执行。CUDA 中添加了 `__shfl_xor_sync` 函数上的 `_sync` 后缀，以明确指示该函数需要同步并指定使用 32 位掩码进行同步的 CUDA 线程。

前向进度保证是 SYCL 社区中的一个活跃主题，并且 SYCL 的未来版本很可能会添加查询以确定设备的前向进度功能，以及用于指定 Kernel 的前向进度要求的属性。不过，目前我们应该意识到，由于独立的线程调度，从 CUDA 移植的语法正确的 SYCL 程序可能无法在所有 SYCL 设备上正确执行。

Barrier 我们应该注意的最后一个微妙的执行模型差异涉及 CUDA 的 `__syncthreads` 函数与 SYCL `group_barrier` 等效函数的比较。CUDA 的 `__syncthreads` 函数同步线程块中的所有未退出的 CUDA 线程，而 SYCL `group_barrier` 函数同步 Work-Groups 中的所有 Work-Items。这意味着，如果某些 CUDA 线程在调用 `__syncthreads` 之前提前退出，则 CUDA Kernel 将正确运行，但不能保证如图 21-7 所示的 SYCL Kernel 将正确运行。

```
std::cout << "WARNING: May deadlock on some devices!\n";
q.parallel_for(nd_range<1>{64, 64}, [=](auto item) {
    int id = item.get_global_id(0);
    if (id >= count) {
        return; // early exit
    }
    group_barrier(item.get_group());
    buffer[id] = id;
}).wait();
```

图 21.7: 可能的 SYCL 屏障死锁

在这种情况下，解决方法很简单：可以将范围检查移到 `group_barrier` 之后，或者在这种特定情况下，可以完全删除 `group_barrier`。但情况并非总是如此，其他 Kernel 可能需要重组以确保所有 Work-Items 始终到达或

始终跳过 group_barrier。

21.3.2 内存模型

从根本上来说，CUDA 和 SYCL 都使用类似的弱有序内存模型。幸运的是，当我们将 CUDA Kernel 迁移到 SYCL 时，我们只需要记住一些内存模型差异。

Barrier 默认情况下，假设传递给 SYCL group_barrier 的组是 Work-Groups，则 CUDA __syncthreads Barrier 函数和 SYCL group_barrier Barrier 函数对内存模型具有相同的效果。同样，假设传递给 SYCL group_barrier 的组是 Sub-Groups，则 CUDA __syncwarp Barrier 函数与 SYCL group_barrier Barrier 函数具有相同的效果。

SYCL group_barrier 接受一个可选参数来指定 Barrier 的 fence_scope，但在大多数情况下，可以省略。可以将更广泛的范围传递给 group_barrier，例如 memory_scope::device，但这通常不是必需的，并且可能会导致 SYCL group_barrier 比 CUDA __syncthreads Barrier 更昂贵。

```
q.parallel_for(nd_range<1>{16, 16}, [=](auto item) {
    // Equivalent of __syncthreads, or
    // this_thread_block().sync():
    group_barrier(item.get_group());

    // Equivalent of __syncwarp, or
    // tiled_partition<32>(this_thread_block()).sync():
    group_barrier(item.get_sub_group());
}).wait();
```

图 21.8: CUDA 和 SYCL 的等效 Barrier

图 21-8 中的代码显示了 CUDA 和 SYCL 的等效 Barrier 语法。请注意，使用 this_thread_block 和 tiled_partition 的较新 CUDA 协作组语法如何具有更接近 SYCL group_barrier 的同步函数。这是 CUDA 和 SYCL 2020 的后续版本变得越来越相似的另一个好例子。

原子和栅栏 CUDA 和 SYCL 都支持类似的原子操作，尽管与 Barrier 一样，我们应该注意一些重要的差异。最重要的区别涉及默认的原子内存顺

序。许多 CUDA 程序都是使用较旧的类似 C 的原子语法编写的，其中原子函数采用指向内存的指针，例如 atomicAdd。这些原子函数是宽松的原子函数，并且在设备范围内运行。这些原子函数还有在不同作用域操作的后缀版本，例如 atomicAdd_system 和 atomicAdd_block，但这些并不常见。

SYCL 原子语法有点不同，它基于 C++20 中的 std::atomic_ref（有关 SYCLatomic_ref 类的详细信息以及它与 std::atomic_ref 的比较，请参阅第 19 章）。如果我们希望 SYCLatomic 与 CUDAatomicAdd 函数等效，我们需要声明 SYCLatomic_ref 具有类似的 memory_order::relaxed 内存顺序和 memory_scope::device 范围，如图 21-9 所示。

```
q.parallel_for(count, [=](auto id) {
    // The SYCL atomic_ref must specify the default order
    // and default scope as part of the atomic_ref type. To
    // match the behavior of the CUDA atomicAdd we want a
    // relaxed atomic with device scope:
    atomic_ref<int, memory_order::relaxed,
               memory_scope::device>
    aref(*buffer);

    // When no memory order is specified, the defaults are
    // used:
    aref.fetch_add(1);

    // We can also specify the memory order and scope as
    // part of the atomic operation:
    aref.fetch_add(1, memory_order::relaxed,
                  memory_scope::device);
});
```

图 21.9: CUDA 和 SYCL 等效原子操作

较新的 CUDA 代码可能使用 CUDA C++ 标准库中的 cuda::atomic_ref 类。cuda::atomic_ref 类看起来更像 SYCLatomic_ref 类，但也有一些重要的区别需要注意：

- CUDAatomic_ref 的范围是可选的，但如果未指定，则默认为整个系统。在所有情况下，SYCLatomic_ref 都必须指定原子范围。
- CUDAatomic_ref 的默认原子顺序是无条件顺序一致性，而 SYCLatomic_ref 可以指定不同的默认原子顺序。通过指定默认的原子顺序，我们的

SYCL 代码可以更加简洁，并且即使原子顺序不是顺序一致性时也可以使用 `+ =` 等方便的运算符。

当我们的代码或算法需要原子时，我们需要记住最后一个问题：SYCL 规范不要求某些原子操作和原子范围，并且可能并非所有 SYCL 设备都支持。对于 CUDA 设备也是如此，但由于 SYCL 设备的多样性，记住 SYCL 尤为重要。有关如何查询 SYCL 设备属性的更多详细信息，请参阅第 12 章，有关 SYCL 设备或上下文可能支持的原子功能的描述，请参阅第 19 章。

21.3.3 其他差异

本节介绍了将 CUDA 代码移植到 SYCL 时需要记住的一些其他差异。

项目类与内置变量 CUDA 和 SYCL 之间较大的风格差异之一是 Kernel 实例识别其在 N 维并行索引空间中的位置的方式。回想一下第 4 章，每个 SYCL Kernel 都必须采用一个项、一个 `nd_item`、一个 `id`，或者在某些情况下采用一个整数参数来标识并行索引空间中的 Work-Items。`item` 和 `nd_item` 类还可用于查询有关并行索引空间本身的信息，例如全局范围、局部范围以及 Work-Items 所属的不同组。

CUDA Kernel 不包含任何参数来标识并行索引空间中的 CUDA 线程。相反，CUDA 线程使用内置变量（例如 `blockIdx` 和 `threadIdx`）来标识并行索引空间中的位置，并使用内置变量（例如 `gridDim` 和 `blockDim`）来表示有关并行索引空间本身的信息。使用协作组的较新 CUDA Kernel 还可以通过调用内置函数（如 `this_thread_block`）隐式构造某些协作组。

这通常只是一个语法差异，不会在功能上影响我们可以编写的代码，尽管这确实意味着 SYCL Kernel 在更多情况下可能会将一个项目或一个 `nd_item` 传递给被调用函数，例如被调用函数是否需要知道 work-item 索引。

上下文 CUDA 和 SYCL 之间的另一个概念差异是 SYCL 上下文的概念。回想一下，SYCL 上下文是一个存储一组 SYCL 设备的 SYCL 应用程序状态的对象。例如，SYCL 上下文可以存储有关内存分配或编译程序的信息。上下文对于 SYCL 应用程序来说是一个重要概念，因为单个 SYCL 应用程序可能支持来自多个供应商的设备，可能使用多个后端 API。

在大多数情况下，我们的 SYCL 程序可能完全不知道上下文的存在，并且本书中的大多数示例程序都不会创建或操作上下文。如果我们确实选择在程序中创建额外的 SYCL 上下文（无论是隐式还是显式），我们需要小心，不要将一个上下文中特定于上下文的 SYCL 对象与不同的 SYCL 上下文一起使用。充其量，不小心使用多个上下文可能会导致我们的程序运行效率低下，比如说，如果我们最终多次编译 SYCL Kernel，每个上下文一次。在最坏的情况下，跨上下文混合 SYCL 对象可能会导致未定义的行为，导致我们的程序变得不可移植或在某些后端或设备上执行不正确。

为了完整起见，请注意 CUDA 也有上下文的概念，尽管 CUDA 上下文仅由较低级别的 CUDA 驱动程序 API 公开。大多数 CUDA 程序也不创建或操作上下文。

错误检查 考虑的最后一个区别与错误检查和错误处理有关。由于 CUDA 继承了 C 语言，CUDA 中的错误通过 CUDA 函数调用的错误代码返回。对于大多数 CUDA 函数，失败的错误代码表示返回错误的函数中存在错误，例如函数的参数不正确。但对于其他一些 CUDA 函数（例如 `cudaDeviceSynchronize`），错误值也可以返回设备上发生的异步错误。

SYCL 还具有同步和异步错误，尽管这两种类型的错误都是使用 SYCL 异常而不是 SYCL 函数的返回值来报告的。有关 SYCL 中错误检测和错误处理的更多信息，请参阅第 5 章。

21.4 CUDA 中的功能尚未在 SYCL 中提供！

到目前为止，我们已经描述了 CUDA 和 SYCL 中都有特征但表达方式不同的情况。本节介绍 CUDA 中的几个功能，但（当前）SYCL 中没有等效功能。这不是详尽的列表，但旨在描述 CUDA 应用程序常用的一些功能，这些功能在迁移到 SYCL 时可能需要更多努力。

请注意，供应商特定的功能是标准化过程的重要组成部分，无论它们是标准的扩展还是在完全供应商特定的 API 中定义。特定于供应商的功能提供了重要的实施经验，并允许功能在完善并纳入标准之前证明其价值。其中许多功能已经在积极开发中，以纳入 SYCL 标准，其中一些功能可能已经作为该标准的扩展提供。

注 87 (参与进来！) 来自用户和开发人员的反馈是标准化过程的另一个重要部分。如果您对新功能有想法，或者您发现其他 API 的扩展或功能很有

价值，请考虑参与其中！*SYCL* 是一个开放标准，许多 *SYCL* 实现都是开源的，因此可以轻松参与不断发展的 *SYCL* 社区。

21.4.1 全局变量

尽管程序员很早就被告知永远不要使用全局变量，但有时全局变量是完成这项工作的正确工具。我们可能选择使用全局变量来存储有用的常量、查找表或我们希望执行数据并行 Kernel 的所有 Work-Items 都可以访问的其他值。

CUDA 支持不同地址空间中的全局变量，因此具有不同的生命周期。例如，CUDA 程序可以在全局内存空间中声明一个对于每个设备都是唯一的 `__device__` 全局变量。这些全局变量可以由主机设置或从主机读取，并由执行 Kernel 的所有 CUDA 线程访问。CUDA 程序还可以在 CUDA 共享内存空间中声明一个 `__shared__` 全局变量（请记住，这相当于在 SYCL 本地内存中声明的变量），该变量对于每个 CUDA 块都是唯一的，并且只能由该块中的 CUDA 线程访问堵塞。

SYCL 尚不支持设备代码中的全局变量，尽管正在开发提供类似功能的扩展。

21.4.2 协作组

如本章前面所述，CUDA 的最新版本支持协作组，这为 Barrier 和洗牌函数等集体操作提供了替代语法。SYCL 组对象和 SYCL 组算法库与 CUDA 协作组有许多相似之处，但仍然存在一些关键差异。

最大的区别在于，SYCL 组功能目前仅适用于预定义的 SYCL Work-Groups 和 Sub-Groups 类，而 CUDA 协作组则更加灵活。例如，CUDA 程序可以创建固定大小的 `tiled_partition` 组，将现有组划分为一组更小的组，或者 CUDA 程序可以将 CUDA warp 中当前活动的 CUDA 线程组表示为 `coalesced_group`。

CUDA 程序还可以创建比 Work-Groups 更大的协作组。例如，CUDA 程序可以创建表示网格中所有 CUDA 线程（相当于全局范围内的所有 Work-Items）的 `grid_group`，或者表示线程块集群中所有 CUDA 线程的 `cluster_group`。为了有效地使用这些更新和更大的组，必须使用特殊的主机 API 函数启动 CUDA Kernel，以确保网格中的所有 CUDA 线程可以协作，或者指定线程块簇尺寸。

SYCL 尚不支持 CUDA 中的所有协作组类型，尽管正在进行扩展以向 SYCL 添加其他组类型。SYCL 2020 中引入的组对象和组算法使 SYCL 能够很好地支持此功能。

21.4.3 矩阵乘法硬件

我们将在本节中描述的最后一个功能是访问矩阵乘法硬件，也称为矩阵乘法和累加 (MMA) 硬件、张量核心或脉动阵列。这些都是专用硬件引擎的不同名称，这些引擎是专门为加速矩阵乘法运算而构建的，而矩阵乘法运算对于许多人工智能 (AI) 工作负载至关重要。如果我们想要定制这些工作负载，那么必须能够访问数据并行 Kernel 中的矩阵乘法硬件以实现峰值性能，这一点非常重要。

CUDA 通过扭曲矩阵乘法和累加 (WMMA) 函数提供对矩阵乘法硬件的访问。这些函数有效地允许扭曲中的 CUDA 线程（相当于 Sub-Groups 中的 Work-Items）协作以在较小的矩阵图块上执行矩阵乘法和累加操作。对于某些设备和算法，这些矩阵图块的元素可以是 32 位浮点型或 64 位双精度型，但更常见的是使用较低精度的类型，例如 8 位字符、16 位半数或专用 AI 类型，例如 bfloat16s (bf16)。

CUDA 和 SYCL 都在积极发展对矩阵乘法硬件的支持。这是一个很好的例子，说明不同的供应商最初如何通过特定于供应商的机制添加对其特定于供应商的功能的支持，然后改进功能，并将常见的最佳实践添加到标准中。

21.5 移植工具和技术

幸运的是，当我们选择将应用程序从 CUDA 迁移到 SYCL 时，它不需要是手动过程，我们可以使用工具来自动化部分迁移。本节将介绍其中一种协助迁移的工具和技术。

21.5.1 使用 dpct 和 SYCLomatic 迁移代码

在本节中，我们将介绍 DPC++ 兼容性工具 (dpct) 和相关的开源 SYCLomatic 工具。我们将使用 dpct 自动将 CUDA 示例迁移到 SYCL，尽管本节中描述的概念同样适用于 SYCLomatic。

图 21-10 显示了我们将要迁移的简单 CUDA 示例的重要部分。此示例

反转缓冲区的块。这在实践中不是一个非常有用的示例，但它有我们的自动迁移工具需要处理的有趣情况，例如 CUDA 共享内存全局变量、Barrier、设备查询、内存分配和初始化、Kernel 调度本身，以及一些基本的错误检查。

```
__shared__ int scratch[256];
__global__ void Reverse(int* ptr, size_t size) {
    auto gid = blockIdx.x * blockDim.x + threadIdx.x;
    auto lid = threadIdx.x;

    scratch[lid] = ptr[gid];
    __syncthreads();
    ptr[gid] = scratch[256 - lid - 1];
}

int main() {
    std::array<int, size> data;
    std::iota(data.begin(), data.end(), 0);

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);
    std::cout << "Running on device: " << deviceProp.name << "\n";

    int* ptr = nullptr;
    cudaMalloc(&ptr, size * sizeof(int));
    cudaMemcpy(ptr, data.data(), size * sizeof(int),
               cudaMemcpyDefault);
    Reverse<<<size / 256, 256>>>(ptr, size);
    cudaError_t result = cudaDeviceSynchronize();
    if (result != cudaSuccess) {
        std::cout << "An error occurred!\n";
    }
    // ...
}
```

图 21.10: 我们将自动迁移一个简单的 CUDA 程序

21.5.2 运行 dpct

因为这是一个简单的示例，所以我们可以简单地调用 dpct 并传递我们想要迁移的 CUDA 源文件。对于更复杂的场景，可以在应用程序构建过程中调用 dpct 来识别要迁移的 CUDA 源文件。请参阅本章末尾的链接以获取更多信息和其他培训材料。

```
$ dpct source_file.cu
NOTE: Could not auto-detect compilation database for file
'source_file.cu' in '/path/to/your/file' or any parent directory.
The directory "dpct_output" is used as "out-root"
Processing: /path/to/your/file/source_file.cu
/path/to/your/file/source_file.cu:38:5: warning: DPCT1001:0: The
statement could not be removed.
    std::cout << "An error occurred!\n";
^
/path/to/your/file/source_file.cu:37:3: warning: DPCT1000:1: Error
handling if-stmt was detected but could not be rewritten.
    if (result != cudaSuccess) {
^
/path/to/your/file/source_file.cu:36:24: warning: DPCT1003:2: Migrated
API does not return error code. (*, 0) is inserted. You may need to
rewrite this code.
    cudaError_t result = cudaDeviceSynchronize();
^
Processed 1 file(s) in -in-root folder "/path/to/your/file"
```

图 21.11: 迁移此 CUDA 程序时的示例 *dpct* 输出

当我们在示例 CUDA 源文件上运行 *dpct* 时，我们可能会看到如图 21-11 所示的输出。我们可以从这个输出中得出一些结论。首先，我们的文件已成功处理，这太棒了！不过，有一些警告表明 *dpct* 无法迁移。对于我们的示例，所有三个警告都是由于 CUDA 和 SYCL 之间的错误检查差异造成的。对于我们的程序，*dpct* 能够生成 SYCL 代码，当程序不生成错误时，该代码将正常运行，但它无法迁移错误检查。

错误检查警告是一个很好的例子，例如 *dpct* 和 SYCLomatic 等迁移工具将无法迁移所有内容。我们应该期望审查和调整迁移的代码以解决任何迁移问题，或者以其他方式改进迁移的 SYCL 代码的可维护性、可移植性或性能。

不过，对于此示例，我们可以按原样使用迁移的代码。

```
$ icpx -fsycl -fsyycl-targets=spir64,nvptx64-nvidia-cuda \
    migrated.cpp -o migrated
$ ./migrated
Running on device: Intel(R) UHD Graphics 770
Success.
$ ONEAPI_DEVICE_SELECTOR=opencl:cpu ./migrated
Running on device: 12th Gen Intel(R) Core(TM) i9-12900K
Success.
$ ONEAPI_DEVICE_SELECTOR=ext_oneapi_cuda:gpu ./migrated
Running on device: NVIDIA GeForce RTX 3060
Success.
```

图 21.12: 编译和运行迁移的 CUDA 程序

图 21-12 显示了如何使用支持 NVIDIA GPU 的 DPC++ 编译器来编译迁移的代码，然后显示我们的迁移程序在 Intel GPU、Intel CPU 和 NVIDIA GPU 上的成功执行。请注意，如果我们要在具有不同设备的不同系统上运行迁移的程序，输出可能会有所不同，或者如果系统中不存在所选设备，则可能无法运行。

21.5.3 检查 dpct 输出

```
void Reverse(int *ptr, size_t size,
            const sycl::nd_item<3> &item_ct1,
            int *scratch) {
    auto gid =
        item_ct1.get_group(2) * item_ct1.get_local_range(2) +
        item_ct1.get_local_id(2);
    auto lid = item_ct1.get_local_id(2);

    scratch[lid] = ptr[gid];
    item_ct1.barrier(sycl::access::fence_space::local_space);
    ptr[gid] = scratch[256 - lid - 1];
}
```

图 21.13: 从 CUDA 迁移的 SYCL Kernel

如果我们检查迁移的输出，我们可以看到 dpct 处理了本章中描述的许多差异。例如，在图 21-13 所示的生成的 SYCL Kernel 中，我们看到 `__shared__` 全局变量 `scratch` 被转换为本地内存访问器并传递到 Kernel 中。我们还可以看到，内置变量 `blockIdx` 和 `threadIdx` 被替换为对 `nd_item` 类实例的调用，并且连续维度的不同约定得到了正确处理，例如，通过将 `threadIdx.x` 的使用替换为调用 `item_ct1.get_local_id(2)`。

```
dpct::device_info deviceProp;
dpct::dev_mgr::instance().get_device(0).get_device_info(
    deviceProp);
std::cout << "Running on device: "
    << deviceProp.get_name() << "\n";
```

图 21.14: 从 CUDA 迁移的 SYCL 设备名称查询

我们还可以看到 dpct 通过使用几个 dpct 实用函数来处理一些主机代

码差异，例如图 21-14 中所示的迁移设备查询。这些辅助函数仅供迁移后的代码使用。为了可移植性和可维护性，我们应该更喜欢直接使用标准 SYCL API 进行额外的开发。

不过，总的来说，dpct 生成的 SYCL 代码是可读的，并且 CUDA 代码和迁移的 SYCL 代码之间的映射是清晰的。尽管通常需要额外的手动编辑，但使用 dpct 或 SYCLomatic 等自动化工具可以节省迁移过程中的时间并减少错误。

21.6 总结

在本章中，我们描述了如何将应用程序从 CUDA 迁移到 SYCL，以使应用程序能够在任何 SYCL 设备上运行，包括使用支持 CUDA 的 SYCL 编译器的 CUDA 设备。

我们首先研究 CUDA 和 SYCL 程序之间的许多相似之处（抛开术语不谈）。我们看到了 CUDA 和 SYCL 如何从根本上使用相同的基于 Kernel 的并行方法，以及相似的执行模型和内存模型，使得将 CUDA 程序迁移到 SYCL 相对简单。我们还探讨了 CUDA 和 SYCL 存在细微语法或行为差异的一些地方，因此在我们将 CUDA 应用程序迁移到 SYCL 时最好记住这些差异。我们还描述了 CUDA 中但 SYCL 中尚未包含的几个功能（尚未！），并且描述了特定于供应商的功能如何成为标准化过程的重要组成部分。

最后，我们研究了几种用于自动化部分迁移过程的工具，并使用 dpct 工具将一个简单的 CUDA 示例自动迁移到 SYCL。我们看到了该工具如何自动迁移大部分代码，生成能正确且可读的代码。迁移后，我们能够在不同的 SYCL 设备上运行迁移的 SYCL 示例，尽管更复杂的应用程序可能需要额外的检查和编辑。

21.7 了解更多信息

将 CUDA 代码迁移到 SYCL 是一个热门话题，还有许多其他资源可供了解更多信息。作者发现以下两个有用的资源：

- 显示如何从 CUDA 迁移到 SYCL 的一般信息和教程 (tinyurl.com/cuda2sycl)
- DPC++ 兼容性工具入门指南 (tinyurl.com/startDPCpp)

22 SYCL 未来发展方向

现在请花点时间感受一下平和与平静，因为我们已经介绍了使用 C++ 和 SYCL 进行编程。所有的部分都已就位。

我们努力确保前几章中的代码示例使用标准 SYCL 2020 功能并在各种硬件上执行，并且我们在少数地方使用了扩展（例如互操作性和 FPGA 特定扩展），我们将其指出。然而，截至 2023 年中期，本尾声中显示的面向未来的代码无法使用任何编译器进行编译。

在这篇尾声中，我们推测未来。我们的水晶球可能有点难以阅读——这个尾声没有任何保证。我们在本书第一版中做出的一些预测实现了，但其他预测却没有实现。

本尾声让我们先睹为快，了解即将推出的 SYCL 功能和 DPC++ 扩展，我们对此感到非常兴奋。我们不保证本尾声中打印的代码示例可以编译：有些可能已经与本书之后发布的编译器兼容，而另一些则可能仅在一些语法调整后才能编译。某些功能可能会作为扩展发布或合并到 SYCL 的未来版本中，而其他功能可能会无限期地保留为实验性功能。与本书相关的 GitHub 存储库中的代码示例可能会随着其发展而更新以使用新语法。同样，我们也会对本书进行勘误，并且可能会不时进行补充。我们建议检查这两个地方的更新（代码存储库和书籍勘误表 - 链接可以在第 1 章前面找到）。

22.1 与 C++11、C++14 和 C++17 更紧密地结合

保持 SYCL 和 C++ 之间的紧密一致有两个优点。首先，它使 SYCL 能够利用 C++ 的最新、最强大的功能来提高开发人员的工作效率。其次，它增加了 SYCL 中引入的异构编程功能成功影响 C++ 未来方向的机会。

SYCL 1.2.1 基于 C++11，SYCL 2020 接口的许多最大改进只能通过 C++14（例如通用 lambda）和 C++17（例如，类模板参数推导——CTAD）。我们预计 SYCL 和 C++ 随着时间的推移会变得越来越接近，并且已经有一些令人兴奋的工作正在进行中。

C++ 标准模板库 (STL) 包含几种与第 17 章中讨论的并行模式相对应的算法。STL 中的算法通常适用于由迭代器对指定的序列，并且从 C++17 开始支持执行策略参数表示它们应该顺序执行还是并行执行。该标准还允许实现定义自己的执行策略，第 18 章中介绍的 oneAPI DPC++ 库 (oneDPL) 利用此类自定义执行策略来使算法能够在 SYCL 设备上执行。其结果是一

种高生产率的异构设备编程方法 - 如果可以仅使用 STL 算法的功能来表达应用程序, oneDPL 就可以在我们的系统中使用加速器, 而无需编写一行 SYCL Kernel 代码! 关于 STL 算法应如何与某些 SYCL 概念 (例如缓冲区) 交互, 以及如何确保我们可能需要的所有标准库类 (例如 std::complex、std::atomic) 可用, 仍然存在悬而未决的问题在设备代码中, 但 oneDPL 是统一主机和设备代码的漫长道路上的第一步。

22.2 采用 C++20、C++23 及其他版本的功能

SYCL 规范故意落后于 C++, 以确保它使用的功能具有广泛的编译器支持。然而, SYCL 委员会成员 (其中许多人还参与了 ISO C++ 委员会) 正在密切关注 C++ 未来版本的开发情况。

采用我们在此讨论的尚未最终确定为规范的 C++ 或 SYCL 功能可能会是一个错误——功能在成为标准之前可能会发生重大变化。尽管如此, 有许多正在讨论的功能可能会改变未来 SYCL 程序的外观和行为方式, 值得讨论。

SYCL 2020 中的一些功能由 C++20 通知 (例如 std::atomic_ref), 其他功能则预先采用到 `sycl::` 命名空间中 (例如 std::bit_cast、std::span)。当我们迈向 SYCL 的下一个正式版本时, 我们希望与 C++20 更加紧密地保持一致, 并合并其中最有用的部分。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 章中探讨了如何使用类似的接口来定义设备范围的屏障, 并且在新的 C++20 语法的上下文中重新检查子组和工作组屏障也可能有意义。

C++23 中最令人兴奋的功能之一是 `mdspan`, 它是一种非拥有数据视图, 它提供指针的多维数组语法和 AccessorPolicy 作为控制对底层数据的访问的扩展点。这些语义与 SYCL 访问器的语义非常相似, 并且 `mdspan` 将使类似访问器的语法能够用于缓冲区和 USM 分配, 如图 EP-1 所示。

```
queue q;
constexpr int N = 4;
constexpr int M = 2;
int* data = malloc_shared<int>(N * M, q);

stdex::mdspan<int, N, M> view{data};
q.parallel_for(range<2>{N, M}, [=](id<2> idx) {
    int i = idx[0];
    int j = idx[1];
    view(i, j) = i * M + j;
}).wait();
```

图 22.1: 使用 *mdspan* 将类似访问器的索引附加到 *USM* 指针

希望 SYCL 正式支持 *mdspan* 只是时间问题。同时，我们建议感兴趣的读者尝试作为 Kokkos 项目一部分提供的开源生产质量参考实现。

22.3 混合 SPMD 和 SIMD 编程

C++ 的另一个令人兴奋的提议功能是 *std::simd* 类模板，它旨在为 C++ 中的显式向量并行性提供可移植接口。采用此接口将明确区分第 11 章中描述的向量类型的两种不同用途：为了程序员的方便而使用向量类型，以及忍者程序员为了低级性能调整而使用向量类型。同一语言中对 SPMD 和 SIMD 编程风格的支持也引发了一些有趣的问题：我们应该如何声明 Kernel 使用哪种风格，以及我们是否应该能够在同一 Kernel 中混合和匹配风格？

我们已经开始以 DPC++ 扩展 (*sycl_ext_oneapi_invoke_simd*) 的形式探索这个问题的潜在答案，它提供了一个新的 *invoke_simd* 函数（在 *std::invoke* 上建模），允许开发人员从 SPMD 中调用显式矢量化 (SIMD) 代码核心。对 *invoke_simd* 的调用充当两种编程风格所隐含的两种执行模型之间的清晰边界，并定义数据应如何在它们之间流动。图 EP-2 中的代码显示了 *invoke_simd* 用法的一个非常简单的示例，调用一个期望接收标量和向量 (*simd*) 参数组合的函数。

```

// Function expects one vector argument (x) and one scalar
// argument (n)
simd<float, 8> scale(simd<float, 8> x, float n) {
    return x * n;
}

q.parallel_for(..., sycl::nd_item<1> it)
    [[sycl::reqd_sub_group_size(8)]] {
    // In SPMD code, each work-item has its own x and n
    // variables
    float x = ...;
    float n = ...;

    // Invoke SIMD function (scale) using work-items in the
    // sub-group x values from each work-item are combined
    // into a simd<float, 8>
    // The value of n is defined to be the
    // same (uniform) across all work-items
    // Returned simd<float, 8> is unpacked
    sycl::sub_group sg = it.get_sub_group();
    float y = invoke_simd(sg, scale, x, uniform(n));
);
}

```

图 22.2: 从 *SPMD Kernel* 调用 *SIMD* 函数的简单示例

`invoke_simd` 采用的方法有几个优点。首先，不会出现令人讨厌的意外——显式调用具有不同执行模型的函数，并且用户负责描述如何来回编组数据。其次，该机制允许细粒度的专业化——可以只编写几行显式矢量化代码（例如，用于性能调整），而不必丢弃其余的 SPMD 代码。最后，扩展很简单——`invoke_simd` 本身可以通过简单的重载来扩展以支持新组或新参数映射，并且可以引入类似的 `invoke_*` 函数来处理与不同上下文的互操作性（例如，用非 SYCL 语言编写的代码）。

22.4 地址空间

SYCL 2020 中引入的通用地址空间支持有可能极大地简化许多代码，因为它允许我们使用常规 C++ 指针，而不必担心正在使用哪种内存。许多现代体系结构为通用地址空间提供硬件支持，因此我们可以期望使用常规 C++ 指针的代码能够在各种机器上工作，并且性能开销最小。

然而，在某些（较旧的或更特殊用途的）体系结构上，通用地址空间支持是一个更复杂的故事。一些硬件可能使用不同的指令来访问不同类型的数据。

内存，要求编译器在编译时识别具体的地址空间（即生成正确的指令）。还可能存在无法表示通用地址空间的 SYCL 后端（例如 OpenCL 1.2）。SYCL 2020 通过一组用于推导地址空间的推理规则来允许此类硬件和后端。

地址空间推导规则继承自 SYCL 1.2.1，SYCL 2020 规范包含一条注释，表明将在 SYCL 的未来版本中重新审视这些规则。尽管在撰写本文时尚不清楚这些规则将如何改变，但 SYCL 的长期想法是明确的：在大多数情况下，我们不应该关心地址空间管理，而应该相信编译器和硬件会做正确的事情。

22.5 特化机制

计划引入编译时查询，使 Kernel 能够根据目标设备的属性（方面）进行专门化（例如，设备类型、对特定扩展的支持、工作组本地内存的大小、子组）大小由编译器选择）。此类查询需要一种当前在 C++ 中不存在的新型常量表达式，它们在编译主机代码时不一定是 `constexpr`，但在目标设备已知时变为 `constexpr`。

用于公开这种“设备常量表达式”概念的确切机制仍在设计中。我们希望它建立在 SYCL 2020 中引入的专门化常量功能的基础上，并且外观和行为与图 EP-3 中所示的代码类似。

```
h.parallel_for(range{1}, [=](id<1> idx) {
    if_device_has<aspect::cpu>([&]() {
        /* Code specialized for CPUs */
        out << "On a CPU!" << endl;
    }).else_if_device_has<aspect::gpu>([&]() {
        /* Code specialized for GPUs */
        out << "On a GPU!" << endl;
    });
});
```

图 22.3: 在 *Kernel* 编译时根据设备方面专门处理 *Kernel* 代码

22.6 编译时属性

SYCL 允许通过将属性列表传递到构造函数来修改某些类（例如缓冲区、访问器）的行为。这些属性已经非常强大，但它们的功能受到以下事实的限制：传递给构造函数的属性直到运行时才知道。允许在编译时声明某些属性有可能显着提高性能，方法是减少运行时检查的数量，并使编译器能够在存在特定属性的情况下积极专门化主机和设备代码。

DPC++ 编译器支持编译时属性的实验性扩展 (`sycl_ext_oneapi_properties`)，并且它已经启用了各种其他扩展：

- 使用超出地址空间范围的信息注释的指针，这可以为 `sycl::multi_ptr` 的未来提供信息 (`sycl_ext_oneapi_annotated_ptr`)
- Kernel 配置控制，可以替代 C++ 属性并增强仅库 SYCL 实现的功能 (`sycl_ext_oneapi_kernel_properties`)
- 所需内存行为和访问控制的描述 (`sycl_ext_oneapi_device_global`、`sycl_ext_oneapi_prefetch`)

我们在编译时属性方面的早期经验非常积极，并且我们一直在寻找越来越多的潜在用例。鉴于其广泛的适用性，我们渴望看到未来 SYCL 规范中采用某些版本的编译时属性。

22.7 总结

SYCL 周围已经充满了兴奋，而这仅仅是开始！我们（作为一个社区）还有很长的路要走，需要持续不断的努力来提炼异构编程的最佳实践，并设计新的语言功能，以在性能、可移植性和生产力之间达到所需的平衡。

我们需要你的帮助！如果 SYCL 中缺少您最喜欢的 C++（或任何其他编程语言）功能，请联系我们。我们可以共同塑造 SYCL 和 C++ 的未来方向。

22.8 更多信息

Khronos SYCL Registry, www.khronos.org/registry/SYCL

H. Carter Edwards et al., “mdspan: A Non-Owning Multidimensional Array Reference,”wg21.link/p0009

D. Hollman et al., “Production-Quality mdspan Implementation,” github.com/kokkos/mdspan
Intel DPC++ Compiler Extensions, tinyurl.com/syckextend