arXiv:2303.08058v2 [cs.DC] 9 May 2023

使用 HPX-Kokkos 和 SYCL 进行恒星并合：使用 SYCL 实现异步多任务运行时系统的方法

预印本

Gregor Daiß
斯图加特大学

Patrick Diehl, **Hartmut Kaiser**
路易斯安那州立大学

Dirk Pfl¨uger
斯图加特大学

2023年5月10日

摘要

从 NVIDIA GPU 到 AMD GPU 和英特尔 GPU：鉴于当前超级计算机中可用加速卡的异构性，可移植性是现代高性能计算应用的关键方面。在 Octo-Tiger 这一模拟双星系统和恒星并合的天体物理学应用中，我们依靠 Kokkos 及其各种执行空间来实现可移植计算内核。进而，我们使用 HPX（一种分布式任务运行时系统）来协调内核启动、CPU 任务和通信。这种组合使我们能够灵活交错执行可移植的 CPU/GPU 计算与通信，从而在各种超级计算机上实现可扩展性。

然而，要让 HPX 和 Kokkos 协同发挥最佳性能，我们需要能够将 Kokkos 内核视为 HPX 任务。否则，我们就无法将异步 Kokkos 内核启动集成到 HPX 的任务图中，而必须通过栅栏命令主动等待它们，白白耗费原本可以用于其他工作的 CPU 时间。借助名为 HPX-Kokkos 的集成层，在某些 Kokkos 执行空间（如 CUDA 执行空间）中，已经可以将 Kokkos 内核视为任务，但在其他执行空间（如 SYCL 执行空间）中还无法做到。
在这项工作中，我们着手让 Octo-Tiger 和 HPX 本身兼容 SYCL。为此，我们对软件进行了多项改动，其中最重要的是引入 HPX-SYCL 集成。该集成使我们能够将 SYCL 事件视为 HPX 任务，进而通过扩展 HPX-Kokkos 的支持范围，使其也能全面支持 Kokkos 的 SYCL 执行空间，从而更好地集成 Kokkos。

我们展示了实现此 HPX-SYCL 集成的两种方式，并使用 Octo-Tiger 及其 Kokkos 内核，在 NVIDIA A100 和 AMD MI100 上对它们进行了测试。我们发现，启用此集成后可获得幅度不大但仍显著的加速比（相关配置下为 1.11x 至 1.15x）；即使只在 Octo-Tiger 的简单单节点场景中运行，此时通信和 CPU 利用率尚未成为问题，也能获得这种提升。我们还发现，在 HPX 调度器中使用事件轮询的集成方式，远优于使用 SYCL 主机任务的替代实现。

关键词： SYCL、Kokkos、HPX、异步多任务、图形处理器、CUDA、HIP、单指令多数据

1 引言

现代 GPU超级计算机，如 ANL 的 Aurora、NERSC 的 Perlmutter 和橡树岭国家实验室的 Frontier，每台都包含数千个计算节点，但采用的图形处理器各不相同，涵盖 NVIDIA® 图形处理器、超威半导体® 图形处理器和 Intel® 图形处理器。要为这类系统开发可移植的高性能计算（HPC）应用程序，既需要能够高效适配这些不同图形处理器（并且最好也能在 CPU 上保持良好性能）的计算内核，也需要高效的通信和工作调度，以避免扩展到更多图形处理器节点时出现瓶颈，从而实现分布式可扩展性。在 Octo-Tiger 中，我们通过结合使用 Kokkos [27] 和用于并行性与并发性的 C++ 标准库（HPX）[16]来解决这一问题。Octo-Tiger 是一种用于模拟双星系统和恒星并合的天体物理学代码 [21], 。借助 Kokkos，我们可以编写可移植的计算内核，利用其中多种内存空间和执行空间，在 CPU 和图形处理器上运行。我们还使用 HPX（一种分布式异步多任务运行时系统）来表达内核之间的依赖关系（通过 HPX future），调用远程计算节点上的方法，重叠通信与计算，并最终扩展至数千个计算节点，例如 CSCS 的 Piz Daint [6] 或橡树岭国家实验室的 Summit [10]。

在先前的工作中，我们将 Kokkos 与 HPX 集成，使我们能够为异步 Kokkos 内核启动和深拷贝获取 HPX future，从而将它们无缝地嵌入 HPX 任务图中 [7]。借助这一集成，当我们需要在主机上获取结果时，无需显式同步 GPU。相反，我们只需为相应的 HPX future 创建一个续延，并在结果可用时自动使用这些结果（例如，在主机上进行后处理或通信）。

然而，这种 HPX-Kokkos 集成仅适用于部分 Kokkos 执行空间（准确地说，是 CUDA®、HIP 和 HPX 执行空间）。要使其正常运行，HPX 必须与相应的底层 API 进行更深入的集成，以便为异步 API 调用获取 HPX future（例如`cudaLaunchKernel`）。

对于不受支持的执行空间，HPX-Kokkos 集成必须先等待（栅栏）GPU 结果返回，然后再返回一个已就绪的虚拟 future（为保持兼容性，尽管此时它已不再是异步的）。这会浪费 CPU 时间，因为等待过程会主动阻塞调用它的 CPU线程，直到 GPU 内核及相关内存拷贝完成；这反过来又会延误其他 CPU 任务，例如通信或启动更多 GPU 内核。值得注意的是，Kokkos 已包含 SYCL™ 执行空间以支持英特尔 GPU，但 HPX-Kokkos 尚不支持它。

在这项工作中，我们通过引入**HPX-SYCL 集成**来解决这一问题，从而将 HPX-Kokkos 支持的执行空间扩展到 SYCL 执行空间，并使用 Octo-Tiger 在 NVIDIA 和 AMD GPU 上对其进行测试并评估其性能（因为我们无法使用当前的英特尔 GPU）。
当然，这种集成也支持在不使用 Kokkos 的情况下同时使用 SYCL 和 HPX，因为它能使 HPX 任务图感知异步 SYCL 操作。这样，用户便可以像往常一样用 SYCL 定义完整的操作图，同时在最后获取一个对应最终结果的 HPX future，并利用 HPX continuation 异步调度通信或其他任意 CPU 任务。

我们研究了实现这种 HPX-SYCL 集成的两种替代方法：SYCL 已经提供了一种方法，可以通过依赖 SYCL 事件的主机任务，在其自身的图中异步调度 CPU 工作。一种集成 HPX 和 SYCL 的方法是直接使用这种 SYCL 主机任务，在其所依赖的事件完成后，将关联的 HPX future 设为就绪状态。不过，我们还展示了另一种集成这些框架的方法：它不依赖这些 SYCL 主机任务，而是在 HPX 调度器中实现事件轮询；我们发现，这种方法的性能远胜于主机任务 这一替代方案。

因此，本工作的贡献如下：

1. 我们添加了 HPX-SYCL 集成。
2. 我们使用 Kokkos SYCL 执行空间和 HPX-SYCL 集成，使 Octo-Tiger 兼容 SYCL。这需要对其他代码仓库进行额外修改（并对 Octo-Tiger 本身进行调整）。
3. 我们使用 Octo-Tiger 在 AMD MI100 和 NVIDIA A100 上收集了运行时数据，不仅使用了 SYCL 执行空间，还使用了分别运行于原生执行空间（CUDA 和 HIP）上的相同 Kokkos 计算内核，以及以纯 CUDA 和 HIP 实现的前代内核。我们收集的部分数据是在启用 HPX-SYCL 集成的情况下获得的，其他数据则是在禁用该集成的情况下获得的。

这些贡献带来了多重益处：Octo-Tiger 用户可以使用新增的 Kokkos 执行空间，而且该空间已全面可用（采用异步而非阻塞方式），未来还有可能让程序运行在英特尔 GPU 和 Aurora 上。HPX 应用开发者则能受益于 HPX-SYCL 集成，因为他们可以将 SYCL 完全集成到自己的应用程序中（既可以使用 Kokkos SYCL 执行空间和 HPX-Kokkos，也可以使用纯 SYCL）。尽管起初将 HPX 与纯 SYCL 结合起来似乎有些奇怪，因为这两种框架都具备基于任务的能力，但它们实际上可以很好地互补：SYCL 负责性能可移植计算内核及其间的数据依赖关系，HPX 则负责通过期值同步结果，并将应用程序分发到多个计算节点上。

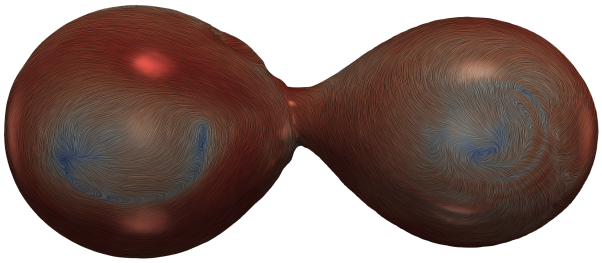


图 1：使用 Octo-Tiger 模拟的双白矮星并合快照。两颗恒星之间形成了聚集带，较小的右侧恒星将质量转移给左侧较大的恒星。颜色表示流的速度大小，红色代表较高速度。

反过来，希望为应用程序添加分布式能力的 SYCL 应用开发者，也可以将 HPX 作为纯 MPI（或类似 Celerity [26] 的框架）之外一个可行的替代方案。

本文其余部分的结构如下：由于我们以 Octo-Tiger 作为示例，下一节将首先介绍 Octo-Tiger 及其软件依赖项（尤其是 HPX 和 Kokkos），以及它的执行模型。随后，在第 3 节中，我们将重点介绍 SYCL 及使其能够与 HPX、HPX-Kokkos 以及由此与 Octo-Tiger 异步协同工作所需的软件补充（尤其是 HPX-SYCL 集成）。接着，我们将在第 4 节中使用 Octo-Tiger 对这一集成进行基准测试和测试。第 5 节将概述相关工作。最后，我们将在第 6 节总结全文并提出未来工作的设想。

2 真实世界应用程序作为基准测试：Octo-Tiger

本研究中推动我们开展开发工作的应用程序是 Octo-Tiger，我们的目标是让它运行于 Kokkos SYCL 执行空间。Octo-Tiger 本身也能从这里展示的工作中受益，获得一个额外的执行空间，从而在未来支持范围更广的平台。

由于 GPU 与 CPU 工作紧密交错，Octo-Tiger 也是测试 HPX-SYCL 等集成方案的理想基准测试，即便在小规模的单节点场景下，每个时间步我们也会启动数千个 GPU 计算内核，并与 CPU 预处理和后处理紧密交错执行。此外，主求解器仍包含 CUDA 和 HIP 内核，因此我们能够比较多个后端的性能。

下面我们将介绍 Octo-Tiger，并简要概述其求解器、数据结构和所需的软件依赖。随后，我们将更详细地介绍其基于所有必需软件的执行模型。

2.1 Octo-Tiger：科学应用程序、数据结构与求解器

Octo-Tiger 是一款 C++天体物理学代码，用于研究双星系统及其最终演变结果，例如恒星并合 [21]。这类恒星系统由两颗受引力束缚的恒星组成。当它们彼此足够接近时，便会相互作用，缓慢交换质量。这种质量转移可能是稳定的，持续数百万年；也可能是不稳定的，进而导致其中一颗恒星瓦解。根据系统的不同，最终结果也各不相同，可能演变为 Ia 型超新星，也可能形成另一颗恒星。图 1 展示了使用 Octo-Tiger 进行的一次双星模拟，其中清晰可见质量转移正在发生。此前，Octo-Tiger 已用于研究北冕座R型变星 [24]以及双多方恒星的并合 [15]。

我们将恒星建模为具有自引力的天体物理流体，采用无黏纳维–斯托克斯方程和牛顿引力。因此，Octo-Tiger 包含两个相互耦合的求解器：一个采用有限体积法的流体动力学求解器，以及一个使用快速多极子方法（FMM）的引力求解器 [20] 。这些求解器运行于三维网格上。Octo-Tiger 使用自适应网格细化，在感兴趣区域（通常是两颗恒星之间的大气层）进行最大程度的细化。为此采用的数据结构是自适应八叉树。为了提高效率，我们不让每个树节点仅包含一个单元，而是让每个树节点包含一个由多个单元组成的完整子网格。这些子网格的大小可在编译时配置；我们使用的默认值为 $8 \times 8 \times 8$，每个子网格包含 512 个单元（较大的子网格会对自适应性、数据分布和快速多极子方法性能产生不利影响，因为 FMM 使用树结构进行近似）。

Octo-Tiger 的可移植性突出，此前曾在 NERSC 的 Cori [13], Piz Daint [6]以及橡树岭国家实验室的 Summit [10] 等超级计算机上运行。最近，我们开始将 NERSC 的 Perlmutter 和理化学研究所的富岳超级计算机作为目标平台。正如下一节将要阐述的那样，它通过结合使用 HPX、Kokkos 和其他框架实现了这种可移植性。

2.2 Octo-Tiger 的软件栈

2.2.1 使用 HPX 的任务与分布式计算

HPX 是一种异步多任务运行时系统（AMT） [16]。它本身以 C++ 实现，同时也实现了所有与并行性和并发性相关的 C++ 20 API（例如`hpx::mutex`等功能）。借助 HPX，我们可以使用 HPX future 构建显式任务图。异步操作会返回期值，这些期值可以通过延续链式连接，形成这样的图。这样，每当任务完成，后续工作和通信就会自动触发。HPX 也适用于分布式场景，使我们能够异步调用远程组件上的函数，并获得期值作为返回结果。一旦相应的任务依赖关系得到满足，

任务图中的 HPX 任务就会由 HPX 工作线程池执行（每个任务由一个工作线程执行）；通常，我们会为每个可用的 CPU核心分配一个工作线程。这意味着，即使我们可能有大量可用任务，执行这些任务的工作线程在任一时刻也只有少数几个。
Octo-Tiger 完全构建于 HPX 之上，利用基于任务的编程框架实现高效的树遍历和分布式计算。为此，八叉树中的上述每个子网格都是一个 HPX 组件。该组件包含其子网格所需的所有数据，并通过远程函数调用（经由 HPX ）和 HPX 通道与其他子网格通信，从而填充其幽灵层并协调计算。这样，每个子网格都成为一个独立的计算单元，便于分布。

因此，所有计算内核每次只对一个子网格（及其幽灵层）进行操作。不过，我们通常可以同时为不同子网格调用计算内核。

2.2.2 使用 Kokkos 和 HPX-Kokkos 的可移植计算内核

Octo-Tiger 的计算内核最初是为单核执行编写的，通常会同时运行多个内核，分别处理不同的子网格。后来，这些内核经过移植，以支持 Vc [19, 22],CUDA 和 HIP。为统一这些独立的内核，我们最终选定了 Kokkos。Kokkos 实现了一种编程模型，用于开发可移植内核 [27]。它包含多个执行空间和内存空间，使我们能够在各种设备（例如超威半导体和 NVIDIA 的 GPU）上运行内核。Kokkos 还已包含实验性的 SYCL 执行空间和内存空间，使其用户能够面向英特尔 GPU。

此外，Kokkos 与 HPX 紧密集成：既有 HPX 执行空间（使用 HPX 工作线程执行 Kokkos 内核），也有 HPX-Kokkos 集成层；后者允许为在受支持的执行空间（CUDA、HIP 和 HPX 执行空间）上运行的 Kokkos 内核返回期值 [7]。借助本工作中引入的 HPX-SYCL 集成，受支持的执行空间列表现在也包含 SYCL 执行空间（因为 HPX-Kokkos 直接使用了本工作开发的 获取 future 功能）。

2.2.3 单指令多数据 - KokkosSIMD 和 std::experimental::simd

Kokkos 内核支持使用 C++ 类型 [23]进行显式 SIMD 向量化。在我们自己的内核中，我们使用 Kokkos SIMD 类型[1], , 因此可以实例化这些类型，例如，在为 CPU 编译时使用 AVX512 类型，而在为图形处理器编译时使用标量双精度类型。同一实现也适用于 `std::experimental::simd`，可用于各种 CPU。在之前的工作中，我们研究了这两个类型库之间的性能差异，并为富士通 A64FX™ CPU 添加了 SVE 类型 [9]。

[1]https://github.com/kokkos/simd-math

2.2.4 工作聚合与内存优化 - CPPuddle

鉴于子网格的默认尺寸较小，每个子网格只有 512 个单元，而且计算内核每次只处理一个子网格，因此每个计算内核的工作负载对图形处理器的性能至关重要。仅调用一个这样的内核，工作量远不足以充分利用图形处理器。随着模拟的进行，可能会创建新的子网格，也可能删除或迁移已有子网格（具体取决于模拟演进过程中的自适应网格细化），因此，静态的工作聚合方法（将子网格划分为若干集合，并始终由一个内核处理）也不适用于 Octo-Tiger。

除了增大子网格本身的尺寸（这会对细化产生负面影响）之外，还有两种应对方法：我们可以依靠并发执行足够多的 GPU 内核（使用多个 GPU 执行器），也可以根据 GPU 的负载，在内核进入调度时动态聚合它们。为实现动态聚合，我们在 Octo-Tiger 中引入了工作聚合执行器 [8]。由于这些执行器也可用于其他 HPX 应用程序，我们将其提取为另一个软件依赖项 CPPuddle。该库还包含用于设备端缓冲区的内存池；当对不同子网格反复运行同一任务时，这些内存池十分实用。如果底层 GPU 执行器当前正忙，这些聚合执行器便会在兼容的内核进入调度时将其成批聚集起来（通常是同一个内核，但运行于不同子网格）。达到用户定义的聚合内核数量上限，或底层 GPU 执行器变为空闲、能够立即执行这些内核时，它们就会作为一个更大的内核启动。更多详情请参阅 [8]。

2.2.5 其他依赖项

Octo-Tiger 还使用了一些其他依赖项，例如用于输入和输出文件的 HDF5 和 Silo。HWLOC 和 Boost 等依赖项也会通过 HPX 间接引入。此外，Octo-Tiger 的一些较旧部分仍在使用 Vc，不过我们正在逐步移除这些部分，因为我们已弃用 Octo-Tiger 中对 Vc 的支持，转而使用带有显式 SIMD 类型的 Kokkos 内核。

2.3 Octo-Tiger 的执行模型

例如，在引力求解器的求解器迭代过程中，Octo-Tiger 会遍历包含其网格结构的八叉树。为此，我们只需调用相邻（或子）树节点上执行求解器的方法。得益于 HPX，这些节点实际位于当前计算节点还是远程

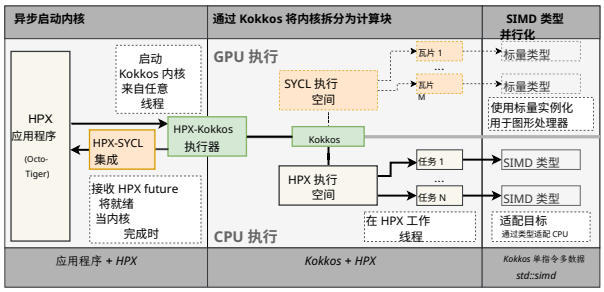


图2：Octo-Tiger 的执行模型，根据 [9]改编，以支持 SYCL 集成及相关执行空间。

节点并不重要，因为每个树节点都是一个 HPX 组件。通过 HPX 调用这些方法会返回期值，我们可以利用它们将相互依赖的任务串联起来。例如，我们可以先收集相邻树节点的数据，以填充当前节点的幽灵层，再利用相关期值串联一个异步 Kokkos 内核启动。
反过来，计算内核会在当前树节点的子网格及其幽灵层上运行。Kokkos 内核本身通过 HPX-Kokkos 异步启动，从而为我们提供另一个期值。每个内核都在 CPU 执行空间或图形处理器执行空间上运行：前者通常是 Kokkos HPX 执行空间，它会将内核拆分为 HPX 任务，由一个或多个工作线程执行；后者则例如 Kokkos CUDA 执行空间。根据执行位置，我们会在内核中实例化适当的单指令多数据类型：在图形处理器上使用标量类型，或在 CPU 上使用支持相应单指令多数据指令的类型，以利用单指令多数据运算和掩码。图2概述了这一执行模型，以及 HPX-SYCL 集成的接入位置。当然，我们已经可以使用 Kokkos SYCL 执行空间，而无需进一步集成，不过只能同步执行。要让上述整套机制像在受支持的执行空间中那样与 SYCL 协同工作，就需要 HPX-Kokkos 能够为底层 SYCL 内核返回一个期值，因此需要 HPX-SYCL 集成。否则，使用期值异步创建任务图的过程就会变成同步操作，从而影响整体性能。

最后，Octo-Tiger 的执行模型还包括图中未显示的一点：我们通常会使用预先分配的 GPU 执行器池，其中每个执行器都能独立执行 GPU 数据传输/内核。我们可以只使用其中一个（完全禁用并发 GPU 内核），也可以使用多达 128 个（理论上还可以更多，但不会带来额外收益）。由于一次处理一个子网格可能不足以充分利用某些设备，即使允许并发 GPU 内核也是如此，因此我们还采用了前文所述的动态工作聚合，利用聚合执行器动态聚合不同子网格上同一内核的内核启动。如 [8]，所示，这对 NVIDIA A100 GPU 有益，对 AMD MI100 GPU 则至关重要。

3 将 SYCL 与 HPX 和 Octo-Tiger 集成

本节介绍我们如何将 SYCL 与 HPX、HPX-Kokkos 以及最终的 Octo-Tiger 集成。我们先简要介绍 SYCL，然后在后续小节中讨论实际的集成工作。首先，我们介绍测试过的两种 HPX-SYCL 集成方案，随后说明对 Octo-Tiger 及其其他依赖项所做的更改。

3.1 SYCL

SYCL 是一种符合 C++ 17 标准、嵌入式的单源领域特定语言（eDSL），为各种加速卡和其他设备提供高级抽象。SYCL 规范由 Khronos 组织负责制定，目前已有多种 SYCL 实现，例如由*Codeplay Software*开发的 ComputeCPP™ 。*hipSYCL* （目前正更名为 Open SYCL）由海德堡大学 [1]开发，支持使用 OpenMP 的所有 CPU 架构、使用 Level Zero 的英特尔 GPU、使用 ROCm 的 AMD GPU，以及使用 CUDA 的 NVIDIA GPU。DPC++ 是 Intel OneAPI 的一部分，支持使用 OpenCL 的英特尔 CPU、使用 Level Zero 或 OpenCL 的英特尔 GPU、使用 AMD ROCm™的 AMD GPU，以及使用 CUDA 的 NVIDIA GPU。值得注意的是，hipSYCL 和 DPC++ 是仅有的支持全部三种 GPU 架构的实现。*triSYCL* 由赛灵思开发，支持使用 OpenMP 或 Intel Thread Building Blocks（TBB）的 Intel 和 AMD CPU，以及使用 OpenMP 的 ARM CPU。此外，还支持通过 OpenCL 使用赛灵思 FPGA。对于本文的研究，我们至少需要一种支持 SYCL 2020 规范 [12]的 SYCL 实现。具体而言，我们需要支持统一共享内存和有序 队列。对 SYCL 主机任务的支持 - 实际上是可选的，因为我们的集成实现中只有一种依赖它。由于 Kokkos 在其 SYCL 执行空间中使用了多种 OneAPI 专属扩展，因此本文后续将重点讨论 DPC++。不过，我们也使用 hipSYCL 对 HPX-SYCL 集成本身进行了测试。

3.2 HPX-SYCL 集成

在这里，我们将进一步探讨 HPX-SYCL 集成。最基本的需求是能够根据 SYCL 事件创建 HPX future。如果事件尚未完成，关联的 future 就不应就绪。事件完成后，也就是说，该事件所需的所有 SYCL 操作都已完成，HPX future 最终应变为就绪状态，从而触发所有依赖此 future 的后续任务。创建此 future 的开销必须很低，并且完全非阻塞，否则我们就无法获得异步 SYCL 内核启动带来的优势。

代码清单 1：使用基本的 HPX-SYCL 集成。

```
sycl :: event event = queue . submit ([&]( sycl :: handler & h) { /* 插入 SYCL 依赖项 */ h. parallel for ( num items , [=]( auto i) { /* 在此处插入数值代码 */ });/ }); // 调用 HPX-SYCL 集成 hpx :: future <void > my future = hpx :: sycl :: experimental :: detail :: get future ( event ); // 添加一个在事件完成后执行的任务 hpx :: future <void > continuation future = my future . then ([& continuation triggered ]( auto && fut ) { /* 插入 CPU 工作 / 通信 / 后处理 */ }); /* 如果内核和续延尚未完成，则挂起当前 HPX 任务。此操作不会阻塞工作线程，只会让其转而处理另一个可用任务。 */ continuation future . get ()
```

有两种不同的实现方式：我们既可以直接使用 SYCL 主机任务，也可以在 HPX 调度器中实现事件轮询方案。

3.2.1 使用 SYCL 主机任务进行集成

SYCL 本身支持创建依赖于 SYCL 事件的主机任务。当所需事件进入状态info::事件命令状态::complete后，便会触发主机任务，并由 SYCL 运行时管理的线程在 CPU 上执行。这个 SYCL 主机任务 可以包含任意 C++ 代码，因此我们可以用它来触发 HPX future。从本质上讲，给定一个 SYCL 事件，我们只需创建一个尚未就绪的期值。通过设置内部 future 数据，我们可以将期值设为就绪状态，从而自动触发用户定义的所有潜在延续。因此，我们将主机任务 用作一种回调，在触发时设置内部 future 数据。

这是最简单的集成形式。不过，它的缺点是必须依赖 SYCL 运行时来高效处理这些异步主机任务的执行。因此，这种方法可能会带来很大的开销（因为所有 CPU 核心此时都应已被 HPX 工作线程占用）。尽管如此，我们仍在此介绍这种方法，因为它似乎是实现集成最直接的方式。然而，结果表明（至少对于 Octo-Tiger 而言），由于前述开销，这并不是一种可行的解决方案。

3.2.2 通过事件轮询进行集成

我们也可以不使用 SYCL 主机任务，而是直接利用 HPX 运行时。由于我们已经有 HPX 调度器，它会在工作线程执行完一个任务、准备获取新任务时被反复调用，因此可以利用它定期轮询 SYCL 事件，并根据事件的执行状态触发相应的回调。当我们需要某个 SYCL 事件对应的期值时，可以先创建一个尚未就绪的期值，并……简单地构造一个期值（此时尚未就绪），并创建一个结构体，其中包含该事件和一个回调 lambda；该 lambda 在执行时会设置期值的数据（从而使期值就绪，并触发可能存在的续延）。我们将这个事件回调 结构体添加到调度器中。为降低开销，我们使用ConcurrentQueue，因为 HPX 已经提供了这种数据结构。借助它，所有线程都可以为不同事件添加各自的事件回调，而无需不必要的锁定。

由调度器调用的轮询函数则有所不同：它每次只会由一个线程执行。也就是说，其他尝试进入的线程不会等待互斥锁，而是立即返回，因为每次只需一个线程进行轮询，因此其他尝试轮询的线程不妨去处理其他任务（只要 HPX 运行时仍在运行，就总会有另一个线程最终前来轮询）。执行轮询函数的线程会尝试从并发队列中获取所有事件，并逐一检查其是否已完成。如果已完成，就执行关联的回调；如果尚未完成，则将事件回调 移入一个向量，以便稍后再次轮询（在轮询函数中，向量中的所有事件也会被轮询一次）。这样，已完成 SYCL 事件对应的 HPX future 最终就会就绪，而无需调用任何阻塞方法，例如事件wait()方法。相反，我们只需在处理其他任务的间隙轮询这些事件；调度器中的集成会自动完成这一操作。图 3 展示了这一过程如何应用于一个期值。代码清单 1 给出了如何使用此功能的示例。

如果我们需要为一个顺序执行的 SYCL 队列获取期值，但又没有可用的 SYCL 事件，我们便提供了一个重载的获取 future，它会插入一个虚拟 SYCL单任务，并使用其 SYCL 事件来获取期值。我们会检查该队列是否顺序执行，因为只有在这种情况下，虚拟内核才会在提交到队列的所有先前操作之后执行，从而使我们能够使用此期值来检查先前提交的命令是否已完成，即使

我们已不再持有它们的事件（例如，这些命令是在外部库中提交的情况）。

我们实现了这一集成（参见拉取请求[2]），并最初使用 DPC++ 和 hipSYCL，通过一些基础的向量加法和流示例进行了测试。要使其正常工作，首先需要解决的难题是使用相应的 SYCL 实现编译 HPX。对于 DPC++，我们只需为 SYCL 相关源文件（例如测试文件和实现事件轮询的源文件）传递 `-fsycl` 标志，并为所有其他文件传递 `-fno-sycl` 标志。最初，我们尝试对 hipSYCL 的 syclcc 包装器采用同样的方法，但未能成功。于是，我们改用 hipSYCL 的 CMake 集成，这种方式开箱即用。

不过，要使集成正常工作，必须考虑 hipSYCL 的默认配置。首先，必须确保 hipSYCL 运行时始终处于运行状态。否则，如果此时没有其他 SYCL 对象存活，整个 SYCL 运行时可能会在轮询函数内部被创建并销毁，因为轮询函数会创建一个临时的事件回调（其中包含一个 SYCL 事件），这会导致 hipSYCL 运行时被重新创建。其次，必须确保内核确实已启动：hipSYCL 的默认调度器支持在多个设备之间自动进行工作分配，因此可能会延迟启动内核，因为我们不会等待事件或命令队列。

只需针对这种用法正确配置 hipSYCL，就能轻松解决这两个问题，因为可以通过适当的环境变量来调整其行为。我们可以将调度器设为直接调度器，以直接启动任何内核/命令，方法是设置 `HIPSYCL_RTSCHEDULER=direct`；同时，设置 `HIPSYCL_PERSISTENT_RUNTIME=1`，确保 hipSYCL 运行时持续运行。不过，为了确保用户无需记住设置这些变量，我们在 HPX 调度器中添加了一个额外的命令队列，只要 HPX 运行时处于运行状态，该队列就会保持运行时处于活动状态。此外，我们还可以使用此队列触发 hipSYCL 中的刷新方法，从而使集成也能与其默认调度器配合工作。

3.2.3 HPX SYCL 执行器

在这种基础的 `getfuture` 集成之上，我们添加了一个 SYCL HPX 执行器。该执行器封装了一个按顺序执行 - SYCL 命令队列，并通过将函数调用封装在`hpx::async` 中来调度它们（双向执行，这意味着它会返回一个 `hpx::future`）或`hpx::apply`（单向执行，不返回期值）。此执行器只接受 SYCL 队列成员函数，并将它们直接传递给底层 SYCL 队列（因此无需手动获取这些调用的期值）。这样做主要是为了方便用户，同时也使其功能与

[2]https://github.com/STEllAR-GROUP/hpx/pull/ 6085

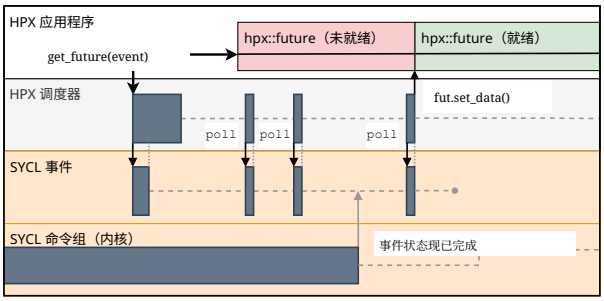


图 3：事件轮询工作方式概览。轮询函数会在其他任务之间触发。关联的 SYCL 事件完成且下一次轮询检测到该事件后，HPX future 就会就绪。将期值设为就绪状态会自动触发任何可能的 HPX 延续。

HPX 中同样支持`hpx::async` 和`hpx::apply`的 CUDA 和 HIP 执行器保持一致。事实上，我们为 SYCL 执行器使用按顺序执行的命令队列，以使其行为与这些其他执行器相同。

与上一节介绍的基本集成一样，这个执行器在使用特定 SYCL 实现编译时也需要进行一些调整。这次，我们需要针对 DPC++进行调整，因为要在此处选择正确的成员函数重载，必须考虑 DPC++ 使用的内部代码位置参数（因此，在检测到使用 DPC++编译时，还需要添加考虑该参数的重载）。

3.2.4 与其他 HPX GPU 执行器的区别

HPX 还包含一个 CUDA 执行器（使用 hipcc 编译时，它也可充当 HIP 执行器）。该执行器还支持事件轮询（当然，也支持 `hpx::async` 和 `hpx::apply`）。它还支持直接使用 CUDA 回调函数。事实上，我们以该执行器为蓝本，实现了 SYCL 执行器和事件轮询 SYCL 集成。不过，两者之间存在一些差异：除了要让所有内容都能通过 SYCL 编译，并更改 SYCL API 调用外，SYCL 执行器（及其集成）还必须兼容不同的 SYCL 实现，因此需要额外的逻辑来适配它们（例如前面提到的变通方法，我们必须据此适配 hipSYCL 或 DPC++）。此外，我们无法实现类似 CUDA 执行器所用的事件池，因为 SYCL 本身不支持在调用中使用我们自有的事件（这样本可以重复使用事件，避免可能开销较大的创建和销毁操作）。因此，我们只能使用 SYCL 队列返回的事件，并希望所用的 SYCL 实现会在内部使用事件池，以降低所需开销。一方面，如果所用的 SYCL 运行时确实支持这一点，就能带来好处，因为我们无需自行实现：事实上，从 0.9.3 版开始，hipSYCL 已为 CUDA 和 HIP 事件提供事件池。另一方面，

如果运行时不支持事件池，我们就无法在 HPX 中自行添加事件池。

3.3 其他软件更改

要让 Octo-Tiger 能够与 SYCL 配合运行，我们还需要进行更多软件方面的补充。

3.3.1 HPX-Kokkos

首先，我们需要修改 HPX-Kokkos 本身（参见拉取请求[3]）。如前所述，HPX-Kokkos 可以为 Kokkos 内核启动返回 HPX 期值，但仅限于受支持的执行空间。因此，我们需要将 SYCL 执行空间添加到受支持的执行空间列表中。这需要添加重载：对于不受支持的执行空间，默认行为是执行栅栏并返回虚拟 future；而我们需要调用 HPX 中正确的获取 future方法。为此，只需将此处的获取 future 调用映射到第 3.2.3 节所述的调用，即可接入其中介绍的基本 HPX-SYCL 集成。我们还需要正确构造 HPX-Kokkos SYCL 执行器实例。请注意，这种执行器不同于上文提到的 HPX-SYCL 执行器；它只是 Kokkos SYCL 执行空间的封装。为使其行为与其他 Kokkos 执行空间保持一致，我们使用按序 命令队列来构造每个 SYCL 执行空间。最后，我们为异步深拷贝 添加一个额外的重载，直接使用内存复制产生的 SYCL 事件来获取关联的 future。

3.3.2 CPPuddle

此外，我们还需要修改 CPPuddle（参见拉取请求[4]）。Octo-Tiger 使用 CPPuddle 管理设备内存池，通过重用已不再使用的旧内存分配，避免不必要的内存分配。这对需要使用大小相近的输入缓冲区的重复 GPU 任务很有帮助。在这里，我们只需添加适当的分配器，即可通过 SYCL USM 内存分配获取设备/主机内存池。

3.3.3 Octo-Tiger

最后，Octo-Tiger 本身也需要进行一些修改，才能与 SYCL 配合使用（参见拉取请求[5]）。在这里，我们切换到适当的 HPX-Kokkos 执行器（底层使用 Kokkos SYCL 执行空间），并使用基于 SYCL 统一共享内存的 CPPuddle 分配器。不过，运行 Octo-Tiger 的标准测试套件时，我们仍然发现使用 SYCL 运行会产生偏差。使用 SYCL 数学函数而非标准库中的函数（即使用 `sycl::sqrt` 而非 `std::sqrt`）解决了这个问题。

3.3.4 Kokkos

虽然要让 Octo-Tiger 在 SYCL 下运行并不一定需要这些修改，但我们也尝试对 Kokkos 做了一些调整。首先，由于我们使用的是有序队列，我们发现可以移除 Kokkos 的 SYCL 执行空间中的一些屏障。此外，为了在 AMD GPU 上使用 Kokkos SYCL 执行空间，我们需要修改 Kokkos 的 CMake 配置，以便正确传递参数。这些更改尚未上游合并，因为我们想先进一步完善并测试。不过，移除部分栅栏的补丁已公开[6]，另一个[7]用于启用 AMD GPU 上的 Kokkos 构建。

4 结果

在本节中，我们评估第 3 节所介绍的软件更改和新增内容对性能的影响。首先介绍测试设置，具体包括所采用的 Octo-Tiger 场景、硬件和软件版本。随后说明实验中使用的参数和补丁，并接着介绍性能测试。

4.1 测试设置

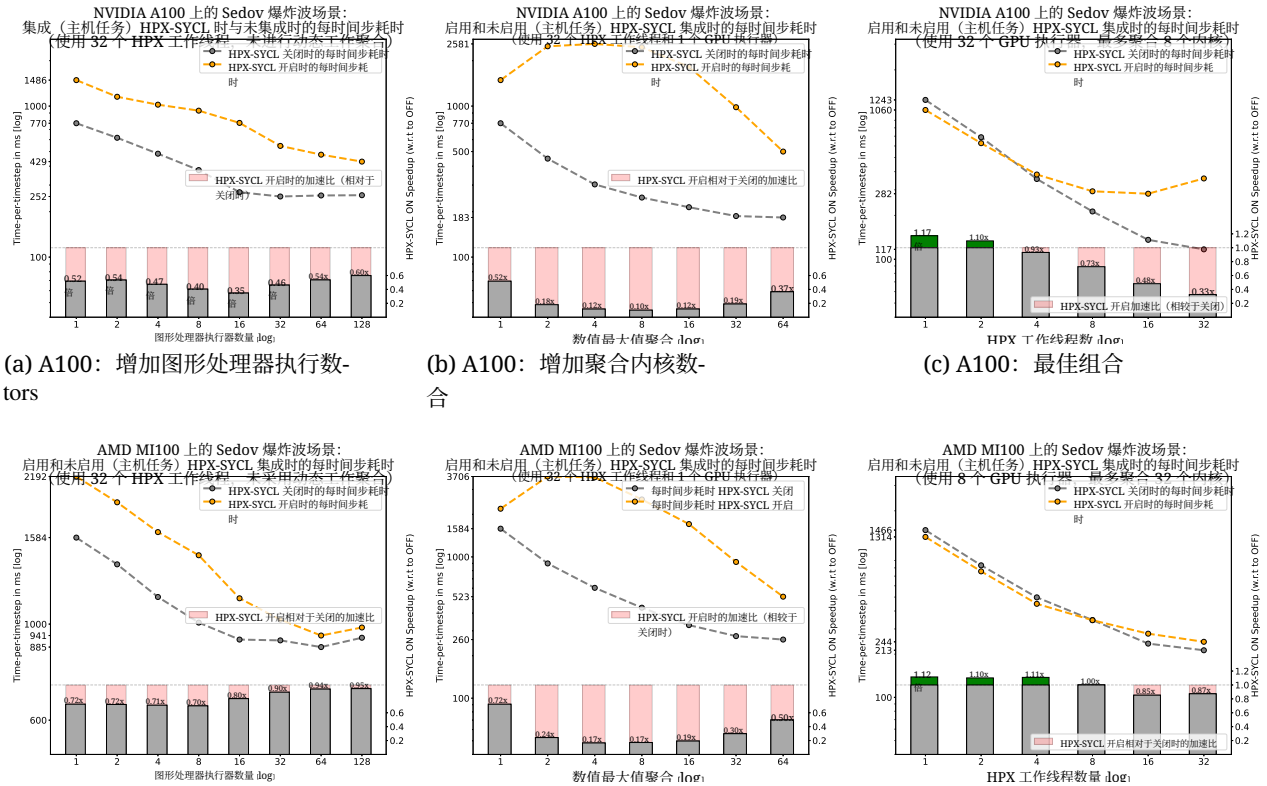
4.1.1 场景与硬件：

我们选择 Sedov–Taylor 爆炸波作为基准测试场景。该场景是最初用于验证 Octo-Tiger 输出的基准测试之一。它仅使用流体动力学求解器，并且具有解析解，因此非常适合测试 Octo-Tiger 这类代码。就我们的目的而言，仅使用流体动力学求解器很有用，因为目前只有该模块完全支持动态工作聚合。这样，我们便能精细调整所启动计算内核的大小（GPU 执行器忙碌时，将子网格分批聚合起来，并在执行器完成先前工作后，将其作为一个内核启动）。因此，我们能够更轻松地考察集成方案在不同大小内核下的开销。引力求解器对这种工作聚合方式的支持尚未完成，因此我们只能采用仅涉及流体动力学的场景。

该场景所需的计算工作量如下：其中包含 512 个叶子子网格，总计 262144 个单元。每个时间步，我们会为每个叶子子网格调用 15 个 GPU 内核，因此每个时间步总共调用 7680 次 GPU 内核（并进行 15360 次 CPU-GPU 数据传输）。

[3] https://github.com/STEllAR-GROUP/hpx-kokkos/pull/13 [4] https://github.com/SC-SGS/CPPuddle/pull/15 [5] https://github.com/STEllAR-GROUP/octotiger/ pull/432

[6] https://github.com/STEllAR-GROUP/OctoTigerBuildChain/blob/sycl_工具链/kokkos_sycl_less_fencing.patch [7] https://github.com/STEllAR-GROUP/OctoTigerBuildChain/blob/sycl_工具链_hip/ kokkos_hip_arch.patch



(a) A100：增加图形处理器执行数-tors

(b) A100：增加聚合内核数-合

(c) A100：最佳组合

(d) MI100：增加图形处理器执行数-执行器

(e) MI100：增加内核数量并进行聚合-聚合

(f) MI100：最佳组合

图 4：开启 SYCL 集成（主机任务版本）（橙色）和关闭（灰色）时的运行情况。在 4a 和 4d 中，我们不使用动态工作聚合，而是只增加 GPU 执行器的数量。在 4b 和 4e 中，我们只使用一个执行器，但增加聚合到一次内核启动中的内核最大数量。在 4c 和 4f 中，我们可以看到最佳组合。所有运行均应用了 Kokkos 优化补丁。

这是假设关闭了上文所述的动态工作聚合，也进一步说明了为什么我们一开始就需要在流体力学模块中采用它。每个时间步的实测运行时间是 15 个时间步的平均值，所有模拟均采用双精度。此前的工作中，在完全相同的场景下（使用相同硬件）测得的最佳计算时间为 86.6 毫秒，测试设备为 NVIDIA® A100 [8]。该运行时间不仅包括所有 GPU 内核，还包括 CPU-GPU 数据传输、CPU 上的后处理、GPU 内核的调度、时间步长的确定，以及值得注意的是，工作聚合所需的全部逻辑。因此，这是一个试图运行大量小型内核的场景：GPU 忙于运行这些内核，而 CPU 则忙于调度它们，并可能对其进行聚合。

我们在两个节点上运行这一场景：第一个节点配备一块 NVIDIA A100 图形处理器和一颗 Intel® Xeon® Platinum 8358 CPU。第二个节点配备一块 AMD MI100 图形处理器和一颗 AMD EPYC ™ 7H12 CPU。两台机器上的 CPU 均使用 32 个 HPX 工作线程，以使运行结果更具可比性（因此将 HPX 在两台机器上都限制为使用 32 个 CPU 核心）。所用软件版本（*git commits*）见表 1。具体而言，我们使用 DPC++ （OneAPI）作为编译器和 SYCL 实现，因为 Kokkos SYCL 执行空间会使用 OneAPI 扩展，例如 SYCL 扩展 oneAPI 入队屏障和 `sycl ext intel usm address spaces`。

4.1.2 参数和配置：

我们通过为 Octo-Tiger 的依赖项应用补丁，在多种软件配置下运行 Octo-Tiger。
第一个补丁[8] 只是关闭了 HPX 中的 HPX-SYCL 集成，以便我们评估其带来的好处。为了确保所有内容都能正确编译，接口必须保持不变，因此，对于给定的 SYCL 事件，需要返回一个 HPX future。关闭集成的补丁会让 HPX 等待 SYCL 事件，然后才通过 `hpx::make ready future()` 返回一个就绪的期值，从而实际上将异步操作变成同步操作。

[8] https://github.com/STEllAR-GROUP/OctoTigerBuildChain/blob/sycl_toolchain/remove_hpx_sycl_集成.patch

Boost	1.75.0	HWLOC	2.7.1
HDF5	1.8.12	Silo	4.10.2
Kokkos	*23a5e94*[1]	HPX-Kokkos	*6f6b655*[2]
JEMALLOC	5.2.1	HPX	*5641895*[3]
CPPuddle	*969902f* [4]	Octo-Tiger	*e969470*[5]
ROCm	5.2.0	CUDA	11.7
DPC++/Intel OneAPI	44c6437684d6		

表 1：软件版本　　实验中使用的版本。由于我们使用了多个实验性拉取请求，因此我们添加了 th　　e 准确的 Git 提交以及相关的拉取请求。

1 部分内容见 https://github.com/kokkos/kokkos/pull/5628 2 部分内容见 https://github.com/STEllAR-GROUP/hpx-kokkos/pull/13 3 部分内容见 https://github.com/STEllAR-GROUP/hpx/pull/6085 4 部分内容见 https://github.com/SC-SGS/CPPuddle/pull/15 5 部分内容见 https://github.com/STEllAR-GROUP/octotiger/pull/432

除非另有说明，默认情况下，以下测试不会应用此补丁（HPX-SYCL 关闭）。

第二个补丁包含前文提到的 Kokkos 优化。在集成的早期测试中，我们发现 Kokkos SYCL 后端包含多个屏障（SYCL 扩展 oneAPI 入队屏障），这削弱了集成带来的收益。幸运的是，我们还发现，使用按顺序 队列时可以去掉其中一些屏障，因为这种队列属性本身就能保证相同类型的顺序。除非另有说明，默认情况下，以下测试会应用此补丁。

我们应用的第三项更改，是在基于主机任务 的 HPX-SYCL 集成与基于事件轮询的集成之间切换。这项更改在 HPX-Kokkos 的编译时完成，因为我们正是在此处调用基本的获取 future 功能。根据此处的配置，我们会调用事件轮询版本或主机任务 版本。

我们考虑的其他参数包括 HPX 工作线程的数量、GPU 执行器的数量，以及单个执行器可以聚合在一起的 GPU 内核最大数量。首先，HPX 工作线程的数量决定了有多少 CPU 线程总体上在处理可用的 HPX 任务，因此该参数实际上控制着 HPX 使用的 CPU 核心数量。其次，GPU 执行器的数量决定了可以并发运行的 GPU 内核/数据传输的数量。最后，GPU 内核聚合的最大数量需要更详细地说明：当遇到一个可聚合的内核（由程序员标记）时，我们会暂停当前任务，并等待其他线程在不同子网格上遇到同一个内核。随后，如果遇到该内核的任务数量达到上限（即最大聚合参数），或者底层 GPU 执行器变为空闲状态（这可能会使聚合后的内核提前启动，从而避免由子网格数量为奇数导致的死锁），我们就会将所有这些任务作为一个更大的聚合内核启动。这种聚合会增大实际在 GPU 上运行的内核规模，从而避免大量细小的计算内核使 GPU 无事可做；同时，由于所需的 API 调用总数更少，也减轻了 GPU 运行时的负担。

动态工作聚合也会影响调用获取 future功能的频率。对于每个叶子子网格，我们会运行五个独立的 GPU 内核，然后才需要将结果传回主机进行后处理，并将结果传递给邻居。

启用工作聚合（即 最大值大于一个内核）后，在聚合过程中遇到的第一个子网格开始时，我们会额外调用一次获取 future （聚合执行器会在内部使用此 future 来检测所用的 GPU 流何时变为空闲）。不过，这仅在每个聚合内核上执行一次。

另一次获取 future 调用用于传递聚合内核的结果。这意味着，当聚合内核的最大数量超过两个时，我们实际上减少了对 HPX-SYCL 集成的调用次数。

4.2 性能测试

在前两项性能测试中，我们先运行启用了 HPX-SYCL 集成的 Octo-Tiger（第一项测试使用主机任务，第二项测试使用事件轮询），然后再使用禁用该集成的 HPX 补丁运行一次，以作比较。之后，我们考察第 3.3.4 节提到的实验性 Kokkos 优化启用和未启用时的性能。最后，我们简要比较这些 Kokkos SYCL 运行与使用 Kokkos CUDA/HIP 执行空间以及 Octo-Tiger 中原生 CUDA/HIP 内核运行时的性能。

4.2.1 测试 1 - 基于 HostTask 的 HPX-SYCL 集成对性能的影响

在第一项测试中，我们考察第 3.2.1 节所述集成的 主机任务 版本。我们在启用和未启用集成的情况下运行 Sedov 爆炸波场景。结果见图 4。与后续测试一样，我们通常考察三种不同的配置：对于第一张图（4a），HPX 工作线程的数量（因而 HPX 使用的 CPU 核心数）固定为 32，并禁用动态工作聚合。随后，我们逐步增加 GPU 执行器的数量，直至达到 128。这样，我们便能观察多个 CPU 核心尝试通过数量不等的 SYCL 命令队列使用该集成时的影响（每个 GPU 执行器都包含一个底层的 依次 SYCL 命令队列）。

在第二张图（4b）中，我们只使用一个 GPU 执行器，但启用动态工作聚合。这样最多可以将 64 个内核聚合到一个更大的聚合 GPU 内核中。不过，通常聚合到单个内核中的内核数量会低于这一上限，因为一旦执行器空闲，聚合内核就会立即启动（即使尚未达到最大聚合容量）。

最后，在第三张图（4c）中，我们考察前两个参数的最佳组合（在 A100 上使用 32 个 GPU 执行器，每次最多聚合 8 个内核）。随后，我们不仅使用 32 个 HPX 工作线程运行这一组合，也尝试使用更少的线程。这通常是最值得关注的测试，因为它不仅测试了我们最有可能在生产运行中采用的配置，还通过减少工作线程，人为降低 CPU 性能；这反过来应该会使 HPX-SYCL 集成更有价值，因为等待 SYCL 结果时（关闭集成的情况下）会阻塞 CPU 线程。另外三张图（4d、4e、4f）对应 MI100 节点，展示了在这台机器上进行的相同实验。这部分实验设置在接下来的两项测试中也保持不变。虽然这两项测试会改变软件配置，但我们改变的参数保持相同。

总的来说，观察测试 1 中各图表的加速比柱状图，可以明显看出，这种集成实际上会显著降低性能，最可能的原因是 SYCL 运行时在内部使用不同的线程处理主机任务。这一点可以从两个方面看出：在图 4c 和 4f 中，使用较少的 HPX 工作线程时，集成带来的加速比更高，因为 HPX 工作线程与 SYCL 运行时的线程之间竞争较少。此外，在严重依赖动态工作聚合的场景（4b、4e）中，我们还观察到更明显的性能下降，因为 CPU 在这里还必须管理工作聚合调度。有趣的是，在 MI100 上结合使用 GPU 执行器和工作聚合效果最佳的运行（4f），在开启集成时，性能优于 A100 上的对应运行（4c）开启后。我们计划在未来的工作中对此作进一步研究。不过，即使在这里，这种集成的益处也不大：只有使用较少的 HPX 工作线程（因此核心数也较少）时，我们才能看到收益。在大多数配置中，这种主机任务集成会损害性能。

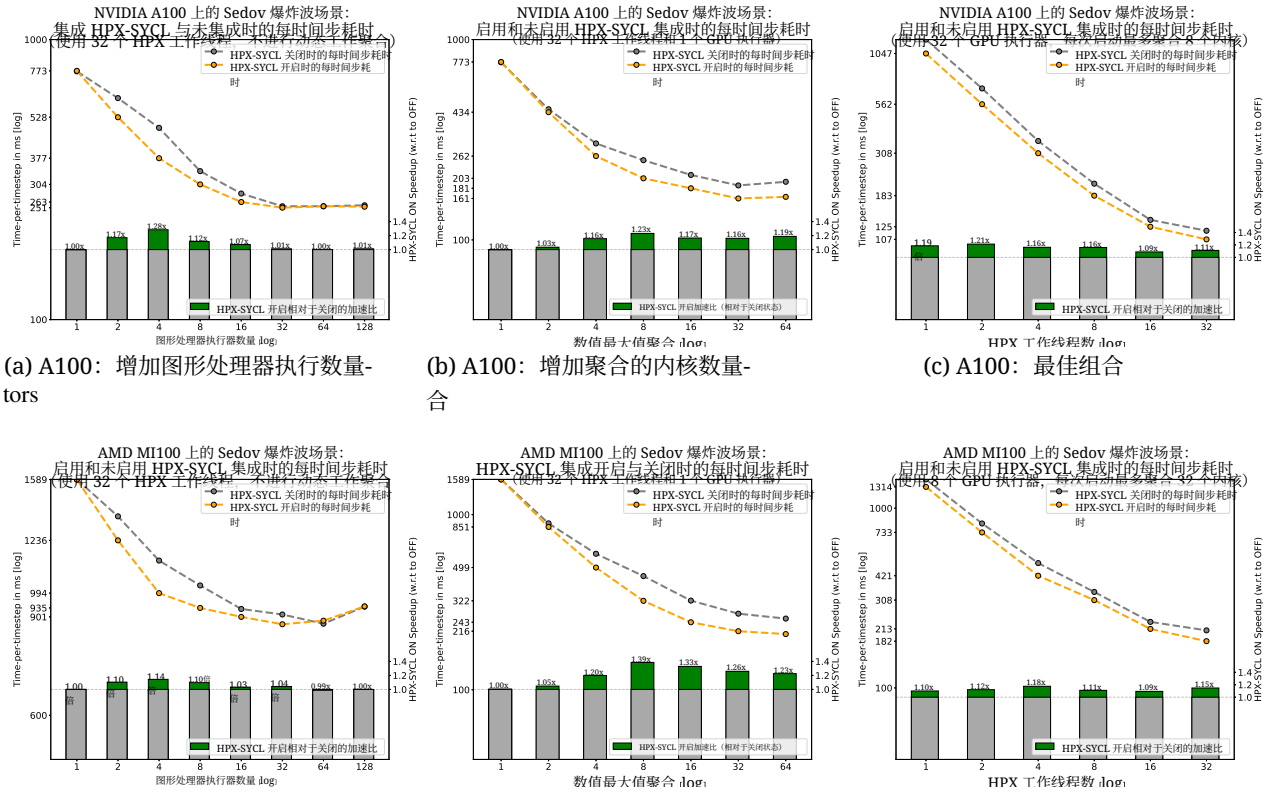
4.2.2 测试 2——事件轮询 HPX-SYCL 集成的性能影响

鉴于主机任务 集成的结果令人失望，我们显然需要寻找替代方案。我们实现的事件轮询版本没有同样的缺点，因为所有工作都由 HPX 线程自身处理，且针对多线程使用进行了优化。不过，持续轮询会带来另一种开销，因此也有必要通过启用和禁用该集成分别运行一次 Octo-Tiger，对其进行检验。

因此，我们现在使用集成的事件轮询版本，重复上一节中的相同测试。结果见图 5。这一次，我们取得了明显的加速比，尤其是在仅使用一个 GPU 执行器（5b、5e）并启用工作聚合的图中。此时，我们从聚合中获益最大，因为随着工作线程在执行其他任务的同时忙于协调工作聚合，CPU 时间变得越来越宝贵。即使是组合效果最佳的图（5c、5f），我们也能看到明显的收益。使用较少的 HPX 工作线程时收益略大一些；不过，即使使用全部 32 个工作线程和最佳组合，我们在 A100 节点上仍获得了 $1.11x$ 的加速比，在 MI100 节点上则获得了 $1.15x$ 的加速比。请注意，MI100 节点上的最佳组合与 A100 节点不同，因为我们从 AMD GPU 上并发使用多个 GPU 执行器中获益较少，因此只使用 8 个 GPU 执行器，并将最多 32 个内核聚合起来。我们在这台机器上使用 HIP 进行的先前测试中也观察到了类似现象，这正是促使我们最初实现动态工作聚合执行器 [8] 的原因之一。

4.2.3 测试 3——Kokkos 修改的性能影响

前两项测试探讨了 HPX-SYCL 集成，但始终应用了 Kokkos 优化补丁（无论是否启用集成）。在本项测试中，我们始终启用（事件轮询）HPX-SYCL 集成，但会切换是否使用 Kokkos 修改补丁。如前所述，该补丁基本上只是在 Kokkos SYCL 执行空间使用顺序队列时跳过几个屏障。将其用于 Octo-Tiger 时，我们没有发现测试的实际结果有任何偏差。不过，它对性能产生了影响。结果见图 6。该补丁始终有益，在最佳组合运行中，加速比约为 1.06 和 1.12。A100 节点上未采用动态工作聚合的加速比（6a）尤为突出。在 NVIDIA 硬件上，我们能更多地受益于并发 GPU 执行器，因此，让它们更充分地获得内核（而不受阻塞）



(a) A100：增加图形处理器执行数量-tors

(b) A100：增加聚合的内核数量-合

(c) A100：最佳组合

(d) MI100：增加图形处理器执行数量-执行器

(e) MI100：增加内核数量并聚合——聚合

(f) MI100：最佳组合

图 5：开启（橙色）和关闭（灰色）SYCL 集成（事件轮询版本）时的运行结果。在 5a 和 5d 中，我们不使用动态工作聚合，而是只增加 GPU 执行器的数量。在 5b 和 5e 中，我们只使用一个执行器，但增加聚合到一次内核启动中的内核数量上限。在 5c 和 5f 中，我们展示了最佳组合。所有运行均应用了 Kokkos 优化补丁。

能带来更大的优势。总体而言，在所有测试配置中，应用该补丁都有益处。

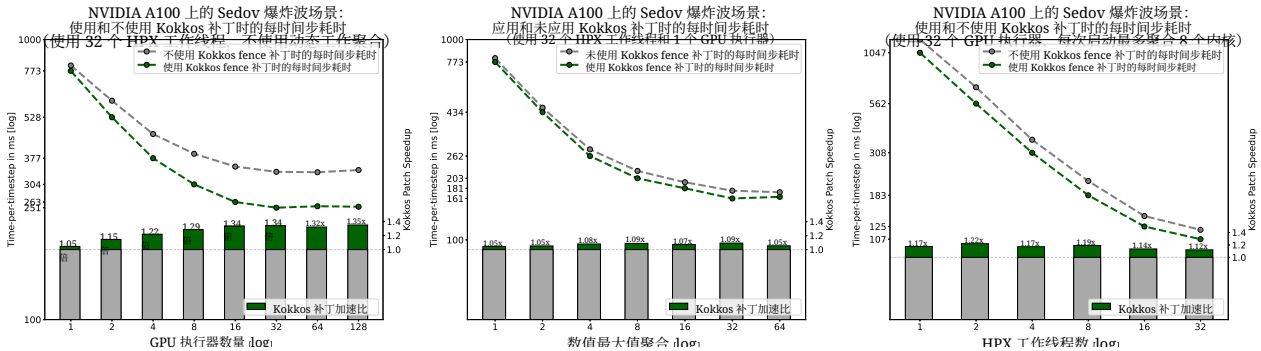
4.2.4 测试 4——使用不同执行后端的性能

在 A100 节点上，我们现在可以为内核使用多个后端：原生 CUDA、使用 CUDA 执行空间的 Kokkos，以及使用 SYCL 执行空间的 Kokkos。为此，有必要在相同场景和相同硬件上，进一步比较这些不同后端运行 Octo-Tiger 的性能。相关结果见图 7（涵盖 A100 和 MI100 节点）。HPX-SYCL 集成（事件轮询）已开启，并且已应用 Kokkos 优化补丁。有趣的是，在这种场景下，Kokkos SYCL 后端似乎可以与其 CUDA 等效版本相媲美。从性能分析器输出中可以看出，SYCL 的平均内核运行时间实际上更短。不过，由于 HPX-CUDA 集成使用事件池，而我们必须从 SYCL 运行时获取 SYCL 事件，因此频繁创建和销毁这些 SYCL 事件又带来了一些运行时开销，导致性能有所损失。值得注意的是，这里的 Kokkos CUDA 执行空间版本比之前的工作稍慢；在之前的工作中，我们使用的是 Clang 12，而非 DPC++ [8]，并且使用的是较旧版本的 Octo-Tiger，其中 Kokkos 内核尚未包含 SIMD 类型（这可能会给图形处理器执行带来一些额外开销）。还值得注意的是，无论使用哪种图形处理器后端，Octo-Tiger 在 A100 图形处理器上的表现都优于在 MI100 上的表现。这表明内核本身还需要针对超威半导体图形处理器进行更多优化。

5 相关工作

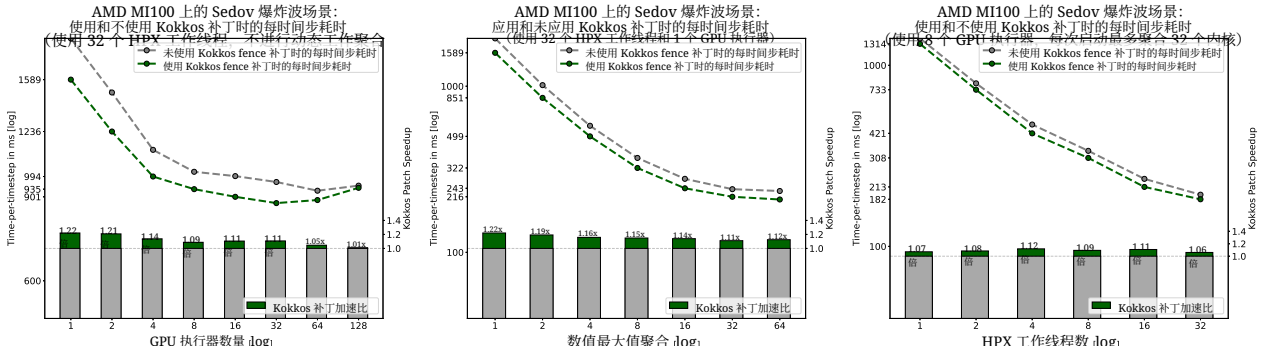
从应用程序的角度来看，其他天体物理学代码也支持 SYCL 抽象层。*DPE-cho*[9]，是一种用于广义相对论磁流体动力学的代码，使用 Intel MPI 和 hipSYCL。*ARGOT*是一种辐射传输代码，使用 Intel DPC++ 支持 GPU（Intel CPU + NVIDIA GPU）和 Intel FPGA [18]。然而，

[9]https://github.com/LRZ-BADW/DPEcho



(a) A100：不断增加图形处理器执行器的数量 (b) A100：不断增加聚合的内核数量 (c) A100：最佳组合

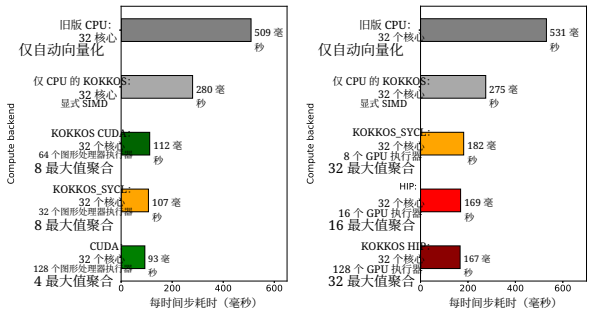
tors 聚合



(d) MI100：不断增加图形处理器执行器的数量 (e) MI100：内核数量递增，聚合 (f) MI100：最佳组合

执行器

图 6：应用 Kokkos 优化补丁（绿色）与未应用补丁（灰色）的运行结果。在 6a 和 6d 中，我们不使用动态工作聚合，而仅增加 GPU 执行器的数量。在 6b 和 6e 中，我们只使用一个执行器，但增加聚合到一次内核启动中的内核最大数量。在 6c 和 6f 中，我们可以看到最佳组合。所有运行均启用了 HPX-SYCL 集成（事件轮询版本）。



(a) NVIDIA 上的最佳运行结果 A100 (b) AMD 上的最佳运行结果 MI100

图 7：在 A100 节点 (7a) 和 MI100 节点 (7b) 上，使用 Octo-Tiger 和所有可用后端运行 Sedov 爆炸场景。我们为每个后端选用了最佳组合，并添加了仅使用 CPU 的运行结果以供比较。

在探讨 SYCL 如何集成到异步多任务系统时，我们重点关注 SYCL API 异步任务的集成。也可以只在一个线程上调用 SYCL API，并在等待事件时阻塞该线程。不过，我们关注的是将 SYCL API 调用集成到异步执行图中。自 10月21日以来，Chapel [4] 便将 Intel DPC++ 作为其 GPU API 模块之一。Unitah [11] 通过 Kokkos SYCL [14]支持 SYCL。Chiu 等人将 SYCL 默认任务图与 CUDA 图执行模型[10] 在大规模机器学习工作负载中的表现进行了比较 [5]。其他值得关注的异步多任务系统包括：Charm++ [17]、Legion [2], 和 PaRSEC [3]。 [25]中给出了详细比较。聚焦于编程模型，Charm++ 和 HPX 非常相近；不过，HPX 符合 C++ 标准，而 Charm++ 是一个使用 C++ 编程语言的库。 [28]比较了使用 HPX 和 Charm++时的开销与使用 MPI 和 OpenMP 时的开销。

这两种代码的应用程序（例如，黑洞和宇宙学）与 Octo-Tiger 不同。

[10]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs

6 结论与未来工作

在这项工作中，我们开始使 Octo-Tiger 和 HPX 本身兼容 SYCL。为此，我们引入了多项软件变更，其中最重要的是 HPX-SYCL 集成。

起初，将这两个框架集成起来似乎是个奇怪的选择，因为它们都具有用于表达执行图的类似功能（一个使用 SYCL 事件，另一个使用 HPX future）。然而，这样做也有优势，因为这两个框架各有所长：借助 SYCL，我们可以快速构建 GPU 内核与数据传输之间的异步依赖关系，如 Octo-Tiger 的流体求解器所示。在该求解器中，我们会先为每个子网格调度 5 个 GPU 内核及其相关的 CPU-GPU 数据传输，然后 HPX 才会再次介入。另一方面，HPX 擅长调度异步 CPU 工作，无论是在本地计算节点还是远程节点上。例如，在使用 SYCL 调度 GPU 工作后，我们通过 HPX 获取一个期值，以确定某个子网格的 GPU 工作何时完成，然后利用这个期值来调度 CPU 后处理以及与相邻子网格的通信，同时始终不阻塞 CPU 主机线程。尽管本项工作只考察了单节点运行，但在分布式运行中，这一点应该会变得更加重要，因为随着使用的节点增多，计算与通信的交错执行会愈发重要，尤其是在 Octo-Tiger 这类基于树的代码中。

虽然我们使用 主机任务 SYCL 功能和事件轮询两种方式实现了 HPX-SYCL 集成，但最终，在 HPX 调度器中使用事件轮询的版本在我们的测试中明显更胜一筹。总体而言，使用该集成时，Octo-Tiger 的加速比提升幅度不大，但效果显著；在最佳配置下（使用 32 个 HPX 工作线程，并采用 GPU 执行器与最大聚合内核数量的最佳组合），加速比范围为 $1.11x$ 至$1.15x$ 。此外，我们使用的 Kokkos 优化补丁颇具潜力，不过在提交时，我们认为它还需要在 Octo-Tiger 以外的代码上进一步测试，然后再向上游提交。结合这些 Kokkos 改动与 SYCL 集成，Octo-Tiger 在使用 Kokkos SYCL 执行空间时的整体运行时行为已可与使用 Kokkos CUDA 执行空间时相媲美。

在未来的工作中，我们希望利用本文奠定基础后带来的两个机会：首先，由于我们更新并优化了 Octo-Tiger 的工具链，使其能够与 SYCL 配合使用，因此现在可以切实地将即将推出的英特尔 GPU 用于未来的 Octo-Tiger 运行。因此，我们希望将来在相应的英特尔硬件上重新运行此处展示的测试。其次，使用 HPX 的一项主要优势是它支持分布式运行。我们希望在 Perlmutter 上更复杂的分布式场景中，重新评估 HPX-SYCL 集成的益处，因为等待 SYCL 结果而阻塞核心（未采用集成时）可能会对性能造成更显著的负面影响，因为这会影响计算与通信的交错执行。

（未采用集成时）可能会对性能造成更显著的负面影响，因为这会影响计算与通信的交错执行。

参考文献

[1] Aksel Alpay 和 Vincent Heuveline。Sycl 超越 OpenCL：hipSYCL 的架构、现状与未来方向。载于 *OpenCL* 国际研讨会论文集，第 1–1 页，2020 年。[2] Michael Bauer 等人。Legion：借助逻辑区域表达局部性与独立性。载于 *SC'12*：国际高性能计算、网络、存储与分析会议论文集，第 1–11 页。IEEE，2012 年。[3] George Bosilca 等人。PaRSEC：利用异构性提升可扩展性。《科学与工程计算》，15(6)：36–45，2013 年。[4] Bradford L Chamberlain 等人。并行编程能力与 Chapel 语言。《国际高性能计算应用期刊》，21(3)：291–312，2007 年。[5] Cheng-Hsiang Chiu 等人。面向大规模机器学习工作负载的 SYCL 任务图并行实验研究。载于 *Euro-Par 2021*：并行处理研讨会：*Euro-Par 2021* 国际研讨会，葡萄牙里斯本，*2021* 年 *8* 月 *30–31* 日，修订版精选论文，第 468–479 页。施普林格，2022 年。[6] Gregor Daiß 等人。从 Piz Daint 到星辰：使用高级抽象模拟恒星并合。载于 国际高性能计算、网络、存储与分析会议论文集，SC '19，美国纽约州纽约市，2019 年。计算机协会。[7]Gregor Daiß 等人。超越分叉-汇合：将性能可移植的 Kokkos 内核与 HPX 集成。载于 *2021* 年 *IEEE* 国际并行与分布式处理研讨会研讨会（*IPDPSW*），第 377–386 页。IEEE，2021 年。[8]G. Daiß、P. Diehl、D. Marcello、A. Kheirkhahan、H. Kaiser 和 D. Pfl¨uger。从基于任务的 GPU 工作聚合到恒星并合：将细粒度 CPU 任务转化为可移植的 GPU 内核。载于 *2022* 年 *IEEE/ACM* 高性能计算性能、可移植性与生产力国际研讨会（*P3HPC*），第 89–99 页，美国加利福尼亚州洛斯阿拉米托斯，2022 年 11 月。IEEE 计算机学会。[9] G. Daiß、S. Singanaboina、P. Diehl、H. Kaiser 和 D. Pfl¨uger。从合并框架到合并恒星：使用 HPX、Kokkos 和 SIMD 类型的经验。载于 *2022* 年 *IEEE/ACM* 第七届极限规模编程模型与中间件国际研讨会（*ESPM2*），第 10–19 页，美国加利福尼亚州洛斯阿拉米托斯，

CA, USA, nov 2022. IEEE Computer Society.

[10] Patrick Diehl 等。Octo-tiger 的新流体力学模块及其在橡树岭国家实验室 Summit 上使用 hpx+cuda 的性能。收录于*2021 年 IEEE 国际集群计算会议（CLUSTER）*，第 204–214 页。IEEE，2021 年。[11] J Davison de St Germain 等。Uintah：一个大规模并行的问题求解环境。收录于第九届国际高性能分布式计算研讨会论文集，第 33–41 页。IEEE，2000 年。[12] Khronos SYCL工作组。SYCL 2020 规范（第 6 次修订），2021 年。[13] Thomas Heller 等。利用数十亿个任务，实现可扩展、可移植的双星并合流体动力学模拟。《国际高性能计算应用杂志》，33(4)：699–715，2019 年。[14] John K. Holmen 等。将 Uintah 移植到异构系统。收录于先进科学计算平台会议论文集，PASC '22，美国纽约州纽约市，2022 年。计算机协会。[15] Kundan Kadam 等。双多方组分双星系统中质量转移的数值模拟。《皇家天文学会月刊》，481(3)：3683–3707，2018 年。[16] Hartmut Kaiser 等。HPX——面向并行性与并发性的 C++标准库。《开放源代码软件期刊》，5(53)：2352，2020 年。[17] Laxmikant V Kale 和 Sanjeev Krishnan。Charm++：一种基于 C++的可移植并发面向对象系统。收录于第八届面向对象编程系统、语言与应用年度会议论文集，第 91–108 页，1993 年。[18]Ryuta Kashino 等。在 OneAPI 环境下，使用图形处理器和 FPGA 加速天体物理模拟。收录于亚太地区高性能计算国际会议，第 84–93 页，2022 年。[19] Matthias Kretz 和 Volker Lindenstruth。Vc：一个用于显式向量化的 C++库。《软件：实践与经验》，42(11)：1409–1430，2012 年。[20] Dominic C Marcello。一种极快且守恒角动量的树算法。《天文学杂志》，154(3)：92，2017 年。[21]Dominic C Marcello、Sagiv Shiber、Orsola De Marco、Juhan Frank、Geoffrey C Clayton、Patrick M Motl、Patrick Diehl 和 Hartmut Kaiser。Octo-Tiger：一种用于恒星并合、采用 HPX并行化的全新三维流体动力学代码。《皇家天文学会月刊》，504(4)：5345–5382，2021 年。[22] David Pfander 等。加速 Octo-Tiger：使用 HPX 在 Intel Knights Landing 上进行恒星并合模拟。收录于国际研讨会论文集 *OpenCL*，IWOCL '18，第 19:1–19:8 页，美国纽约州纽约市，2018 年。ACM。[23]Damodar Sahasrabudhe、Eric T Phipps、Sivasankaran Rajamanickam 和 Martin Berzins。面向异构架构的 Kokkos 可移植单指令多数据原语。载于国际加速器指令编程研讨会，第 140–163 页。施普林格，2019 年。[24] Jan E Staff 等人。挖掘在双白矮星并合中的作用。,《天体物理学杂志》，862(1):74，2018 年。[25] Peter Thoman 等人。面向高性能计算的任务型并行编程技术分类。《超级计算杂志》，74(4):1422–1434，2018 年。[26] Peter Thoman 等人。Celerity 高级 API：用于加速器集群的 C++20。《国际并行编程杂志》，50(3-4):341–359，2022 年。[27]Christian R. Trott 等人。Kokkos 3：面向百亿亿次时代的编程模型扩展。《*IEEE 并行与分布式系统汇刊*》，33(4):805–817，2022 年。[28]Nanmiao Wu 等人。使用 Task Bench 量化 Charm++和 HPX 的开销，2022 年。

A 补充材料

Octo-Tiger 的构建脚本可在 GitHub[11]上获取。对于在 A100 节点上的运行，我们使用了此代码库的 Git 分支*SYCL 工具链*；对于在 MI100 节点上的运行，则使用了*SYCL 工具链 HIP*。

[11]https://github.com/STEllAR-GROUP/OctoTigerBuildChain