arXiv:2303.02543v2 [cs.DC] 8 Mar 2023

面向异构分布式平台的 RuntimeSupport 性能可移植性

Polykarpos Thomadakis[1*]和 Nikos Chrisochoides[11]
弗吉尼亚州诺福克市 Old Dominion 大学计算机科学系，邮编 23529，美国。

通讯作者。电子邮件：pthom001@odu.edu；其他作者：nikos@cs.odu.edu；

摘要

对于高性能计算而言，硬件异构性将长期存在。当前，大规模系统的每个计算节点都配备了多个 GPU加速器，并且预计还将纳入更多专用硬件。这一计算生态系统的转变带来了许多提升性能的机会；然而，也增加了针对这类架构进行编程的复杂性。本文介绍了一种运行时框架，可在实现异构系统轻松编程的同时，高效利用硬件资源。该框架集成于一个分布式、可扩展的运行时系统中，以促进异构节点间的性能可移植性。除了设计之外，本文还介绍了该实现及所执行的优化：在单个设备上性能最高提升 300%，并在配备四个 GPU 的节点上实现线性可扩展性。该框架在分布式内存环境中提供了可移植的抽象，使具备不同能力的设备之间能够进行高效的节点间通信。与 MPI+CUDA 相比，对于大型消息，其性能最高提升 20%，同时将小消息的开销控制在 10% 以内。此外，我们在分布式 Jacobi 代理应用中进行的性能评估结果表明，该软件仅产生极小的开销，并可实现最高 40% 的性能提升。这得益于库级优化，以及通过创造机会来利用应用特定优化（如过度分解）所实现的改进。

1 引言

摩尔定律近年来的放缓正在导致计算生态系统发生大规模变革。用户和供应商正从使用具有相对同质化 CPU架构的计算节点，转向每个节点配备多个图形处理器的系统。这一趋势预计在可预见的 future 中仍将持续，并纳入更多类型的异构设备，包括FPGA、片上系统（SoC）以及用于人工智能的专用硬件[1]。新的计算生态系统为显著提升性能、能效、可靠性和安全性奠定了基础；因此，高性能计算（HPC）系统正在针对传统工作负载和现代工作负载进行调整与优化。

要充分利用极端异构性，需要采用新的技术和抽象，以应对生产力、可移植性和性能方面日益增加的复杂性。新方法应允许用户以统一方式表达其应用程序工作流，隐藏底层架构的特性，同时处理由性能可移植性引发的问题。其中一个问题是管理各种设备上的数据。在大多数情况下，为了执行针对加速器显式优化的内核，需要在设备之间传输数据；因此，框架需要分配相应的内存，查找参与此类事务的设备，启动传输并监控其进度。不同设备上同一数据的一致性也是一个问题。必须保证应用程序始终使用数据的最新版本，而不论使用的是哪个设备。此外，由于这些操作会产生大量开销，因此应异步执行，并与有价值的工作重叠，从而进一步增加复杂性。

另一个值得关注的问题是任务（计算）执行的编排。只有在相应数据已移至目标设备后，任务才应开始异步执行；并且只有在不与其他任务冲突时才能执行，因此需要轻量级同步。任务调度和负载均衡也应受到高度重视，以保持可用设备处于满负荷状态并充分利用它们。调度器应了解每台设备的负载以及每个任务的数据局部性，从而高效地指定每项计算的执行位置。最后，能够处理所有这些问题的框架应为应用程序提供友好的高级接口，同时也应开放底层访问能力，使专家能够根据特定需求优化应用程序。此外，提供底层访问选项对于在多个节点上高效使用分布式内存框架至关重要。

在多个异构节点上利用并协调数据移动与任务执行，会增加需要处理的问题数量。因此，一个完整的运行时框架还应采用相同的数据与工作负载抽象方式，促进对分布式异构节点的无缝使用，不受底层硬件影响，并与单节点中使用的性能可移植性层紧密集成。当前高性能计算领域的趋势遵循 MPI+CUDA 编程模型，

这会导致代码复杂、性能欠佳，或两者兼有。采用这种方式的用户需要在向远程节点发送数据或从远程节点接收数据之前，显式地在主机与设备之间传输数据。此外，他们还需要对内存传输操作（主机-GPU 和网络）使用异步操作，并使其重叠执行，以避免浪费计算周期。这就引出了正确性以及涉及两类数据传输和异步内核调用的同步问题。即使能够正确处理所有这些问题，最终得到的应用程序也只能针对其开发所用的特定硬件高效运行（甚至可能完全无法运行）。因此，只有原生支持并抽象异构节点的分布式框架，才能创建不受其实现所用硬件限制、可扩展的应用程序。

在 [2]，中，我们介绍了多计算机应用并行运行时环境（PREMA），这是一个面向分布式同构平台的可扩展运行时系统。它采用高级抽象，简化了动态非规则应用的分布式编程。在本研究中，我们扩展了 PREMA，使其能够支持在异构节点上无缝、高效且具有性能可移植性的分布式应用开发。首先，我们介绍一种异构任务框架，用于优化单个节点上异构任务的并行执行。该任务框架提供了一种编程模型，可自动利用异构设备。与其他系统不同，我们的框架不要求应用选择任务运行的设备；相反，应用只需选择设备类型，框架便负责将任务调度到最优的计算设备上。接下来，我们将 PREMA 与异构任务框架集成，使其能够统一管理和利用异构节点。除了介绍最终产品的设计与实现，我们还介绍了有助于实现高性能的优化。微基准测试和代理应用的评估结果表明，我们的系统开销低，且性能具有可扩展性。

1.1 多核应用并行运行时环境

PREMA 是一种旨在为大规模计算集群提供运行时支持的系统。它采用两级并行方法，使用消息传递接口 (MPI) 进行节点间通信，并使用 Pthreads 或 Argobots[3] 进行节点内协调。该系统采用移动对象这一构造，这些对象是全局可寻址且与位置无关的容器，用于保存应用程序数据。移动对象支持移动对象驱动 (MOD) 编程模型，使本地或远程移动对象能够通过远程方法调用（称为处理程序）进行交互。无论数据位于本地还是远程，都可以统一调用移动对象上的处理程序。这种方法抽象了工作调度、负载均衡和通信开销的复杂性，使应用程序能够

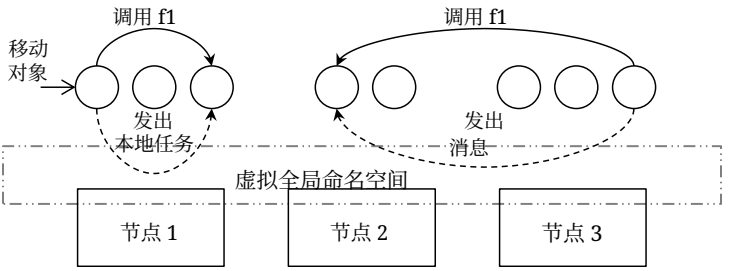


图 1 PREMA 的移动对象驱动（MOD）模型。应用程序通过本地或远程移动对象之间的方法调用来表达。虚拟全局命名空间提供统一的高层接口，用于发出本地或远程工作。运行时系统负责在本地运行任务，或根据目标移动对象的位置发送主动消息。图改编自 [4]。

利用可用的计算能力，而无需显式处理并发。图1 展示了 MOD 模型的一个示例。

PREMA 通过并发运行互不冲突的处理程序来提取共享内存并行性，同时在计算节点之间隐式迁移移动对象，以提供分布式内存负载均衡。因此，硬件内存空间和处理单元实现了虚拟化，从而支持处理程序间并行性（多个处理程序并行运行），并在同一节点中的线程之间共享移动对象工作负载。此外，它还提供了一个模块，便于试验和开发新的两级负载均衡/调度策略。

1.2 贡献

本文致力于应对如何以可移植且高性能的方式高效利用分布式异构计算节点这一挑战。本文引入了一个异构任务框架，作为可扩展的可移植性层。此外，本文还介绍了将该框架集成到分布式内存系统中的方案，从而形成一种能够原生处理配备异构设备的分布式节点的设计。本文的主要贡献如下。

- 一种全新的异构任务框架设计与实现，用于开发性能可移植应用程序，使其能够在单核到多核、多设备（*CPU*、图形处理器）平台上高效扩展，无需进行任何代码重构。
- 将分布式运行时与异构任务框架创新性地集成，为跨分布式异构计算节点扩展提供端到端解决方案，同时提供高级且抽象的编程模型。
- 一系列内存、调度和线程性能优化带来了显著提升，在单个GPU上最高可达 300%，并在多GPU平台上实现线性可扩展性；这些优化可直接应用于类似的系统和应用。

- 通过将新的运行时框架与过度分解等广泛研究的优化方法相结合，在端到端分布式异构代理应用（例如 Jacobi 求解器）中实现最高 40% 的加速比[5]。

2 相关工作

一些系统经过改造，可在其工作流中高效利用图形处理器，同时也出现了一些旨在为其使用创建新标准的新系统。Charm++ [6]、HPX [7], 和 X10 [8] 等系统引入了用于支持图形处理器的新接口。然而，这些系统要求用户显式处理内存传输请求、设备平台管理、任务分配和工作队列等问题，以优化性能。相比之下，本文提出的工作为异构任务和数据提供了统一抽象，并独立于目标设备后端，隐式处理调度、负载均衡和延迟重叠。StarPU [9], 、OmpSs [10], 和 Par-Sec [11] 提供了不同的高级方法，用于高效利用分布式异构系统。然而，它们的编程模型更适用于工作流遵循规则模式、且主要可通过静态方式推断的应用。另一方面，PREMA采用动态的消息驱动编程模型，更适合不规则应用。SYCL 和 DPC++/oneAPI [12], ，以及最新版本的 OpenMP，都是近期为现代 C++提供性能可移植接口的尝试，可用于面向异构设备。然而，用户仍需处理多设备系统的负载均衡、调度和工作队列，并需要将其与另一个面向分布式节点的运行时解决方案结合使用。

3 设计与实现

遵循关注点分离原则，我们引入了一个新的抽象兼容层，使 PREMA 能够以统一的方式访问不同的异构设备。该层以独立异构任务框架的形式实现，负责处理多种设备共存所带来的所有问题。PREMA 将此框架集成为与异构设备交互的首选方式。因此，除了这个底层兼容层之外，无需修改 PREMA 的实现，即可轻松扩展它以利用更多设备类型。此外，PREMA 还通过高级接口封装并公开了其中一些功能，使用户能够以受控且安全的方式使用此类设备。图2 展示了软件栈的高层表示，以及不同层之间的交互方式。在接下来的章节中，我们首先详细介绍异构框架层及其功能，然后重点说明它如何与 PREMA 集成，从而提供一种分布式异构感知运行时。

3.1 异构任务框架

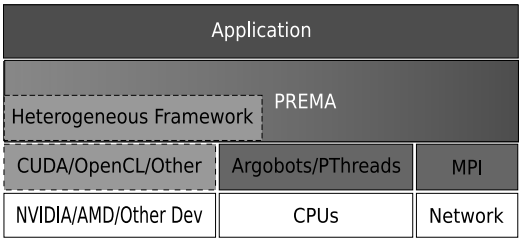


图 2 异构感知 PREMA 的高级表示。硬件设备/接口位于底层，通过将 PREMA 与 MPI、PThreads 和 Argobots 集成来加以利用（仅限 CPU；参见 [2], [4]），以及异构任务框架（在当前工作中）。应用位于其上层，并通过一个简单而强大的接口充分利用这些功能。

异构任务框架的编程模型建立在两种简单的抽象之上：异构对象（异构对象）和异构任务（异构任务）。异构对象以统一的方式表示用户定义的数据对象，这些对象驻留在异构计算节点的一个或多个计算设备上（例如 CPU、GPU、FPGA）。应用程序将此类对象视为数据的不透明容器，无需了解其物理位置。异构任务封装了一个非抢占式计算内核，该内核会运行至完成，并实现中粒度并行计算。与异构对象类似，异构任务由应用程序以统一方式定义和处理，不受其执行设备的影响。图3 展示了一个使用任务框架实现 DGEMM 的示例。该示例展示了针对图形处理器的 DGEMM 执行请求；不过，只需更改第 53 行中的设备目标，就可以指定其他设备，而无需改动其余代码。代码清单顶部的 kernel() 宏会展开为 CUDA、OpenCL 和/或其他已定义的后端，以适配供应商提供的实现。

3.1.1 异构对象

在不同异构设备上处理同一数据的副本，可能会导致容易出错且难以维护的应用程序代码。通常，应用程序需要管理设备之间的数据传输，为相应设备使用正确的指针，并跟踪其一致性。异构对象是一种抽象机制，可自动处理这些问题，将同一数据的不同副本维护在一个引用中。底层系统控制异构对象，以确保在需要时，目标设备能够获得数据的最新版本。例如，从图形处理器访问当前驻留在 CPU 上的对象，会自动触发将底层数据从主机传输到相应设备。类似地，从另一个设备访问同一对象时，会发起从图形处理器到该设备的数据传输，并且可能会先将数据暂存于主机。最后，运行时系统保证设备之间的数据一致性

```
// Device-independent kernel implementation
kernel(dgemm, (device_global double* A, device_global double* B, device_global double* C,
long N), {
      parallel_region(
        int ROW = kernel_group_id_y * kernel_local_size_y + kernel_local_id_y;
        int COL = kernel_group_id_x * kernel_local_size_x + kernel_local_id_x;
          double local_sum = 0;
        if (ROW < N && COL < N) {
            for (int i = 0; i < N; i++) {
                local_sum += A[ROW * N + i] * B[i * N + COL];
            }
            C[ROW * N + COL] = local_sum;
        }
      )
    })
    int main() {
    const int N = 1024;
      // Allocate NxN matrices A, B, C
     hetero_object<double> m_A(N,N);
    hetero_object<double> m_B(N,N);
    hetero_object<double> m_C(N,N);
    // Get write access to data and populate them
    double *A = m_A.request_data(false, true).get();
    double *B = m_A.request_data(false, true).get();
... 
    // Release data access from host
      m_A.release();
    m_B.release();
      {
          hetero_task task;
        // Set the input/output matrices A, B, C
        task.arg(m_A).read();
        task.arg(m_B).read();
        task.arg(m_C).write().dim_x();
        // Set the thread dimensions
        task.set_threads({32, 32, 1}, {32, 32, 1});
        // Set the target device type
        task.device(device::GPU);
        // Set the kernel to execute
        task.submit(dgemm);
      }
  }
```

图 3 使用任务框架的 DGEMM 应用示例。

计算设备，并跟踪最新副本或过时副本，妥善处理它们。

为获得最佳性能，异构对象所捕获的内存应主要通过异构任务进行访问和修改。然而，应用程序也可以在指定所请求访问的类型以维持一致性后，显式请求在主机上访问底层数据。该方法将从拥有最新版本数据的设备触发（如有需要）异步传输，并立即返回一个 future。该 future 可用于查询传输状态，并在传输完成后提供对原始数据的访问。在此状态下，可保证异构对象的数据在主机端始终有效，从而阻止会修改这些数据的任务执行，直至用户显式将其控制权交还给运行时系统。由于应用程序无法直接访问分配给不同设备的内存，我们的框架会监控每个设备的内存使用量。当某个设备的内存接近耗尽时，运行时系统将自动开始将部分用户数据卸载到主机或其他设备。我们目前采用最近最少使用（LRU）策略来确定应卸载哪个异构对象，以释放设备的内存。应用程序可以显式请求从所有设备中移除一个异构对象，以帮助运行时清理部分内存；否则，异构对象将在离开作用域时被释放。在这两种情况下，只有当没有任务和其他操作引用该异构对象时，才会将其移除。

3.1.2 异构任务

异构任务（hetero task）是不透明的结构，用于整合表征计算任务的参数。通过异构任务，应用程序可定义要执行的内核、输入/输出数据参数、所请求的处理单元（例如，CPU 中的线程）、任务依赖关系以及目标设备类型。此外，应用程序还可以请求分配仅在内核执行期间可用的临时共享内存区域，这对应于其他图形处理器编程 API（CUDA、OpenCL）中的本地/共享内存概念。

异构任务独立于底层目标硬件，因此无论目标是 CPU、GPU 还是其他设备类型，都能以统一的方式描述应用程序工作流。它们所代表的计算内核以类似于 OpenCL 内核的方言定义，并针对每种目标设备进行相应转换。任务的输入/输出数据参数定义为其需要访问的异构对象，并为每个对象指定所需的访问类型（读、写、读写）。这些信息用于发出适当的数据传输、维护一致性并推断任务依赖关系。提交任务以供执行并不会立即执行相应的内核；相反，运行时系统会将任务执行请求加入队列，并立即将控制权交还给用户。异构任务框架提供了查询内核执行状态或等待其完成的方法。此外，任务依赖关系既可以由用户显式定义，也可以由运行时隐式推断。

应用程序可以使用异构任务的 *add dependency()* 方法，**显式定义任务依赖图** 。设置好依赖关系后，应用程序便可一次性提交相应的任务以供执行。这种方法能够让运行时系统提升性能，同时减轻应用程序确保正确性的负担。为进一步减少确保正确性所需的工作量，框架还支持根据每个异构任务访问的参数进行**隐式任务依赖检测**。假设应用程序按正确的顺序提交任务，就能确保冲突任务按正确顺序执行；独立任务则会自动充分利用并行性，并避免竞态条件。

3.1.3 执行模型

异构任务由任务执行框架异步执行。提交执行的任务会被添加到任务执行请求列表中。随后控制权会立即返回应用程序，应用程序可以继续发出更多任务或执行其他工作。运行时的一个独立组件（可选择在单独的线程中运行）会检查任务执行请求，并在完成确保正确性所需的步骤后，最终安排这些任务执行。

执行任务的第一步，是根据其数据参数推断该任务与其他任务之间的依赖关系。运行时会维护一份列表，记录当前已提交或正在运行、且以每个异构对象为目标的任务；如果新任务以与现有访问类型冲突的方式访问这些异构对象，就会据此设置其依赖关系。至少存在一个尚未完成的依赖关系的任务会被放入另一个阻塞任务队列；否则，它们会直接加入调度器的可运行工作池。系统会定期检查阻塞任务的依赖关系是否已解除，并将所有依赖关系均已解除的任务移至调度器的可运行任务池。

检查完所有阻塞任务后，调度器即可开始调度可运行任务。在此阶段，调度器会根据用户的设备类型偏好决定任务的执行顺序及其运行设备（也就是说，用户只需选择设备类型，而调度器会选择具体的设备ID）。随后，运行时将预留设备资源，并发出数据传输请求，将输入/输出异构对象传输到所选设备上以供访问。此外，如果条件允许，运行时还会利用目标设备API提供的特性（例如CUDA流或OpenCL命令队列），自动尝试重叠执行不同操作。当任务的所有未完成数据传输均已完成后，计算内核将被提交到目标设备的工作池中。运行时会定期检查已提交任务是否完成，以更新处于等待状态的依赖任务的状态。

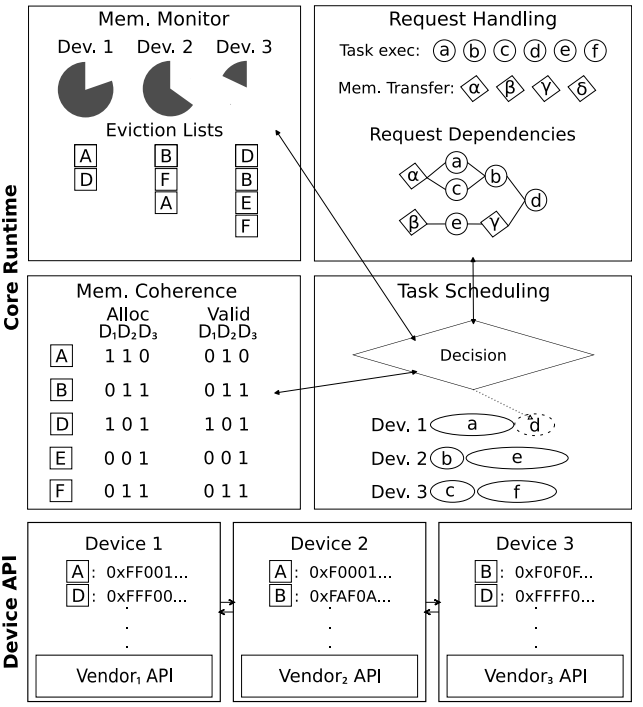


图 4 异构任务框架软件栈及其操作的高级表示。核心运行时执行的操作包括：(a) 内存监控，用于跟踪可用的设备内存，并在资源不足时释放未使用的对象；(b) 内存传输和任务执行请求处理，在安全时分发这些请求；(c) 确保多个设备上同一异构对象的不同副本之间的内存一致性；(d) 任务调度，以减少内存传输并优化整体执行时间。设备 API 提供了一个抽象的“设备类”，可统一封装不同厂商接口的实现。设备 API 将异构对象和异构任务等高级抽象映射到特定设备的实际构造。

3.1.4 调度器

随着异构计算设备和工作负载不断增加，调度和负载均衡预计将变得更加复杂。为了适应不同的使用场景，调度器的实际实现被设计为模块化，并与异构任务框架的其余部分分离。我们将调度器提供为一个抽象类，只需实现两个操作。*push()* 操作会将新的可运行任务添加到调度器的工作池中，而 *pop()*操作则返回下一个要执行的任务及其应运行的设备。抽象调度器类支持用户根据需要开发简单或复杂的自定义数据结构和策略。

3.1.5 实现

异构任务框架采用 C++ 编程语言实现，充分发挥其性能优势并利用面向对象的设计。该框架分为三个软件层开发，以便轻松集成新的设备类型、编程 API 和调度策略（见图4）。

设备 API是最底层，封装了异构设备供应商提供的各种操作。它由抽象的 C++类组成，这些类通过虚方法公开所需操作，用于：(a) 同步发出任务并管理此类设备中的数据；(b) 查询其硬件规格；以及检索异步操作状态的方法。目前，我们为 GPU 提供 CUDA 和 OpenCL 原生支持。设备 API 为任务框架的所有抽象提供底层的、特定于供应商的实现，例如将异构对象映射到内存位置，将异构任务和任务执行请求映射为实际的内核调用、内存传输请求等。

下一层是**核心运行时**层，它为异构对象和异构任务提供底层实现，监控数据不同副本之间的一致性，并检测和强制执行任务依赖关系。它利用设备 API 协调数据传输、确保任务正确执行，并发出不同操作完成的信号。该层充当应用偏好、调度器与负载均衡策略以及设备 API 之间的“黏合剂”。

顶层是**应用层**，由一个精简的 API 组成，通过高级接口展现任务框架的功能。在当前实现中，通过使用宏来定义内核；这些宏会展开，为每个可用目标实现一个内核版本。

3.2 PREMA 中的异构性

在 PREMA 中集成异构性，是满足百亿亿次级时代机器负载处理需求的关键。应用应能够在远程处理程序执行期间，轻松使用和传输设备内存。朝这一方向迈出的一步是允许 PREMA 显式地（目前仅支持 CUDA）或通过我们引入的异构任务框架抽象，发送和接收位于 GPU 设备中的缓冲区。显式方式允许用户使用 GPU，而不必局限于我们的异构任务框架，从而便于与遗留 CUDA 代码互操作。它还提供了一种将异构性集成到分布式系统之上的基础方式，可作为性能评估的基准，因为这种方式引入的开销尽可能小。

3.2.1 显式处理设备

在显式方法中，应用程序可以直接调用 CUDA API 的不同 GPU 操作来分配/释放内存、发起传输并

执行内核。PREMA 提供了一个用于调用远程处理程序的函数，该处理程序以 GPU 缓冲区作为参数；此函数需要指定远程进程的 ID 、要传输的缓冲区及其大小、源设备和目标设备的 ID，以及接收方要调用的处理程序（主机函数）。PREMA 会在远程 GPU 之间传输缓冲区，并在传输完成后调用处理程序。随后，处理程序可以安全地调用任何针对该缓冲区的 GPU 相关操作。但是，应用程序需要确保处理程序不会在内核完成之前返回，因为通过处理程序传输的所有缓冲区都会在处理程序返回时被释放，包括 GPU 缓冲区。等待所有设备操作完成（例如通过 $cudaDeviceSynchronize()$）足以保证正确性；但是，这种方法会损害 PREMA 的时间片轮转能力，使其无法在 GPU 操作进行期间切换到其他任务。因此，用户应采用更复杂的方法，在不阻塞的情况下查询操作状态（例如通过$cudaEvents$），并定期让出线程控制权，以便 PREMA 运行后台任务。

3.2.2 利用异构任务框架

为了促进 PREMA 与异构设备之间更高层次的交互，我们引入了一组扩展，使其能够直接利用异构任务框架提供的抽象。与显式的远程处理程序调用 API 相比，用户只需提供要执行的处理程序、目标进程 ID，以及作为参数传递（传输）的异构对象。由于异构对象负责处理底层数据的位置，用户无需指定其位置。该框架会自动决定在目标进程上用于存储所接收缓冲区的设备。远程方法调用的异构对象传输完成后，指定的处理程序将在目标端调用。应用程序可以调用利用该对象的任务，并在任何可用的设备类型上执行。此外，系统保证：任何作为异构任务执行或消息传递操作目标的异构对象，都将一直存在到所有此类操作完成，即使处理程序更早返回也是如此。此外，任务框架还将确保，处于网络传输过程中的异构对象不会有其他任务开始执行。代码示例见图5，其中展示了如何在异构 PREMA 中实现一系列两个分布式 DGEMM 调用。两个移动对象创建矩阵 A、B 和 C，然后根据其数据创建移动指针，并相互交换这些指针。接下来，每个移动对象在自身上调用第一次 DGEMM，并将结果发送给调用第二次 DGEMM 的远程对象。请注意，无需显式处理用于网络传输的数据缓冲区（第 29、55 行）。此外，在将结果发送给远程移动对象（第 29 行）之前，应用程序也无需显式等待 DGEMM 任务完成（第 27 行）。最后，尽管第一次 DGEMM 的结果存储在局部变量中（第 20、24 行），且处理程序可以在异步发送完成之前返回，但数据仍将被正确传输。

```
// PREMA mobile object handler
DEFINE_MP_HANDLER(execute_second_dgemm_handler) {
    hetero_object<double> m_B = get_hetero_object<double>();
    hetero_task task;
    // Set the input/output matrices A  B  C  ,,  }
    task.arg(this->m_A).read();
    task.arg(m_B).read();
    task.arg(this->m_C).write().dim_x();
    task.set_threads({32, 32, 1}, {32, 32, 1});
    task.device(device::GPU);
    task.submit(dgemm);
}

// PREMA mobile object handler
DEFINE_MP_HANDLER(execute_first_dgemm_handler) {
    task.set_threads({32, 32, 1}, {32, 32, 1});
    task.device(device::GPU);
    task.submit(dgemm);

    prema::mp_send(this->other_mp, execute_second_dgemm_handler, m_C);
}

int main() {
    prema::init();
    const int N = 1024;
    // Allocate NxN matrices A, B, C
    hetero_object<double> m_A(N,N);
    hetero_object<double> m_B(N,N);
    hetero_object<double> m_C(N,N);
    // Populate A  B ,
    mobile_object_data my_data(m_A, m_B, m_C);
    // my_mp is a reference to the remote mobile object
    prema::mobile_ptr my_mp(my_data);
    // other_mp is a reference to the remote mobile object
    prema::mobile_ptr other_mp = /*get remote mobile ptr*/;
    my_data.other_mp = other_mp;

    prema::mp_send(my_mp, execute_first_dgemm_handler);
    prema::shutdown();
}
```

Fig. 5 一系列两次 DGEMM 调用的示例，其中第一次调用的结果是需要用第二次调用，并使用异构感知 PREMA。

PREMA 和任务框架将确保数据保持一致，并按正确顺序更新。

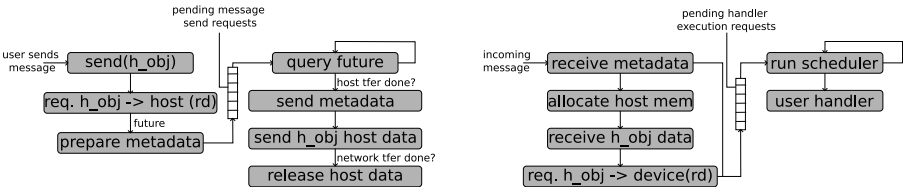


图 6 PREMA 中无 GPU 感知互连时的异构数据发送（左）和接收（右）。

PREMA 提供的异构对象还具备另一项理想功能：能够在可能分布于不同设备上的数据之间执行“放入”和“获取”操作。一项新的扩展允许用户为异构对象创建全局指针，即可由分布式系统中的所有进程引用的唯一标识符。当应用程序需要将数据存储到远程异构对象中或从中检索数据时，只需提供该对象的全局指针以及待读取或写入数据的位置（异构对象或指针），并提供一个回调函数。该回调函数会在目标端触发，以表示操作已完成。

3.2.3 实现

根据底层通信库和目标设备硬件的能力，内存传输的实际实现有所不同，以利用异构性感知通信底层。当应用程序使用异构对象，且通信底层不具备异构性感知能力时，内存传输的实现包括以下步骤（另见图 6）：

发送方：

1. 从设备异步读取内存到主机，并返回一个 future。2. 将消息（包括 future 和处理程序元数据）推送到待发送消息队列。3. 定期查询待发送消息队列。4. 当消息的 future 完成后，发送两条消息：元数据和异构对象。5. 消息传输完成后，释放对主机数据的访问权限。

接收方：

1. 接收元数据。2. 使用元数据中的信息为异构对象数据做好准备。3. 在准备好的数据中接收异构对象。4. 请求在设备中分配异构对象。5. 执行用户处理程序。

PREMA 将自动请求将异构对象数据从设备异步读取到主机。任务框架将

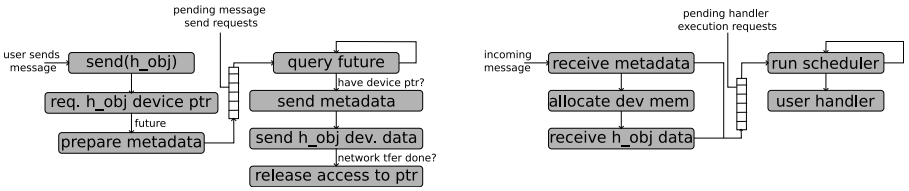


图7 在 PREMA 中不使用 GPU 感知互连时异构数据的发送（左）和接收（右）。

确保所有先前提交的冲突任务完成后才开始设备到主机传输，并在 PREMA 完成网络传输之前阻止任何新任务运行。接下来，PREMA 会准备一条封装处理程序元数据（例如数据指针、目标等）的头消息，并将其推入待发送消息请求队列。该头消息还会包含上一步返回的 future，以便检查传输进度。PREMA 会定期检查传出消息请求队列。当消息的 future 表示设备到主机传输已完成时，该消息便可通过网络发送。PREMA 将使用 MPI 作为通信底层，异步发送两条消息，一条包含元数据，另一条包含异构对象数据。最后，两条消息传输完成后，PREMA 将释放针对异构对象的读请求，并通知任务框架，其他任务可以安全地修改或删除该对象。

在接收端，一旦检测到并收到第一条元数据消息，就会利用其中携带的信息分配所需内存，以存储异构对象的实际数据。接下来，第二条包含异构对象实际数据的消息会被接收到新分配的缓冲区中。随后，系统会向任务框架发送请求，以便将接收到的数据分配到设备中；元数据消息及该请求则会被加入待处理程序执行请求队列。最后，调度器会选取一项待处理程序执行请求，并运行相应的用户处理程序。请注意，在这种情况下，PREMA 不会等待主机到设备传输完成后才启动处理程序，因此可以运行与异构对象本身无关的代码，使传输与代码执行重叠。此外，由于异构对象的数据将通过异构任务使用，任务框架会确保任何任务执行都延迟到传输完成之后。

“put”操作几乎完全相同，唯一的小区别是接收方无需分配新的主机内存。由于目标端已存在该异构对象，PREMA 会在主机侧请求写访问；获得访问权限后，便会直接将数据接收到该异构对象的主机内存中。请注意，此处不会执行从设备到主机的传输，因为接收到的数据会覆盖原有数据。该请求仅用于确保网络传输不会与该异构对象上运行的其他任务发生冲突。

如果通信库/硬件和计算设备支持分布式设备之间的直接传输（目前仅针对 CUDA-OpenMPI 进行了测试），则可以跳过主机暂存步骤。因此，在用户使用异构对象的情况下，实现过程如下（另见图 7）：

发送方：
1. 异步请求访问设备内存，并返回一个 future。2. 将消息（包括 future 和处理程序元数据）推送到待发送消息队列。3. 定期查询待发送消息队列。4. 消息的 future 完成后，发送两条消息：来自主机的元数据，以及直接来自设备的异构对象。5. 消息传输完成后，释放对设备数据的访问权限。

接收方：
1. 接收元数据。2. 根据元数据信息，为设备中的异构对象数据做好准备。3. 在准备好的数据中接收异构对象。4. 请求在设备中分配异构对象。5. 执行用户处理程序。

PREMA 将自动异步请求对当前设备中异构对象指针的读取权限。同样，任务框架会确保不会与活动任务发生冲突，但不会进行任何传输。与非 GPU 感知的情况相同，系统会准备一条元数据消息，其中包含根据上一步请求生成的 future，并将其添加到消息发送请求中。一旦 PREMA 可以安全地访问设备内存，消息便可通过网络发送。首先发送元数据消息，随后发送设备中的数据。在特定条件下，某些 MPI实现可以直接以 CUDA设备内存为目标，本例中便使用了这种方式。消息传输完成后，PREMA 将释放对异构对象的读取权限。这样一来，其他任务/消息传递操作就能安全地修改该异构对象。

在接收端，一旦收到元数据消息，就会请求任务框架在设备上创建一个能够容纳传入数据的虚拟异构对象。随后，元数据消息和虚拟异构对象会一同加入待处理程序队列，等待执行。异构对象分配完成后，PREMA 会将实际数据直接接收到设备中已分配的内存，并通知任务框架无需继续保留对设备指针的访问权限。最后，调度器会选取一个待处理程序执行请求并运行它。同样，处理程序可以在实际数据已经

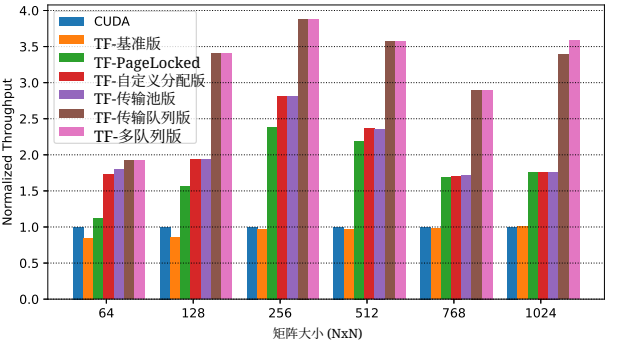


图8 在单个图形处理器上运行矩阵乘法基准测试时，逐步应用不同优化对异构任务框架性能的影响。我们在多种矩阵大小下评估性能提升，并以 CUDA 实现的性能进行归一化。**CUDA**：基线 CUDA 实现。**TF- Baseline**：未应用任何优化的任务框架（TF）。**TF-Pagelocked**：使用页锁定主机内存的 TF。**TF-CustomAlloc**：应用自定义内存分配器优化后的 TF。**TF-TPools**：使用请求池的 TF。**TF-TferQueue**：引入独立内存传输队列后的 TF。**TF-MultQueue**：TF 的最终版本，使用多个队列进行内核调用。

接收到异构对象之前启动，此时由任务框架通过根据需要延迟任务来确保不会发生冲突。

与前一种情况类似，这是“put”操作，唯一的区别在于，接收方可以直接在异构对象当前所在的位置接收传入数据。由于目标端已经存在该异构对象，PREMA 将请求对其当前设备位置进行写访问。获得访问权限后，它会将数据直接接收到异构对象的设备内存中。同样，接收时会覆盖数据；因此，该请求可确保网络传输不会与在此异构对象上运行的任何其他任务发生冲突。

4 性能评估

本节研究了能够提升异构感知 PREMA 和任务框架所提供的不同操作性能的优化，并展示了优化后实现应用于代理应用时的性能。我们使用了一个小型 16 节点集群，每个节点配备两颗 Intel Xeon Gold 6130 20 核 CPU（2.1 GHz）和四块 NVIDIA Tesla V100 GPU。

4.1 异构任务框架

我们使用一个简单的双精度矩阵乘法基准测试，评估此框架针对 NVIDIA GPU 的不同性能优化技术，并将 100 次迭代中达到的吞吐量与纯 CUDA 实现进行比较。在每次迭代中，三个矩阵都分配在

设备上，输入数据会被传输，并执行计算内核。请注意，结果不会复制回主机，以防纯 CUDA 实现出现阻塞。我们逐步应用优化，并展示其效果。如图8所示，基准柱表示该框架在朴素 CUDA 实现的基础上增加了一些开销，对于较小的矩阵尤其明显。PREMA 通过所介绍的优化技术消除了部分开销。请注意，异构框架自动应用的许多优化也可以直接在 CUDA 实现中完成。不过，这需要用户付出更多努力，且应用程序代码会相对更加庞大、复杂。此外，还需要重写代码，才能在非 NVIDIA 设备上运行。

4.1.1 页锁定主机内存

为充分利用相应硬件的带宽能力，英伟达 GPU 要求传输到 GPU 的主机数据必须位于页锁定内存区域中。此外，这是设备到主机传输相对于主机实现异步的唯一方式。因此，应用程序需要以一种特殊方式显式分配和释放内存，这会增加代码复杂度，并带来远高于常规内存分配的开销。我们将这一优化集成到框架中，从而减轻应用程序和 PREMA 显式处理这一负担的压力。为此，框架会在初始化阶段分配一大块页锁定内存，之后将其用作主机内存分配的内存池，避免后续昂贵的页锁定主机内存分配请求。

对于不同的矩阵大小，该优化带来了显著的性能提升，幅度介于 30% 和 145% 之间（图 8；TF-PageLocked）。具体而言，较小矩阵（64x64）的提升幅度最小，为 30%；其次是较大矩阵（768x768 和 1024x1024），提升约为 70%。在 128x128 的情况下，提升幅度达到 90%；而在使用 256x256 矩阵（120%）和 512x512 矩阵（145%）时，提升最为显著，超过 100%。

4.1.2 自定义设备内存池

分配和释放设备内存是开销较大的操作，可能需要主机与设备之间进行同步。此外，这两个函数运行前，可能需要等待先前已发出的异步操作完成。所提出的运行时系统为每个设备使用一个自定义内存分配器，以避免不断请求新内存分配（或释放）所带来的开销。在运行时系统初始化期间，会请求分配每个设备（主机除外）的大部分可用内存，并由自定义内存分配器管理返回的内存。需要分配内存时，会使用自定义分配器，而不是设备库提供的分配器。

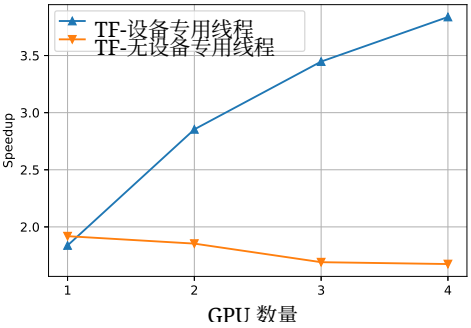


图 9 异构任务框架在 64x64 矩阵乘法基准测试中的性能，测试使用多个 GPU设备，并比较每个设备配备和未配备专用线程的情况。性能以相对于简单 CUDA 实现的加速比表示。

这种优化最显著的提升体现在最小矩阵（64x64）上，额外提升了 60%。随着矩阵尺寸增大，观察到的提升幅度逐渐下降：对于从 128x128 到 512x512 的矩阵，分别为 25%、12% 和 6%。对于更大的矩阵，提升微乎其微（图8；TF-自定义分配版）。

4.1.3 启用 GPU 并发操作

到目前为止，实现的所有优化都着眼于提升 CPU 与 GPU 操作的重叠程度；然而，默认情况下，GPU 上运行的进程仍然是串行的。我们利用设备实现提供的多个执行流（OpenCL命令队列或适用于 NVIDIA GPU 的 CUDA流），在不同的 GPU 操作之间启用隐式并发。我们的实现使用多个流来提交计算内核，并使用两个流来执行内存传输操作（每个方向各一个）。我们发现，通常使用五个计算内核流便足以充分发挥设备的并发内核执行能力。此外，我们还提供了一个环境变量，允许应用程序在不重新编译的情况下更改这一设置。

当内存传输与内核执行之间的重叠程度足够大时，这种优化在大多数矩阵大小下都能带来显著效果。对于所有大于 64x64 的矩阵大小，情况都是如此。由于将数据传输到设备的开销很大，其中很大一部分可以与前一次迭代的内核执行重叠。因此，对于大小为 128x128–1024x1024 的矩阵，结果分别显示出 75%、50%、50%、85% 和 100% 的性能提升。另一方面，64x64 矩阵的性能提升为 10%，这是因为需要传输的数据量较少（图8；TF-传输队列版、TF-多队列版）。在这一方面采取的最后一项优化，是在每个数据传输方向上使用多个流。不过，这并未进一步提升性能，因为设备硬件仅支持两个复制引擎。

4.1.4 其他优化

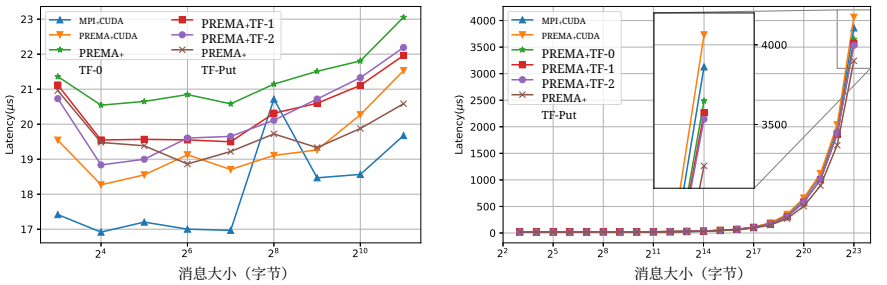
我们引入了请求内存池，以减轻系统调用和线程同步带来的影响。每个活动线程都维护各自的请求池；请求对应的操作完成后，其内存会在池中回收。我们还针对用于向设备提交请求的队列实施了另一项优化。最初，我们使用 C++ STL 提供的结构来实现此类队列，并用互斥锁加以保护。随后，我们将这些队列替换为自定义链表，避免为存储新元素而分配节点。此外，我们将互斥锁定步骤移至检查队列大小之后，以消除不必要的加锁操作。这项优化不如前文讨论的优化重要，提升幅度约为 2%，但有助于获得更稳定的延迟，尤其是在使用专用线程时（图8；TF-传输池版）。

4.1.5 综合应用

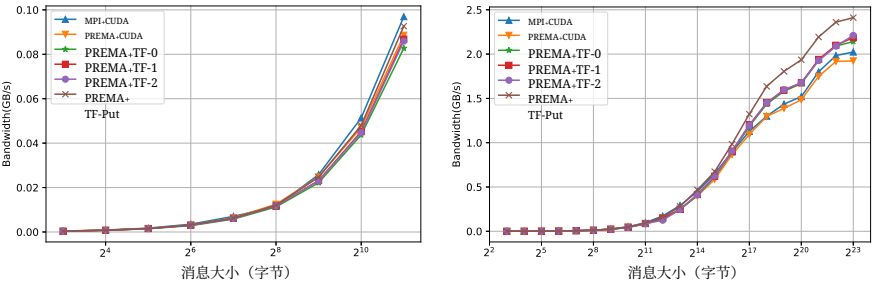
总而言之，我们实施了一系列优化，这些优化对任务框架在特定基准测试中的性能产生了影响，影响幅度因矩阵大小而异。最重要的优化包括自动使用页锁定主机内存，以及引入多个流。不过，自定义设备内存池也显著提升了较小矩阵场景下的性能。这一系列优化带来的整体性能提升可超过 300%，具体取决于矩阵大小。我们的框架能够以极少的用户参与提供所有这些优化，并且无需修改应用程序代码便可持续改进。需要注意的是，随着每次计算所需的内存传输次数减少，性能提升幅度会进一步增大。

4.1.6 多设备平台

上述结果展示了异构任务框架在单个设备上的性能。然而，我们的框架能够自动利用多个设备。这正是为每个设备使用专用线程的重要性得以体现之处。当仅使用单个 GPU 时，专用线程并无帮助（如图 9 所示），但要扩展到单个设备以上，它们至关重要。图 9 显示，随着添加更多设备，该框架几乎可以线性扩展，在 4 个 GPU 上实现最高 3.8 的加速比。在一个和两个 GPU 设备的情况下观察到的超线性加速比，是前文针对单个设备提出的优化所产生的效果。随着添加更多设备，这些优化的影响逐渐减弱，加速比也随之变为线性。必须注意的是，我们可用的硬件限制了该框架的能力。特定机器中的所有设备都通过同一条总线与主机相连。因此，无论使用一个还是四个设备，最大的内存传输吞吐量都是恒定的。对于该基准测试，最多可以同时将 2 个 64x64 矩阵传输到任意设备。如果具备更强大的



(a) 小消息（左）和整体（右）的延迟。



(b) 小消息（左）和整体（右）的带宽。

图 10 在不使用网络设备间直接传输的乒乓基准测试中，对不同优化技术的延迟（a）和带宽（b）进行性能评估。各项优化与 MPI+CUDA 实现的基线性能进行比较。**PREMA+CUDA**：直接将 PREMA 与 CUDA 集成，不涉及异构任务框架。**PREMA+TF-0**：PREMA 与异构任务框架的基线集成。**PREMA+TF-1**：在 PREMA+TF-0 的基础上引入内存池。**PREMA+TF-2**：在 PREMA+TF-1 的基础上，优化了对极小消息避免进行消息缓冲区复制的处理。**PREMA+TF-Put**：在 PREMA+TF-2 优化后执行的远程放置操作。

硬件，我们预计该框架在多设备环境中的表现会好得多。

4.2 异构PREMA

为了优化 PREMA 异构性感知版本的性能，我们针对远程处理程序调用的实现所带来的开销进行了一系列优化实验。这些调用涉及异构内存，且分别在不使用和使用任务框架（不使用专用线程）的情况下进行。我们在一个简单的节点间通信乒乓基准测试上评估这些优化，并将其与 MPI+CUDA 实现进行比较。该基准测试运行 100 次乒乓迭代，消息大小从 8 字节到 8 MB 不等，并报告每种消息大小对应的平均延迟和带宽。

4.2.1 设备消息接收缓存

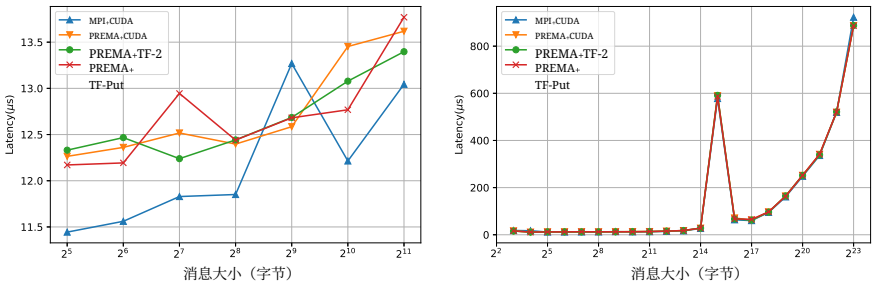
在 PREMA 中，消息是单边且异步的，并且会被隐式接收，以便在目标端调用指定的任务。因此，接收方无法指定存储消息缓冲区的内存区域（如 MPI）。PREMA 必须动态分配内存来接收传入的消息缓冲区，并在调用应用程序处理程序时将其提供给处理程序。在我们最初的实现中，由于没有任务框架，每条传入消息都要分配新的设备内存，导致性能不佳；与相应的 MPI实现相比，延迟最高增加了十倍。我们通过专门为已接收消息的缓冲区在设备上分配缓存，避免了这种情况。即将接收新的消息缓冲区时，如果可行，就从缓存而非设备 API 请求内存。借助该缓存，我们实现的性能开销控制在 MPI 所实现性能的 10% 以内（图10；PREMA+CUDA）。还需要注意的是，对于大小为 256B 的消息，MPI 性能中始终出现明显的“尖峰”；而在使用设备缓存时，我们的实现似乎未受到这一现象的影响。

4.2.2 预分配异构对象

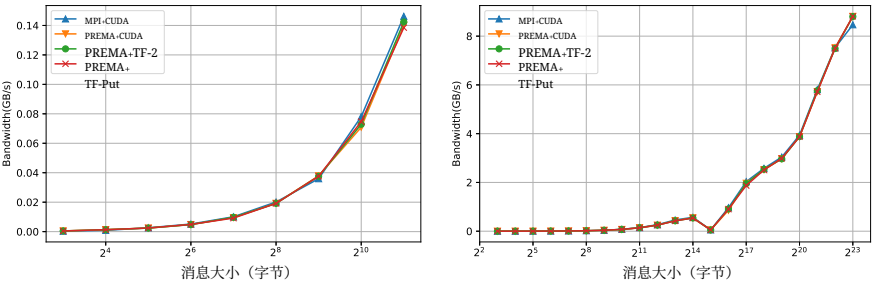
异构对象会自动利用设备内存池，从而隐式解决显式管理设备内存时遇到的问题，使性能达到 MPI+CUDA 实现的 25% 以内（图 10；PREMA+TF-0）。不过，我们仍可进一步降低因不断分配和释放临时异构对象而产生的一些延迟，这些对象用于封装指向设备内存的消息缓冲区。具体而言，我们发现，为异构对象分配的数据结构用于记录在不同设备上针对该对象执行的各项操作，这些数据结构会显著影响通信性能。由于这些结构可以提前分配，我们采用预分配的半初始化异构对象池来减轻这一影响。这项优化使小消息的延迟降低了约 10%，令 PREMA 的性能开销相较于 MPI+CUDA 降至 15% 以内，如图 10 （PREMA-TF-1）所示。此外，对于较大的消息，其性能比 MPI+CUDA 高出近 10%，这一点在放大查看 8MB 消息的延迟时可以看到。

4.2.3 避免消息缓冲区复制

第 3.2.3 节所述的节点间消息传输在采用主机暂存时，包含一项针对小缓冲区大小的优化，只发送一条消息。大小不超过 512 字节（消息头 + 缓冲区大小）的消息，将实际应用程序数据/缓冲区附加在消息末尾。这有助于优化小消息的性能，因为即使是微不足道的开销也很明显。然而，使用异构对象时，这会引入一次额外的复制。如前所述，请求访问异构对象的底层数据会隐式地将数据复制到由框架维护的主机缓冲区中



（a）小消息的延迟（左）和整体延迟（右）



（b）小消息的带宽（左）和整体带宽（右）

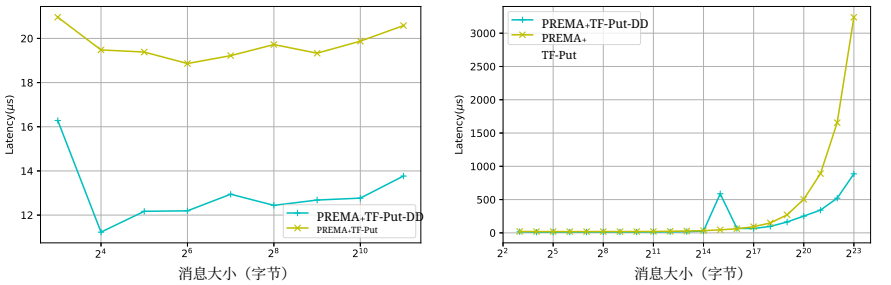
图 11 在采用网络设备间直接传输的乒乓基准测试中，从 (a) 延迟和 (b) 带宽两个方面对不同优化技术进行了性能评估。将这些优化与 MPI+CUDA 实现的基线性能进行了比较。**PREMA+CUDA**：将 PREMA 直接与 CUDA 集成，无需涉及异构任务框架。**PREMA+TF-2**：将 PREMA 与异构任务框架集成，并采用本节介绍的所有优化。**PREMA+TF-Put**：在 PREMA+TF-2 优化之后执行的远程放置操作。

。为了将数据附加到消息头末尾，PREMA 需要将数据从框架位置复制到消息头中。

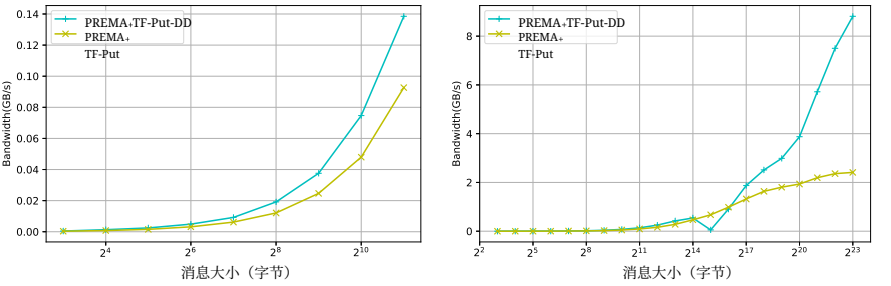
异构对象 API 中引入了一种新方法，允许用户请求将其底层数据复制到主机上指定的内存区域。PREMA 利用此功能请求框架将设备数据直接传输到消息头缓冲区的末尾。此更改略微缩短了通信关键路径，并使较小消息的延迟降低了最多 5%（图 10；PREMA-TF-2）。

4.2.4 Put 操作

另一种用于进一步提升设备间网络通信性能的操作是 put 操作。如前所述，put 操作使 PREMA 能够利用异构对象中现有且经过页面锁定的内存。通过利用这些优化，put 操作的性能超过了所有先前使用



(a) 小消息（左）和整体（右）的延迟



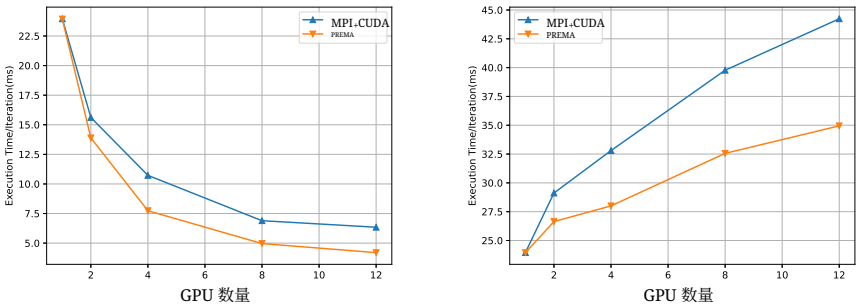
(b) 小消息（左）和整体（右）的带宽

图 12 使用主机暂存或设备到设备（DD）网络传输进行乒乓基准测试时，(a) 延迟和 (b) 带宽方面的最佳性能比较。

任务框架，因为在接收方，从主机到设备的传输速度要快得多。图10（PREMA-TF-Put）显示，对于大于 64 字节的消息，其性能达到并超过了 PREMA+CUDA 实现；对于大于 8KB 的消息，其性能甚至超过了 MPI+CUDA，对于 8MB 的消息，性能最高可提升 20%。

4.2.5 直接传输至设备

目前介绍的优化针对的是基于 PREMA 的异构性通用实现，此时通信库/硬件并不具备异构性感知能力。不过，正如3.2.3节中详细提到的，PREMA 可以利用已集成设备到设备直接通信支持的硬件/库的能力。按照前文介绍的流程，异构性感知硬件所产生的延迟可以显著降低。图11 展示了各项操作优化版本的性能，以及在支持设备到设备直接通信时 MPI+CUDA 的性能。如图12所示，在这种情况下，对于小消息，性能最高可提升 100%；对于大消息，最高可提升 200%，相比之下，采用主机暂存的性能较低。



(a) Jacobi3D 强扩展 (b) Jacobi3D 弱扩展

图 13 Jacboid3D 代理应用的强扩展（左）和弱扩展（右）性能。

4.2.6 总结

总体而言，PREMA 中为处理异构性而引入的三种操作——传输 CUDA 缓冲区、异构对象和远程放置操作——都显著简化了应用程序开发，节省了数百行样板代码，同时与 MPI+CUDA 相比，延迟和带宽方面的开销仍处于合理范围（10%–15%）。此外，值得注意的是，其中一些操作（例如 put 操作）在处理大消息时的性能优于 MPI+CUDA（最高可达 20%；见图10、11），这是因为它们隐式利用了异构对象的页锁定内存。页锁定内存会强制操作系统将某个虚拟内存地址锁定到特定的物理地址，从而允许 CUDA GPU 使用直接内存访问，并显著提高读写操作的吞吐量。

4.3 代理应用：Jacobi3D

为评估分布式内存框架的性能，我们在 PREMA 上改编了一个 Jacobi3D 代理应用 [13]。该代理应用在由长方体分解并封装为移动对象的三维域中，使用 GPU 执行固定次数的雅可比方法迭代。每次迭代中，移动对象都会交换 halo 数据，将 GPU 数据打包并传输给邻居。在接收端，数据会被解包到 GPU 中；收到所有 halo 后，便执行雅可比更新。

图13 （左）展示了在 1024x1024x768 数据域上（强扩展）的异构 PREMA与 MPI+CUDA 对应实现的执行时间。PREMA实现的性能最高可提升 23%。在弱扩展测试中，数据域大小根据分布式 GPU 的数量增加。其性能最高比 MPI+CUDA 实现提升 25%（图13；右）。观察到的性能提升源于自动重叠消息传递、主机与设备间的内存传输以及内核调用。

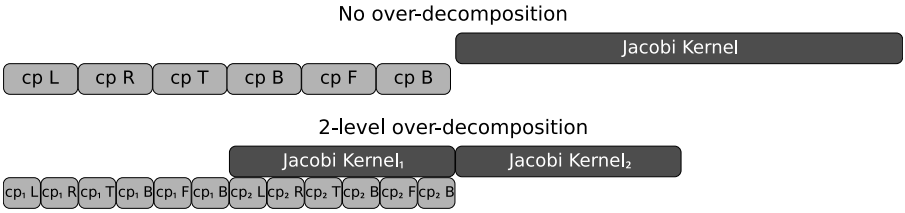
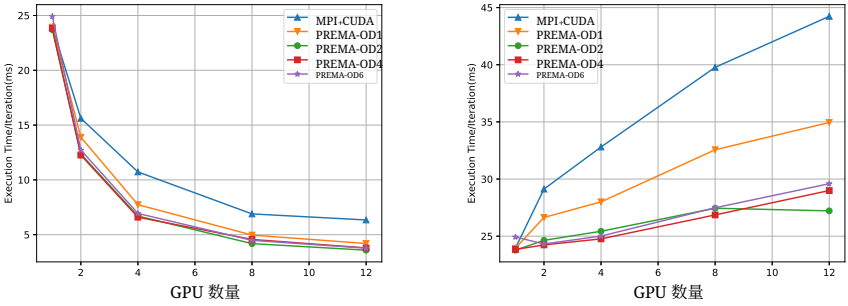


图 14 无过度分解和采用两级过度分解时 Jacobi3D 执行时间线的示例。每次内核调用都需要将六个光环缓冲区（左（L）、右（R）、上（T）、下（B）、前（F）和后（B））复制到图形处理器中。通过将数据域（以及内核调用）分解为两部分，PREMA 可以将第二部分的内存传输与第一部分的内核调用重叠，从而节省大量时间。



(a) 强 g 扩展 g 采用过度分解 P解 (b) 弱扩展性 g 与过度分 P解

图 15 Jacboid3D 代理应用在不同过度分解级别下的强扩展性（左）和弱扩展性（右）性能。

4.4 过度分解

PREMA 应用常用的一种性能优化方法是过度分解。过度分解用于将数据域拆分成比处理单元数量更多的块，从而使 PREMA 能更灵活地对工作负载进行负载均衡并隐藏延迟。这种方法的有效性已在先前针对异构平台的研究中得到证明 [2, 4]。在异构环境中，主机到设备以及设备到主机的内存传输会被拆分成流水线式的小块，因此更容易与后续内核调用重叠。图14展示了所实现的执行时间线示例。图 15 展示了同一 Jacobi3D 代理应用中过度分解的效果。本基准测试尝试了不同程度的过度分解，每种程度的性能都优于 MPI实现，也优于未进行过度分解的 PREMA 实现（OD1）。过度分解程度为 2 时性能最佳，相较 MPI实现最高提升 40%，相较初始 PREMA 实现则提升约 20%。

4.5 讨论

除了所引入抽象所带来的终端用户生产力提升和易用性外，该框架还提供了众多定量改进。总体而言，本文展示了以下改进：

1. 自动管理多个硬件设备间的内存传输和任务执行依赖，免去了代码重构的需要，并降低了复杂性。2. 通过在单个GPU上隐式利用特定硬件优化，计算性能最高可提升 300%。3. 可配置的调度器通过优化数据局部性并在多个设备间均衡分配工作负载，实现多 GPU 节点的无缝利用，并达到线性可扩展性。4. 自动利用底层网络接口潜在的异构性感知，无需显式编写代码即可发挥此功能的优势，性能最高可提升三倍。5. 用于分布式设备间通信的可移植抽象，免去了显式监控设备与主机之间以及节点间传输的需要，从而确保数据一致性和传输完成。尽管这种方法对小消息会带来一定开销（不超过 MPI 的 10%），但对于大消息也能带来出色性能，最高可提升 20%。6. 借助这些优化，基于 PREMA 实现的代理三维 Jacobi 应用性能最高提升了 30%。7. 引入过度分解后，分布式 Jacobi 的性能相比 MPI+CUDA 实现进一步提升 40%，这是通过隐式实现操作流水线化和延迟重叠达成的。

5 结论与未来工作

本文介绍了一种高效的任务框架，可处理多设备异构节点上的性能可移植性。该框架能够自动将应用扩展到多个设备，同时高效管理调度、负载均衡、任务依赖关系、内存传输和延迟隐藏，实现最高达 300% 的性能提升。将该框架集成到分布式运行时系统后，形成了一个完整的库，可利用由多个异构节点组成的百亿亿次级高性能计算系统。此外，本文还介绍了一系列优化及其评估结果，表明这些优化将开销降低至 MPI 的 10% 以内。代理应用上的评估结果显示，该工作的最终成果延迟低、可扩展性强（相较于 MPI+CUDA，性能最高提升 40%），同时提供了简单统一且独立于目标硬件的接口。未来，我们计划进一步提升任务框架的性能，并提供编译器支持，以自动生成设备

内核。此外，我们还计划扩展 PREMA 的隐式负载均衡层，以适应异构设备的工作负载，并试验多种负载均衡和调度策略。

声明

伦理审批

不适用。

利益冲突

作者声明不存在利益冲突。

作者贡献

P.T. 撰写了论文主体。两位作者共同构思并设计了这项工作。P.T. 开发了本研究所用的软件和基准测试。两位作者均审阅了论文。

资助

本研究部分由多米尼恩奖学金、老道明大学的 Richard T. Cheng捐赠基金以及美国国家科学基金会的资助项目支持：CCF-1439079、CNS- 1828593。

数据和材料的可用性

作为本项工作一部分开发的软件目前尚未公开，但未来将会公开。

参考文献

[1] J. Ang、A. A. Chien、S. D. Hammond、A. Hoisie、I. Karlin、S. Pakin、J. Shalf 和 J. Vetter，《重新构想先进科学计算的协同设计：重新构想协同设计 ASCR 研讨会报告》，2021年10月。技术报告。美国能源部科学办公室。https://doi.org/10.2172/1822199。[2]P. Thomadakis、C. Tsolakis 和 N. Chrisochoides，《用于并行和自适应应用程序的多线程运行时框架》，《工程与计算》，第 38 卷，第 4675–4695 页，2022年10月。[3] S. Seo、A. Amer、P. Balaji、C. Bordage、G. Bosilca、A. Brooks、A. Castello、D. Genet、T. Herault、P. Jindal、L. Kale、S. Krishnamoorthy、J. Lifflander、H. Lu、E. Meneses、M. Snir、Y. Sun 和 P. H. Beckman，《Argob-ots：轻量级线程/任务框架》，*IEEE*《并行与分布式系统汇刊》，2016年10月/2017年。

[4]P. Thomadakis 和 N. Chrisochoides，《面向非结构化动态百亿亿次级时代应用的运行时支持》，《超级计算杂志》，2023年1月。[5] N. Chrisochoides，《并行自适应偏微分方程计算动态负载均衡的多线程模型》，《应用数值数学》，第 20 卷，第 4 期，第 349–365 页，1996年。[6] L. V. Kale 和 S. Krishnan，《Charm++：一种基于 c++ 的可移植并发面向对象系统》，SIGPLAN Not.，第 28 卷，第 91–108 页，1993年10月。[7] H. Kaiser、T. Heller、B. Adelstein-Lelbach、A. Serio 和 D. Fey，《Hpx：全局地址空间中的任务型编程模型》，载于《第八届分区全局地址空间编程模型国际会议论文集》，PGAS '14，（美国纽约州纽约市），第 6:1–6:11 页，ACM，2014年。[8] P. Charles、C. Grothoff、V. Saraswat、C. Donawa、A. Kielstra、K. Ebcioglu、C. von Praun 和 V. Sarkar，《X10：一种用于非均匀集群计算的面向对象方法》，SIGPLAN Not.，第 40 卷，第 519–538 页，2005年10月。[9] C. Augonnet、S. Thibault、R. Namyst 和 P.-A. Wacrenier，《Starpu：异构多核架构任务调度的统一平台》，Concurr. Comput.: Pract. Exper.，第 23 卷，第 187–198 页，2011年2月。[10] A. Duran、E. Ayguad´e、R. M. Badia、J. Labarta、L. Martinell、X. Martorell 和 J. Planas，《Ompss：异构多核架构编程方案》，《并行处理快报》，第 21 卷，第 02 期，第 173–193 页，2011年。[11]G. Bosilca、A. Bouteiller、A. Danalis、T. Herault、P. Lemarinier 和 J. Dongarra，《Dague：用于高性能计算的通用分布式有向无环图引擎》，《并行计算》，第 38 卷，第 1 期，第 37–51 页，2012年。下一代并行编程模型扩展。[12] B. Ashbaugh、A. Bader、J. Brodman、J. Hammond、M. Kinsner、J. Pennycook、R. Schulz 和 J. Sewall，《Data parallel c++：通过扩展提升 SYCL 的生产力和性能》，载于《OpenCL 国际研讨会论文集》，IWOCL '20，（美国纽约州纽约市），计算机协会，2020年。[13]P. P. Laboratory-UIUC，《Charm++ jacobi3d 代理程序》。https://github.com/UIUC-PPL/charm/tree/jchoi/hips22/examples/charm++/cuda/gpudirect/jacobi3d/mpi，2022年。