GPU负载均衡编程模型

Muhammad Osama
mosama@ucdavis.edu 加州大学戴维斯分校 美国加利福尼亚州戴维斯

SerbanD. Porumbescusdporumbescu@ucdavis.edu 加州大学戴维斯分校 美国加利福尼亚州戴维斯

John D. Owens
jowens@ucdavis.edu 加州大学戴维斯分校 美国加利福尼亚州戴维斯

arXiv:2301.04792v1 [cs.DC] 12 Jan 2023

摘要

我们提出了一种面向图形处理器的细粒度负载均衡抽象，将负载均衡与工作处理解耦，并旨在通过可编程接口支持静态调度和动态调度，以实现新的负载均衡调度。在此之前，要释放图形处理器在不规则问题上的潜力，唯一的方法是通过特定于应用程序且紧密耦合的负载均衡技术来平衡工作负载。

借助我们的负载均衡开源框架，我们希望提高程序员在图形处理器上开发不规则并行算法时的生产力，同时通过允许快速试验各种现有负载均衡技术，改善此类应用程序的整体性能特征。因此，我们还希望，通过在我们的抽象中将负载均衡与工作处理分离，管理现有代码并将其扩展到未来架构将变得更加容易。

***CCS*概念：** •**计算方法** → 共享内存算法。

关键词： 负载均衡、稀疏计算、图形处理器、调度

1 引言

图形处理器（GPU）擅长处理规则细粒度并行问题，并且通常就是为此类问题而设计的，例如通用矩阵乘法（GEMM）。在 GEMM 这类规则问题中，相邻线程具有相似或完全相同的工作负载，通常能够达到 GPU 理论峰值性能的近 100%。更具挑战性的是这样的应用程序：虽然具有充足的细粒度 并行性，但并行性却不规则 。在此类应用程序中，相邻线程以锁步方式运行时，将具有不同的工作负载——或许需要完成不同数量的工作——因此，要在 GPU 这样的高度并行机器上实现高效执行，将面临重大挑战。

考虑稀疏矩阵向量乘法（SpMV），其输入为稀疏矩阵 $\mathbf{A}$ 和稠密向量 x。SpMV 计算输出向量 $y = \mathbf{A}x$，这是细粒度并行中的一种不规则模式。与通用矩阵乘法（GEMM）不同，SpMV 中的稀疏矩阵各行可能存在不规则性：矩阵各行的非零元素数量可能不同。将每行简单地映射到一个 GPU线程，可以显露这种不规则性：相邻线程可能被分配不同数量的非零元素进行处理，导致同一线程束中的线程[1] 必须等待负责处理大量非零元素的线程。这种由不规则性造成的不平衡——具体而言，当工作未在并行执行单元之间均匀分配，进而导致某些执行单元空闲而其他执行单元承担更多工作时——称为负载不均衡问题。

当前的实现通过针对应用程序的负载均衡技术来解决图形处理器上的负载不均衡问题。这些技术旨在均匀分配工作，使每个线程获得相同数量的工作项，从而实现最高性能（例如，Merrill 和 Garland 的负载均衡稀疏矩阵-向量乘法实现 [20]）。这些负载均衡技术通常与应用程序本身紧密耦合。这些实现中的负载均衡组件既复杂，往往也共同构成了应用程序性能的最大影响因素。我们的工作将当今针对应用程序的负载均衡算法推广为一种简洁、模块化且强大的抽象，可用于许多复杂的不规则工作负载。

在构建我们的抽象的过程中，我们发现了目前用于图形处理器上稀疏、不规则应用程序的常见负载均衡方法：面向特定应用程序的框架，例如 GraphIt [5], Gunrock [29],和 Graph- BLAST [31]；来自底层 CUDA 库（如 ModernGPU [3] 和 CUB [24]）的技术；以及应用程序中其他手工编写的负载均衡算法实现，例如稀疏矩阵-向量乘法/稀疏矩阵乘法 [10, 14, 20], 、三角形计数 [13, 16], 和广度优先搜索 [6, 21]。我们将展示，

分发声明“A”（已批准公开发布，不限分发）。

允许为个人或课堂使用而免费制作本作品部分或全部内容的数字或纸质副本，但不得为营利或商业利益而制作或分发副本，且副本首页必须附有本声明和完整引文。本作品中第三方组件的版权必须得到尊重。其他用途请联系所有者/作者。*PPoPP '23，2023*年*2*月*25*日至*3*月*1*日，加拿大魁北克省蒙特利尔© 2023 版权归所有者/作者所有。ACM 国际标准书号 979-8-4007-0015-6/23/02。

https://doi.org/10.1145/3572848.3577434

[1]一个 CUDA 线程束由 32 个线程组成，它们以锁步方式执行指令。线程束中的线程不会发生分歧，并以单指令多线程（SIMT）方式运行。

借助一种简单、直观且强大的抽象，这些负载均衡调度方案可以扩展，以支持比其原本针对的特定问题更通用的不规则工作负载。我们通过将基于稀疏线性代数的负载均衡应用于数据中心型图遍历内核来展示这一点。

编写高性能负载均衡代码十分复杂，主要原因是这类代码必须承担多种职责。除了其他任务，它还必须从特定数据结构中读取数据，对这些数据执行用户定义的计算，并以负载均衡的方式调度这些计算。我们抽象方法的关键洞见，是将工作负载映射（负载均衡任务）与工作执行（用户定义的计算）分开处理；我们将稀疏格式（如压缩稀疏行（CSR））映射到称为工作**原子**、**瓦片**和**集合**的简单抽象组件。这些基本组件以可组合的 C++ 范围和基于范围的 for 循环表示，并用于构建负载均衡调度。程序员随后可以使用这些 API 构建负载均衡的高性能应用程序和基础操作。通过这种表达方式，我们可以重构现有的、依赖于应用程序的负载均衡技术，以应对不规则性，并使其更加通用、可移植且可编程。我们的工作贡献如下：

1. 我们提出了一种面向 GPU 非规则并行工作负载的新型抽象。该抽象在较高层次上使程序员能够以最少的代码开发稀疏不规则并行算法，同时实现高性能。
2. 我们设计并实现了一组直观的 API，可在我们开源的 GPU 负载均衡框架中使用。该框架基于所提出的抽象，并采用 CUDA-C++ ranges 和基于范围的 for 循环构建。
3. 我们展示了实现新型负载均衡调度方案的便捷性：我们实现了一种基于协作组的新型负载均衡调度方案，该方案见第 5.2 节，是对以往线程级、线程束级和线程块级负载均衡调度方案的泛化 [30]。
4. 作为基准测试，我们在 SuiteSparse 矩阵集合 [11] 上实现了最先进的 SpMV 性能；相较于 cuSparse 的最先进实现，我们使用简单的启发式方法和 3 种 GPU 负载均衡调度方案，取得了几何平均加速比 2.7× 。

2 设计目标

我们的编程模型专注于细粒度嵌套数据并行这一广泛类别。任务级并行的负载均衡需要采用不同的方法，超出了本文的讨论范围。本节将阐述负载均衡抽象的设计目标：

实现高性能。首先，也是最重要的，我们工作的目标是让不规则应用程序达到现有负载均衡算法的高性能。我们的抽象不能以显著增加开销或降低性能为代价。我们通过将抽象的性能与现有硬编码实现的性能进行比较，来衡量是否实现了高性能。

可组合且可编程的接口。 重要的是，我们不希望将用户限制在会接管整个系统的库接口上。程序员更愿意采用能够融入现有控制结构的新软件组件，而不是被迫采用新的控制结构。我们希望用户能够：(1) 保持对 GPU 内核边界（内核启动）的控制；(2) 能够添加新的负载均衡算法；(3) 根据现有负载均衡 API 组合出新的负载均衡原语。我们通过比较该抽象与现有实现的代码行数（LOC）来衡量其可编程性，并通过使用现有 API 实现一种新的负载均衡算法来展示其可组合性。

可扩展至新应用程序。 我们的目标是将应用专用负载均衡技术解耦，并将其扩展到新的不规则并行领域。我们的抽象旨在促进现有负载均衡技术在新应用程序中的复用。我们使用稀疏矩阵-向量乘法（SpMV）作为基准应用程序，并采用三种不同的负载均衡技术实现，其中一些技术此前曾用于实现并行图分析内核 [5, 6, 10, 29]。

促进优化探索。我们的抽象的一个关键目标是，通过切换用于平衡工作量的底层负载均衡算法，促进针对给定应用程序探索优化方案。我们希望鼓励用户尝试启发式方法和新的负载均衡技术，以找到最适合其应用程序需求的方案。我们通过优化稀疏矩阵在多种不同负载均衡技术下的稀疏矩阵-向量乘法性能响应，来衡量这一目标是否达成。

非目标

除了上述设计目标外，我们还定义了以下非目标：

面向其他并行架构。虽然我们相信，所汲取的经验也适用于其他并行架构，但我们明确以英伟达的 CUDA 架构和编程模型 [23] 为目标。我们的抽象充分利用了 CUDA 的计算层次结构：线程、线程束和线程块映射到物理流式多处理器上；同时也利用 CUDA 的超额订阅模型，即分配比处理器数量更多的工作，以充分占用

底层硬件，以及 CUDA 在第5.2节中介绍的协作组编程模型 [18], 第5.2，以实现高性能。

多***GPU***支持。 本文着重研究单个GPU的负载不均衡问题，不考虑单节点多GPU或多节点系统，尽管这些都是未来值得探索的方向。

3 我们的负载均衡抽象

我们的GPU负载均衡抽象背后的关键洞见是 将关注点分离 ，即将工作项到处理单元的映射与工作执行分开。我们将该抽象划分为三个关键概念（如图1所示），分别描述实现的不同方面：（1）定义工作；（2）定义GPU线程、线程束或线程块之间的工作负载均衡；以及（3）定义均衡后的工作及每个线程的工作执行和计算。这种分离使我们能够清晰地划分应用程序开发者与负载均衡库开发者之间的工作，并通过组合不同的负载均衡技术与稀疏不规则算法，探索各种优化方案。侧栏 1 给出了一个实际示例，说明了我们提出负载均衡抽象的动机。

3.1 稀疏数据结构输入

我们从以某种稀疏数据结构形式表示的输入数据开始。这类数据结构包括但不限于压缩稀疏行（CSR）和坐标格式（COO）。我们这一抽象阶段的目标是将输入数据格式映射到一种通用的数据框架和词汇，以供下一阶段使用。这套词汇包含三个简单的组成部分，共同描述输入数据：

1. 一个**工作原子**，即将被调度到处理器上的单个工作单元（例如，稀疏矩阵中的一个非零元素）。我们假设所有工作原子在执行时的开销都相同。

2. 一个**工作瓦片**，即由一组工作原子表示的逻辑实体（例如，稀疏矩阵的一行）。工作瓦片在执行时的开销可能各不相同。正如我们在引言中强调的那样，从逻辑上 看，工作最适合按工作瓦片并行化，但从效率上 看，通常最适合按工作原子并行化，而工作瓦片与工作原子之间的映射可能既昂贵又复杂。

3. **分块集合**，即一组共同构成整个待处理问题（例如稀疏矩阵）的工作块。在我们的抽象中，分块集合中的瓦片必须彼此独立（因此可以在多个处理器上并行运行）。

这种稀疏格式与原子/瓦片/瓦片集之间的映射由用户定义。尽管我们尚未实现

侧边栏 1 一种针对稀疏不规则工作负载进行负载均衡的现有主流方法的实际示例。

考虑开源 CUDA CUB 库中提供的一种图形处理器上的稀疏矩阵-向量乘法实现 [24]。CUB 实现并维护了 Merrill 和 Garland 论文中提出的稀疏矩阵-向量乘法算法 [20]。第 5.2.1 节详细介绍的基于归并的稀疏矩阵-向量乘法，是一种基于压缩稀疏行格式且实现了完美负载均衡的稀疏矩阵-向量乘法：每个线程分得均等的工作量，而工作量由矩阵行总数与非零元素总数之和决定。在参考实现中，这种高效且先进的实现包含 1,100 行代码（LoC）（其中内核代码为 503 行），分布在 3 个文件中（此外还需要一个包含额外 234 行代码的第 4 个文件来执行分段修正步骤）。相比之下，该参考实现中稀疏矩阵-向量乘法的实际计算仅通过一个单一的for 循环和 4–5 行代码即可表达！将工作项以负载均衡的方式映射到处理单元所需的代码行数，与表达所需计算所需的代码行数之间的巨大差异，正是本研究的主要动机。此外，CUB 的实现专用于稀疏矩阵-向量乘法算法；即使是在同一计算领域中，要将其用于其他算法，也需要进行大量重写。Yang 等人将合并路径负载均衡从稀疏矩阵-向量乘法扩展到稀疏矩阵-稠密矩阵乘法（SpMM）实现，就是这种重写的一个实例 [30]。两项工作中的负载均衡算法相同，但应用于不同的计算，这也说明了复用的必要性。

我们相信，这里的映射抽象足够灵活，能够表达文献中各种现有的稀疏数据格式 [12] ，使其适用于我们抽象的下一阶段的负载均衡。此外，我们已在负载均衡库的实现中纳入了几种常见的稀疏格式（压缩稀疏行格式、压缩稀疏列格式、坐标格式），用户可以直接选择和使用，无须自行实现。给定到原子、瓦片或瓦片集的映射后，我们接下来就可以实现负载均衡算法，从计算的角度来看，该算法能够透明地对工作原子或瓦片进行并行处理。

3.2 定义负载均衡

通过使用一种抽象来表示不同粒度层级（即分块集合、原子和瓦片）的工作负载，我们可以更轻松地将计算均匀分配到图形处理器的可用资源上。给定用户定义的输入瓦片集、相应的原子和瓦片序列，以及用户选择的分区算法，我们的负载均衡阶段会输出分配给各处理器 ID 的原子和瓦片子序列（即原子或瓦片将由哪些处理器处理）。

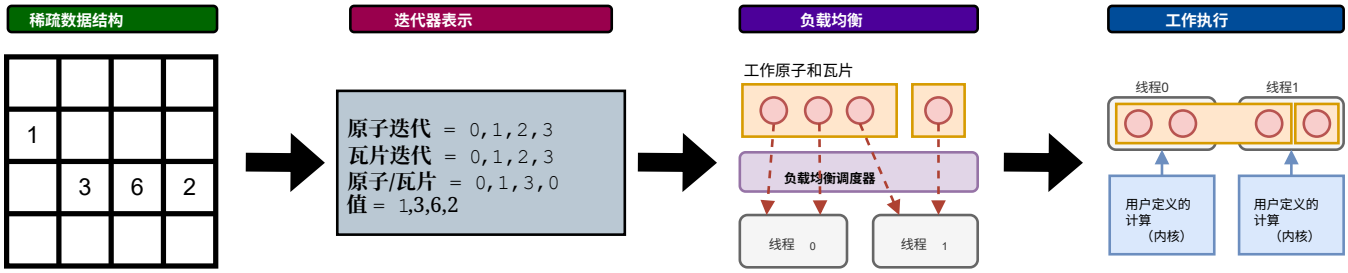


图 1. 负载均衡是由抽象的三个关键概念构成的简单流水线：(1) 以迭代器表示的稀疏数据结构，(2) 将工作分配到各线程上的负载均衡算法，以及 (3) 消耗均衡后的工作并在各线程上执行的用户定义计算。

将子序列分配给处理器 ID 的结果，对于在各处理单元之间有效平衡工作负载至关重要，而且通常取决于具体问题和数据集。用户必须指定所需的序列。理想情况下，预言机 可以接收这些序列，并为每个处理单元选出最优的子序列。寻找这样的预言机仍是一个开放问题，因此我们提供了次优方案：用户可以从一组预定义的调度方案中进行选择和试验，也可以自行实现调度方案。通常，负载均衡算法设计者必须在调度成本与更优调度带来的收益之间权衡。调度方案可以非常简单，例如将处理单元分配给原子数量任意的瓦片（如稀疏矩阵中非零元素数量任意的行）；也可以更加复杂、成本更高，采用更全面的工作负载处理方式（如考虑稀疏矩阵中非零元素数量不一的多行之间的工作负载）。

3.3 定义工作执行

负载均衡抽象的最后一个组成部分描述不规则并行计算本身。上一阶段接收负载不均衡的工作并对其进行负载均衡；接下来，这一阶段通过对负载均衡后的工作执行计算来处理它。可表达的计算范围十分广泛，唯一的限制是如何在 CUDA 内核中处理以序列表示的负载均衡后工作。由于该框架并不假定会控制内核，因此凡是能在 CUDA 内核中编写的内容，也都能在我们的框架中实现。例如，程序员可以表达对每个工作原子或每个工作瓦片执行的数学运算，也可以构建协作算法：这些算法不仅处理分配给各线程的工作，还会将结果与邻近线程的结果结合起来，以实现并行归约或扫描等更复杂的算法。我们使用这一抽象在框架中实现的实际示例（见第4.3节和第5.3节）包括但不限于稀疏线性代数内核，例如稀疏矩阵和稀疏张量收缩，以及数据中心并行图算法，例如基于邻域遍历内核构建的单源最短路径（SSSP）和广度优先搜索（BFS）。

我们预计，库的典型用户只会为抽象的这一阶段编写自己的代码，并使用库中已有的标准数据结构和负载均衡调度方案。不过，这些用户也可以实现自定义数据格式和负载均衡调度方案。

4 高层框架实现

我们的 GPU 负载均衡框架使用 C++17 和 CUDA 实现了第 3 节所述的抽象。在我们的系统中，程序员使用 CUDA/C++ 来开发不规则并行算法并实现新的负载均衡调度方案。根据可组合 API、可扩展性和复用性这些设计目标，本节及下一节将介绍 API 的实现细节，以及如何利用它开发新的应用程序，从而促进复用框架中可用的高性能负载均衡技术。我们还将探讨一种基于 CUDA 协作组模型的新型负载均衡方法（第 5.2 节）。此外，我们还将说明如何利用本研究促进针对特定应用程序（例如稀疏矩阵-向量乘法 SpMV）的优化探索。

4.1 稀疏数据结构的实现

我们的框架使用简单的 C++ 迭代器，将稀疏数据结构（例如 COO、CSR、CSC）转换为工作原子、工作瓦片和瓦片集（第 3.1 节）。C++ 迭代器是指向某个元素范围内元素的对象，并能通过一组运算符遍历该范围内的元素。例如，计数_迭代器是一种表示指向一系列连续值的指针的迭代器 [1]。我们的框架要求用户使用 C++定义三个重要的迭代器：(1) 遍历所有工作原子的迭代器；(2) 遍历工作瓦片的迭代器；(3) 遍历每个工作瓦片中原子数量的迭代器。（我们的库已支持多种常见的稀疏数据结构。）借助这些迭代器，负载均衡调度便可确定负载均衡的工作，并将其分配到

```
// 用于原子和瓦片的简单迭代器。
计数_迭代器<整数> 原子_迭代（0，非零元）；
计数_迭代器<整数> 瓦片_迭代（0，        行）；
// 遍历瓦片 i 内的原子。
自动原子_每个_瓦片 =创建_变换_迭代器(
  瓦片_迭代器，
  [瓦片_迭代器，行_偏移量]
  __主机__ __设备__(常量整数& i) {
    返回 (行_偏移量[瓦片_迭代器[i  + 1]]-
          行_偏移量[瓦片_迭代器[i]]);
});
```

清单 1. 使用 C++17 在我们的框架中表示的压缩稀疏行（CSR）格式。CSR 格式使用三个数组描述矩阵：(1) 非零值的列索引；(2) 行的范围（行偏移量）；(3) 非零值。由于 CSR 数据结构不包含指向原子和瓦片索引（非零元素和行）的数组，因此在上面的清单中，我们将原子迭代器和瓦片迭代器定义为简单的计数迭代器，分别从 0 计数到非零元素总数（nnz），以及从 0 计数到矩阵的总行数（行数）（第 2–3 行）。遍历每个工作瓦片中原子数量的迭代器使用变换迭代器表示，该迭代器会针对每个 tileid 计算所提供函数中的表达式。对于 CSR，这只需用下一个瓦片的偏移量减去当前瓦片的行偏移量（第 5–11 行）。

底层硬件。列表 1 展示了我们的抽象如何将常用的压缩稀疏行格式表示为框架内的一个分块集合。

4.2 实现负载均衡调度 ules

或许最直接的调度方式是将每个工作瓦片分配给一个图形处理器线程。这种方法在文献和实践中很常见 [3, 10, 21, 26, 30]；尽管当瓦片之间存在显著的负载不均衡时，这种策略效果不佳，我们仍以此为例，说明在我们的框架中如何定义负载均衡。

输入包括上一阶段的三个迭代器，以及一个原子数和瓦片数。负载均衡算法的开发者随后实现`tiles()` 和`atoms()` 过程调用，这些调用会返回当前线程要处理的瓦片和原子的 C++ 范围，从而在分配的处理器 ID 与工作负载片段之间建立映射。清单 2 展示了线程映射调度的完整示例。尽管算法简单，但对于具有粗粒度并行（每个瓦片包含少量原子）的均衡工作负载，例如用稠密向量乘以稀疏向量，它能够实现高性能。此外，我们的抽象并不局限于简单的调度算法，第 5.2 节给出了更复杂的负载均衡算法示例。

```
1 class schedule_t { 2 // 构造线程映射调度。3 __
host__ __device__4 schedule_t(atoms_it_t atoms_it, 5
tiles_it_t tiles_it, 6 atoms_it_t atoms_per_tile_it, 7 size_t
num_atoms, size_t num_tiles) : 8 m_atoms_it(atoms_it),
m_tiles_it(tiles_it), 9 m_atoms_per_tile_it(atoms_per_tile_
it), 10 m_num_atoms(num_atoms), 11 m_num_til
es(num_tiles) {} 12 // 此线程要处理的瓦片范围。 13 // 按网
格维度进行步进。14 __host__ __device__ auto tiles() {
15 auto begin =m_tiles_it(blockDim.x * blockIdx.x
16 +threadIdx.x); 17 auto end = m_tiles_
it(m_num_tiles); 18 return range(begin, end) 19 .
step(gridDim.x * blockDim.x); 20 } 21 // 此线程要处理的原
子范围。22 __host__ __device__ auto atoms( 23 const
std::size_t& tile) { 24 auto begin =m_atoms_per_tile_it[
tile]; 25 auto end = m_atoms_per_tile_it[tile + 1]; 26
return range(begin, end).step(1); 27 } 28 }; 29 using
schedule_t = thread_mapped_schedule_t;
```

列表 2。 以 C 语言表达的线程映射负载均衡算法++ 范围，包含作为迭代器定义的原子和瓦片，见列表 1。每个瓦片都映射到一个线程，其中线程 ID 对应于该瓦片在分块集合中的索引。线程会按顺序处理瓦片内的所有原子。处理完一个瓦片后，线程会映射到下一个瓦片，该瓦片通过将索引按内核的网格大小步进获得。

4.3 实现工作执行

我们的框架旨在明确让用户掌控 内核启动边界。掌控 CUDA 内核边界意味着用户负责维护和配置启动参数，并实现用于定义应用程序的 CUDA 内核。尽管这一设计决策牺牲了便捷性和简洁性，却显著提升了用户通过我们的抽象能够表达的内容的灵活性。作出这一设计决策的原因如下。(1) 用户无需在现有工作流/库中添加复杂依赖，因此代码维护会更加简单、可扩展，因为他们不必依赖我们的框架来纳入新的 CUDA 构造和功能。(2) 用户可以在使用我们的负载均衡 C++ranges 时，在内核中自由表达 CUDA 所允许的一切。这使得

表达内容更加灵活，因为用户现在可以指定多个负载均衡工作域、基于范围的 loops，甚至可以融合多项计算，在单个内核中构建更复杂的算法。(3) 可以使用高层 API 构建更简单的高层抽象，这些抽象确实 掌控内核边界，并以牺牲灵活性为代价提供更简单的 API。

作为此阶段的输入，用户使用负载均衡后的 C++ 范围来实现其计算。这可以通过多种方式完成，但最常见的模式之一是嵌套的基于范围的 for 循环，它遍历所有已分配的瓦片范围和原子范围。列表 3 展示了一个简单的 CUDA 内核示例，该内核使用压缩稀疏行格式和列表 1 和 2中描述的线程映射负载均衡算法实现稀疏矩阵-向量乘法算法。在此示例中，每个线程中的外层 `for` 循环遍历稀疏矩阵中分配给该线程的行（瓦片），内层循环则依次处理每一行中分配的非零元素（原子）。在第 5.3 节中，我们将实现并讨论更复杂的内核和计算。

```
1 // 实现负载均衡的稀疏矩阵-向量乘法内核。2 __global__void spmv(const size_t rows, 3 const size_t cols, const size_t nnz, 4 const int* offsets, const int* indices, 5 const float* values, const float* x, 6 float* y) { 7 // 配置负载均衡。 8 // 输入：为压缩稀疏行格式定义的迭代器。 9 schedule_t config( 10 atoms_iter, tile_iter, 11 atoms_per_tile_it, 12 nnz, rows); 13 // 使用基于范围的 for 循环处理各行。 14 for (auto row : config.tiles()) { 15 type_t sum = 0; 16 // 使用基于范围的 for 循环处理各原子。 17 for (auto nz : config.atoms(row)) 18 sum += values[nz] * x[indices[nz]]; 19 y[row] =sum; 20 } 21 } 22 // 启动稀疏矩阵-向量乘法内核。 23 constexpr size_t blocks = 256; 24 size_t grid = (rows + blocks - 1) / blocks; 25 spmv<<<grid, blocks>>>(rows, cols, nnz, 26 offsets, indices, values, x, y);
```

清单 3。使用基于范围的嵌套`for` loops，在我们的负载均衡抽象中实现的稀疏向量矩阵乘法（SpMV）。稀疏矩阵采用基于 CSR 的格式表 x 示稠密输入向量， y 表示稠密输出向量（$y = Ax$）。第 9–12 行使用清单 2中实现的负载均衡调度，以及清单 1 中定义的迭代器，构造待处理的负载均衡工作。第 14 行和第 17 行展示了每个线程内的 `for`loops，这些循环遍历分配给该线程的稀疏矩阵行，并依次处理每行中分配的工作原子。第 18 行展示了对每个工作原子（非零元素）执行的实际计算，第 19 行则将结果写入稠密输出向量 y。

5 实现细节

5.1 灵活、可组合的 CUDA 范围

使用我们的 API 将负载均衡原语与应用程序组合起来，是我们框架中的一项有意设计选择；我们通过使用 CUDA 支持的 C++ranges 来实现这一点。我们的框架并不拥有 内核边界（内核启动），因此我们的 API 专注于内核内部，并将功能限制在内核之中。这使程序员能够构建并维护自己的内核，同时受益于我们框架的负载均衡能力。这主要通过设备级 C++ 函数和类实现，这些函数和类带有 CUDA 的__device__ 关键字标记。[2] 我们实现并公开了几种不同类型的专用范围，它们在实现负载均衡调度方案时尤其有用：

- step_range：一种按 step 步长从 begin 迭代到 end 的范围。适用于定义需要自定义步进范围的负载均衡调度方案，或用于处理每个线程固定数量的工作项（可通过 step 定义）。
- infinite_range：一种从 begin 迭代到无穷大的范围。适用于定义持久内核模式下的负载均衡调度方案 [32],，在这种模式下，内核会持续运行，直到所有工作都被处理完毕或算法收敛。
- grid_stride_range：一种特化的 step range，它使用 CUDA 内核的网格大小，以 step 为步长从 begin 迭代到 end。也支持 block 和 分别以线程块大小或 warp 大小为步长进行迭代的 warp 步幅变体。

[2]使用 __device__ 关键字修饰的方法可让 CUDA 编译器生成设备可调用入口点，从而使代码能够在内核 [23]中调用。

5.2 实现非平凡负载均衡

正如我们在第 5.1 节所述，我们可以将现有的负载均衡技术解耦，并将其表示为一组 C++ 范围。为说明这种抽象的潜力，我们首先解耦并表示一种名为合并路径的先进负载均衡算法 [17]，该算法此前用于对基于 CSR 的稀疏矩阵-向量乘法和稀疏矩阵乘法进行负载均衡 [20, 30],；此外，我们还实现了另外三种负载均衡算法（线程束映射、块映射和组映射），程序员都可以通过我们的库使用这些算法。我们新提出的组映射算法采用每组一个瓦片的调度，其中组定义为

任意大小的线程集合（不局限于线程束或块大小）。我们的组映射调度是每线程、每线程束或每块一个瓦片调度的推广 [5, 21]，采用 CUDA 的协作组编程模型 [18]。

5.2.1 合并路径负载均衡。从稀疏矩阵的角度来看，合并路径假设矩阵中的每个非零元素和每个新行的工作量相等，然后将 nnzs+rows 份工作均匀分配给工作线程集合。随后，每个线程会在 CSR 矩阵的非零元素索引和行偏移量中执行二维二分搜索，以找到它需要处理的行和非零元素的起始位置。接着，线程从起始位置开始，依次处理各行和非零元素，直至完成分配给自己的工作 [20]。

我们在抽象中将此算法实现为负载均衡调度，并分两步进行描述：**(1) 设置：** C++ schedule 类的初始化步骤会计算每个线程的工作单元数，按照上述方法执行二分搜索，并将每个瓦片和原子的起始位置存储在线程局部变量中。**(2)范围：** 算法的第二步会为每个线程构建待处理的范围，并将其划分为"完整"瓦片和"部分"瓦片 [20]。如果线程的原子范围完全位于一个瓦片内，则该范围为"完整"，并通过简单的嵌套循环进行处理。如果线程的范围跨越瓦片边界，则线程会在单独的嵌套循环中处理其工作。

由于我们将负载均衡方法（第 4.2 节及以上）与工作执行（第 4.3 节）解耦，因此可以使用这种 merge-path 实现来实现稀疏矩阵-向量乘法，以及任何其他可将工作划分为瓦片和原子的算法，例如用于实现广度优先搜索的图邻域遍历算法 [29]。同样重要的是，merge-path 调度如今不再局限于基于 CSR 的稀疏格式。要支持其他格式，只需构建必要的、稍复杂一些的迭代器，使其能够统计每个瓦片中的原子数量（CSR 实现通过清单 1 中的行偏移数组完成这项计算）。

5.2.2 线程束级和线程块级负载均衡。线程束级或线程块级负载均衡调度的目标，是为每个线程束或线程块分配相同份额的瓦片，然后按顺序处理这些瓦片。每个瓦片中的工作原子由线程束或线程块中的可用线程并行处理。每个线程以线程束或线程块的大小为步长，处理下一个工作原子，直到所有工作处理完毕。

不同处理单元之间的不均衡由硬件调度器负责处理。该调度器依赖 CUDA 的超额订阅模型：程序员可以启动比图形处理器在任一时刻实际能够调度的更多线程束或线程块。当处理单元完成其工作后，新的处理单元会从超额订阅池中进行调度 [5, 21]。

5.2.3 组级负载均衡。组级负载均衡是 warp 级和 block 级调度方案的泛化。与上述方法不同，它不要求组的大小必须等于一个 warp 或一个 block，而是利用 CUDA 的协作组（CG）编程模型 [18]，允许程序员指定大小任意且可动态调整的组。在这些组内，CG 模型不仅允许对组的同步行为进行精细控制，还支持简单的组级并行集体操作，例如归约或扫描。我们利用这一强大工具实现通用的组级负载均衡调度；当组的大小等于一个 warp 或一个 block 时，上述 warp 级和 block 级调度方案便可直接实现。

我们的调度方案将工作瓦片分配给一个组，每个组处理分配给自己的等份瓦片，计算每个瓦片包含的原子数量，并将结果存储在暂存存储器（CUDA 的 共享内存）中。随后，该组执行并行前缀和。这是一种广泛使用的并行算法，它以一个数组为输入，并生成一个新数组，其中任意位置的元素都是此前所有元素的和 [4]。我们将此前缀和数组用于两个目的：（1）前缀和数组的最后一个元素表示该组需要处理的工作原子总数；（2）前缀和数组中每个和的位置对应于这些原子所属的工作瓦片。该调度方案的设置阶段会在暂存存储器中为每个组构建前缀和数组，而其范围循环则返回每个线程要处理的原子。如有需要，可通过简单的 get_tile(atom_id) 操作获取对应的瓦片；该操作会在前缀和数组中执行二分搜索，以找到正在处理的原子所属的瓦片。

依赖 CG 模型实现这种负载均衡调度器有一个独特优势：可以根据问题的形态和底层硬件架构配置组大小（即直接映射到硬件的软件构造）。例如，现在只需使用一个简单的编译时常量，就能面向 warp 大小不是 32 个线程的 GPU（AMD 的 GPU 架构支持 warp 大小为 64 [2]），也可以配置组大小，使其与问题结构完美匹配。

5.3 应用领域

我们的工作定义（第 3.1 节）、可组合 API（第 5.1 节）以及多种复杂的高性能负载均衡调度方案（第 5.2 节）共同构成了一个灵活且可扩展的框架，并为特定应用优化留出了充足空间。在清单 3中，我们已经演示了如何使用稀疏矩阵-向量乘法算法

```
1 // ... 在 CUDA 内核内部。 2 // 遍历所有已分配的行。 3 for (auto row : config.tiles()) { 4 // 遍历矩阵 B 的所有列。 5 for (auto col : range(size_t(0), B.cols) 6 .stride(size_t(1))) { /// < 新循环 7 float sum = 0; 8 // 遍历所有已分配的非零元素。 9 for (auto nz : config.atoms(row)) 10 sum += values[nz]* B(nz, col); 11 // 将总和写入矩阵 C。 12 C(row, col) =sum; 13 } 14 }
```

清单 4。在清单3 中介绍的 SpMV 外层套用一个简单循环，即可表示稍复杂一些的负载均衡 SpMM 计算。

我们的框架。一个简单而自然的扩展是实现稀疏矩阵-矩阵乘法（SpMM）。清单 4 展示了所需的微小改动：在清单**B** 矩阵的列上再增加一层循环，并将其套在清单 3 中的现有代码外，即可实现 SpMM。还可以扩展此实现以支持 Gustavson 通用稀疏矩阵-矩阵乘法（SpGEMM），这需要使用两个内核和一个分配阶段；第一个内核计算输出行的大小，以便为输出稀疏矩阵分配内存，第二个内核则执行乘加运算。

除了稀疏线性代数之外，我们还可以利用框架来处理其他领域的应用程序。列表 5 展示了如何使用我们的组级负载均衡调度实现图原语单源最短路径（SSSP）。SSSP 在 GPU 上的性能很大程度上取决于能否实现良好的负载均衡 [5, 29],但如果程序员从我们的库中选择一种负载均衡调度，负载均衡的具体细节就会完全隐藏起来。此外，在一个应用领域（例如稀疏线性代数）中使用的相同调度方案，也很容易在这一不同的应用领域中重用。

6 评估

我们的目标是展示，基于负载均衡抽象构建的框架，能够为稀疏不规则问题同时带来高性能和更好的可编程性。以下评估使用我们的 SpMV 实现作为基准，并将其与英伟达开源 CUB 库和商用闭源 cuSparse 库提供的最先进实现进行比较。我们还考虑并实现了若干其他应用程序进行评估，包括 SSSP、广度优先搜索和稀疏矩阵乘法。我们发现，这些应用程序得出了类似的高层次结论。因此，本文的评估重点关注 SpMV。我们的测试语料库包含约完整的 SuiteSparse 矩阵集合 [11]，涵盖范围广泛的稀疏矩阵

```
// ... 位于 CUDA 内核内部。
// 遍历所有分配的边进行处理。
for(auto 边 : config.原子()) {
  自动 源节点 =配置.get_瓦片(边);
  // G 是图数据结构
  自动 邻居节点 = G.get_邻居节点(源节点, 边);
  自动 权重 =G.get_边_权重(边);
  浮点数 源节点_距离 = 距离[源节点];
  浮点数 邻居节点_距离 =源节点_距离 + 权重;
  // 检查目标节点是否已被
  // 声明为某个节点的子节点。
  浮点数 恢复_距离 =
    原子最小值操作(&(dist[neighbor]), neighbor_dist);
  // 将邻居节点加入前沿。
  如果(邻居节点_距离 < 恢复_距离)
    输出_前沿[邻居节点] =真;
}

// ... CUDA 内核之外。
// 循环直到前沿为空。
```

清单 5. 使用我们的负载均衡调度表示的并行单源最短路径（SSSP）图原语。

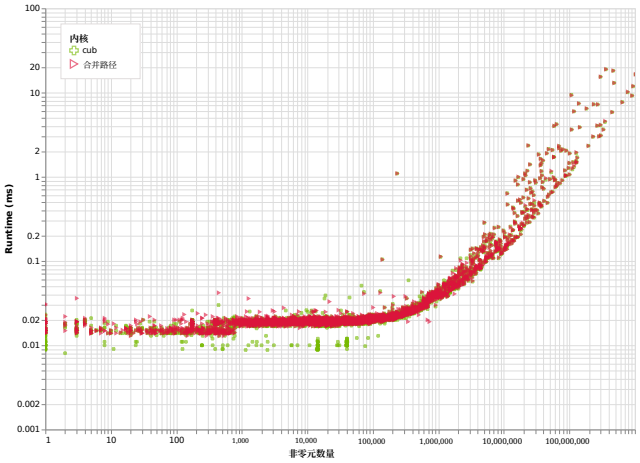


图 2. 稀疏矩阵-向量乘法运行时间对比：我们的合并路径稀疏矩阵-向量乘法实现与 CUB 在所有 SuiteSparse 数据集上的表现。对于所有数据集，我们的运行时间几乎与 CUB 完全一致。少数几个 CUB 更快的数据集，是因为 CUB 针对单列稀疏矩阵（即稀疏向量）采用了一种简单的启发式方法。

这些矩阵来自许多不同的高性能计算领域。所有实验均在一台基于 Ubuntu 20.04 LTS 的工作站上运行，该工作站配备 NVIDIA Tesla V100 GPU 和 CUDA 11.7。

6.1 性能开销

我们的首要目标是确保抽象中的元素不会给基于这些元素开发的现有负载均衡技术和算法带来任何额外的性能开销。为此，我们

在 SuiteSparse 集合上，将我们采用 merge-path 调度的稀疏矩阵-向量乘法实现，与英伟达的 CUB 库提供的实现进行运行时间比较 [24]（该实现也用于 Merrill 和 Garland 的 merge-path 稀疏矩阵-向量乘法论文 [20]）。如前所述，与我们的设计不同，CUB 将 merge-path 调度算法硬编码在实现中，并未将工作负载均衡与实际的稀疏矩阵-向量乘法计算解耦。若不对实现进行重大修改，CUB 的方法无法复用于任何其他不规则并行问题。

图 2 绘制了我们的方法与 CUB 实现中非零元素数量（即总工作量）和运行时间之间的关系。使用我们的抽象时，我们的实现只有极小的性能开销：与 CUB 相比，几何平均减速为 2.5%，且 92% 的数据集性能达到 CUB 的至少 90%。图 2 显示，除少数非零元素少于 100,000 的数据集外，我们的实现几乎与 CUB 完全匹配。进一步调查后发现，当给定输入矩阵的列数等于 1（即稀疏向量）时，CUB 会采用一种简单的启发式方法来启动线程映射稀疏矩阵-向量乘法内核。与我们更通用的实现不同，对于稀疏向量-向量乘法计算这类负载均衡完美的工作负载，CUB 简单（但专用）的线程映射稀疏矩阵-向量乘法内核没有负载均衡开销。

6.2 性能提升

我们还将本研究与 NVIDIA 用于稀疏计算的厂商库 cuSparse 进行了比较。图 3展示了我们分别采用各调度算法实现的 SpMV 与 cuSparse 最先进实现之间的性能表现。切换到我们的任一实现所需的代码改动都很少；对于合并路径和线程映射，我们只需更新一个 C++ 枚举（标识符），即可选择所需的负载均衡调度方案。

随后，我们将这些调度算法整合为一个 SpMV 实现（图 4），其性能明显优于 cuSparse。这主要得益于我们能够利用多种可用的负载均衡调度方案，快速试验不同的启发式方案。此处，除非行数或列数小于阈值 α，且给定矩阵的非零元素数小于阈值 β（对于 SuiteSparse，我们选择 $\alpha = 500$ 和 $\beta = 10000$），否则我们使用合并路径。在这种情况下，我们改用线程映射或组映射负载均衡。与 cuSparse 相比，我们的系统最高性能加速比达到 39×，几何平均性能加速比为 2。7×。

我们的框架不仅让程序员能够高效、简单地表达计算（即无需担心负载均衡算法），而且还能快速……使用多种调度算法优化给定的应用程序，只需对代码进行少量修改。

负载均衡算法	NVIDIA/CUB	我们的工作
合并路径	503	36
线程映射	22	21
组映射	N/A	30
warp 映射	N/A	30（免费）
块映射	N/A	30（免费）

表 1. 英伟达 CUB 库与我们针对采用合并路径、线程映射和组映射（线程束映射和块映射与组映射使用完全 相同的代码）负载均衡算法实现的稀疏矩阵-向量乘法应用程序的代码行数（LOC）对比。我们仅统计非注释代码行数，并使用遵循 Chromium 风格指南的 clang-format 工具进行格式化；这些代码行用于内核实现 [15], 。

6.3 代码行数（LOC）

我们只需以极低的代码复杂度即可实现这些性能提升。表1 展示了我们的框架与英伟达 CUB 库中最先进的开源合并路径和线程映射实现的代码行数（LOC）对比。正如前文所示，我们实现了相同性能，同时对于合并路径和线程映射调度算法，所需代码行数分别为 14× 和 1× 。在简单的线程映射调度基础上，仅需增加 ~15 行代码，即可实现我们的合并路径方案。

此外，我们在同样的 30 行代码内，将相同的稀疏矩阵-向量乘法计算扩展到新颖的组映射负载均衡调度（也可专门用于块映射和*warp* 映射负载均衡）。

7 相关工作

负载均衡是图形处理器在稀疏不规则并行问题上实现高性能的关键。多种高性能计算应用程序会在 GPU 上采用复杂的负载均衡算法。例如，高性能稀疏矩阵向量乘法（SpMV）会采用合并路径 [20]（本文将对此进行详细讨论）或非零元拆分算法，将稀疏矩阵中的非零元素均匀分配到各个线程上 [3, 9, 26]。稀疏矩阵乘法（SpMM）和稀疏矩阵化张量乘 Khatri-Rao 积（SpMTTKRP）则采用分箱和打包算法 [14, 22, 30],，尝试将长度相近的工作分到同一组，以便一并处理。

有些应用程序会主动执行工作来平衡给定输入的负载，而另一些则将输入存储为效率更高、已实现负载均衡或分区的格式。这些格式包括用于稀疏张量矩阵乘积-张量乘积的 F-COO 格式（坐标格式的一种变体）

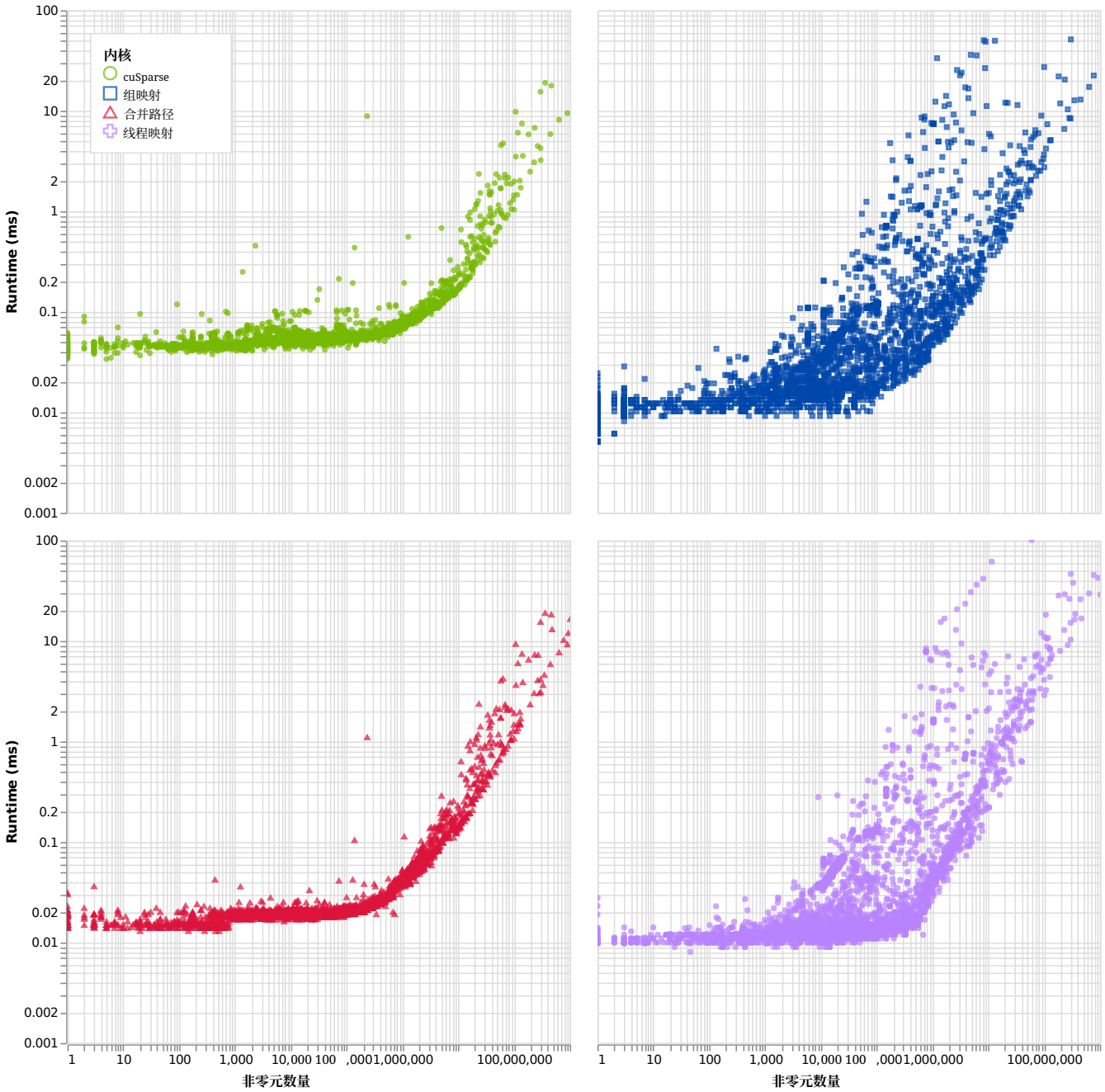


图 3. 使用 3 种负载均衡调度方案与英伟达的 cuSparse 库，对 SuiteSparse 的所有数据集进行稀疏矩阵-向量乘法的完整性能分析。此性能比较展示了针对给定数据集及其非零元素数量，采用不同方法对稀疏矩阵-向量乘法进行负载均衡所产生的影响。随后在图 4 中，我们利用这一发现选出速度最快的调度方案，从而提升整体性能。此外，我们只需对代码进行极少改动，就能实现这 3 种不同的稀疏矩阵-向量乘法实现。

以及用于稀疏张量张量乘法（SpTTM）的格式，在这种格式中，每个线程要处理相同数量的非零元素 [19]。

上述许多图形处理器负载均衡算法以及其他新颖技术，最初都在图中进行了描述

分析领域。Davidson 等人以及 Merrill 和 Garland 首次提出了用于单源

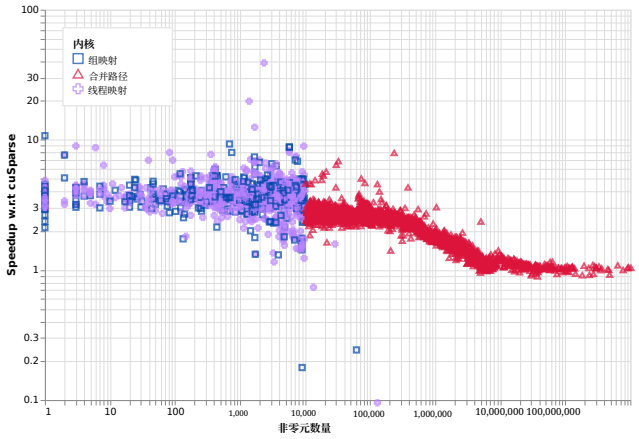


图4。使用启发式方法（第6.2节）选择合适的负载均衡调度后，我们的框架在 SuiteSparse 上执行稀疏矩阵-向量乘法相较于 cuSparse 的加速比。

最短路径（SSSP）和广度优先搜索（BFS）的线程束级、线程块级以及线程—线程束—CTA 动态负载均衡技术 [10, 21]。对数基数分桶（LRB）是一种尤为有效的技术，它根据对数级工作量估算对工作进行分桶，应用于三角形计数图算法等 [13, 16]。Gunrock、GraphIt 和 GraphBLAST 是图分析库，实现了多种不同的图算法，例如 BFS、SSSP、PageRank、图着色等，并建立在前文所述的负载均衡技术之上 [5, 29, 31]。尽管其中许多负载均衡技术都很有效，且具有高性能实现，但它们都将工作负载调度与应用程序本身紧密耦合。我们的框架旨在将这两个关注点分离，使应用程序独立于负载均衡算法，从而能够以简洁的方式表达。我们的方法还允许在框架中实现这些先前提出的技术，并将其用于最初目标应用之外的其他应用。

针对不规则工作负载的通用负载均衡，面向 GPU 的研究相对较少。其中大多数研究都专注于提供一种以任务并行为核心的单一动态负载均衡方案，通常会使用基于图形处理器队列的数据结构。Cederman 和 Tsigas 提出了一种基于任务的方法，使用无锁和基于锁的方法对八叉树划分工作负载进行负载均衡 [7]。Tzeng 的两项研究提出了任务管理框架，分别使用单体任务队列和支持任务窃取与捐赠的分布式队列来实现任务负载均衡 [27, 28]。CUIRRE 是一个用于在 GPU 上对不规则应用进行负载均衡和特征分析的框架，也采用了任务池方法 [33]，；近期推出的 Atos 则是一个面向异步算法的任务并行图形处理器动态调度框架 [8]。上述研究都在 GPU 上部署了集中式或分布式的类队列数据结构，并各自决定如何划分和更新队列。除最新的 Atos 研究外，早期的大多数研究都采用粗粒度并行方法，将任务有效地分发到 GPU。我们的工作利用了较新的 GPU 架构，这些架构通过细粒度并行方法（对工作原子而非工作瓦片进行并行化）能够得到更有效的利用。与我们的抽象不同，上述研究依赖于单一的负载均衡方案；而我们的抽象能够灵活适配多种静态和动态负载均衡技术，并允许在框架中实现新的调度方案。

8 结论

本文提出了一种面向稀疏不规则并行问题的图形处理器负载均衡编程模型。我们的模型基于这样一种理念：将工作负载映射与工作执行分离。未来，我们希望将模型扩展至多图形处理器环境，并实现跨越图形处理器边界、覆盖多个设备和节点的负载均衡调度，以应对大规模并行问题。当前工作仅关注负载均衡，但我们也发现，局部性是实现高性能的另一个关键因素。我们希望探索一种正交模型，将缓存和局部性抽象融入现有的负载均衡框架。

致谢

本材料所述工作由国防高级研究计划局（DARPA）根据合同编号 HR0011-18-3-0007 以及美国国家科学基金会根据合同编号 OAC-1740333 资助。本文材料中表达的任何观点、发现、结论或建议均属于作者，不一定反映美国政府的观点。分发声明“A”（批准公开发布，不限分发）。我们感谢英伟达的 Michael Garland 和 Duane Merrill 对框架提供指导。我们还感谢 Toluwanimi Odemuyiwa、Jonathan Wapman、Matthew Drescher 和 Muhammad Awad 就这项工作进行的研究讨论并提供反馈。我们也感谢超威半导体公司（Jalal Mahmud 和 AMD Research）提供差旅经费支持，使我们能够参加会议并展示这项工作。

参考文献

[1] 2017。 国际标准化组织国际标准 *ISO/IEC 14882:2017(E) - C* 编程语言++。技术报告。国际标准化组织（ISO）。 https://isocpp.org/std/the-standard。
[2] 超威半导体公司。2022年。HIP 编程指南 v5.2。（2022年6月）。 https://docs.amd.com/bundle/HIP-Programming-Guide-v5.2/page/Introduction.html

[3] Sean Baxter。2013–2016。Moderngpu：GPU计算的模式与行为。（2013–2016）。http://moderngpu.github.io/moderngpu。[4] Guy E. Blelloch。1990。前缀和及其应用。技术报告 CMU-CS-90-190。卡内基梅隆大学计算机科学学院。[5] Ajay Brahmakshatriya、Yunming Zhang、Changwan Hong、Shoaib Kamil、Julian Shun 和 Saman A marasinghe。2021。使用 GraphIt 为 GPU 编译图应用程序。载于 IEEE/ACM代码生成与优化国际研讨会（CGO 2021）。248–261。https://doi.org/ 10 1109 51591 2021 9370321 。/CGO 。 。[6] F. Busato 和 N. Bombieri。 。BFS-K：一种适用于 Kepler GPU架构的高效广度优先搜索实现。IEEE并行与分布式系统汇刊 26，7（2015年7月），1826–1838。https://doi.org/ 10.1109/TPDS.2014.2330597[7] Daniel Cederman 和 Philippas Tsigas。2008。图形处理器上的动态负载均衡。载于 Graphics Hardware（GH '08）。57–64。https://doi.org/10.2312/EGGH/EGGH08/057-064[8] Yuxin Chen、Benjamin Brock、Serban Porumbescu、Aydın Buluç、Katherine Yelick 和 John D. Owens。2022。Atos：一种用于图分析的任务并行GPU调度器。载于国际并行处理会议论文集（ICPP 2022）。https://doi.org/10.1145/ 3545008.3545056 arXiv:2112.00132[9] Steven Dalton、Sean Baxter、Duane Merrill、Luke Olson 和 Michael Garland。2015。使用归并路径优化 GPU 上的稀疏矩阵运算。载于 IEEE国际并行与分布式处理研讨会。 407 416 10 1109 2015 98 – 。[10] https://doi.org/ 。/IPDPS。 。Andrew Davidson、Sean Baxter、Michael Garland 和 John D. Owens。2014。用于单源最短路径的工作高效并行GPU方法。载于第28届 IEEE国际并行与分布式处理研讨会论文集（IPDPS 2014）。349–359。https: //doi.org/ 10 1109 2014 45 。/IPDPS。 。Timothy A. Davis 和 Yifan Hu。2011。佛罗里达大学稀疏矩阵集合。ACM数学软件汇刊 38，1，文章 1（2011年12月），25 10 1145 2049662 2049663 页。https://doi.org/ 。/ 。[12] Salvatore Filippone、Valeria Cardellini、Davide Barbieri 和 Alessandro Fanfarillo。2017。GPGPU 上的稀疏矩阵-向量乘法。ACM数学软件汇刊 43，4，文章 30（2017年1月），49页。https://doi.org/ 10 1145 3017994 。/ James Fox、Alok Tripathy 和 Oded Green。2019。通过对数基数分桶改进不规则应用的调度。2019年 IEEE高性能极限计算会议（2019），1–7。https://doi.org/ 10 1109 2019 8916333 。/HPEC。 。Trevor Gale、Matei Zaharia、Cliff Young 和 Erich Elsen。2020。用于深度学习的稀疏GPU内核。载于高性能计算、网络、存储与分析国际会议论文集（SC '20）。文章 17，14页。https://doi.org/10.1109/SC41405. 2020.00021[15] 谷歌。2023。Chromium C++ 风格指南。（2023）。https://chromium.googlesource.com/chromium/src/+/HEAD/styleguide/c++/c++.md[16] Oded Green、James Fox、Alex Watkins、Alok Tripathy、Kasimir Gabert、Euna Kim、Xiaojing An、Kumar Aatish 和 David A. Bader。2018。对数基数分桶与向量化三角形计数。载于 IEEE高性能极限计算会议（HPEC 2018）。1–7。https://doi.org/ 10 1109 2018 8547581 。/HPEC。 。Oded Green、Robert McColl 和 David A. Bader。2012。GPU归并路径：一种GPU归并算法。载于第26届 ACM国际超级计算会议论文集（ICS '12）。331–340。https: //doi.org/ 10 1145 2304576 2304621 。/ 。Mark Harris 和 Kyrylo Perelygin。2017。协作组：灵活的 CUDA线程编程。（2017）。https://developer.nvidia.com/ blog/cooperative-groups/ [19] Bangtian Liu、Chengyao Wen、Anand D. Sarwate 和 Maryam Mehri Dehnavi。2017。一种用于稀疏张量的统一优化方法 GPU 上的操作。IEEE国际集群计算会议（ 2017 47 57 10 1109 2017 75 年9月），–。[20] https://doi.org/。/CLUSTER.。Duane Merrill 和 Michael Garland。2016。基于归并的并行稀疏矩阵-向量乘法。载于国际高性能计算、网络、存储与分析会议（SC '16）。678– 689 10 1109 2016 57 。https://doi.org/。/SC.。[21] Duane Merrill、Michael Garland 和 Andrew Grimshaw。2012年。可扩展的 GPU 图遍历。载于第17届 ACM SIGPLAN并行编程原理与实践研讨会论文集（PPoPP ' 12 117 128 10 1145 2145816 2145832 ）。–。[22] https://doi.org/。/。Israt Nisa、Jiajia Li、Aravind Sukumaran-Rajam、Richard Vuduc 和 P. Sadayappan。2019年。GPU 上的负载均衡稀疏 MTTKRP。载于 IEEE国际并行与分布式处理研讨会（IPDPS 2019 123 133 10 1109 2019 00023 ）。–。[23] https://doi.org/。/IPDPS.。英伟达公司。–。CUDA C 编程指南。（2007年12月–2022年）。https://docs.nvidia.com/cuda/PG-02829-001_v12.0。[24] 英伟达公司。2023年。CUB：CUDA C 的协作原语++。（2023年）。https://nvlabs.github.io/cub/。[25] Muhammad Osama、Serban D. Porumbescu 和 John D. Owens。2022年。GPU 负载均衡编程模型。（2022年12月）。https://doi.org/ 10 5281 7465053 。/zenodo。[26] Markus Steinberger、Rhaleb Zayer 和 Hans-Peter Seidel。2017年。GPU 上全局均匀、局部自适应的稀疏矩阵-向量乘法。载于国际超级计算会议论文集（ICS 2017）。ACM，13:1–13:11。https://doi.org/10.1145/3079079.3079086 [27] Stanley Tzeng、Brandon Lloyd 和 John D. Owens。2012年。具有依赖关系解析功能的 GPU 任务并行模型。IEEE《计算机》45卷，第8期，2012年，34–41，10.1109/2012（8月），–。https://doi.org/。/MC.255 [28] 2010 Stanley Tzeng、Anjul Patney 和 John D. Owens。...。GPU 上非规则并行工作负载的任务管理。载于高性能图形学会议论文集（HPG '10）。29–37。https://doi.org/10.2312/EGGH/HPG10/029-037[29] Yangzihao Wang、Yuechao Pan、Andrew Davidson、Yuduo Wu、Carl Yang、Leyuan Wang、Muhammad Osama、Chenshan Yuan、Weitang Liu、Andy T. Riffel 和 John D. Owens。2017年。Gunrock：GPU图分析。ACM并行计算汇刊4卷，第1期（2017年8月），3 1 3 49 10 1145 3108140 : – :。https://doi.org/。/ Carl Yang、Aydın Buluç 和 John D. Owens。2018年。GPU 上稀疏矩阵乘法的设计原则。载于 Euro-Par 2018：第24届欧洲并行与分布式计算国际会议论文集，Marco Aldinucci、Luca Padovani 和 Massimo Torquati（编）。672 687 10 1007 978 3 319 96983 1_48 –。[31] https://doi.org/。/- - - - Carl Yang、Aydın Buluç 和 John D. Owens。...。GraphBLAST：GPU 上高性能、基于线性代数的图框架。ACM数学软件汇刊48卷，第1期，文章1（2022年2月），51页。https://doi.org/10.1145/3466795[32] Lingqi Zhang、Mohamed Wahib、Peng Chen、Jintao Meng、Xiao Wang 和 Satoshi Matsuoka。2022 年。面向迭代型内存受限 GPU 应用程序的持久化内核。CoRR（ 2022 2204 02064 年4月）。arXiv:。[33] Tao Zhang、Wei Shu 和 Min-You Wu。...。CUIRRE：用于负载均衡和表征 GPU 上不规则应用的开源库。《并行与分布式计算杂志》74卷，第10期（2014年10月），2951–2966。https://doi.org/10.1016/j.jpdc.2014.07.004

A 制品说明

我们提供了负载均衡框架 loops 的源代码，以及用于评估本文所报告结果的测试框架。

A.1 要求

1. **操作系统** Ubuntu 18.04、20.04、Windows。
2. **硬件** NVIDIA GPU（Volta 微架构或更新版本）。
3. **软件** CUDA 11.7 或更高版本以及 cmake 3.20.1 或更新版本。4. **编译** NVCC（随 CUDA 提供）、g++和 gcc、msvc，支持 C++14标准。5. **输出** 用于生成第6节中图表的逗号分隔值（CSV）文件。6. **磁盘空间** 需要 886 GB 来存储整个 SuiteSparse 矩阵集合 [11]压缩和未压缩版本。若仅使用数据集的子集运行测试，则可显著减少所需空间。

7. **代码许可证** Apache 2.0。

A.2 如何访问

主代码仓库托管于 GitHub：https://github.com/gunrock/loops。我们的框架也以 Zenodo 存档形式提供：https://doi.org/10.5281/zenodo.7465053 [25]。代码仓库中的 README Markdown 文件提供了详细且格式规范的说明，下面则列出其摘要。

A.3 入门

在构建 loops 之前，请确保系统已安装 CUDA 工具包和 cmake，并已将它们的路径导出到系统的 PATH 中。其他外部依赖项（例如 thrust、CUB 等）会由 cmake 自动获取。

```
cd loops mkdir build &&cd build cmake -
DCMAKE_CUDA_ARCHITECTURES =70..
make -j$( nproc )
```

A.3.1 合理性检查。 在 cmake 的构建目录中运行以下命令：

```
二进制目录/ loops . 稀疏矩阵-向量乘法 . merge_路径 \
 -m ../ 数据集 / 切萨皮克 . mtx \
 -- 验证 -v
# 预期输出
# 耗时 ( 毫秒 ): 0.063328
# 矩阵: 切萨皮克 . mtx
# 维度: 39 x 39 (340)
# 错误: 0
```

A.4 复现结果

我们提供以下说明，以便重新生成本文中呈现的结果。

1. 在运行脚本中，将 DATASET_DIR 更新为所有已下载数据集的路径（设置为包含 MM 目录的目录路径；MM 目录中包含若干存放 .mtx 文件的子目录）：scripts/run.sh。

   - 您可以更改 DATASET_FILES_NAME 的路径，该文件包含所有数据集的列表（默认指向 datasets 目录中的 suitesparse.txt 文件）。

2. 在 scripts 目录中使用 run.sh 启动完整运行：cd scripts && ./run.sh。请注意，完整运行最长可能需要 3 天（该运行会使用四种不同的算法，对整个 SuiteSparse 矩阵集合数据集进行四次遍历；主要瓶颈是从磁盘加载文件）。

   - 警告：部分矩阵的运行预计会失败，因为尽管它们被标记为 .mtx 文件，却不符合正确的 MatrixMarket 格式。这些矩阵以及无法装入图形处理器的矩阵会导致运行时异常或类型溢出，可以安全忽略。

3. 若要运行 N 个数据集，只需调整此处的停止条件（默认设置为 10）：run.sh#L22；或者完全移除此 if 条件，以便对所有可用的 .mtx 文件运行：run.sh#L22-L26。

此外，我们还提供了预先生成的结果（以 CSV 文件的形式），无需运行所有实验即可绘制第 6 节中的图表。这些预先生成的结果位于代码仓库的 docs 目录下。

A.5 预期输出和图表

上述运行的预期输出是与 run.sh 位于同一目录中的 CSV 文件。这些文件可以替换 docs/data 中现有的 CSV 文件，并可使用 Python Jupyter 笔记本评估结果。该 Python 笔记本包含生成图表的说明。以下是其中一个 CSV 文件的输出示例：

```
内核，数据集，行，列，非零元素，耗时merge -path，144，
144649，144649，2148786，0.07202merge -path，08 线程块，
300，300，592，0.0170898merge -path，1138_bus，1138，
1138，4054，0.0200195
```