Julia 中适用于任意类型和运算符的高性能可移植 GPU 基元

Emmanuel Pilliat雷恩大学、*Ensai*、法国国家科学研究中心、*CREST—UMR 9194，F-35000*法国雷恩
emmanuel.pilliat@ensai.fr

arXiv:2603.18695v1 [cs.DC] 19 Mar 2026

摘要—**Kokkos 和 RAJA 等可移植 GPU 框架降低了跨架构开发的负担，但与厂商优化的库相比，在基础并行原语上通常会产生可测量的开销。我们介绍 KernelForge.jl，这是一个 Julia 库，通过两层可移植架构实现扫描、映射归约和矩阵-向量原语：KernelIntrinsics.jl 为线程束级洗牌操作、内存栅栏和向量化内存访问提供与后端无关的抽象，而 KernelForge.jl 则完全基于这些接口构建高性能算法。在 NVIDIA A40 和 AMD MI300X 上的评测显示，KernelForge.jl 在 A40 上的扫描和映射归约内核执行时间与 CUB 相当或更短，并且在大多数测试配置下，矩阵-向量运算的吞吐量与 cuBLAS 相当——这证明了作为概念验证，可移植的即时编译抽象能够在不牺牲通用性的前提下实现厂商级吞吐量。**

索引词—**GPU 计算、性能可移植性、并行原语、Julia、扫描、映射归约、矩阵—向量乘积、CUDA、ROCm、即时编译。**

I. 引言

现代科学计算、机器学习和数据分析日益依赖 GPU 加速来达到所需的性能水平。然而，GPU 计算领域已从由单一厂商主导的生态系统，演变为拥有多种竞争架构的异构环境。英伟达的 CUDA 平台、AMD 的 ROCm、英特尔的 oneAPI，以及新兴架构各自提供了不同的硬件能力和编程模型。这种多样性造成了根本性的矛盾：针对某一种架构优化的应用程序，在其他架构上可能无法编译，或性能不佳。

传统做法是为每家厂商的平台维护独立实现，这会带来高昂的开发和维护成本。研究团队和软件开发者长期面临两难：编写可移植但性能可能不佳的代码，或维护多个专用实现以最大限度地提升性能，却增加复杂性。高性能计算领域已对性能与可移植性之间的这一矛盾进行了广泛记录 [1], [2]。

与手动优化的厂商实现相比，现有的性能可移植性框架（如 RAJA 和 Kokkos）可能会产生高达 100% 的开销 [3], [4] ，这是因为抽象层生成的代码并非最优。对于扫描、归约、矩阵—向量或向量—矩阵运算等基础并行原语而言，这一性能差距尤为突出；它们是复杂算法的构件。这些原语支撑着广泛的工作负载，从稀疏线性代数求解器到图分析和排序例程；即使这一层级的开销不大，也会在更高层的应用中不断累积。现有框架的另一项局限是依赖提前编译，这使其难以与 Python 和 Julia 等动态高级语言集成，而这些语言正日益主导科学计算工作流。

具体而言，在 Julia 生态系统中，GPU 编程已通过 CUDA.jl [5] 和 KernelAbstractions.jl [6]等软件包迅速成熟。然而，即使 Julia 原生的 GPU 编译器，由于底层代码生成方面的差异，相较于供应商实现仍会引入持续开销 [7], ；而 Julia 中的可移植 GPU 库在核心原语方面尚未弥合这一性能差距。弥合这一差距将惠及不断壮大的 Julia 高性能计算和机器学习社区，该社区依赖这些原语作为基础设施。

我们表明，与通常的预期相反，性能可移植性既不必以牺牲性能为代价，也不需要掌握复杂的底层语法。通过精心设计并利用 Julia 的高级抽象、元编程能力以及即时编译（JIT），可以实现与手工优化的供应商库在吞吐量上相当的可移植性和性能。

我们提出 **KernelForge.jl** [8],，这是一个面向 Julia 的 GPU 库，实现了基本并行原语，其吞吐量可媲美经供应商优化的实现。KernelForge.jl 的架构围绕三个目标：性能，即在核心原语上匹配供应商优化库；灵活性，即支持任意结合算子和任何 Bitstype 元素——包括矩阵—向量乘积，而供应商库仅限于标准数值运算；以及可移植性，即通过后端无关的抽象表达所有算法，并将供应商特定功能隔离在一个精简的扩展层中。当前实现通过 CUDA.jl 支持 NVIDIA GPU，并通过 AMDGPU.jl 支持 AMD GPU；扩展至英特尔 oneAPI 和其他后端时，仅需提供底层内在函数的后端特定实现，无需更改算法层。本工作的主要贡献如下：

- 一种两层架构：其中 **KernelIntrinsics.jl** [9]为任意类型提供用于向量化内存访问、可配置内存栅栏和线程束级操作的低级、可移植内置功能，而 **KernelForge.jl**仅通过 KernelIntrinsics.jl 和 KernelAbstrac- tions.jl 实现高性能并行算法的

实现。特定于后端的功能通过 Julia 的包扩展机制和 @device_override 限定在 KernelIntrinsics.jl 的扩展模块中，从而确保面向新架构时无需修改算法层。

- 通过实证展示，即时编译的高级抽象在扫描、归约和矩阵-向量原语方面，性能可媲美、在某些情况下甚至超过手工优化的供应商库；测试问题规模从 10_6 到 10_9 个元素，并在 NVIDIA A40 和 AMD MI300X 上进行评估。在 A40 上，我们报告通过 CUDA 事件测得的纯内核执行时间，以及完整流水线耗时（包括内存分配、主机端分派和内核执行）；在 MI300X 上，我们仅报告完整流水线耗时。
- 通过与 CUB、CUDA.jl、Accel- eratedKernels.jl [10], 以及 Kokkos（仅扫描）进行比较评估，结果表明，KernelForge.jl 在 A40 上的扫描、归约和矩阵-向量原语方面，性能可媲美经过供应商优化的实现；在 MI300X 上也实现了有竞争力的吞吐量（该平台上没有扫描和映射归约的供应商优化基准）。

KernelForge.jl 提供了一种基于 Julia 原生多重分派的轻量级调优机制，可在编译时选择特定于架构的参数，且无需更改算法层。

我们强调，KernelForge.jl 是一项概念验证，而非可用于生产环境的成熟库。目前的实现已全面支持英伟达 GPU；通过 AMDGPU.jl 提供的 AMD 支持也已可用，但尚未针对 MI300X 上的扫描和映射归约提供厂商优化的基准测试——本次评估未包含 rocPRIM 基准测试，因此，MI300X 上这些原语的结果证明的是可移植性和正确性，而不能据此声称该平台不存在可移植性开销。矩阵-向量和向量-矩阵原语是例外：AMDGPU.jl 会在内部将 LinearAlgebra.mul! 分派到 rocblas_gemv，从而为这些操作提供有效的 rocBLAS 基准。除 CUDA 和 ROCm 之外的后端尚未经过测试。本文的主要贡献在于架构设计和实证研究：证明可移植性与性能之间的权衡并非不可避免，精心设计的高级抽象层能够在核心并行原语上达到与厂商优化实现相当的吞吐量。

II. 背景

A. GPU 架构

图形处理器执行成千上万个线程，这些线程组织成线程束——执行的原子单元——以及线程块，线程束会被调度到各个流式多处理器（SM）上 [11]。内存层次结构以容量换速度：全局内存容量大但速度慢，共享内存作用域限于线程块且速度快，而寄存器则属于单个线程，速度最快。线程束内的线程可以通过洗牌指令直接交换数据，而无需访问内存 [12]；线程块内的线程束通过共享内存进行同步 [13]；跨线程块协调则需要全局内存栅栏或原子操作 [14], [15]。

内存访问性能很大程度上取决于合并访问——连续且对齐的线程束访问可通过一次事务完成——以及是否发出宽加载（例如 CUDA 中的 128 位 float4），以最大化每笔事务的带宽 [16], [17]。

这三种机制——线程束级原语、内存栅栏和向量化内存访问——是实现与供应商竞争的并行原语所需的底层能力，也是 KernelIntrinsics.jl（第 IV 节）的重点。要进一步达到峰值性能，还需要根据硬件特性调整线程块数量和每线程处理项数；这些特性包括线程束宽度和 L2 缓存大小，且会因架构而异。

B. Julia 中的跨架构 GPU 编程

Julia 的 GPU 生态系统由分层的软件栈构成。在最底层，供应商专用软件包——面向英伟达的 CUDA.jl、面向 AMD ROCm 的 AMDGPU.jl、面向英特尔的 oneAPI.jl，以及面向苹果的 Metal.jl——提供对各平台编程模型和供应商库的全面访问。其上，KernelAbstractions.jl [6] 提供了一种后端无关的内核语言：通过 Julia 的多重分派和即时编译，同一个内核定义可以编译到任何受支持的后端。

KernelAbstractions.jl 提供了一种编程模型，用户可通过 @kernel 宏定义内核。内核由后端参数化（例如 CUDABackend()、ROCBackend()），并在首次调用时针对目标设备进行编译。该框架提供工作组级抽象——包括工作组索引、局部索引和 @synchronize 屏障——这些抽象在英伟达设备上分别映射为线程块、线程局部索引和__syncthreads()，在其他后端上也有类似映射。共享内存通过 @localmem 分配，在 CUDA 上映射为 __shared__ memory，在 AMD 上则映射为本地数据共享内存。

清单 1：KernelAbstractions.jl 中的复制内核，摘自 GPUArraysCore.jl。仅需一个源代码定义，该内核即可编译为 NVIDIA PTX、AMD GCN、Intel SPIR-V 或 Apple AIR。

```
1 @kernel function 复制_内核!(dst, src)2 I  =
@index(Global)3     dst[I] =src[I]4 end
```

清单 1 展示了这种编程模型的简洁性。该抽象涵盖了 GPU 编程领域的很大一部分，使许多工作负载能够使用可移植内核。然而，要让基础原语达到与厂商竞争的性能水平，仍有几项必不可少的能力未得到公开：

- **线程束级洗牌操作**。目前没有适用于 shfl_sync、shfl_down_sync 或其等效操作的可移植抽象。需要在寄存器级数据

在线程束内进行交换的内核——这对快速归约和扫描至关重要——必须退而使用供应商特定内建函数，从而破坏可移植性。

- **有序内存访问**。不支持用于控制线程、线程块或设备间内存操作可见性顺序的细粒度内存栅栏（threadfence()、threadfence_block()）。解耦回看等块间协调协议需要这些栅栏，例如单遍扫描算法 [18] 中使用的协议。

- **向量化加载与存储**。没有机制可以生成 64 位或 128 位加载/存储指令。复制和带宽受限内核要达到峰值内存带宽，就需要这一能力。

这些不足促使我们开发了**KernelIntrinsics.jl** 层（第 IV 节），该层提供全部三种能力的可移植、按后端分派的实现，使 KernelForge.jl 能够构建具有厂商竞争力的算法，同时不牺牲可移植性。

C. 并行原语

除了复制操作——它是衡量其他原语时实际采用的带宽上限（参见图 1）——还有几种基础操作构成了 GPU计算的支柱，其作用类似于线性代数中的 BLAS [19] 。我们重点关注三种操作：映射归约、扫描以及矩阵–向量/向量–矩阵乘积。

映射归约 操作通过映射函数 $f: T \to S$转换每个元素，然后使用交换且结合的运算符 op 合并所有转换后的值（这里要求满足交换律，而扫描只要求满足结合律），以计算 $\mathrm{op}(f(\mathrm{src}[1])\ f(\mathrm{src}[2])\ldots, f(\mathrm{src}[n]))$。高效的图形处理器归约采用分层方式：线程首先在寄存器内进行归约，然后通过洗牌操作在线程束内进行归约，接着在一个线程块内借助共享内存跨线程束进行归约，最后跨线程块进行归约。利用线程束级原语的实现，相比仅使用共享内存的基于树的归约，速度最高可提升 $7.8\times$ [12], [20]。

扫描 （前缀和）操作放宽了交换律要求，接受一个结合算子 op（不一定满足交换律），并计算一系列部分归约：输出 $\mathrm{dst}[i] = \mathrm{op}(\mathrm{src}[1], \mathrm{src}[2], \ldots, \mathrm{src}[i])$ 包含截至位置 i for all $1 \le i \le n$ 的累积结果。扫描是最重要的并行原语之一：它支持流压缩、基数排序、直方图计算和稀疏矩阵运算 [21], [22]。当前最先进的方法是采用解耦回看的单遍并行前缀算法 [18], ，通过仅需约 $2n$ 的数据搬运（读取 n 个输入、写入 n 个输出），并通过策略性地执行冗余工作，将局部计算与全局前缀传播延迟相分离，从而实现接近复制操作的吞吐量。该协议要求通过全局内存进行线程块间通信，并谨慎使用内存栅栏和状态标志——而这正是现有可移植框架所缺少的能力，参见第 II-B 节。

最后，我们将**矩阵–向量** 和**向量–矩阵** 乘积视为不同的原语。给定一个 $n \times p$ 矩阵 A、一个向量 $x \in T^n$、一个映射函数 $f\colon T \times T \to S$，以及定义在 S 上的结合归约算子 op，矩阵–向量乘积计算向量 $y \in S^p$，其中 $y[j] = \mathrm{op}_{i=1}^{n} f(x[i], A[i,j])$，并对 A 的各行进行归约。向量–矩阵乘积则对称地计算 $z \in S^n$，其中 $z[i] = \mathrm{op}_{j=1}^{p} f(A[i,j], x[j])$，并对各列进行归约。令 $f = \times$ 且 op 为 $= +$，即可得到标准 BLAS GEMV 操作；但这一广义表述支持任意代数结构，例如用于最短路径算法的热带半环（其中 $f = +$ 且 op 为 $= \min$），或用于保证数值稳定性的对数空间运算。这两个操作在图形处理器上并不 对称：由于数组按列主序存储，矩阵–向量乘积沿列进行步长为 1（合并）的读取，而向量–矩阵乘积必须沿行读取，从而产生跨步访问模式。要让两种方向都实现高吞吐量，需要采用不同的内核策略，详见第 V-C 节。

III. 相关工作

A. C++ 性能可移植性框架

GPU 计算中的性能可移植性通过各种 C++框架得以实现，各框架体现了不同的设计理念。RAJA [23] 通过执行策略将算法实现与执行策略解耦，从而实现可移植性。Kokkos [24] 提供了一个覆盖面更广的框架，对执行空间和内存空间都进行了抽象。其他追求类似目标的框架还包括 OpenACC、OpenMP 和 SYCL。尽管各有不同，这些框架都有一个根本性限制：它们依赖提前编译，因此难以与 Python 和 Julia 等动态语言集成。

性能研究揭示了一种一致的规律：与手工优化的供应商实现相比，可移植框架通常会产生额外开销。Martineau 等人 [3] 发现，与针对特定架构编写的代码相比，性能会下降 5–30%；Artigues 等人 [4] 报告称，与原生 CUDA 相比，在 NVIDIA V100 上性能下降了 2–3× 。Davis 等人 [25] 证实，在 V100、A100 和 H100 GPU 上的五种代理应用中，Kokkos 的表现均略逊于 CUDA。Sedova 等人 [1]得出结论，非可移植优化对于生产级分子动力学代码至关重要。这些研究印证了传统观点：使用抽象层时，可移植性开销不可避免。

B. 供应商优化库

对于一维并行原语（复制、归约、扫描），英伟达的 CUB [26] 和 Thrust 库提供了高度优化的 CUDA 实现，通过精心利用线程束级操作、共享内存分 bank 模式和指令级并行，达到接近峰值的性能。AMD 的 rocPRIM 和英特尔的 oneDPL 则为各自的平台提供了类似的功能。多个厂商特定库彼此 API 不兼容，这凸显了

一个根本性挑战：目前要实现峰值性能，必须针对每家厂商分别编写实现。

对于矩阵–向量运算，cuBLAS、rocBLAS 和 oneMKL 等供应商库提供了经过优化的 GEMV 例程，但仅支持采用固定算术规则的标准数值类型（$f = \times$、$\oplus = +$）。需要非标准代数结构的应用——例如用于最短路径问题的热带半环，或用于提高数值稳定性的对数空间运算——无法使用这些库，只能转而采用自定义内核，从而无法享受供应商级别的优化。KernelForge.jl 的广义公式涵盖了标准 GEMV，同时将其扩展到任意类型和运算符，并且不会牺牲标准数值场景下的性能。

C. Julia GPU 生态系统

Julia GPU 生态系统通过多个软件包提供全面的 GPU 支持。CUDA.jl 同时为英伟达 GPU 提供高级数组抽象和低级内核编程功能。KernelAbstractions.jl [6] 通过统一的内核语法和后端分派，将可移植性扩展到多个后端（CUDA、ROCm、oneAPI、Metal）。AcceleratedKernels.jl [10] 构建于 KernelAbstractions.jl 之上，为常见运算提供可移植的实现。然而，现有的 Julia GPU 库更重视通用性而非优化：CUDA.jl 的实现偏向简洁，而 AcceleratedKernels.jl 则专注于广泛的可移植性，而不是达到供应商库的性能水平。

D. 我们的贡献

KernelForge.jl 与上述框架有两点不同。首先，与采用提前编译的 C++ 可移植性层不同，它利用 Julia 的即时编译，在调用处生成针对具体数据类型、运算符和问题规模特化的代码，从而消除抽象开销；现有研究报告的可移植性开销为 5–30%，这部分开销也因此得以避免。其次，与优先拓展 API 覆盖面的 Julia 图形处理器库不同，KernelForge.jl 专注于一组范围有限的原语——扫描、映射归约和矩阵—向量乘积——并通过双层架构对其进行优化：KernelIntrinsics.jl 将所有供应商特定功能（线程束洗牌、内存栅栏、向量化加载）限制在一个精简的扩展层中，而算法层则完全构建于这些可移植抽象之上。在 A40 上，这种设计在全部三种原语上都达到了与 CUB 和 cuBLAS 相当的吞吐量；在 MI300X 上，相较于现有的 Julia 基准实现，它在映射归约和扫描上实现了有竞争力的性能，并在宽矩阵配置下的矩阵—向量运算中达到与rocBLAS相当的性能。

IV. 跨架构内核内在函数的设计

A. 洗牌操作

线程束内的洗牌操作允许某个通道根据源索引或掩码，直接从另一个通道的寄存器中读取值。然而，厂商实现的 shuffle 内建函数仅针对一组有限的类型提供支持——通常是 Float32 和 Int32 等 32 位原语类型。

CUDA.jl 通过一种递归机制部分解决了这一限制，该机制将洗牌操作扩展到 Int64 或复数等类型，但仍不支持元组或四元数等更复杂的复合类型。

KernelIntrinsics.jl 利用 Julia 的 @generated 函数，在编译时递归分解任意复合类型，从而推广了这一方法。关键在于，一旦为单个 32 位原语（例如 UInt32）定义了 shuffle 内建函数，就可以将任意复合类型分解为其组成的原语字段，并分别对每个字段执行洗牌操作，从而支持该类型。我们通过递归定义对这一方法进行形式化：*Bitstype* 是指不超过 64 位的具体原语、由 Bitstype 组成的元组，或字段均为 Bitstype 的结构体。借助 @generated 函数，编译器会静态展开针对结构体字段和元组元素的递归，为每种具体类型生成专用的零开销代码。根据此定义，KernelIntrinsics.jl 可对任意 Bitstype 执行洗牌操作，完全屏蔽特定供应商的类型限制，且没有运行时开销。

这种通用的 shuffle 机制完全通过寄存器实现快速的线程束级通信，可处理任意复杂的数据类型，无需借助共享内存。KernelForge.jl 基于这一能力提供了高层接口：用户只需编写 mapreduce(f, op, src) 或 scan(f, op, src)，即使 src 的元素是复杂结构、元组或其嵌套组合，也能获得正确且高效的结果。

B. 有序内存访问

线程存储一个元素时，该写入不会立即对所有其他线程可见。该值首先写入 L1 缓存，此时只有同一工作组中的线程能够看到它。随后，该值会传播到 L2 缓存，而 L2 缓存由多个工作组共享。这会造成潜在的通信危险：在更新后的值到达 L2 缓存之前，另一个工作组中的线程可能会从自己的 L1 缓存中读取到过时的值。

考虑一种生产者-消费者模式：一个工作组将数据写入全局内存，然后设置一个标志以示完成。对标志的存储采用释放语义，可确保之前的所有写入对随后观察到该标志的任何线程都可见。对相应加载采用匹配的获取语义，可确保之后的所有读取都能获取最新值。释放-获取对共同在工作组之间建立先行发生关系，而无需使用完整的系统范围栅栏。

KernelIntrinsics.jl 通过 @access 宏公开了这一模式：@access flag[i] = 0x01 会发出释放存储，而 x = @access flag[i] 会发出获取加载。在英伟达 GPU 上，它们会直接降低为 PTX 内存排序注解——例如，存储操作使用 st.release.gpu.global，而 GPU 作用域栅栏使用 fence.acq_rel.gpu。

在 AMD GPU 上，等效语义通过 GCN 作用域位（sc1/sc0）结合显式缓存刷新和失效指令（buffer_wbl2、buffer_inv）来表达。这两种降低形式都会在 KernelIntrinsics.jl 的测试套件中于汇编层面进行验证，从而确认 Julia 的编译流水线不会引入意外的栅栏或重排序。对于无法使用细粒度顺序标注的后端，@access 会回退到完整的内存栅栏；这种方式的精确性较低，但无需更改算法层，仍可保持相同的正确性保证。

C. 向量化内存访问

图形处理器内核的性能受计算吞吐量或内存带宽限制 [27], [28]。对于能够使内存带宽饱和的大型问题，向量化加载至关重要：每个线程发出宽事务（例如，加载四个 Float32 值的 128 位加载），而非标量加载，以最大限度地提高带宽利用率，尤其是对于能够放入 L2 缓存的问题。KernelForge.jl 使用静态参数 Nitem 定义内核，用于控制每个线程处理的元素数量，并通过 Julia 的 Val 类型在编译时进行特化，且没有额外开销。最优 Nitem 不一定由向量加载宽度决定：扫描内核每个线程处理 16 个 Float32 值，因为依次处理更多元素可以将同步开销分摊到各通道和线程束上。图 1 通过复制内核展示了这一点：128 位加载可最大限度地提高带宽，而 KernelForge.jl 的性能与 CUDA.jl 内部的 libcuda 实现相当或更优。

向量化加载要求连续内存访问。对于跨步子数组，KernelForge.jl 会退回到将各个元素逐一加载到元组中，放弃向量化加载，但保留每线程处理多个元素的结构及其同步优势。

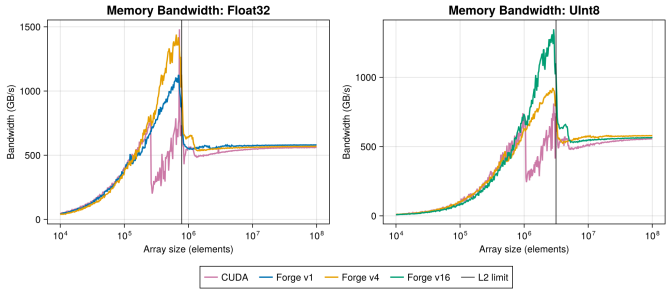

图 1：**向量化复制的带宽（GB/s）随问题规模的变化**。通过使用 CUDA.@profile 对内核计时测得的实际带宽，展示了 CUDA.jl（其内部调用 libcuda）以及每个线程处理 1、4 和 8 个元素时 KernelForge.jl 的表现。虚线表示 L2 缓存大小除以 2 ×sizeof(element)。采用 128 位加载时可达到峰值带宽，此时 KernelForge.jl 的表现优于 CUDA.jl。

列表 2：KernelForge.jl 中每个线程处理 4 个元素的向量化复制内核。vload 和 vstore! 会透明地处理对齐。

```
@内核函数 vcopy!(dst, src) I =@索引(Global)
    vals =vload(src,I,4)
vstore!(dst,I,vals) # 如果 I == ndrange 结束
```

D. 对齐约束

对于适合放入 L2 缓存的问题，向量化加载与存储可带来显著的性能提升，但也会施加严格的对齐要求。PTX 指令集架构规定，任何内存访问的地址都必须按访问大小的倍数对齐 [29]。例如，ld.v4.f32 指令会加载 16 字节，因此起始地址必须按 16 字节对齐。实际上，这意味着执行 128 位 Float32 元素加载的线程，只能从索引为 4 的倍数的元素处开始读取（使用从 0 开始的索引）。从索引 2 或 3 开始加载会违反此约束，并导致未定义行为或硬件故障。

更一般地，对于一次加载 Nitem 个元素、每个元素大小为 s 字节的向量化加载，字节地址必须是 Nitem $\times s$的倍数。这给矩阵运算带来了一个关键问题：由于矩阵的行数 n 可能不是 Nitem 的倍数，第二列（以及后续列）的起始地址即使第一列已正确对齐，也无法保证满足对齐约束。

一种自然的解决方案是在运行时将每次未对齐加载分解为一系列更小且正确对齐的加载。例如，若 Nitem $= 4$，且起始索引 i 的未对齐量为 2（即，$(i-1) \bmod \text{Nitem} = 2$），则该加载可拆分为在 i处进行的标量加载、在 $i+1$处进行的 2 元素向量化加载，以及在 $i+3$处进行的标量加载。KernelIntrin- sics.jl 通过函数 vload_pattern 将此形式化；该函数是一个 @generated 函数，可针对静态已知的对齐模式生成最优加载序列。该模式表示为一个整数元组，其各项之和为 Nitem（例如，上述示例中的 (1，2，1)）。外层 vload 函数随后通过编译时生成的开关表，在运行时分派到相应的 vload_pattern 特化，从而仅分支选择发生在运行时，而每个加载序列本身都已完全特化。

关键在于，这一机制对用户完全隐藏。如代码清单 2 所示，在 KernelForge.jl 中，向量化复制内核只需调用 vload 和 vstore!—— 对齐处理、模式分解和特化均由库透明地管理，并在编译时通过多重分派完成。

V. 快速并行原语的设计

A. 映射归约

MapReduce 原语（第 II-C 节）采用固定线程块网格实现（例如，每个架构经过调优后使用 100 个线程块，每个线程块含 256 个线程）。每个线程以总线程数为步长遍历输入数组，并在寄存器中累积部分结果。这种固定网格策略无需扩大启动配置，即可处理任意大小的数组。

在每个线程块内，部分结果采用分层方式进行归约：首先使用洗牌操作在每个线程束内的线程之间进行归约，然后通过共享内存在各线程束之间进行归约，最终每个线程块得到一个部分值。KernelIntrinsics.jl（第 IV-A 节）提供了线程束级洗牌操作，并支持任意复合类型，且不会产生运行时开销。

块间聚合无需进行第二次内核启动。每个线程块将其部分结果写入全局内存，并设置一个 UInt8 完成标志；该标志会在启动前初始化为零。通过 @access 宏发出的释放语义存储可确保，在其他线程块观察到该标志之前，部分结果已对所有其他线程块可见。指定的线程块通过获取语义加载（同样使用 @access）不断检查所有标志，并在确认所有部分结果均已可见后执行最终归约。这种单次启动设计避免了 CUDA.jl 所采用的双内核序列。

该实现根据输入维数在三条路径之间进行选择。一维数组采用上述策略。二维数组则分派到矩阵-向量和向量-矩阵内核（第 V-C 节），这些内核已针对沿两个轴的合并访问和线程束级并行进行了优化。高维数组和任意迭代器则回退到通用归约内核；与专用路径相比，这会带来适度的性能开销。

B. 扫描

扫描原语使用任意结合算子，对输入数组执行包含式或排除式前缀归约。它与映射归约共享相同的构件——向量化加载、线程束级洗牌操作，以及通过 @access 实现的释放/获取标志同步——并借助 Merrill 和 Garland 的解耦回看算法 [18]，实现最佳单遍吞吐量。

输入被划分为每个包含 $256 \times$Nitem 个元素的瓦片，每个线程块处理一个瓦片。每个线程使用 vload（第 IV-C 节）加载 Nitem 个元素，在寄存器中完全累积局部前缀，然后参与线程束洗牌归约和共享内存交换，以计算瓦片聚合值。因此，全局内存对每个元素恰好读取一次。完成后，指定线程存储瓦片聚合值，并通过释放存储（@access flag[i] =PARTIAL）设置 部分状态标志。

全局前缀传播通过回看阶段进行。单个线程束会向后扫描前面 32 个瓦片（在 AMD wavefront-64 硬件上为 64 个），通过获取加载对每个状态标志进行自旋等待（@access flag[j]），直到至少有一个部分标志变为可见。如果前面的 32 个（或 64 个）瓦片都只设置了部分标志，线程束就会通过基于 shuffle 的归约来归约它们的聚合值，并继续回看下一组瓦片。一旦线程束遇到带有 前缀 标志的瓦片——这表示截至该瓦片的完整包含式前缀已可用——就能立即完成当前瓦片的包含式前缀，无需检查任何更早的瓦片。完成的前缀会写入全局内存，而该瓦片会通过释放存储设置自身的前缀标志，从而解除对所有后续瓦片的阻塞。该方案的正确性依赖于 @access 所建立的释放 - 获取顺序（第 IV-B 节）：若没有这一顺序，瓦片可能会观察到已设置的标志，而关联的聚合值在远程 L1 缓存中仍然是陈旧的。

获得包含式前缀后，每个线程完全在寄存器中计算其最终输出值，并使用向量化存储将其写回目标数组，因此全局内存对每个元素恰好写入一次。

由此得到的实现适用于任意结合算子，包括四元数乘法等非交换情形，并且在标准类型上的吞吐量可与 CUB 媲美（第 VII 节）。

C. 矩阵-向量与向量-矩阵乘积

矩阵-向量与向量-矩阵乘积是神经网络推理和训练的核心，通常由 cuBLAS 等高度优化的专有库提供支持。然而，这些库受供应商锁定，且仅支持标准数值运算。KernelForge.jl 提供开放源码实现，在 CUDA 上针对标准类型实现了与 cuBLAS 相当的吞吐量，同时支持任意元素类型和结合算子对。这种通用性使其能够支持超越标准 $(\times,\ +)$ 半环的应用场景，例如用于最短路径计算的热带半环运算，或用于提升数值稳定性的对数空间累加，同时不牺牲标准数值场景下的性能。

由于矩阵采用列主序存储，水平归约（矩阵向量乘积）和垂直归约（向量-矩阵乘积）会沿不同轴访问内存，因此需要使用具有不同合并访问策略的不同内核（第 II-C 节）。最佳线程组织方式还取决于矩阵形状。对于高而窄的矩阵，每一列都类似于一个独立的一维归约：映射归约中使用的相同固定网格线程块步进方式可以直接应用，每列分配少量线程块（默认值：100，可针对不同架构进行调整）。对于宽而短的矩阵，则必须沿两个维度分配工作，以确保所有线程都处于忙碌状态；图 2 展示了向量-矩阵乘积情形下的这种布局。KernelForge.jl 会在内核启动时根据矩阵形状选择适当的策略，并通过 Julia 的 Val 类型静态解析分派，使编译器能够分别对每条路径进行专门化处理，且不会产生运行时开销。

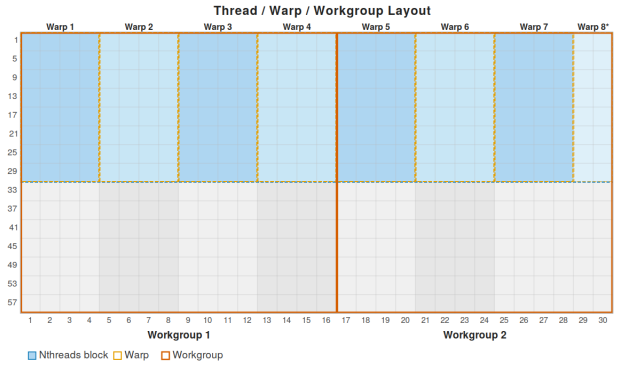

图 2：**宽而短的矩阵的向量-矩阵线程组织**。 x 轴表示列，y 轴表示行。每个线程束（黄色虚线边界）分配 4 个连续列；线程沿行方向跨步，蓝色和灰色区域分别对应第一和第二个行步幅，每个线程在每个步幅中加载 4 个元素。因此，每个包含 128 个线程的工作组（橙色实线边界）覆盖 16 列。这种布局在保持合并内存访问的同时，使所有线程在多个步幅中都保持活跃。

VI. 正确性验证

使用灵活的操作符实现快速并行原语会带来显著的测试复杂性，因为开发过程中自然会出现两类截然不同的错误。

第一类关注算法正确性：验证大型数组上的结果，处理诸如包含 31 或 33 个元素、跨越线程束边界的边界情况，并确保不存在竞态条件。我们不采用超出本文范围的形式化证明，而是采取全面的经验性方法，测试各种数组大小和标量类型。

第二类关注编译有效性。Julia 在生成 PTX 或 GCN 代码之前会先通过 LLVM 编译；任何 LLVM 无法正确降级的指令都会引发错误，而这类错误往往难以诊断——从明确的 IR 故障，到无声的设备级故障，后者会表现为虚假的越界访问或未对齐内存操作；其中任何一种都可能在没有明确错误消息的情况下导致 Julia 会话崩溃。用户提供的运算符、具有非标准对齐方式的自定义结构体，或具有复杂内存布局的数组视图，都可能触发此类故障。我们的测试套件 [8] 涵盖了这三种情况：自定义运算符、特意设置为未对齐的结构体，以及连续和非连续数组视图。

关键的是，同一套测试套件无需修改即可在英伟达硬件上运行，在 AMD 上也取得了有竞争力的结果，这验证了 KernelForge.jl 无需任何特定后端测试路径，就能在 A40（CUDA）和 MI300X（ROCm）上产生正确结果。这种跨架构验证证实，KernelIntrinsics.jl 抽象层的可移植性保证在实践中确实成立：正确性并非某个后端所独有的属性，而是共享算法层的特性。

VII. 性能评估

我们将 KernelForge.jl 与多个参考实现进行比较：CUDA.jl，它提供开源的映射归约和累积实现；AcceleratedKernels.jl，一个跨架构的 Julia库；以及 cuBLAS，作为矩阵-向量和向量-矩阵运算的参考实现（对于具体类型，这些运算由 CUDA.jl 在内部调用）。此外，我们还使用 [8] 的 perf 文件夹中提供的 nvcc 基准测试直接对 CUB 进行基准测试。对于扫描原语，我们还纳入 Kokkos [24], [30], ，这是一个广泛使用的 C++ 性能可移植性框架，作为另一项比较对象。

A. 实验设置

a) 硬件平台： 实验在两个图形处理器平台上进行。**NVIDIA A40** 是一款 Ampere 代图形处理器，配备 48 GB GDDR6 内存，峰值内存带宽为 696GB/s，在 Ubuntu 22.04 上运行 CUDA 12.8。**AMD MI300X** 是一款 CDNA3 代图形处理器，配备 192 GB HBM3 内存，峰值内存带宽为 5.3 TB/s，在 Ubuntu 24.04 上运行 ROCm。这两个平台代表了 GPU 领域中截然不同的两类硬件：一个是拥有成熟 CUDA 工具链的主流数据中心 GPU，另一个是具有高内存带宽但软件生态系统尚不成熟的加速器。

b) 测量方法： 性能通过两个互补的度量指标进行评估。

图形处理器执行时间 衡量设备上内核执行所需的时间，不包括启动开销，反映生成代码的质量，也代表了内核被反复调用的生产场景。对于使用 CUDA 的 Julia 库，该指标通过 CUDA.@profile 宏获取，其内部依赖 CUDA 事件；对于 CUB，则直接使用 CUDA 事件。在 AMD 后端上，计时使用 HIP 事件（ AMDGPU.HIP.HIPEvent），测量设备端的耗时，其中包括内核启动开销。与原始 HIP 事件相比，CUDA.@profile 宏能够以更高精度隔离单个内核片段；然而，AMDGPU.jl 中尚无成熟的等效性能分析基础设施。对于提供预分配接口的所有库，均不计入内存分配；在 AMD 后端上，这一点尤为重要，因为 AMDGPU.jl 的设备内存分配开销远高于其 CUDA 对应项。因此，对于未提供预分配接口的库，其基准测试结果应解读为可实现内核时间的上限，而非直接比较结果。该度量指标在两个平台上均有报告。

端到端流水线时间用于衡量包含 CPU 端开销在内的总墙钟时间：临时内存分配、内核启动、从设备到主机的结果传输，以及 CUDA.jl API 调用。由于 Julia 的即时编译模型，首次编译产生的开销 [31]不计入其中，因为这些开销会在重复调用中摊销。该度量指标仅在 A40 上报告；在 ROCm 上，CPU 端计时的方差显著更高，因此不

予报告。请注意，端到端流水线无法直接与 CUB 的计时结果进行比较，因为后者仅衡量设备端执行。

c) 调优参数： 要实现与供应商优化库相当的性能，需要仔细调节各项参数，例如每个线程处理的元素数量、每个线程块中的线程数量，以及每个流式多处理器上的线程块数量。这些参数取决于硬件特定的特性，例如 L2 缓存大小、寄存器文件容量和内存带宽（见图 1），并且每项参数都必须从二的幂次组成的离散集合中选择，从而形成组合式调优空间。对于矩阵-向量和向量-矩阵内核而言，这一点尤为明显，因为它们包含多个相互依赖的参数，用于控制在线程、线程束和线程块之间划分工作。

KernelForge.jl 通过架构分派层次结构（A40 <: Ampere <: AbstractArch）解决了这一问题，该层次结构利用 Julia 的多重分派，在编译时选择默认参数，类似于 CUB 针对不同 PTX 版本的调优策略 [32]。第 VII-B 节中使用的参数经过手动调优，以匹配 A40 上供应商的性能；若要在新架构上达到相同性能，则需要针对该目标进行同等程度的调优工作。自动调优可以减轻这一负担，也是未来工作的一个自然方向，但不在本文的研究范围内。

B. A40 上的结果

a) 映射归约：图 3 展示了 MapReduce 原语的基准测试结果，对比了 KernelForge.jl 与 CUDA.jl、AcceleratedKernels.jl 和 CUB 在输入规模$n \in \{10^7, 10^8\}$ 下的表现，以及两种数据类型：Float32 和自定义的 UnitFloat8 Julia 类型（CUB 使用 UInt8）。

在 Float32 上，各实现于这些规模下均取得了相近的吞吐量。KernelForge.jl 的表现与 CUB 相当，略优于 CUDA.jl 和 AcceleratedKernels.jl。单次启动标志式设计（第 V-A 节）在较小规模下优势更为明显，因为此时块间同步开销占主导地位，如表 III 所示。

我们还评估了自定义 8 位类型的性能。UnitFloat8 使用 256 个均匀间隔的级别对值进行编码 $[-1, 1]$ ；求和前会将元素提升为 Float32，以避免溢出。由于 UnitFloat8 在 KernelForge.jl 中实现，且 CUB 没有对应类型，因此 CUB 基准测试采用原始 UInt8 求和，作为参考下界。正确性通过检查结果符号是否与 CPU 上的参考 Float64 计算一致来验证。

KernelForge.jl 在原生 UInt8 上达到了与 CUB 相当的表现，在 $n = 10^8$ 时，性能比 CUDA.jl 快 $1.8\times$ ，比 AcceleratedKernels.jl 快 $3.8\times$ 。值得注意的是，这种提升不会带来可测量的额外开销：在这一规模下，内核受内存带宽限制，因此额外的算术运算完全被内存延迟掩盖。这得益于向量化内存访问：每个事务加载多个 8 位元素，使有效带宽与 32 位运算相当。

b) 扫描： 图 4 展示了输入规模为 $n \in \{10^7, 10^8\}$ 、包含两种数据类型 Float32 和 Float64 时，前缀扫描原语的基准测试结果。

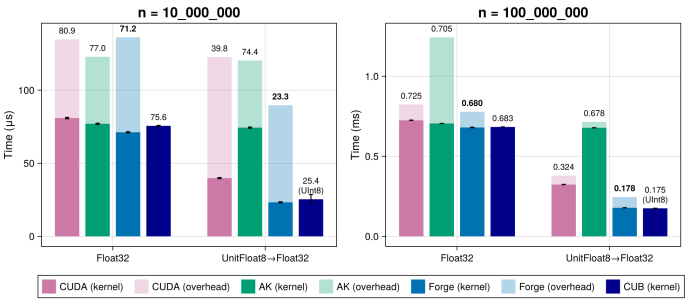

图 3：**跨实现归约基准测试（A40）**。对四种并行求和实现进行比较：CUDA.jl、AcceleratedKernels.jl、KernelForge.jl 和 CUB（使用 nvcc 编译）。结果展示了两种输入规模： $n = 10^7$ （左）和 $n = 10^8$（右），以及两种数据类型：Float32 和 UInt8。深色柱表示内核执行时间；浅色柱包含启动开销。误差条表示不同运行之间的波动。由于 UnitFloat8 是 Julia 专用类型，CUB 基准测试使用虚拟 UInt8 求和作为参照。

在所有配置下，KernelForge.jl 的表现都在测量误差范围内与 CUB 相当，证实了解耦式回顾算法（第 V-B 节）能够针对这两种类型实现厂商级吞吐量。CUDA.jl 采用多次启动的“先归约后扫描”策略，在 $n = 10^8$ 下，Float32 慢 $3.4\times$ ，Float64 慢 $3.9\times$ 。

AcceleratedKernels.jl 的默认扫描使用顺序的线程块间累加阶段，在规模较大时会成为瓶颈。这一点在 $n = 10^8$ 的 Float64 上最为明显，其速度比 KernelForge.jl 慢 $14.9\times$ 。在 $n = 10^9$ 处，差距进一步扩大：KernelForge.jl 在 Float64 上相较于 AcceleratedKernels.jl 的加速比最高可达 $\mathbf{140}\times$ （表 IV）。AcceleratedKernels.jl 确实提供了更适合大规模输入的解耦回看变体；由于该变体并非默认实现，我们未在此处对其进行基准测试。

Kokkos 在 $n = 10^8$ 的 Float64 上达到 7.4ms，相较于 KernelForge.jl 和 CUB 慢 $2.6\times$ 。Kokkos 从源代码构建（版本 4.6.x，这是基准测试时的最新版本），使用了 -DCMAKE_BUILD_TYPE=Release、-DKokkos_ARCH_AMPERE86=ON 以及 C++20。Kokkos 基准测试报告临时存储使用量为零，这与以下事实一致：Kokkos::parallel_scan 在 CUDA 上会通过 Kokkos::Impl::CudaScan 执行——这是一个自定义实现，不会调用 cub::DeviceScan，因此不会使用解耦回看算法。在所有测试规模（$n = 10^7$、10^8、10^9）下，这一差距都保持一致，证实性能差异源于算法选择，而非抽象开销。

c) 向量-矩阵与矩阵-向量运算： 图 5 和图 6 对比了 KernelForge.jl 与 cuBLAS 在 Float32 向量-矩阵乘法和矩阵-向量乘法中的表现（矩阵-向量乘法：行向量 $\times$ 矩阵；向量-矩阵乘法：矩阵 $\times$ 列向量

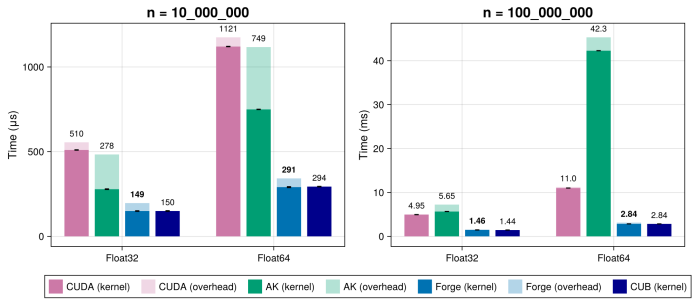

图 4：**不同实现的扫描基准测试（A40）**。实现与归约基准测试相同。每种算法均使用两种数据类型进行测试：Float32 和 Float64。作为参考，Kokkos 在 Float64 下，对 $n = 10^7$、10_8 和 10^9 的耗时分别为 0.84ms、7.4 ms 和 73.4 ms，相较于 KernelForge.jl 和 CUB，开销为 a $2.6\times$ 。

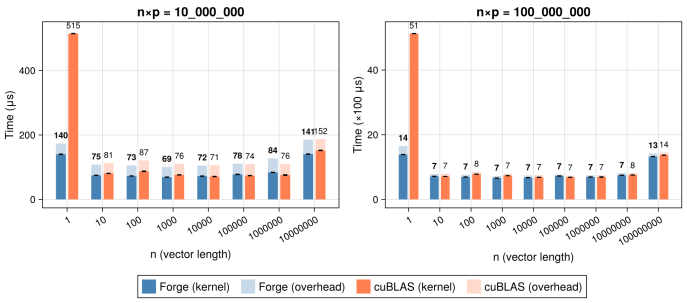

图 5：**不同矩阵形状下的向量矩阵乘积基准测试（A40）**。通过 CUDA.jl 使用的 cuBLAS 与 KernelForge 在 Float32 向量矩阵乘法中的吞吐量对比。总输入数据量 $n \times p$ 固定为 10^7（左）和 10^8 （右），同时改变长宽比，以评估不同内存访问模式下的性能。

），其中总输入规模 $n \times p$ 在所有长宽比下固定为 10^7 和 10^8,。完整结果见表 V 和表 VI。

对于所有非退化长宽比，KernelForge.jl 在 $n \times p = 10^8$ 时的吞吐量与 cuBLAS 相当。在 $n \times p = 10^7$ 时，差异视形状而定，幅度在 $\sim 15\%$ 以内，双方均无持续优势。这是第 V-C 节所述线程划分方案的直接结果；其静态参数通过架构分派系统针对 A40 进行了调优。

退化情形 $n = 1$ （向量-矩阵乘法）和 $p = 1$ （矩阵-向量乘法）可简化为普通内存复制，而 cuBLAS 不会为此调用优化路径；在这些配置下，KernelForge.jl 的内核时间最多可降低 $3.7\times$。列出这些情形仅为完整起见，实际使用时应优先采用专用复制内核。

在 $n \times p = 10^9$ 上，对于矩阵向量乘积，除 $(n,\ p) = (10^4, 10^5)$ 外，KernelForge.jl 在所有长宽比下都能达到与 cuBLAS 相当的性能；在该矩阵形状下，它 $1.45\times$ 较慢（$9591.6\,\mu$s，而 $6591.4\,\mu$s）。此矩阵形状并未针对 A40 进行专门调优，重新调整静态划分参数或许可以缩小部分差距。不过，cuBLAS 也可能针对该矩阵形状利用了张量核心指令（例如 4×1 MMA 分块），在这种情况下，如果无法以类似的硬件级访问方式执行，可能就无法达到与其相当的性能。

C. AMD MI300X 上的结果

我们在 AMD MI300X 上评估 KernelForge.jl，并与 AMDGPU.jl 和 AcceleratedKernels.jl 进行比较（参见表 I–VIII ）。与英伟达环境不同，CUDA.@profile 可以隔离各个内核片段，而 ROCm 的计时依赖 HIP 事件（AMDGPU.HIP.HIPEvent），测量设备端的耗时，其中包括内核启动开销以及任何设备端内存分配。我们不采用 CPU 端流水线计时，因为它在 ROCm 上的方差要大得多——在最坏情况下，如果在计时区域内分配临时缓冲区或目标缓冲区，标准差甚至可能超过均值。

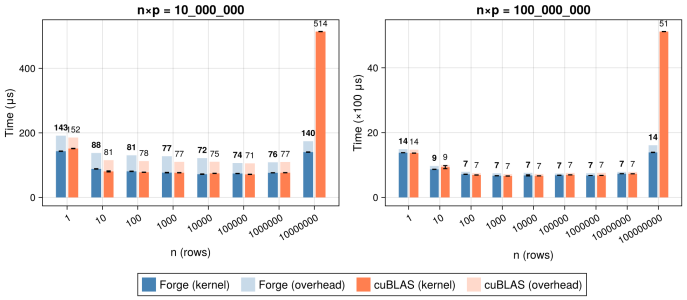

图 6：**不同矩阵形状下的矩阵-向量乘积基准测试（A40）**。通过 CUDA.jl 使用 cuBLAS 与 KernelForge 执行 Float32 矩阵向量乘法的吞吐量对比。实验设置与图 5 相同，总输入规模固定为 $n \times p$ 10^7 （左）和 10^8 （右）。

为缓解这一问题，我们使用预分配的临时缓冲区对 KernelForge.jl 和 AcceleratedKernels.jl 进行基准测试——这些缓冲区分别通过 KernelForge 的 get_allocation 接口或 AcceleratedKernels.jl 的 temp 关键字参数提供；这两种方式都能以统一的方式用于所有原语。AMDGPU.jl 不提供等效功能，因此基准测试时采用内部内存分配；其报告的耗时应理解为该库在具备预分配接口时所能达到性能的上限，而不能视作其内核吞吐量的单独度量。因此，与 AMDGPU.jl 的比较主要体现了 KernelForge 的内存分配转发设计相较于缺少此功能的库所具有的实际优势，而非算法本身的优越性。与 CUDA.jl 调度至 cuBLAS 的情况类似，AMDGPU.jl 会在内部将 LinearAlgebra.mul! 调度至 rocblas_gemv，因此它实际上是矩阵-向量和向量-矩阵原语的 rocBLAS 基准。

a) 映射归约： KernelForge.jl 在所有测试规模下，对 Float32 和 UInt8 的表现都优于两个基线。在 n =10^9 的 Float32 测试中，其速度比 AMDGPU.jl 快 $6.3\times$，比 AcceleratedKernels.jl 快 $1.4\times$。对于 UInt8，其优势

更为明显：在 $n = 10^9$ 时，KernelForge.jl 的速度比 AMDGPU.jl 快 $17.7\times$，比 AcceleratedKernels.jl 快 $3.6\times$；这与在 A40 上观察到的向量化内存访问优势一致。对于 UnitFloat8，在 $n = 10^9$ 时，KernelForge.jl 的速度比 AMDGPU.jl 快 $11.3\times$，但并未始终优于 AcceleratedKernels.jl。在 A40 上，尽管内部会提升为 Float32，UnitFloat8 和 UInt8 的吞吐量仍相近；但 MI300X 上并非如此，这表明 ROCm 编译器流水线对类型提升的处理方式有所不同。针对 AMD 硬件上混合精度类型的进一步调优留待未来工作。

表 I：AMD MI300X 上的映射归约基准测试：平均内核时间（μ秒） $\pm$ 标准差，通过 HIP 事件测量。AK：AcceleratedKernels.jl。UF8：UnitFloat8，U8：UInt8。AMDGPU 的时间包含临时分配；KernelForge 和 AK 的时间不包含临时分配。

n	Type	*AMDGPU*	*AK*	*KernelForge*
10^6	F32	220.5 ± 9.8	52.2 ± 5.0	31.8 ± 2.0
	UF8→F32	143.9 ± 3.8	58.4 ± 5.8	72.6 ± 12.4
	U8	110.5 ± 3.5	50.4 ± 5.2	32.7 ± 1.8
10^7	F32	125.5 ± 3.5	52.6 ± 4.5	35.7 ± 2.0
	UF8→F32	140.2 ± 3.9	92.6 ± 4.1	82.3 ± 13.1
	U8	213.4 ± 8.3	51.2 ± 4.1	33.9 ± 1.9
10^8	F32	669.1 ± 5.2	158.4 ± 3.7	122.2 ± 2.2
	UF8→F32	646.0 ± 4.3	217.8 ± 11.0	332.8 ± 6.4
	U8	637.2 ± 3.1	133.9 ± 3.0	54.1 ± 1.6
10^9	F32	6029 ± 20	1360 ± 11	961 ± 5
	UF8→F32	5675 ± 16	1599 ± 16	501 ± 5
	U8	5670 ± 10	1144 ± 3	319 ± 3

b) 扫描：在 $n \geq 10^7$，KernelForge.jl 的表现与两个基准实现相当或更优，而且规模越大，优势越明显。在 $n = 10^8$，对于 Float32，KernelForge.jl 比 AMDGPU.jl 快 $2.7\times$，比 AcceleratedKernels.jl 快 $3.2\times$；对于 Float64，则分别快 $1.9\times$ 和 $2.3\times$。在 $n = 10^9$，对于两种类型，KernelForge.jl 相比两个基准实现的优势都扩大到约 $20\times$——这与 A40 的结果一致：在大规模情况下，AcceleratedKernels.jl 的线程块间顺序累积成为主要瓶颈。AMDGPU.jl 和 AcceleratedKernels.jl 的耗时几乎相同，这在预期之中，因为 AMDGPU.jl 在内部将扫描实现委托给 AcceleratedKernels.jl。在 $n = 10^6$，三种实现的测量结果都处于彼此的测量噪声范围内。

表 II：AMD MI300X 上的扫描基准测试：平均内核时间（μs） $\pm$ 标准差，通过 HIP 事件测量。AK：AcceleratedKernels.jl。F32：Float32。F64：Float64。AMDGPU 的耗时包含临时分配；KernelForge 和 AK 的耗时不包含临时分配。

n	Type	*AMDGPU*	*AK*	*KernelForge*
10^6	F32	110.6 ± 3.2	41.3 ± 2.1	38.8 ± 2.1
	F64	56.0 ± 11.8	47.5 ± 1.9	52.5 ± 9.1
10^7	F32	158.0 ± 3.2	141.8 ± 2.4	98.6 ± 2.3
	F64	208.5 ± 6.7	258.8 ± 7.5	146.9 ± 13.8
10^8	F32	2103 ± 10	2454 ± 26	779 ± 28
	F64	2568 ± 26	3103 ± 18	1329 ± 37
10^9	F32	156267 ± 57	159883 ± 55	7467 ± 12
	F64	270080 ± 518	275408 ± 431	13056 ± 20

c) 向量-矩阵与矩阵-向量运算：对于宽矩阵（小 n、大 p），KernelForge.jl 相较于 AMDGPU.jl 实现了显著加速。对于 $n \times p = 10^8$、$n = 10^3$、$p = 10^5$ 条件下的矩阵-向量乘法（MatVec），KernelForge.jl 的速度提升了 $3.6\times$（参见表 VII）；类似地，在小n 和大 p 时，向量-矩阵乘法（VecMat）也实现了显著加速（参见表 VIII）。然而，随着 n 增大、p 减小，AMDGPU.jl 的性能会变得有竞争力，甚至更快；对于 $n \times p = 10^8$ 下的正方形或高而窄的矩阵形状，KernelForge.jl 则没有持续稳定的优势。

这些结果证实，KernelForge.jl 的算法设计能够正确移植到 AMD 硬件上，并且无需任何 AMD 专属调优，即可在映射归约和扫描方面实现有竞争力或更优的性能。矩阵运算结果揭示了将针对 A40 调优的参数迁移到不同架构时的局限性，也说明了未来开展针对各架构的调优，并针对特定长宽比重写内核的必要性。

VIII. 结论

我们介绍了 KernelForge.jl，这是一个 Julia 库，展示了通常被认为彼此互斥的三项特性——性能、灵活性和可移植性——能够在基础 GPU 并行原语中同时实现。KernelForge.jl 采用双层架构，将可移植的算法逻辑与后端特定内在函数分离；在英伟达和 AMD 硬件上，其扫描、映射归约和矩阵-向量运算的性能可媲美供应商优化实现，同时支持任意元素类型和运算符，超出供应商库所提供的范围。

我们将这项工作视为一项概念验证；目前仍存在若干局限。移植到其他后端（Apple Metal、Intel oneAPI）需要为每个目标扩展 KernelIntrinsics.jl，不过相关工作仅限于精简的内在函数层。更关键的是，静态调优参数是针对 A40 优化的，需要针对不同架构重新调优；自动网格搜索则留待未来工作。最后，要在矩阵-矩阵乘积上达到供应商级性能将困难得多，因为供应商库利用了张量核心指令，而这需要超越线程束级原语的特化。

附录

表 III 报告了第 VII 节中各图对应的原始内核计时。

表 III：A40 上的 MapReduce 内核基准测试：CUDA.jl、加速内核、KernelForge 和 CUB 的平均内核时间（μs） ± 标准差（参见图 3）。

n	Type	*CUDA*	*AK*	*KernelForge*	*CUB*
10^6	F32	10.1 ± 0.5	11.9 ± 0.3	6.1 ± 0.2	9.4 ± 0.4
	UF8→F32	7.3 ± 0.2	11.0 ± 0.2	4.9 ± 0.2	8.0 ± 0.3
10^7	F32	80.9 ± 0.5	77.0 ± 0.4	71.2 ± 0.4	75.6 ± 0.3
	UF8→F32	39.8 ± 0.4	74.4 ± 0.4	23.3 ± 0.4	25.4 ± 3.3
10^8	F32	724.9 ± 1.2	705.1 ± 0.3	679.9 ± 1.8	683.2 ± 0.7
	UF8→F32	323.7 ± 1.0	678.1 ± 0.3	178.4 ± 0.7	175.2 ± 0.3
10^9	F32	7207 ± 9	6972 ± 2	6562 ± 2	6809 ± 2
	UF8→F32	3310 ± 5	6719 ± 1	1718 ± 3	1724 ± 3

表 IV：A40 上的扫描内核基准测试：平均内核时间（μ秒） ± 标准差（参见图 4），针对 CUDA.jl、加速内核、KernelForge 和 CUB。

n	Type	*CUDA*	*AK*	*KernelForge*	*CUB*
10^6	F32	60.0 ± 0.4	21.2 ± 0.2	21.5 ± 0.4	20.7 ± 1.3
	F64	134.2 ± 0.6	64.7 ± 0.5	34.4 ± 1.0	38.9 ± 0.7
10^7	F32	509.9 ± 1.1	278.4 ± 1.9	149.4 ± 1.6	149.5 ± 2.0
	F64	1120.6 ± 1.0	749.5 ± 0.9	290.6 ± 2.6	293.6 ± 1.3
10^8	F32	4948 ± 2.2	5655 ± 0.6	1460 ± 3.5	1435 ± 3.9
	F64	11002 ± 4.2	42285 ± 5.6	2841 ± 7.5	2837 ± 3.9
10^9	F32	49322 ± 5	423868 ± 1482	14553 ± 10	14287 ± 7
	F64	109724 ± 11	3944795 ± 41	28327 ± 22	28291 ± 12

表 V：A40 上的向量-矩阵乘法内核基准测试：平均内核时间（μs） ± 标准差（参见图 5）。各行按总输入规模 $n \times p$ 分组。

n	p	*KernelForge*	*cuBLAS*
1	10^6	14.2 ± 0.3	53.5 ± 0.2
10	10^5	8.1 ± 0.2	10.1 ± 0.3
100	10^4	7.3 ± 0.1	6.6 ± 0.1
10^3	10^3	6.7 ± 0.1	11.2 ± 0.2
10^4	100	9.6 ± 0.2	12.3 ± 0.2
10^5	10	10.8 ± 0.4	7.7 ± 0.3
10^6	1	21.0 ± 0.3	18.2 ± 0.3
1	10^7	140.0 ± 0.7	514.7 ± 0.4
10	10^6	74.7 ± 0.3	80.7 ± 0.3
100	10^5	72.7 ± 0.3	87.1 ± 0.9
10^3	10^4	69.2 ± 0.3	76.1 ± 0.4
10^4	10^3	72.5 ± 0.4	71.0 ± 0.4
10^5	100	77.8 ± 0.6	73.8 ± 0.4
10^6	10	83.8 ± 0.4	75.7 ± 0.8
10^7	1	140.9 ± 0.5	152.0 ± 0.6
1	10^8	1386.5 ± 4.3	5127.4 ± 3.0
10	10^7	713.9 ± 0.4	710.5 ± 0.3
100	10^6	693.6 ± 0.4	779.0 ± 2.0
10^3	10^5	659.5 ± 0.4	734.4 ± 1.3
10^4	10^4	673.2 ± 0.4	683.4 ± 1.0
10^5	10^3	724.5 ± 3.1	683.3 ± 0.8
10^6	100	693.3 ± 1.7	692.1 ± 6.7
10^7	10	746.4 ± 1.5	755.9 ± 3.6
10^8	1	1324.8 ± 1.7	1368.1 ± 1.6
10	10^8	7104.6 ± 1.0	7083.6 ± 2.9
100	10^7	6903.7 ± 1.3	6889.4 ± 2.2
10^3	10^6	6567.2 ± 1.8	7301.5 ± 7.4
10^4	10^5	6670.5 ± 0.9	6812.7 ± 4.4
10^5	10^4	7228.4 ± 8.4	6592.3 ± 1.6
10^6	10^3	6954.2 ± 35.5	6770.5 ± 1.9
10^7	100	6927.2 ± 47.5	6830.4 ± 4.2
10^8	10	7259.4 ± 20.2	7716.6 ± 61.8

表 VI：A40 上的矩阵向量乘法内核基准测试：平均内核时间（μ秒） $\pm$ 标准差（参见图 6）。各行按总输入规模 $n \times p$ 分组。

n	p	***KernelForge***	***cuBLAS***
1	10^6	20.3 ± 0.5	18.5 ± 0.4
10	10^5	9.0 ± 0.3	13.0 ± 0.2
100	10^4	7.5 ± 0.2	7.2 ± 0.2
10^3	10^3	7.5 ± 0.3	10.5 ± 0.2
10^4	100	7.9 ± 0.2	11.4 ± 0.2
10^5	10	5.4 ± 0.2	4.4 ± 0.2
10^6	1	13.6 ± 0.3	53.4 ± 0.2
1	10^7	143.2 ± 0.5	151.5 ± 0.6
10	10^6	88.4 ± 0.7	80.6 ± 1.6
100	10^5	80.9 ± 0.6	78.0 ± 0.3
10^3	10^4	76.9 ± 0.8	76.6 ± 0.5
10^4	10^3	72.1 ± 0.7	74.5 ± 0.3
10^5	100	74.1 ± 0.4	71.4 ± 0.4
10^6	10	76.4 ± 0.3	76.6 ± 0.3
10^7	1	140.5 ± 0.6	514.0 ± 0.4
1	10^8	1380.4 ± 1.1	1366.8 ± 1.7
10	10^7	866.5 ± 2.6	934.8 ± 47.6
100	10^6	713.7 ± 2.4	692.1 ± 1.1
10^3	10^5	669.1 ± 1.6	661.1 ± 1.4
10^4	10^4	679.6 ± 17.7	665.2 ± 0.9
10^5	10^3	682.9 ± 0.8	696.0 ± 1.2
10^6	100	676.2 ± 0.5	678.4 ± 0.6
10^7	10	732.4 ± 0.4	729.3 ± 0.3
10^8	1	1390.0 ± 4.2	5117.6 ± 2.1
10	10^8	8785.5 ± 12.9	15063.6 ± 186.2
100	10^7	7061.4 ± 10.5	6814.6 ± 4.5
10^3	10^6	6584.1 ± 8.5	6686.6 ± 10.1
10^4	10^5	9591.6 ± 20.2	6591.4 ± 24.0
10^5	10^4	6902.6 ± 4.5	6794.1 ± 11.6
10^6	10^3	6620.4 ± 1.9	6843.1 ± 2.9
10^7	100	6707.5 ± 3.1	6735.0 ± 4.1
10^8	10	7278.7 ± 0.9	7259.3 ± 0.7

表 VII：AMD MI300X 上的矩阵-向量乘法基准测试：平均内核时间（μs） $\pm$ 标准差，通过 HIP 事件测量。各行按总输入规模 $n \times p$ 分组。AMDGPU 的时间包含临时分配；KernelForge 的时间不包含临时分配。

n	p	***KernelForge***	***AMDGPU***
1	10^7	263.7 ± 2.9	53558 ± 430
10	10^6	388.6 ± 3.1	5681.5 ± 39.6
100	10^5	262.1 ± 2.9	835.6 ± 7.5
10^3	10^4	133.8 ± 3.9	148.0 ± 1.9
10^4	10^3	189.3 ± 3.4	174.5 ± 116.8
10^5	100	30.9 ± 1.2	115.9 ± 8.7
10^6	10	33.7 ± 1.4	29.3 ± 14.5
10^7	1	43.7 ± 1.9	1021.3 ± 6283.3
1	10^8	2797.6 ± 10.3	737748 ± 4006
10	10^7	4440.5 ± 9.8	75250 ± 386
100	10^6	1961.0 ± 4.8	10572 ± 44
10^3	10^5	307.3 ± 43.2	1096.5 ± 3.5
10^4	10^4	251.6 ± 7.1	141.1 ± 2.1
10^5	10^3	155.1 ± 2.0	163.0 ± 8.6
10^6	100	142.3 ± 2.4	170.4 ± 17.0
10^7	10	201.6 ± 3.7	159.8 ± 36.7
10^8	1	285.7 ± 4.4	13218 ± 23618
10	10^8	43906 ± 171	752899 ± 4023
100	10^7	19112 ± 39	105736 ± 403
10^3	10^6	2140.2 ± 4.9	10869 ± 23
10^4	10^5	1410.7 ± 21.4	1294.3 ± 8.6
10^5	10^4	1412.5 ± 17.1	1086.2 ± 30.9
10^6	10^3	1219.1 ± 13.7	1011.9 ± 42.7
10^7	100	1299.1 ± 7.1	1162.9 ± 17.0
10^8	10	1872.9 ± 22.0	27329 ± 233531

表 VIII：AMD MI300X 上的向量-矩阵乘法基准测试：平均内核时间（μs） $\pm$ 标准差，通过 HIP 事件测量。按总输入规模 $n \times p$ 分组的行。AMDGPU 的时间包含临时分配；KernelForge 的时间不包含临时分配。

n	p	***KernelForge***	***AMDGPU***
1	10^7	36.9 ± 1.1	448.4 ± 17.3
10	10^6	29.8 ± 1.3	133.2 ± 16.8
100	10^5	27.8 ± 1.2	108.9 ± 8.3
10^3	10^4	30.5 ± 5.0	82.8 ± 2.3
10^4	10^3	28.5 ± 1.2	103.0 ± 3.1
10^5	100	48.9 ± 2.0	137.4 ± 1.6
10^6	10	52.6 ± 8.9	98.8 ± 3.9
10^7	1	56.9 ± 10.2	104.8 ± 2.3
1	10^8	283.6 ± 3.5	4740.0 ± 156.5
10	10^7	145.6 ± 2.7	686.1 ± 94.7
100	10^6	162.5 ± 8.0	169.4 ± 13.7
10^3	10^5	137.2 ± 12.5	175.0 ± 6.8
10^4	10^4	131.9 ± 12.3	164.4 ± 12.6
10^5	10^3	130.0 ± 1.9	185.8 ± 1.7
10^6	100	187.3 ± 3.2	148.2 ± 3.4
10^7	10	156.0 ± 2.7	158.6 ± 2.5
10^8	1	212.4 ± 2.8	232.1 ± 3.4
10	10^8	1410.6 ± 12.8	6877.0 ± 82.9
100	10^7	1452.2 ± 29.6	1247.6 ± 30.2
10^3	10^6	1180.1 ± 28.3	1495.2 ± 14.5
10^4	10^5	1179.4 ± 42.0	1079.9 ± 8.5
10^5	10^4	1040.6 ± 44.0	1112.3 ± 4.5
10^6	10^3	1176.1 ± 7.9	1640.3 ± 4.3
10^7	100	1212.6 ± 29.1	1159.2 ± 53.8
10^8	10	1411.9 ± 13.1	1242.1 ± 66.4

参考文献

[1] A. Sedova、J. D. Eblen、R. Budiardja、A. Tharrington 和 J. C. Smith，《面向生物与材料科学的高性能分子动力学模拟：性能可移植性的挑战》，载于 *2018 年 IEEE/ACM 高性能计算性能、可移植性与生产力国际研讨会 (P3HPC)*。IEEE，2018 年，第 1–13 页。[2] S. J. Pennycook、J. D. Sewall 和 V. W. Lee，《性能可移植性度量指标的影响》，*Future Generation Computer Systems*，第 92 卷，第 947–958 页，2019 年。[3] M. Martineau、S. McIntosh-Smith 和 W. Gaudin，《使用 tealeaf 评估现代并行编程模型的性能可移植性》，*Concurrency and Computation: Practice and Experience*，第 29 卷，第 15 期，e4117 页，2017 年。[4] V. Artigues、K. Kormann、M. Rampp 和 K. Reuter，《评估用于实现粒子网格代码的性能可移植性框架》，*Concurrency and Computation: Practice and Experience*，第 32 卷，第 11 期，e5640 页，2020 年。[5] T. Besard、C. Foket 和 B. De Sutter，《高效的可扩展编程：释放 Julia 在 GPU 上的潜力》，*IEEE Transactions on Parallel and Distributed Systems*，第 30 卷，第 4 期，第 827–841 页，2019 年。[6] V. Churavy，《KernelAbstractions.jl》。 [在线]。可访问：https://github.com/JuliaGPU/KernelAbstractions.jl [7] W. F. Godoy、P. Valero-Lara、T. E. Dettling、C. Trefftz、I. Jorquera、T. Sheehy、R. G. Miller、M. Gonzalez-Tallada、J. S. Vetter 和 V. Churavy，《评估高级编程模型的性能与可移植性：百亿亿次级节点上的 Julia、Python/Numba 和 Kokkos》，载于 2023 年 IEEE 并行与分布式处理国际研讨会系列会议 (IPDPSW)。IEEE，2023 年，第 373–382 页。[8] “KernelForge.jl”，可访问 https://github.com/epilliat/KernelForge.jl。[9] “KernelIntrinsics.jl”，可访问 https://github.com/epilliat/ KernelIntrinsics.jl。[10] A.-L. Nicusan、D. Werner、S. Branford、S. Hartley、A. J. Morris 和 K. Windows-Yule，《Acceleratedkernels.jl：基于统一转译代码库的跨架构并行算法》，arXiv 预印本 arXiv:2507.16710，2025 年。[11] J. Nickolls 和 W. J. Dally，《GPU 计算时代》，IEEE Micro，第 30 卷，第 2 期，第 56–69 页，2010 年。[12] S. G. De Gonzalo、S. Huang、J. G´omez-Luna、S. Hammond、O. Mutlu 和 W.-m. Hwu，《自动生成线程束级原语和原子指令，以实现快速且可移植的 GPU 并行归约》，载于 2019 年 IEEE/ACM 代码生成与优化国际研讨会 (CGO)。IEEE，2019 年，第 73–84 页。[13] B. A. Hechtman 和 D. J. Sorin，《探索面向海量线程吞吐量处理器的内存一致性》，载于第 40 届计算机体系结构国际年会论文集，2013 年，第 201–212 页。[14] S. Xiao 和 W.-c. Feng，《通过快速屏障同步实现块间 GPU 通信》，载于 2010 年 IEEE 并行与分布式处理国际研讨会 (IPDPS)。IEEE，2010 年，第 1–12 页。[15] K. Wang、D. Fussell 和 C. Lin，《GPU 上快速细粒度全局同步》，载于第二十四届编程语言与操作系统架构支持国际会议论文集，2019 年，第 793–806 页。[16] X. Mei 和 X. Chu，《通过微基准测试剖析 GPU 内存层次结构》，IEEE Transactions on Parallel and Distributed Systems，第 28 卷，第 1 期，第 72–86 页，2016 年。[17] M. Rhu、M. Sullivan、J. Leng 和 M. Erez，《面向高能效 GPU 架构的局部性感知内存层次结构》，载于第 46 届 IEEE/ACM 微体系结构国际年会论文集，2013 年，第 86–98 页。[18] D. Merrill 和 M. Garland，《采用解耦回看的单遍并行前缀扫描》，英伟达，技术报告 NVR-2016-002，2016 年。[19] L. S. Blackford 等，《更新版基本线性代数子程序集 (BLAS)》，ACM Transactions on Mathematical Software，第 28 卷，第 2 期，第 135–151 页，2002 年。[20] M. Harris、S. Sengupta 和 J. D. Owens，《使用 CUDA 实现并行前缀和（扫描）》，载于 GPU Gems 3，H. Nguyen 编。Addison Wesley，2007 年，第 851–876 页。[21] G. E. Blelloch，《前缀和及其应用》，卡内基梅隆大学计算机科学学院，宾夕法尼亚州匹兹堡，技术报告 CMU-CS-90-190，1990 年。[22] ——，《作为基本并行操作的扫描》，IEEE Transactions on Computers，第 38 卷，第 11 期，第 1526–1538 页，1989 年。[23] R. D. Hornung 和 J. A. Keasler，《RAJA 可移植性层：概述与现状》，劳伦斯利弗莫尔国家实验室，加利福尼亚州利弗莫尔，技术报告 LLNL-CONF-653873，2014 年。[24] H. C. Edwards、C. R. Trott 和 D. Sunderland，《Kokkos：通过多态内存访问模式实现众核性能可移植性》，Journal of Parallel and Distributed Computing，第 74 卷，第 12 期，第 3202–3216 页，2014 年。[25] J. H. Davis 、 P. Sivaraman 、 J. Kitson 、 K. Parasyris 、 H. Menon、I. Minn、G. Georgakoudis 和 A. Bhatele，《检验 GPU 编程模型的性能可移植性》，载于第 39 届 ACM 超级计算国际会议论文集，2025 年，第 776–791 页。[26] D. Merrill，《CUB：CUDA UnBound 库》，NVIDIA Research，2015 年，现为 NVIDIA CCCL（CUDA 核心计算库）的一部分。[在线]。可访问：https://nvlabs.github.io/cub/ [27] S. Williams、A. Waterman 和 D. Patterson，《Roofline：多核架构的直观性能模型》，Communications of the ACM，第 52 卷，第 4 期，第 65–76 页，2009 年。[28] C. Yang、T. Kurth 和 S. Williams，《面向 GPU 的分层 Roofline 分析：加速 NERSC-9 Perlmutter 系统的性能优化》，Concurrency and Computation: Practice and Experience，第 32 卷，第 20 期，e5547 页，2020 年。[29] NVIDIA Corporation，PTX 指令集架构 8.8 版，2024 年。 [在线]。可访问：https://docs.nvidia.com/cuda/parallel-thread-execution/ [30] C. R. Trott、D. Lebrun-Grandi´e、D. Arndt、J. Ciesko、V. Dang、N. Ellingwood、R. Gayatri、E. Harvey、D. S. Hollman、D. Ibanez 等，《Kokkos 3：面向百亿亿次级时代的编程模型扩展》，IEEE Transactions on Parallel and Distributed Systems，第 33 卷，第 4 期，第 805–817 页，2021 年。

[31] T. Besard、C. Foket 和 B. De Sutter，《有效的可扩展编程：释放 Julia 在 GPU 上的潜力》，*IEEE Transactions on Parallel and Distributed Systems*，第 30 卷第 4 期，第 827–841 页，2018 年。[32] 英伟达，《CUB 调优》，https://nvidia.github.io/cccl/cub/tuning.html，CUDA 核心计算库文档。