# 高层编程模型的性能与可移植性评估：Julia、Python/Numba 和 Kokkos 在百亿亿次级节点上的表现

William F. Godoy、Pedro Valero-Lara、T. Elise Dettling、Christian Trefftz、Ian Jorquera、Thomas Sheehy、Ross G. Miller、Marc Gonzalez-Tallada、Jeffrey S. Vetter 橡树岭国家实验室{godoywf}, { valerolarap}, {dettlingte}, {trefftzci}, {jorqueraid}, {sheehytb}, {rgmiller}, {gonzaleztalm}, {vetter} @ornl.gov Valentin Churavy 马萨诸塞州理工学院 vchuravy@mit.edu

arXiv:2303.06195v1 [cs.DC] 10 Mar 2023

摘要—**我们探讨高层编程模型的性能与可移植性：基于 LLVM 的 Julia 和 Python/Numba，以及 Kokkos 在高性能计算（HPC）节点上的表现。这些节点包括 Frontier 测试平台 Crusher 系统中的 AMD Epyc CPU 和 MI250X 图形处理器（GPU），以及橡树岭领导力计算设施 Wombat 系统中的 Ampere 基于 Arm 的 CPU 和 NVIDIA A100 GPU。我们将 CPU 上手工编写的稠密矩阵乘法算法的默认性能与供应商编译的 C/OpenMP 实现进行比较，并将各 GPU 上的性能分别与 CUDA 和 HIP 进行比较。我们并不着重于内核优化本身，而是选择这种朴素方法，以模拟科学研究中的探索性工作，并将其作为性能下限，从而隔离各编程模型的影响。Julia 和 Kokkos 在 CPU 上的表现与 C/OpenMP 相当，而 Julia 实现在 GPU 上的表现可与 CUDA 和 HIP 媲美。我们发现，Julia 的单精度实现和 Kokkos 在 NVIDIA A100 GPU 上存在性能差距，Python/Numba 在所有场景中也存在性能差距。我们还讨论了半精度支持、生产力、性能可移植性指标以及平台就绪度。随着第一代百亿亿次级系统投入使用，我们希望本研究能增进对高层、高生产力 HPC 语言的理解，并为其发展方向提供参考。**

索引词—**Julia、Python/Numba、Kokkos、OpenMP、LLVM、性能、可移植性、高性能计算、百亿亿次级计算、图形处理器**

## I. 引言

过去十年中，Python [1], Julia [2],和 R [3] 等高级动态语言一直处于人工智能/机器学习（AI/ML）、数据分析和交互式计算工作流的前沿。为实现性能和可扩展性、支撑底层低级计算的传统高性能计算（HPC）框架，均采用编译型语言编写：C语言、C++和 Fortran。与此同时，使用这些语言编写的并行编程模型旨在应对目标高性能计算硬件日益增强的异构性，而这类硬件主要由高线程数的 CPU 和图形处理器（ GPU ） [4] 构成。

主要编译器厂商纷纷采用 LLVM [5] ，这推动了统一工作，旨在让多种语言和编程模型都能实现性能可移植代码。Julia 和 Python/Numba [6] 复用 LLVM 的模块化特性，通过生成中间表示（LLVM 中间表示）来实现性能，同时保留其高级动态编程模型。类似地，基于指令的标准方法（例如 OpenMP [7], OpenACC [8]）为高性能计算编译型语言提供了更高级别的性能可移植模型；而元编程方法（例如 Kokkos [9], Raja [10], [11],Thrust [12]）则为面向 C语言++ 应用程序提供了强大的可移植接口。这些高级模型依赖高度优化的供应商后端（例如 OpenMP、CUDA [13], HIP [14]），其性能可移植性和开销权衡已成为活跃的研究领域 [11]。尽管如此，高级编程模型仍是端到端协同设计流程的一种有吸引力的替代方案，因此，随着人们日益重视科学工作流的性能、可移植性和生产力，它们对于弥合人工智能/机器学习、数据科学和高性能计算融合过程中的差距至关重要 [15]。

本研究比较了 Julia、Python/Numba 和 Kokkos 高级编程模型在驱动即将到来的百亿亿次级系统的 CPU 和图形处理器架构上的性能、可移植性和生产力。我们分析了托管于橡树岭领导力计算设施 (OLCF)[1]的两套系统的单节点可扩展性：Wombat 使用 Arm Ampere Neoverse 中央处理器和 2 块 NVIDIA A100 GPU；Crusher 配备 AMD EPYC 7A53 CPU 和 8 块 MI250X GPU，并作为 Frontier 的测试平台。Frontier 是 TOP500 榜单上的首个百亿亿次级系统。[2] 我们使用 Julia、Python/Numba 和 Kokkos 实现手写的稠密矩阵通用矩阵乘法 (GEMM) 代码，并将其性能与 C语言在多线程 CPU（OpenMP）和单 GPU（CUDA/HIP）系统上的性能进行比较。GEMM 是基本线性代数子程序 (BLAS) 中的重要内核， [16] 广泛应用于多个深度学习人工智能框架；现代图形处理器架构针对这些框架进行了大量优化，并采用了张量核心。 [17]–[20]。

[1]https://www.olcf.ornl.gov/[2] https://www.top500.org/

选择手写 GEMM 实现的原因是：i) 在简单内核中隔离各编程模型和环境；ii) 提供一个性能下限参照点，模拟快速原型开发中针对稠密矩阵的定制科学计算内核，这类内核包含大量向量乘加运算。本文展示了半精度（条件允许时）、单精度和双精度浮点运算实现的结果。我们采用统一的指标分析生成的代码实现，以评估这些高级方法的生产力和性能可移植性。

本文其余部分安排如下：第二节总结了评估这些高级编程模型性能与可移植性的相关工作。第三节介绍在 Crusher 和 Wombat 节点上开展的数值实验。第四节展示性能结果并进行后续讨论，第五节分析性能可移植性。第六节总结本研究。附录 A 描述了本研究所用的可复现制品。

## II. 相关工作

近期研究尝试探究可移植的高级编程模型与其对应的高度优化、特定于供应商的实现之间的性能差距。我们根据高级实现的性质对这类工作进行分类。

*a)* 动态语言：Julia 提供了面向科学计算和数据科学、以 LLVM 为目标的动态编译前端。其在高性能计算领域的应用仍处于积极探索和社区参与阶段 [21]。Ranocha 等人 [22] 对其双曲偏微分方程（PDE）求解器 Trixi.jl 进行了大规模评估。他们得出结论：尽管 Julia 在大规模运行时面临类似甚至更复杂的挑战，但其性能与传统高性能计算语言相当。与此同时，Tomasi 和 Giordano [23]探讨了 Julia 在天体物理学应用中的不足与优势。Lin 和 McIntosh-Smith [24] 使用内存受限和计算受限型迷你应用表明，在多种 CPU/图形处理器高性能计算硬件配置下，Julia 的性能与传统编译型语言相当或略逊一筹。Faingnaert 等人 [25] 提供了经过优化的 Julia 通用矩阵乘法内核，其性能可与 cuBLAS 和 CUTLASS 实现相媲美。Ko 等人介绍了 DistStat.jl [26]，，这是 Julia 中统一的统计计算环境，旨在实现性能可移植性，并已在大型云系统上经过测试。最近，Giordano 等人 [27] 发现，Julia 的消息传递接口（MPI） [28]MPI.jl [29] 在基于 Arm 的富岳富士通 A64FX 系统上具有竞争性的系统性能。Gmys 等人 [30] 得出结论，与 Chapel [31]相比，Julia 和 Python/Numba 在多线程并行化的可扩展性方面仍存在差距。

近期利用 Numba 发挥 Python 性能优势的工作并不多。Mattson 等人介绍了 PyOMP [32]，它是面向 Numba 的 OpenMP 实现，初步结果与绕过 Python 全局解释器锁（GIL）的 C语言实现不相上下。近期关于 Python/Numba GPU 实现的研究主要针对支持 NVIDIA CUDA 的硬件。例如，Oden [33] 指出，与 C CUDA 相比，Numba 的 CUDA 实现因 Python 的性能局限而存在性能差距；而 Di Domenico 等人 [34]在使用 Python 评估 NASA 高级超级计算部门的并行基准测试内核时，展示了令人鼓舞的性能。Python/Numba 最近已弃用 AMD GPU 支持，[3] 而 PyCUDA、PyOpenCL [35],和 CuPy [36] 则通过字符串接口传递 C 或 C++ 自定义内核代码进行编译，从而在运行时访问 NVIDIA 和 AMD GPU 硬件。

*b)* 元编程：元编程已成为代码可移植性和性能可移植性领域的重要研究方向。Kokkos [9]，、Raja [10]，[11]，和 Thrust [12]是该领域的重要成果。它们提供并行分发选项，无须指定目标系统的任何细节。尽管 Kokkos 已应用于许多应用程序，其性能可移植性仍是一个活跃的研究课题 [37]–[40]。Kokkos 依赖高度优化的后端（例如 OpenMP、OpenACC [41],CUDA/HIP），这些后端基于模板实例化。因此，这会妨碍部署内核特定优化（例如，为线程块数量和每个线程块的线程数选择合适的值，以及选择数据传输与计算的重叠程度）。模板会设定这类优化，而这些优化发生在实际代码生成阶段之前。Kokkos 的每个后端都是对通用前端的优化，也就是说，针对目标后端或设备（CPU 或图形处理器），会采用高度专业化的并行计算和内存管理技术。这些技术可能不同于本研究使用的参考实现，从而影响性能。

*c)* 基于指令的语言：基于指令的语言如今已在高性能计算领域无处不在。OpenMP 和 OpenACC 是共享内存和加速器编程领域事实上的标准。编译器和运行时技术解决了代码可移植性问题，事实证明，它们足以支持代码在众多高性能计算系统之间移植。然而，对于性能可移植性而言，情况并非如此。这两项标准，尤其是 OpenMP，通过增加专用构造来指导编译器了解系统层面提供的特性（例如，统一虚拟内存、面向目标设备的厂商专有功能），从而提高了自身的复杂性 [42]–[45]。

*d)* 运行时库：CUDA [13]， HIP [14],OpenCL [46],和 SyCL [47] 已成为高性能计算领域常见的编程框架。例如，Bertoni 等人 [48]研究了 OpenCL 在英特尔和 NVIDIA 系统上的性能可移植性。同样，Halver 等人 [49]评估了 OpenCL 在分子动力学应用程序中的可移植性。对于 SyCL，也开展了类似研究

[3]https://github.com/numba/numba/pull/6991

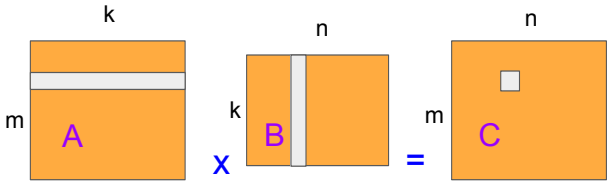



图 1：手工编写的简单 GEMM 内核矩阵乘法示意图。

以探索其在人工智能模型 [50]和稀疏线性代数内核 [51]中的可移植性。总体而言，这些工作都存在一些局限，因为程序员会使用供应商特定的运行时原语来实现高性能。这些原语无法在不同的高性能计算配置之间移植，因此必须通过条件编译对其进行标注，这增加了整个部署可移植代码过程的复杂性。对于性能可移植性，同样的问题也会因同样的原因而出现——目标特定的运行时原语。

## III. 实验

本节介绍针对 CPU 和 GPU 编程模型采用的不同并行化策略。图 1 展示了手工编写的 GEMM 内核示例，即两个稠密矩阵的乘积。多线程 CPU 代码实现采用粗粒度映射，为每个线程分配较大的子组件。为确保计算工作负载相当，这些子组件根据语言采用行主序（例如 Python 默认的 NumPy 数组）还是列主序（例如 Julia），分别对应整行或整列。向量化 GPU 代码实现采用细粒度映射，为每个线程分配较小的子组件。这些子组件是由 GPU 编程模型中的二维线程网格定义的单个元素。本节其余部分将重点介绍不同编程模型、编译和环境配置之间的差异。

### *A. CPU* 实现

C/OpenMP、Julia 和 Python/Numba 中多线程 CPU 并行化的粗粒度策略，与在基于串行*for* 循环的实现上使用元编程指令的方法类似。另一方面，Kokkos 要求完全从头编写匿名 lambda 函数实现。图 2 展示了 C/OpenMP 中典型的单层并行*for* 循环，它通过编译指示对串行代码版本进行最少修改即可实现并行。正如预期的那样，多维矩阵的索引线性化由程序员负责，并使用非安全内存访问；而线程私有变量必须显式标注，以便编译器找到更好的优化。线程数由 OMP NUM THREADS_ _环境变量控制。其他线程策略则通过绑定线程进行控制，具体使用 OMP_PROC_BIND=true 和 OMP_PLACES=threads 环境变量，如附录 A 所示。

(a) C/OpenMP

```
#include "omp.h"
...
#pragma omp parallel for default(shared) \
            private(i,      k, j, temp)
for (i  = 0; i  < A_行; i++)
  for (k  = 0; k  < A_列; k++)
    温度  = A[i * A   _列  + k];
    for (j  = 0; j  < B_列; j++)
      C[i * B   _cols  + j] += temp * B[k * B   _cols  + j];
```

(b) Kokkos

```
Kokkos::parallel_for( "AxB=C",          mdrange_策略( {0,       0}, {M,
↪   N}),  KOKKOS _LAMBDA ( int m  ,  int n ){
    float tmp  = 0.0;
    for ( int k  = 0; k  < K; k++ )
      tmp  += A(m,      k) * B(k,      n);
    C(m, n)  = 临时变量;
  }
);
```

(c) Julia

```
导入 Base.Threads: @threads

函数 gemm(A, B, C)
...
    @线程 for j in 1:B_cols
        for l in 1:A 列_
            @inbounds 温度  = B[l,    j]
            对于 i in 1:A 行_
                @inbounds C[i,    j] += temp * A[i,    l]
            end
        end
    end
end
```

(d) Python/Numba

```
from numba import njit, prange
import numpy as np

@njit(parallel=True, nogil=True, fastmath=True)
def gemm(A: np.ndarray, B: np.ndarray, C: np.ndarray):
...
    for i in prange(0, A_行数):
        for k in range(0, A_列数):
            临时  = A[i,    k]
            for j in range(0, B_列数):
                C[i, j] += temp * B[k, j]
```

图 2：本研究所用编程模型的 CPU 多线程、粗粒度简单通用矩阵乘法内核。

图 2b 展示了 Kokkos 的编程模型，它使用 C语言++ lambda 函数来指定结果矩阵中一个元素的计算。Kokkos 旨在实现架构无关，使程序员能够通过模板特化摆脱供应商或目标特定编程模型的底层细节。在实践中，目标架构通过编译时的 KOKKOS_DEVICES 标志来定义。例如，要为 NVIDIA 图形处理器生成二进制代码，必须使用 KOKKOS_DEVICES=Cuda。随后，针对目标架构编译 Kokkos 库和源代码。

等效的 Julia 实现如图2c所示。Julia 提供了更高级别的实现，使用内置的 Threads 模块。如图所示，只需添加 @threads 宏，即可立即并行化外层循环

无需进一步指定。这种方法的缺点是，在整个可执行文件运行期间，Julia 中的线程数不可更改，因为它通过 Julia 可执行文件的参数 -t 或 JULIA_NUM_THREADS 环境变量进行配置。由于其数值计算特性，Julia 支持原生多维数组和强类型，并可与底层多线程后端实现互操作。@inbounds 宏可避免数组访问时的额外边界检查。该设置可在可执行文件级别进行配置，但为便于说明，此处将其保留在代码中。所有运行均使用 JULIA_EXCLUSIVE 环境变量来控制线程策略，并严格按顺序将线程绑定到核心。

如图 2d 所示，Python/Numba 提供了类似但稍具侵入性的方法，使用元编程装饰器标记即时编译区域。必须将预期的并行 for 循环修改为使用 prange 关键字。尽管 Numba 支持 NumPy 数组，但在即时编译区域中不要求强类型。NUMBA_NUM_THREADS 环境变量可用于选择线程数，但目前没有设置线程绑定/固定策略的机制（不同于 C/OpenMP 和 Julia）。

图 2 所示的手写 GEMM 内核在 OLCF 提供的两种不同 CPU 系统上运行：Wombat（Arm + NVIDIA）和 Crusher（AMD）。表 I 列出了 CPU 规格、所需的 C/OpenMP 和 Kokkos 编译标志、最新版本的 Julia 和 Python/Numba，以及环境变量（即 ENV）。我们在相应的基于 LLVM 的厂商编译器（例如 ArmClang、AMDClang）上选择了针对目标的标志，以便针对不同矩阵大小所定义的一系列系统工作负载，充分利用所有可用核心，实现最高节点内性能。总体而言，对于多线程代码，Julia 和 Python/Numba 采用了与 OpenMP 类似的方法，而 Kokkos 则为 CPU 和图形处理器提供了可移植统一 API。

表 I：CPU 实验规格。

| 编程语言/系统<br>型号 | Wombat（Arm）<br>Ampere Altra<br>80 核，1 个非统一内存访问域 | Crusher（AMD）<br>AMD Epyc 7A53<br>64 核，4 个非统一内存访问域 |
|---|---|---|
| **C语言 OpenMP** | | |
| 编译器<br>标志 | ArmClang22<br>-O3 -fopenmp | AMDClang14<br>-O3 -fopenmp<br>-march=native |
| **C语言++ Kokkos** | v3.6.01 | |
| KOKKOS_设备<br>KOKKOS_架构 | OpenMP<br>Armv8-TX2 | Zen 3 |
| 编译器<br>标志 | ArmClang++22<br>-O3 -fopenmp | AMDClang++14<br>-O3 -fopenmp<br>-march=native |
| **Julia**<br>ENV | v1.7.2<br>Julia 独占=1 | v1.8.0-rc1 |
| **Python**<br>**Numba**<br>ENV | v3.9.9<br>v0.55.1<br>NUMBA_优化=3(默认) | |

*B.* 图形处理器实现

图形处理器实现采用细粒度方法，将计算任务映射到单个线程，以计算一个元素，从而得到矩阵。CUDA 和 HIP 的简单矩阵乘法内核如图 3a 所示。HIP 紧密沿用 CUDA 内核模型，不过网格的定义依据启动的线程总数，而非线程块。

(a) CUDA/HIP

```
int row =blockIdx.y * blockDim.y +threadIdx.y; int col =
blockIdx.x * blockDim.x +threadIdx.x; double sum = 0.0; if(
row < A_rows && col < B_cols ) { for(int i = 0; i < n; i
++) { sum += A[row * n + i] *B[i *k + col]; } C[row * k +
col] =sum; }
```

(b) Julia CUDA.jl

```
using CUDA ...

行 = (blockIdx().x - 1) * blockDim().x + 线程Idx().x
列 = (blockIdx().y - 1) * blockDim().y + 线程索引().y

sum = zero(eltype(C))
如果行 <= size(A, 1) 且列 <=size(B, 2)
    对于 i，范围为 1:size(A , 2)
        @inbounds 求和 += A[row , i] * B[i , col]
    end
    @inbounds C[row , col] = sum
end
返回 nothing
```

(c) Julia AMDGPU.jl

```
使用 AMDGPU
. . .
row =(workgroupIdx().x - 1) *
        工作组维度().x + 工作项索引().x
col =(工作组索引().y - 1) *
       工作组维度().y + 工作项索引().y

sum = zero(eltype(C))
如果行 <= size(A, 1) 且列 <=size(B, 2)
    for i in 1:size(A , 2)
        @inbounds sum += A[row, i] * B[i, col]
    end
    @inbounds C语言[行，列] = 和
end
```

(d) Python/Numba CUDA

```
from numba import cuda
from numba.cuda.cudadrv.devicearray import 设备多维数组
import numpy as np

@cuda.jit
定义 gemm(A: 设备多维数组, B: 设备多维数组, C: C语言.
↪  设备多维数组）:
. . .
    i, j = cuda.grid(2)
    如果 i < C.shape[0] 且 j < C.shape[1]:
        tmp = 0。
        对于 k in range(A.shape[1]):
            tmp += A[i, k]* B[k, j]
        C[i, j] = tmp
```

图 3：本研究所用编程模型的图形处理器细粒度手写 GEMM 内核。

Julia GPU 编程模型分别使用面向 NVIDIA 和 AMD 图形处理器的厂商专属实现 CUDA.jl [52], [53] 和 AMDGPU.jl [54]。它们提供高层

机制，用于在图形处理器设备上定义多维数组（CUArray 和 ROCArray）。Julia 还提供 KernelAbstractions.jl [55] 软件包，可编写可移植的内核，同时仍依赖 CUArray 或 ROCArray。图 3b 和图 3c 分别展示了 CUDA.jl 和 AMDGPU.jl 中对应的内核实现。如图所示，由于线程映射与 CUDA/HIP 模型十分相似，熟悉这些编程模型的用户可以轻松完成过渡。Julia 的优势在于使用多维数组，并在设备内核代码中增添了功能，从而为图形处理器编程提供强大的语法。

Python/Numba 提供了一个非常简单的接口来访问 CUDA 内核功能，如图 3d 所示。与 CUDA/HIP 模型不同，它提供了一个简单的 cuda.grid 映射函数，用于将行列坐标映射到图形处理器线程上。与 Julia 类似，Python 通过 devicearray 接口支持在设备内核中使用多维矩阵语法。如第二节所述，目前已弃用 Python/Numba 对 AMD 图形处理器的支持。

表 II：GPU 实验规格。

| 编程语言/系统<br>模型 | Wombat（NVIDIA）<br>A100 Ampere | Crusher（AMD）<br>MI250X |
|---|---|---|
| **C CUDA/HIP**<br>编译器<br>标志 | nvcc v11.5.1<br>-arch=sm 80 | hipcc v14.0.0<br>–amdgpu-target=gfx908 |
| **C语言++ Kokkos** | v3.6.01 | |
| KOKKOS_设备<br>KOKKOS_架构<br>编译器<br>标志 | Cuda<br>Ampere80<br>CUDA v11.5.1<br>-expt-extended-lambda<br>-Xcudafe<br>-arch=sm 80 | Hip<br>Vega908<br>HIP v14.0.0<br>–amdgpu-target=gfx908 |
| **Julia** | v1.7.2<br>CUDA.jl | 1.8.0-rc1<br>AMDGPU.jl |
| **Python**<br>**Numba**<br>标志 | v3.9.9<br>v0.55.1 | 不支持 |

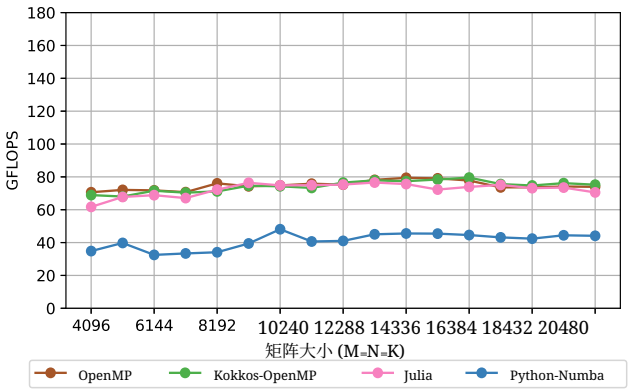
## IV. 结果

本节介绍第三节所述实验的结果。所有数据均通过多次（至少 5 次或 10 次）运行 GEMM 内核并排除初始预热步骤后获得。排除预热步骤也会去除 Julia 和 Python/Numba 中初始通信（线程和图形处理器）以及即时编译的开销。由于节点用途专一，这些结果最有可能反映实际性能；我们没有进行详尽的变异性分析，只呈现预期平均值。我们认为，变异性本身是系统的一项特性，而非编程模型本身造成的影响，因为这正是本次比较的目标。尽管如此，可复现的产物

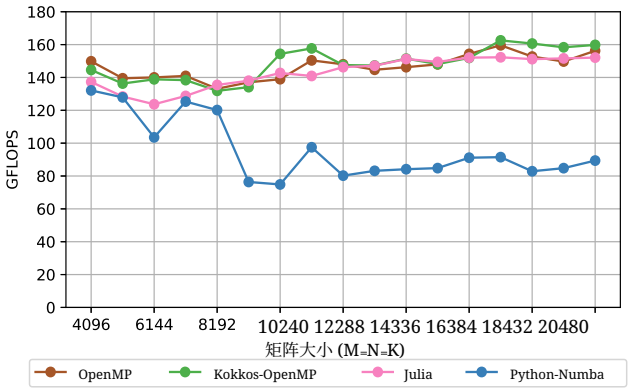
详见附录 A，供独立验证。

### A. CPU 性能

*a) Crusher AMD EPYC 7A53*：图 4 展示了该系统上多线程 CPU 实现的结果 Crusher 在（a）双精度和（b）单精度下的结果。总体而言，Kokkos/OpenMP 和 Julia 线程的性能与供应商提供的 ArmClang C/OpenMP 实现相当，而 Python/Numba 的性能仍然较差。OpenMP 和 Julia 使用环境标志将线程绑定到 CPU 资源，如表 I 所示；Python/Numba API 不支持此选项。




(a) 双精度 (FP64)



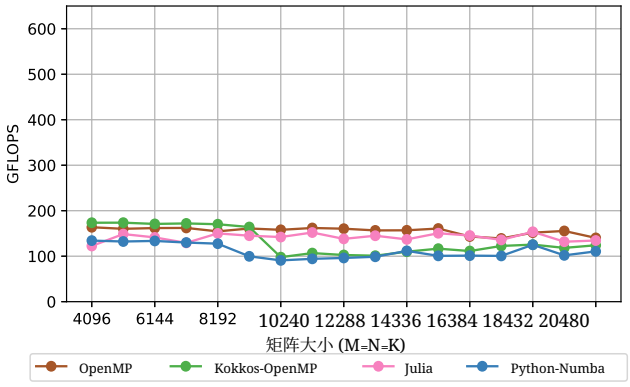

(b) 单精度 (FP32)

图 4：Crusher 使用 64 个线程跨 4 个非统一内存访问区域时的多线程中央处理器性能。

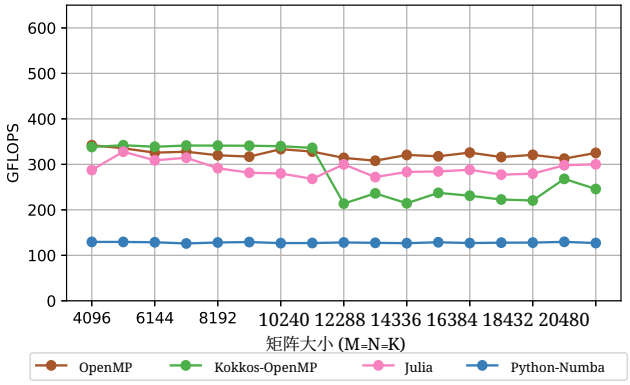
*b) Wombat Arm Altra Ampere*：Arm CPU 上的性能结果如图 5 所示，分别对应 (a) 双精度和 (b) 单精度。值得注意的是，使用 OpenMP 后端的 Kokkos 在这两种情况下都出现了性能下降。与此同时，Julia 的性能几乎与供应商 OpenMP 实现持平。

Python/Numba 区域不支持结合 NumPy Float16 随机数功能使用半精度浮点数（FP16），因此输入矩阵均填入了 1。Julia 的半精度支持仍在积极开发中。我们在 Crusher AMD CPU 上获得的性能非常低（本文未报告），预计随着 Julia 原生 FP16 支持日趋成熟，性能将会提升。[4] Arm 上的 Julia 线程实现运行顺畅，并达到了预期的性能水平，如图所示

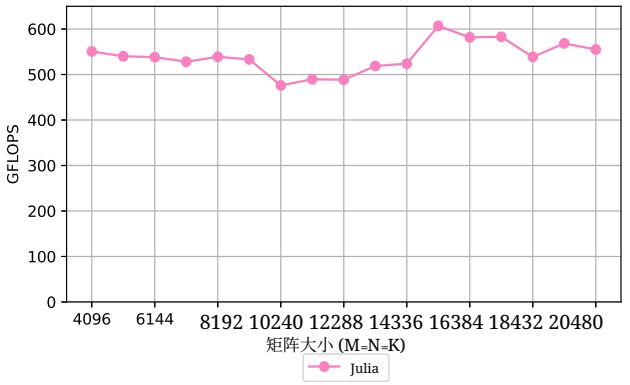
[4]https://github.com/JuliaLang/julia/issues/45542




(a) 双精度 (FP64)




(b) 单精度 (FP32)




(c) 半精度 (FP16)

图 5：使用 80 个线程时 Wombat 的多线程中央处理器性能。

图 5c。文献中也有一篇近期文章讨论了 Julia 在 Arm 系统上的 FP16 性能 [27]。

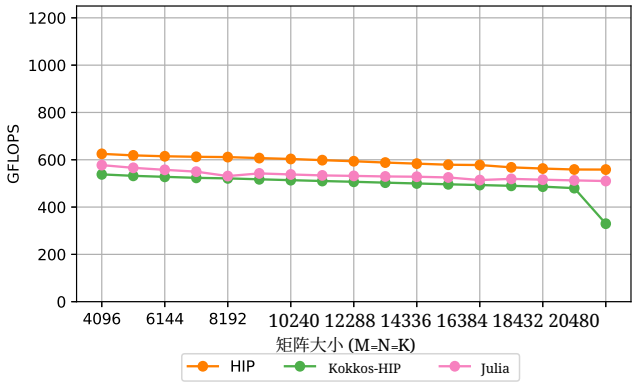
### *B. GPU*性能

*a) Crusher AMD MI250X*：图6展示了在 Crusher 的 AMD MI250X GPU 上，HIP、Kokkos-HIP 和使用 AMDGPU.jl 的 Julia 在不同浮点精度下执行简单通用矩阵乘法的性能。该 GPU 与 OLCF Frontier 系统中的图形处理器相似（表 II）。Python/Numba 不支持 AMD 图形处理器。如图6a 所示，在双精度运行中，供应商提供的 HIP 实现性能最高，其次是使用 AMDGPU.jl 的 Julia 和 Kokkos/HIP。两者的性能都很有竞争力，但在所有矩阵规模下仍无法媲美 HIP，因为引入的开销似乎是固定的。Kokkos 在最大规模下出现了可重复的性能下降，这可能需要在该系统上进一步调查。图6b展示了单精度浮点运算的性能。与预期一样，所有模型的性能都有所提升，但 Kokkos + HIP 的性能持续下降，这同样需要进一步调查。有趣的是，使用 AMDGPU.jl 的 Julia 性能略高于供应商提供的 HIP 实现，不过随着矩阵规模增大，差异变得很小，这可能只是该特定系统上的波动所致。最后，图6c展示了 Julia AMDGPU.jl 在半精度乘法中的性能结果，其中数据存储在单精度矩阵中（图1c）。与单精度运行相比，性能没有明显提升。尽管如此，其他编程模型并未提供无缝的半精度支持，而 Julia 目前支持在 AMD 图形处理器上进行随机数生成和内核计算。

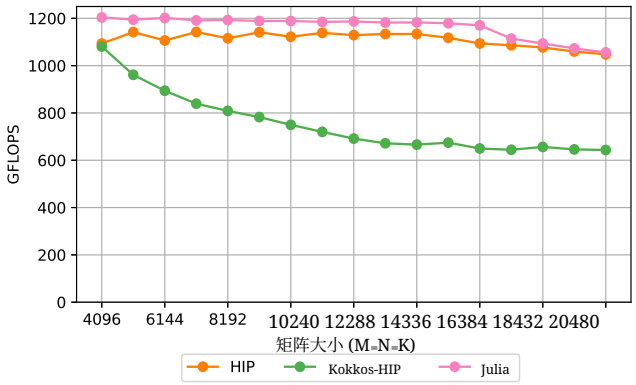
*b) Wombat NVIDIA A100*：图7展示了在 Wombat 的 NVIDIA A100 GPU（表 II）上，采用不同浮点精度时，(1) 使用 CUDA、Kokkos/CUDA 和 CUDA.jl 的 Julia，以及 (2) 使用 CUDA 的 Python/Numba 的简单通用矩阵乘法性能。图 7a 中所示的双精度运行结果表明，与供应商提供的 CUDA 实现相比，使用 CUDA.jl 的 Julia 存在恒定开销。此处未展示生成的低级并行线程执行（PTX）指令集架构（ISA），但其显示展开循环指令存在差异：CUDA.jl 为 2 条，原生 CUDA 为 4 条。要针对不同的内核工作负载生成更有效的启发式模型，还需进行更深入的研究。使用 CUDA后端的 Kokkos 和 Python/Numba 性能始终较差，这引发了对配置和/或实际图形处理器运行情况的疑问。我们使用 NVIDIA 的 nvprof 分析器对 Kokkos 和 Python/Numba 进行了验证，以确认图形处理器活动。图7b展示了单精度运行的性能。正如预期，供应商提供的 CUDA 实现性能显著提升，而其他实现仍存在差距。双精度运行切换为单精度运行后，Julia、Kokkos/CUDA 和 Python/Numba 的性能提升幅度均较小，约为 10%。最后，我们展示了受支持的 Julia（使用 CUDA.jl）和 Python/Numba 实现的半精度结果。由于 numpy 的 float16 类型不支持随机数生成，必须修改 Python 半精度实现。尽管如此，我们发现，与单精度实现相比，半精度并未带来性能提升。

## V. 性能可移植性

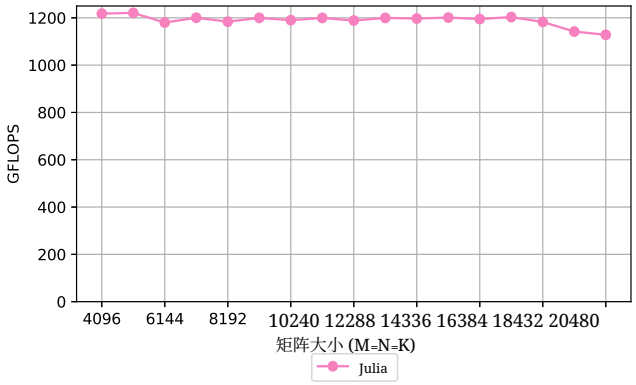
尽管目前尚无衡量性能可移植性的公认指标，但我们参考了当前文献中针对并行应用程序提出的一些方案。然而，




(a) 双精度（FP64）

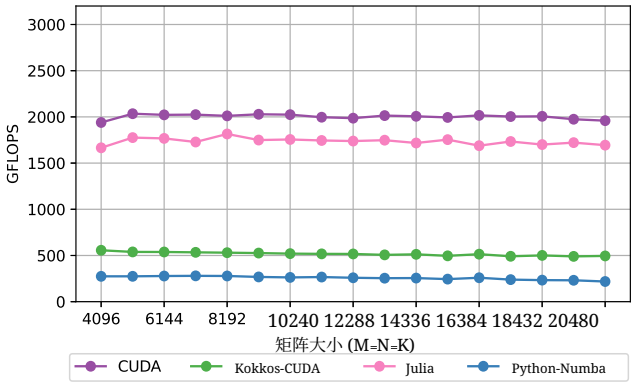



(b) 单精度（FP32）

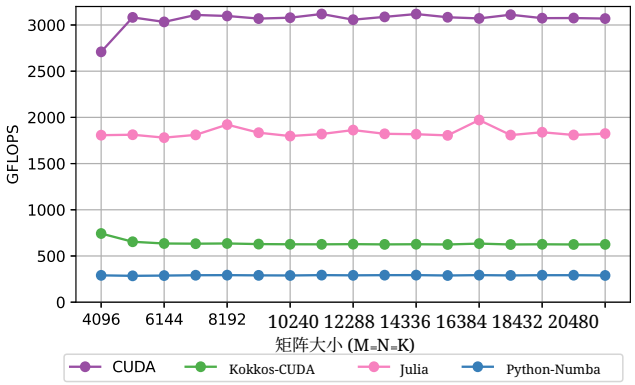



(c) 半精度（FP16）

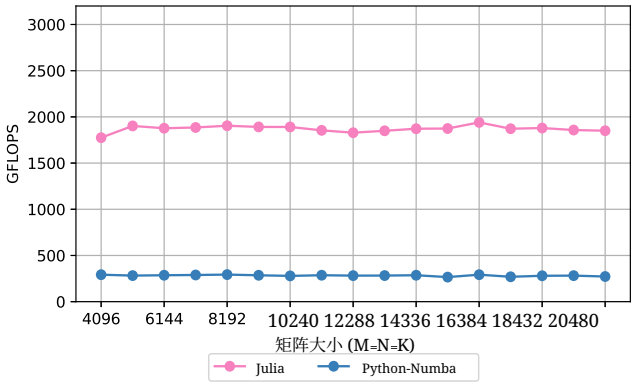
图 6：在 Crusher AMD MI250X 图形处理器上，使用 $32 \times 32$ 线程块大小时的简单通用矩阵乘法性能。




(a) 双精度 (FP64)




(b) 单精度 (FP32)




(c) 半精度 (FP16)

图 7：使用 $32 \times 32$ 线程块大小时，Wombat NVIDIA A100 上的简单通用矩阵乘法性能。

本研究重点评估不同编程模型的性能可移植性。

最早尝试实现这一目标的工作之一，发生在美国能源部关于性能可移植性的会议上 [56],，会上提出了不同的定义。Pennycook 等人 [57] 提出了一个独特的定义，此后该定义被高性能计算社区采用：“对……的测量

应用程序针对给定问题的性能效率，该问题能够在给定集合中的所有平台上正确执行。”此后，人们定义了一系列不同的指标和公式 [58]。最近，这一定义得到扩展，将编程模型的性能可移植性定义为可移植模型的平均性能与

不可移植模型的平均性能之比 [11]。

用于计算编程模型性能可移植性的度量指标 $M$ 定义如下：

$$\Phi_M = \frac{\sum_{i \in T} e_i(a)}{|\,T\,|}, \tag{1}$$

其中，在我们的案例中，$e_i(a)$ 表示可移植编程模型 $M$（即 Kokkos、Julia 或 Python-Numba）执行矩阵乘法的性能效率，除以供应商特定实现的性能。例如，Julia 在 MI250X AMD GPU 上的性能效率可表示如下：

$$e_{MI250x} = \frac{Julia\ Performance}{HIP\ Performance}. \tag{2}$$

表 III 展示了该简单内核在各个硬件目标上的计算效率，以及根据公式 (1) 得出的整体编程模型效率。在 CPU 分析中，我们使用供应商提供的 C/OpenMP 性能作为特定于架构的参考模型；在 NVIDIA 和 AMD 图形处理器的分析中，则分别使用 CUDA 和 HIP 性能作为特定于架构的参考模型。除 $e_{A100}$ 外，Kokkos 和 Julia 的性能效率相近。鉴于 AMD 图形处理器不受支持，根据性能结果，Python/Numba 的数值最低。就该简单内核的编程模型效率而言，Julia 得分最高，其次是 Kokkos 和 Python/Numba。比较双精度与单精度分析时，未发现显著差异；对于所有模型，单精度浮点计算的可移植性略低。

表 III：Kokkos、Julia 和 Python/Numba 在各项实验中的性能效率。

| 架构 | Kokkos | Julia | Python/Numba |
|---|---|---|---|
| **双精度** | | | |
| $e_{Epyc\ 7A53}$ | **0.994** | 0.912 | 0.550 |
| $e_{Ampere\ Altra}$ | 0.854 | **0.907** | 0.713 |
| $e_{MI250x}$ | 0.842 | **0.903** | - |
| $e_{A100}$ | 0.260 | **0.867** | 0.130 |
| $\Phi_M$ | 0.738 | **0.897** | 0.348 |
| **单精度** | | | |
| $e_{Epyc\ 7A53}$ | **1.014** | 0.976 | 0.655 |
| $e_{Ampere\ Altra}$ | 0.836 | **0.900** | 0.400 |
| $e_{MI250x}$ | 0.677 | **1.050** | - |
| $e_{A100}$ | 0.208 | **0.600** | 0.095 |
| $\Phi_M$ | 0.684 | **0.882** | 0.288 |

## 六、结论

我们研究了高生产力、动态语言 Julia 和 Python/Numba，将其作为 LLVM 的高级接口，并将它们与可移植的 C/OpenMP 和 Kokkos 实现进行了比较。本文展示了在百亿亿级节点架构上的性能结果和效率指标，测试对象是简单的手写 gense 矩阵乘法。所用架构包括 Wombat（采用 Arm Ampere CPU 和 2 个 NVIDIA A100 GPU）以及 Crusher（Frontier 的测试平台，配备 AMD EPYC 7A53 CPU 和 8 个 MI250X GPU）。双精度和单精度浮点运算的结果表明，默认 Julia 实现在这些平台上的性能相当。对于 CPU，Julia 的性能与采用基于 LLVM 的 ArmClang 和 AMDClang 供应商编译器的 C/OpenMP 相当。对于 AMD GPU，Julia 的 AMDGPU.jl 性能与 HIP 相当。Julia 在生产力和性能方面的优势，源于其从设计之初便充分利用 LLVM。尽管如此，在 NVIDIA A100 GPU 上的单精度浮点运算仍存在性能差距，还需进一步研究默认生成的底层 PTX ISA 代码。我们发现，Python/Numba 实现仍缺乏必要支持，无法在这些系统上达到相当的 CPU 和 GPU 性能，而且 AMD GPU 支持已弃用。Kokkos 提供了一种值得关注的性能可移植性方案，但其效果仍取决于后端以及若干编译和策略设置。总体而言，Julia 和 Python/Numba 编程模型提供了高生产力的 CPU 和 GPU API，便于编写 LLVM 编译的内核。除此之外，它们强大的 HPC 数据分析和工作流生态系统、对半精度浮点数的无缝支持，以及与供应商后端的互操作性，都为科学发现过程增添了价值。未来工作应继续探索它们在更复杂 HPC 工作负载中的应用；随着基于 LLVM 的实现及其支持生态系统日趋成熟，它们有望在异构硬件上实现所需的性能可移植性。

## 致谢

本研究得到百亿亿级计算项目（17-SC-20-SC）的支持。该项目由美国能源部科学办公室与国家核安全管理局合作开展。本研究使用了橡树岭国家实验室橡树岭领导力计算设施的资源，该设施由美国能源部科学办公室根据合同编号 DE-AC05-00OR22725 提供支持。

## 参考文献

[1] G. Van Rossum 等，“Python 编程语言”。载于 *USENIX* 年度技术会议，第 41 卷第 1 期。加利福尼亚州圣克拉拉，2007年，第 1–36 页。[2] J. Bezanson、A. Edelman、S. Karpinski 和 V. B. Shah，“Julia：数值计算的新方法”，《SIAM评论》，第 59 卷第 1 期，第 65–98 页，2017 年 1 月。[3] R. Ihaka 和 R. Gentleman，“R：用于数据分析和图形绘制的语言”，《计算与图形统计学杂志》，第 5 卷第 3 期，第 299–314 页，1996 年。

[4] J. S. Vetter、R. Brightwell、M. Gokhale、P. McCormick、R. Ross、J. Shalf、K. Antypas、D. Donofrio、T. Humble、C. Schuman、B. Van Essen、S. Yoo、A. Aiken、D. Bernholdt、S. Byna、K. Cameron、F. Cappello、B. Chapman、A. Chien、M. Hall、R. Hartman-Baker、Z. Lan、M. Lang、J. Leidel、S. Li、R. Lucas、J. Mellor-Crummey、P. Peltz Jr.、T. Peterka、M. Strout 和 J. Wilke，“极端异构性 2018——极端异构时代的高效计算科学：美国能源部高级科学计算研究办公室极端异构性研讨会报告”，2018 年 12 月。[5] C. Lattner 和 V. Adve，“LLVM：用于持续程序分析与转换的编译框架”，载于《国际代码生成与优化研讨会》，2004 年。CGO 2004。IEEE，2004 年，第 75–86 页。[6] S. K. Lam、A. Pitrou 和 S. Seibert，“Numba：基于 LLVM 的 Python 即时编译器”，载于《高性能计算 LLVM 编译器基础设施第二届研讨会论文集》，2015 年，第 1–6 页。[7] OpenMP 架构审查委员会，“OpenMP 应用程序接口 5.2 版”，2021 年 11 月。 [在线]。网址：https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf[8] OpenACC 架构审查委员会，“OpenACC 应用程序接口 3.1 版”，2020 年 11 月。[在线]。网址：https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf[9] H. Carter 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”，《并行与分布式计算期刊》，第 74 卷第 12 期，第 3202–3216 页，2014 年，专题：面向高性能计算的领域特定语言与高级框架。 [在线]。网址：https://www.sciencedirect.com/science/article/pii/S0743731514001257[10] D. A. Beckingsale、J. Burmark、R. Hornung、H. Jones、W. Killian、A. J. Kunen、O. Pearce、P. Robinson、B. S. Ryujin 和 T. R. Scogland，“Raja：面向大规模科学应用程序的可移植性能”，载于《2019年 IEEE/ACM 高性能计算性能、可移植性与生产力国际研讨会（P3HPC）》，2019 年，第 71–81 页。[11] A. Marowka，“OpenACC、OpenMP、Kokkos 和 RAJA 的性能可移植性”，载于《HPC Asia 2022：亚太地区高性能计算国际会议》，线上会议，日本，2022 年 1 月 12–14 日。ACM，2022 年，第 103–114 页。 [在线]。网址：https://doi.org/10.1145/3492805.3492806[12] NVIDIA，“Thrust（CUDA C++ 模板库）的 API 参考指南。”2022 年 5 月。 [在线]。网址：https://docs.nvidia. com/cuda/thrust/index.html[13] ——，“CUDA 工具包文档——v11.7.0”，2022 年 5 月。 [在线]。网址：https://developer.nvidia.com/cuda-toolkit[14] AMD，“AMD ROCm v5.2 发布”，2022年6月。 [在线]。网址：https://rocmdocs.amd.com/en/latest/Current Release Notes/Current-Release-Notes.html#amd-rocm-v5-2-release[15] T. Ben-Nun、T. Gamblin、D. S. Hollman、H. Krishnan 和 C. J. New-burn，“工作流已成为新的应用程序：性能、可移植性和生产力方面的挑战”，载于《2020年 IEEE/ACM 高性能计算性能、可移植性与生产力国际研讨会（P3HPC）》，2020 年，第 57–69 页。[16] L. S. Blackford、A. Petitet、R. Pozo、K. Remington、R. C. Whaley、J. Demmel、J. Dongarra、I. Duff、S. Hammarling、G. Henry 等，“一组更新的基本线性代数子程序 BLAS”，《ACM 数学软件汇刊》，第 28 卷，151，2002 2，135 no. pp. – 。[17] H. Kim、H. Nam、W. Jung 和 J. Lee，“面向图形处理器的卷积神经网络框架性能分析”，载于《2017年 IEEE 系统与软件性能分析国际研讨会（ISPASS）》，2017 年，第 55–64 页。[18] X. Li、G. Zhang、H. H. Huang、Z. Wang 和 W. Zheng，“基于图形处理器的卷积神经网络性能分析”，载于《2016年第 45 届并行处理国际会议（ICPP）》，2016 年，第 67–76 页。[19] S. Markidis、S. W. D. Chien、E. Laure、I. B. Peng 和 J. S. Vetter，“英伟达张量核心的可编程性、性能与精度”，载于《2018年 IEEE 并行与分布式处理国际研讨会（IPDPSW）》，2018 年，第 522–531 页。[20] C. Brown、A. Abdelfattah、S. Tomov 和 J. Dongarra，“AMD 图形处理器上稠密线性代数算法的设计、优化与基准测试”，载于《2020年 IEEE 高性能极限计算会议（HPEC）》，2020 年，第 1–7 页。[21] V. Churavy、W. F. Godoy、C. Bauer、H. Ranocha、M. Schlottke-Lakemper、L. R¨ass、J. Blaschke、M. Giordano、E. Schnetter、S. Omlin、J. S. Vetter 和 A. Edelman，《通过 Julia 编程语言连接高性能计算社区》，已提交审稿，2022年。[22] H. Ranocha、M. Schlottke-Lakemper、A. R. Winters、E. Faulhaber、J. Chan 和 G. J. Gassner，《使用 Trixi.jl 进行自适应数值模拟：Julia 在科学计算中的案例研究》，《JuliaCon 会议论文集》，第1卷，第1期，第77页，2022年。[23] M. Tomasi 和 M. Giordano，《迈向科学计算的新解决方案：Julia 案例》，2018年12月。[24] W.-C. Lin 和 S. McIntosh-Smith，《将 Julia 与面向高性能计算的性能可移植并行编程模型进行比较》，载于《2021年高性能计算系统性能建模、基准测试与仿真国际研讨会（PMBS）》，2021年，第94–105页。[25] T. Faingnaert、T. Besard 和 B. De Sutter，《图形处理器上的灵活高性能 GEMM 内核》，IEEE 并行与分布式系统汇刊，第33卷，第9期，第2230–2248页，2022年。[26] S. Ko、H. Zhou、J. Zhou 和 J.-H. Won，《Diststat.jl：迈向高性能统计计算环境的统一编程》，2020年10月。[27] M. Giordano、M. Kl¨ower 和 V. Churavy，《生产力与性能兼得：A64FX 上的 Julia》，载于《2022年 IEEE 集群计算国际会议（ CLUSTER ）》，2022 年，第 549–555 页。[28] M. Snir、W. Gropp、S. Otto、S. Huss-Lederman、J. Dongarra 和 D. Walker，《MPI——完整参考：MPI 核心》。麻省理工学院出版社，1998 年，第1卷。[29] S. Byrne、L. C. Wilcox 和 V. Churavy，《MPI.jl：消息传递接口的 Julia 语言绑定》，《JuliaCon 会议论文集》，第1卷，第1期，第68页，2021年。 [在线]。网址：https://doi.org/10 21105 00068 。/jcon。J. Gmys、T. Carneiro、N. Melab、E.-G. Talbi 和 D. Tuyttens，《用于并行元启发式算法的高生产力、高性能编程语言比较研究》，《群智能与进化计算》，第57卷，第100720页，2020年9月。[31] B. L. Chamberlain、D. Callahan 和 H. P. Zima，《并行可编程性与 Chapel 语言》，《国际高性能计算应用期刊》，第21卷，第3期，第291–312页，2007年。[32] T. G. Mattson、T. A. Anderson 和 G. Georgakoudis，《PyOMP：Python 中的多线程并行编程》，《科学与工程计算》，第23卷，第6期，第77–80页，2021年。[33] L. Oden，《比较 C-CUDA 与 Python-Numba 进行 GPU 计算的经验教训》，载于《2020年第28届 Euromicro 并行、分布式与网络处理国际会议（PDP）》，2020年，第216–223页。[34] D. Di Domenico、G. G. H. Cavalheiro 和 J. V. F. Lima，《使用 Python 实现的 NAS 并行基准测试内核：聚焦图形处理器的性能与编程工作量分析》，载于《2022年第30届 Euromicro 并行、分布式与网络处理国际会议（PDP）》，2022年，第26–33页。[35] A. Kl¨ockner、N. Pinto、Y. Lee、B. Catanzaro、P. Ivanov 和 A. Fasih，《PyCUDA 和 PyOpenCL：基于脚本的 GPU 运行时代码生成方法》，《并行计算》，第38卷，第3期，第157–174页，2012年。[36] R. Nishino 和 S. H. C. Loomis，《CuPy：兼容 NumPy 的 NVIDIA GPU 计算库》，第31届神经信息处理系统会议，第151卷，第7期，2017年。[37] K. Teranishi、D. M. Dunlavy、J. M. Myers 和 R. F. Barrett，《Sparten：利用 Kokkos 在节点内并行化二阶方法，以拟合泊松数据的典范多元张量模型》，载于《2020年 IEEE 高性能极限计算会议（HPEC 2020）》，美国马萨诸塞州沃尔瑟姆，2020年9月22日至24日。IEEE，2020年，第1–7页。[38] R. Halver、J. H. Meinke 和 G. Sutmann，《Ewald 库仑求解器的 Kokkos 实现及其性能可移植性分析》，《并行与分布式计算杂志》，第138卷，第48–54页，2020年。[39] S. Rajamanickam、S. Acer、L. Berger-Vergiat、V. Q. Dang、N. D. Ellingwood、E. Harvey、B. Kelley、C. R. Trott、J. Wilke 和 I. Yamazaki，《Kokkos 内核：性能可移植的稀疏/稠密线性代数与图内核》，CoRR，卷号 abs/2103.11991，2021年。[ ] 2103 11991 在线。网址：https://arxiv.org/abs/。[40] J. A. Ellis 和 S. Rajamanickam，《使用 Kokkos 内核对稀疏深度神经网络进行可扩展推理》，载于《2019年 IEEE 高性能极限计算会议（HPEC 2019）》，美国马萨诸塞州沃尔瑟姆，2019年9月24日至26日。IEEE，2019年，第1–7页。[41] P. Valero-Lara、S. Lee、M. G. Tallada、J. E. Denny 和 J. S. Vetter，《KokkACC：通过 OpenACC 增强 Kokkos》，载于《第9届使用指令进行加速器编程研讨会（WACCPD@SC 2022）》，美国得克萨斯州达拉斯，2022年11月13日至18日。IEEE，2022年，第32–

42。 [在线]。网址：https://doi.org/10.1109/WACCPD56842.2022。00009[42] S. Catal´an、T. Usui、L. Toledo、X. Martorell、J. Labarta 和 P. Valero-Lara，《迈向自动调优的任务型 SpMV（LASs 库）》，载于《OpenMP：现代系统上的可移植多级并行——第16届 OpenMP 国际研讨会，IWOMP 2020，美国得克萨斯州奥斯汀，2020年9月22日至24日，论文集》，计算机科学讲义系列，K. F. Milfeld、B. R. de Supinski、L. Koesterke 和 J. Klinkenberg 编，第12295卷。施普林格，2020年，第115–129页。

[43] [ ] 10 1007 978 3 030 58144 2 8 在线。网址：https://doi.org/。/-----P. Valero-Lara、D. Andrade、R. Sirvent、J. Labarta、B. B. Fraguela 和 R. Doallo，《面向多核处理器的大型三对角系统快速求解器（lass 库）》，IEEE Access，第7卷，第23 365–23 378页，2019年。

[44] [ ] 10 1109 2019 2900122 在线。网址：https://doi.org/。/ACCESS。。P. Valero-Lara、S. Catalan、X. Martorell、T. Usui 和 J. Labarta，《sLASs：基于 OmpSs 中实现的 OpenMP 扩展、全自动调优的线性代数库（LASs 库）》，《并行与分布式计算杂志》，第138卷，第153–171页，2020年。 [在线]。网址：https://doi.org/

[45] 10 1016 2019 12 002。/j.jpdc.。。P. Valero-Lara、S. Catalan、X. Martorell 和 J. Labarta，《使用 OmpSs 区域优化的 BLAS-3（lass 库）》，载于《第27届 Euromicro 并行、分布式与网络处理国际会议（PDP 2019）》，意大利帕维亚，2019年2月13日至15日。IEEE，2019年，第25–32页。 [在线]。网址：https://doi.org/10.1109/EMPDP。2019.8671545[46] J. E. Stone、D. Gohara 和 G. Shi，《OpenCL：异构计算系统的并行编程标准》，《科学与工程计算》，第12卷，第3期，第66–73页，2010年5月。[47] 《SYCL 2020规范第5次修订》，2022年5月。 [在线]。网址：https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf [48] C. Bertoni、J. Kwack、T. Applencourt、Y. Ghadar、B. Homerding、C. Knight、B. Videau、H. Zheng、V. A. Morozov 和 S. Parker，《跨英特尔和 NVIDIA 平台评估 OpenCL 基准测试的性能可移植性》，载于《2020年 IEEE 并行与分布式处理国际研讨会系列会议（IPDPSW 2020）》，美国路易斯安那州新奥尔良，2020年5月18日至22日。IEEE，2020年，第330–339页。

[49] [ ] 10 1109 50202 2020 00067 在线。网址：https://doi.org/。/IPDPSW。。R. Halver、W. Homberg 和 G. Sutmann，《使用 OpenCL 在异构并行架构上实现分子动力学的函数可移植性》，《超级计算杂志》，第74卷，第4期，第1522–1533页，2018年。 [在线]。网址：https://doi.org/

[50] 10 1007 11227 017 2232 2 。/s---M. Tanvir、K. Narasimhan、M. Goli、O. E. Farouki、S. Georgiev 和 I. Ault，《使用 SYCL-DNN 实现人工智能模型性能可移植性的探索》，载于《IWOCL'22：OpenCL 国际研讨会》，英国布里斯托尔，2022年5月10日至12日。ACM，2022年，第23:1–23:3页。 [ ] 10 1145 3529538 3529999 在线。网址：

[51] https://doi.org/。/。T. Sabino 和 M. Goli，《使用 SYCL 实现高度可参数化 TRSM 算法性能可移植性的探索》，载于《IWOCL'21：OpenCL 国际研讨会》，德国慕尼黑，2021年4月，S. McIntosh-Smith 编。ACM，2021年，第5:1–5:10页。 [在线]。网址：https://doi.org/

[52] 10 1145 3456669 3456694 。/。T. Besard、C. Foket 和 B. De Sutter，《高效且可扩展的编程：释放 Julia 在 GPU 上的潜力》，IEEE 并行与分布式系统汇刊，2018年。[53] T. Besard、V. Churavy、A. Edelman 和 B. De Sutter，《面向异构与分布式平台的快速软件原型开发》，《工程软件进展》，第132卷，第29–46页，2019年。[54] J. Samaroo、V. Churavy、W. Phillips、A. Ramadhan、J. Barmparesos、J. TagBot、L. R¨ass、M. Schanen、T. Besard、A. Smirnov、T. Arakaki、S. Antholzer、Alessandro、C. Elrod、M. Raayai 和 T. Hu，《JuliaGPU/AMDGPU.jl：v0.4.1》，2022年8月。 [在线]。网址：https://doi.org/

[55] 10 5281 6949520 。/zenodo。V. Churavy、D. Aluthge、L. C. Wilcox、J. Schloss、S. Byrne、M. Waruszewski、J. Samaroo、A. Ramadhan、Meredith、S. Schaub、J. Bolewski、A. Smirnov、C. Kawczynski、C. Hill、J. Liu、O. Schulz、Oscar、P. Haraldsson、T. Arakaki 和 T. Besard，《JuliaGPU/KernelAbstractions.jl：v0.8.3》，2022年6月。[在线]。网址：https://doi.org/

[56] 10 5281 6742177 。/zenodo。美国能源部，《美国能源部卓越中心性能可移植性会议》，2016年4月。 [在线]。网址：https://asc.llnl.gov/sites/asc/files/2020-09/COE-PP-Meeting-2016-FinalReport 0.pdf [57] S. Pennycook、J. Sewall 和 V. Lee，《性能可移植性度量指标的影响》，《未来一代计算机系统》，第92卷，第947–958页，2019年。 [在线]。可访问：https://www.sciencedirect.com/ science/article/pii/S0167739X17300559

[58] A . Marowka，"迈向更好的性能可移植性指标"，载于第*29*届 *Euromicro* 并行、分布式和网络化处理国际会议，巴利亚多利德，西班牙，*2021*年*3*月*10*日至*12*日。IEEE，2021年，第181–184页。 [在线]。可访问：https://doi.org/10.1109/PDP52278.2021.00036

## 附录 A

### 可复现性制品描述

本研究使用的代码托管在 GitHub 上：https: // github.com/williamfgc/simple-gemm。每种实现都有自己的目录：C语言、Kokkos、Julia 和 Python。脚本目录包含在 OLCF系统上进行各项实验的配置。图8和图9展示了在 Wombat 上运行 C/OpenMP 和 Julia 实验的脚本示例。

```
#!/bin/bash

可执行文件=../simple-gemm/c/gemm-dense-openmp64-armclang
模块 load ARM_编译器_For_HPC/22.0
Ms=( 4096 5120 ... 19456 20480 )

export OMP_PROC_BIND=true
export OMP_PLACES=线程
export OMP_NUM_THREADS=80

for M in "${Ms[@]}"; do
   salloc -N 1 -p Ampere -t 10:00:00 srun -n 1 -c $t \
      $可执行文件 $M $M $M  5 >
         Ampere-ARMClang→  22  _5 _  2> 1 -${t}t-${M}M s线程.log &
done
```

图8：Wombat CPU C/OpenMP 启动脚本。

```
#!/bin/bash

加载 nvhpc-nompi/22.1 模块
module load julia/1.7.3
导出 JULIA_CUDA_USE_BINARYBUILDER=false
GemmDenseCUDADIR=../../simple-gemm/julia/GemmDenseCUDA
可执行文件=$GemmDenseCUDADIR/gemm-dense-cuda.jl

Ms=( 4096 5120 ... 19456 20480 )

for M in "${Ms[@]}"; do

   salloc -N 1 -p Ampere -t 10:00:00 --gres=gpu:1 \
      srun -n 1 julia -O3 --project=$GemmDenseCUDADIR \
      $可执行文件 $M $M $M 5 \
      > A100-Julia1_7_3-${M}M_5s_F64.log  2>&1 &
done
```

图 9：Wombat 图形处理器 Julia 启动脚本。

表 I 和表 II 描述了用于生成所有实验的软件栈和编译标志。Kokkos 实现位于 simple-gemm/cpp/Kokkos/ 中，并使用其自己的编译框架。