便携式高性能内核生成用于计算流体动力学代码的 DaCe

Måns I. Andersson[1,3][0000−0002−6384−2630], Martin Karp[1][0000−0003−3374−8093], Niclas Jansson[2][0000−0002−5020−1631], and Stefano Markidis[1][0000−0003−0639−0639]

[1] KTH Royal Institute of Technology
[2] PDC Centre for High Performance Computing
[3] mansande@kth.se

摘要 随着新的高性能计算（HPC）加速器的出现，如 Nvidia 和 AMD 的 GPU，有效针对各种硬件架构已经成为 HPC 应用程序开发者的重大挑战。HPC 系统中不断增加的硬件多样性通常需要开发特定于架构的代码，这阻碍了大规模科学应用的可持续性。在这项工作中，我们利用以数据为中心的并行编程框架 DaCe 来自动化生成高性能内核。DaCe 能够为多核处理器和各种加速器自动生成代码，减轻了开发者为每个新架构重写代码的负担。我们的研究通过将自动代码生成应用于计算流体动力学（CFD）中使用的关键计算内核，展示了 DaCe 的能力。具体而言，我们关注的是基于 Fortran 的求解器 Neko，它采用谱元方法，依赖于小张量运算。我们详细描述了利用 DaCe 的状态数据流多图（SDFG）表示来制定这个计算内核，并讨论这种方法如何促进高性能代码生成。此外，我们概述了将 DaCe 生成的代码无缝集成到 Neko 求解器中的工作流程。我们的结果突出了生成代码在多个平台上的可移植性和性能，包括 Nvidia GH200、Nvidia A100 和 AMD MI250X GPU，具有竞争力的性能结果。通过展示自动代码生成的潜力，我们强调了使用便携式解决方案确保大规模科学应用长期可持续性的可行性。

Keywords: 性能可移植性生产力 · 谱元法 · 计算流体力学。

1 介绍

在当前高性能计算领域，加速器的使用增加，尤其是图形处理单元（GPU），已成为现代超级计算机的重要特征。位于洛斯阿拉莫斯国家实验室的 Roadrunner 超级计算机作为首个突破每秒千万亿次浮点运算性能障碍的

系统，标志着一个重要里程碑。该超级计算机配备了 6,480 个 AMD Opteron 双核处理器和 12,960 个 IBM PowerXCell 8i 加速器，引领了一个以异构计算能力为特征的新时代。随着 GPU 的出现，Nvidia 的 CUDA 编程框架成为优化科学应用的关键工具，带来了性能提升。今天，我们通过 Frontier 超级计算机实现了 exascale 的能力，该系统配备了 AMD MI250x GPU。然而，这一转变引入了另一种挑战，即 ROCm HIP，这是需要利用这些加速器全部功能的另一个编程接口。同时，其他超级计算机也采用了现场可编程门阵列（FPGA），每种都要求不同的界面。因此，软件开发者面临着不断适应和重写代码以满足最新计算设备及其相关编程接口的需求。

应对这一新加速器的支持和利用挑战，但高性能计算应用程序需要选择和使用能够使应用开发者无需重写应用程序的有效编程系统或方法。一种可能性是使用编译器指令或代码注释来指示加速器的并行化或移植需求。在这种范式中，编译器生成必要的代码转换。OpenMP 是这种方法的一个示例。然而，它伴随着显著的性能限制——由于缺乏算法洞察力，编译器无法优化特定代码段。这一限制推动了 MLIR 的开发和采用。

我们在本研究中采用的一种替代策略是构建一个单源、面向对象的科学应用程序框架。该框架包含了一个通用接口，适用于可针对不同硬件架构的各种实现方式的目标计算内核，这些目标被称为“后端”。TensorFlow 是这种方法的一个显著示例，它支持多种专为特定加速器定制的后端。我们的研究将这一范式扩展到了计算流体动力学领域，重点以 Neko CFD 代码作为说明性示例 [8–10, 12]。这种策略的缺点是程序员需要使用多种编程模型和语言来编写代码。我们在本工作中采用的方法是 DaCe，这是一个数据为中心的编程框架 [5]，它允许我们首先将算法表达为 SDFG，使我们能够解耦算法制定和性能优化以及移植到不同架构，并且其次自动生成每个系统的代码。本工作的主要目标是实现大规模计算流体动力学代码 Neko 的自动高效 HPC 代码生成，并评估这种方法的有效性。本工作的主要贡献如下：

- 我们将主要的 Neko 计算内核，矩阵评估，表示为 DaCe SDFG，并生成针对不同加速器系统的优化代码。
- 我们设计并开发了一个工作流，将 DaCe 代码集成到 Neko 计算流体动力学求解器中。
- 我们评估了 DaCe 生成的代码在不同加速系统上的性能，包括 Nvidia GH200、Nvidia A100 和 AMD MI250x GPU 系统，并将它们与手工编写的 Neko 内核进行了比较。

2 背景

在本节中，我们讨论了背景材料和相关前期研究，并将我们在使用自动代码生成以及将其整合到一个大型流体动力学代码 Neko 中的工作置于上下文中。

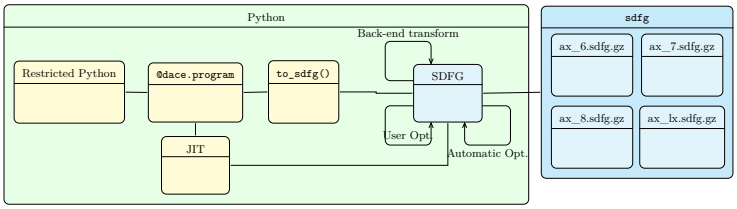

图 1. Ax 内核 (`ax.sdfg`) 的 SDFG 是从一个受限的 Python 形式生成的。一般来说，SDFGs 可以即时生成和编译。但在这种情况下，我们生成 SDFG 应用优化变换，并将变换后的 SDFGs 以压缩格式存储。

数据驱动的编程框架 DaCe [5] 被用于生成优化代码并集成到 CFD 代码中。DaCe 编程框架在苏黎世联邦理工学院积极开发，并已在多个应用中使用 [2, 3, 14]。完全由 DaCe 开发的最著名的应用程序是材料科学研究用的 OMEN 代码。OMEN 代码因其能够实现全机 Summit 仿真而获得了著名的戈登贝尔奖。DaCe 框架基于一种称为 SDFG 的中间表示，该表示旨在表达数据流和并行性移植性。该中间表示由高级 Python 前端生成，并生成程序的图形图表示。在中间表示形式上生成内核的过程可以在 figure 1中找到。

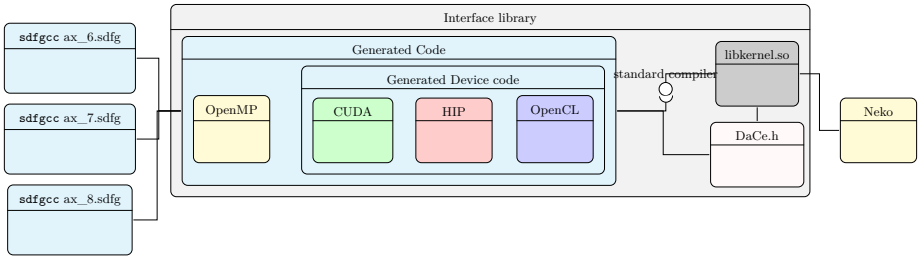

图 2. 该库随附了多个 ax.sdfg（每个多项式阶次一个）文件。可以在使用 `sdfgcc` 编译之前应用额外的优化传递。编译配置最好通过 DaCe 环境变量来控制。

SDFG 拥有自己的代码生成器和支持多个硬件后端的库 figure 2。

2.1 猫和小张量核

Neko 是一个新的更新版 CFD 代码，其根源来自 Nek5000。它在大规模运行中表现出色的并行性能，包括全规模 LUMI [8] 的运行。该代码已通过多个测试进行验证，并应用于不同的重大挑战，包括 Flettner 旋转器和瑞利-贝纳德对流案例。与它的前身不同，Neko 代码是用现代 Fortran 编写的，采用了动态内存管理和面向对象的结构，利用一个统一的接口来实现不同的后端。Neko 基于谱元法。在此方法中，模拟被划分为多个单元，这些单元使用高阶基函数进行离散化。Neko 代码的两个主要组成部分是矩阵乘法和聚集-散布操作。矩阵乘法通常尺寸较小，因为其大小取决于谱元的阶数。

3 谱元方法

本文研究了出现在 CFD 代码 Neko 中的单个测试核的性能移植性。我们将调查一个简化的子问题，该子问题是 Neko 使用的完整 Navier-Stokes 求解器的一部分，即生成 Poisson 方程矩阵自由评估的计算部分，这是主要组成部分之一。以下是 Poisson 方程的标准弱形式公式化，它是任何有限元方法 (FEM) 求解器的起点。

$$\int_\Omega \nabla u : \nabla v \,\mathrm{d} = \int_\Omega f v \,\mathrm{d}, \quad \forall v \in V, \tag{1}$$

其中 Ω 是域，v 是 V 中的任意测试函数。谱元素法是一种基于特定元素选择的有限元方法，它使用 N^{th} 阶 Legendre 多项式 l_i 在 Gauss-Lobatto-Legendre (GLL) 求积点 ξ 上进行插值，这些求积点位于参考单元上。

$$l_i(\xi) = \frac{N(1-\xi^2 L_N(\xi))}{(N+1)(\xi-\xi_i)L_N(\xi_i)}, \quad \xi \in [-1,1]. \tag{2}$$

谱元素解 u_h 可以表示为参考单元（u_h^e）上的张量积，该解存在于 V_h 中，

$$u_h^e(\xi,\eta,\gamma) = \sum_{i,j,k}^{N} u_{ijk} l_i(\xi) l_j(\eta) l_k(\gamma). \tag{3}$$

4 方法论

我们将与 Neko（v0.9.99）最新发布的内核进行比较。这依赖于为每个支持的后端编写的手写代码。基线实现是 Neko 中使用的 CUDA 和 HIP 实

现。Neko 提供了两种算法，这些算法在运行时基于时间自动调整选择，该选择可以通过将 NEKO_AUTO 调整设置为首选方法（1D 或 K 步）来执行，这两种实现之间的主要差异与如何最好地利用 GPU 的共享内存有关。

4.1 与猫集成

在 figure 2中，我们有一个图表来展示从生成的 SDFG 到最终链接到 Neko 的库编译的不同步骤。Neko 是为了性能可移植性而设计的——也就是说，它的创建初衷是每个硬件后端都可以通过少量样板代码添加。在 Listing 1.1中，我们展示了如何在 Neko 中使用 C 到 Fortran 接口，每个后端实现都有一个对应的执行调用包装器。由于重点在于加速器卸载，我们在 Python 代码中标注了存储类型.GPU_全局，以表示生成的 SDFG 的输入参数位于设备上。

```
interface
 subroutine dace_x_helm(handle, wd, ud, &
      dxd, dyd, dzd, dxtd, dytd, dztd, &
      h1d, g11d, g22d, g33d, g12d, g13d, g23d, nelv, lx) &
      bind(c, name='__dace_ax_helm')
   use, intrinsic :: iso_c_binding
   type(c_ptr), value :: handle
   type(c_ptr), value :: wd, ud
   type(c_ptr), value :: dxd, dyd, dzd
   type(c_ptr), value :: dxtd, dytd, dztd
   type(c_ptr), value :: h1d, g11d, g22d, g33d, g12d, g13d, g23d
   integer(c_int) :: nel, lx
 end subroutine dace_ax_helm
end interface
```

Listing 1.1. The interface with a backend library using device pointers. In this case for the DaCe implementation of the kernel. The DaCe implementation does not require any boilerplate wrapping code for different GPU backends.

4.2 内核在 DaCe 中的表示

DaCe 原则上旨在尽可能接近数学表述来表示内核，并将优化工作委托给中间表示 SDFG 的类似编译器的基础设施。为了与当前代码库配合，我们从一个现有实现开始并对其进行了简化。重新设计的实现在 Listing 1.2中可以找到，请注意该内核被简单地拆分成了两个遍历所有元素的并行循环（e1 和 e2）。

```
1  dtype = dace.float64
2  @dace.program()
3  def ax(wd : dtype[ne,lx,lx,lx],  ud : dtype[nel,lx,lx,lx],
4       dxd  : dtype[lx,lx],          dyd  : dtype[lx,lx],
5       dzd  : dtype[lx,lx],          dxtd : dtype[lx,lx],
6       dytd : dtype[lx,lx],          dztd : dtype[lx,lx],
7       g11d : dtype[nel,lx,lx,lx],  g22d : dtype[nel,lx,lx,lx],
8       g33d : dtype[nel,lx,lx,lx],  g12d : dtype[nel,lx,lx,lx],
9       g13d : dtype[nel,lx,lx,lx],  g23d : dtype[nel,lx,lx,lx])
10      h1d  : dtype[nel,lx,lx,lx]):
11
12     # Temp arrays
13     stmp   = np.empty((nel,lx,lx,lx),dtype=dtype)
14     rtmp   = np.empty((nel,lx,lx,lx),dtype=dtype)
15     ttmp   = np.empty((nel,lx,lx,lx),dtype=dtype)
16     urtmp  = np.empty((nel,lx,lx,lx),dtype=dtype)
17     ustmp  = np.empty((nel,lx,lx,lx),dtype=dtype)
18     uttmp  = np.empty((nel,lx,lx,lx),dtype=dtype)
19
20     # First map over all elements for each spatial dimension
21     for e, k, j, i in dc.map[0:ne,0:lx,0:lx,0:lx] @ScheduleType.GPUdevice :
22         rtmp[e,k,j,i] = 0.0
23         stmp[e,k,j,i] = 0.0
24         ttmp[e,k,j,i] = 0.0
25
26         # Sequential most inner loop
27         for l in range(lx):
28            rtmp[e,k,j,i] = rtmp[e,k,j,i]+dxd[l,i]*ud[e,k,j,l]
29            stmp[e,k,j,i] = stmp[e,k,j,i]+dyd[l,j]*ud[e,k,l,i]
30            ttmp[e,k,j,i] = ttmp[e,k,j,i]+dzd[l,k]*ud[e,l,j,i]
31
32         G00 = g11d[e,k,j,i]; G01 = g12d[e,k,j,i]; G02 = g13d[e,k,j,i]
33         G11 = g22d[e,k,j,i]; G12 = g23d[e,k,j,i]; G22 = g33d[e,k,j,i]
34         H   = h1d[e,k,j,I]
35         rtmp_s = rtmp[e,k,j,I]
36         stmp_s = stmp[e,k,j,I]
37         ttmp_s = ttmp[e,k,j,i]
38
39         urtmp[e,k,j,i] = H * (G00*rtmp_s + G01*stmp_s + G02*ttmp_s])
40         ustmp[e,k,j,i] = H * (G01*rtmp_s + G11*stmp_s + G12*ttmp_s])
41         uttmp[e,k,j,i] = H * (G02*rtmp_s + G12*stmp_s + G22*ttmp_s])
42
43     # Second map over all elements for each spatial dimension
44     for e2, k2, j2, i2 in dc.map[0:ne,0:lx,0:lx,0:lx]@ScheduleType.GPUdevice:
45         wd[e2,k2,j2,i2] = 0.0
46         for l2 in range(lx):
47            wd[e2,k2,j2,i2] = wd[e2,k2,j2,i2] + dxtd[l2,i2]*urtmp[e2,k2,j2,l2]
48            wd[e2,k2,j2,i2] = wd[e2,k2,j2,i2] + dytd[l2,j2]*ustmp[e2,k2,l2,i2]
49            wd[e2,k2,j2,i2] = wd[e2,k2,j2,i2] + dztd[l2,k2]*uttmp[e2,l2,j2,i2]
```

Listing 1.2. The DaCe kernel written in the Python frontend, adapted from the standard kernel but simplified to easily be expressed with the DaCe formalism. Note the temporary arrays added on lines 12 to 17.

SDFG IR 基于一组原始元素，这些元素在本节中简要描述。存在一种可读的 SDFG IR 的可视化表示，但本文大多数情况下将省略这种表示，原始元素的可视化表示将在括号内描述。基本的原始元素是状态（方形/矩形），并且它们是相连的。数据移动由*内存连接器*（箭头）处理，这表明在用箭头标记的 `sdfg` 中容器之间数据的移动。并行性通过映射（分叉六边形）表示，

在最简单的情况下，映射内的所有内容都可以进行尴尬的并行执行或使用写冲突解决（虚线）。DaCe 编程模型通常会导致需要在优化步骤中移除临时数据容器（椭圆）的形式。任务轻量级程序（六边形）是 `sdfg` 的基本单位，并未在 figure 3中显示，因为其被认为过于细化而对本研究不感兴趣。

4.3 具有 SDFG 的优化

在接下来的部分中描述的并行化策略是通过使用贪婪方法对 SDFGs 进行 GPU 优化而得到的结果，尽可能地合并映射，并引入共享内存容器。DaCe

表 1. 下一节中将使用的变换的简要概述。

映射变换	
映射融合	Fuses two consecutive maps, if the maps align.
映射折叠	Collapses two nested maps into one. The new map has the union of the dimensions of the original maps
映射扩展	Expands a map into the different iterators and enables a hierarchical view of parallelism.
地图瓦片	Implements the orthogonal tiling, which is a type of nested map fission that creates tiles in every dimension of the matched map.
剥采开采	Allows loop-fracturing which can allow vectorization and improved cache usage with loop-blocking.
瓦普平铺	A specific transform that combines StripMining and MapTiling.
数据变换	
本地存储	Introduces a local transient array for caching data
流变换	
状态融合	Fuses two states into a single state.
映射到 `For` 循环	Converts map to for-loop.

提供的以数据为中心的编程模型迫使程序员首先以通用的方式编写内核，在创建 SDFG 时过早的优化常常妨碍了通过 SDFG 变换来提高性能的能力。一些最常见的转换总结在 Table 1中。本工作集中在三个方面：1) 融合所有元素的两个映射；2) 沿着空间维度 i、j 和 k 进行块 铺砌；3) 共享内存变换以移除在将内核 DaCe 转换时引入的瞬态数据容器。在 figure 3中，我们展示了描述于 Listing 1.3中的 passes 之前和之后的人可读 SDFG 格式，值得注意的是共享内存变换用颜色突出显示。

```python
def ax_3D_optimization_1(sdfg : dc.SDFG, lx_val : int)
    sdfg.apply_gpu_transformations()
    sdfg.apply_transformations(MapExpansion) # Expand Maps 'e','i','j','k'
    sdfg.apply_transformations(MapCollapse)  # Recollapse Maps 'i' and 'j'
    sdfg.apply_transformations(MapCollapse)  # Recollapse Maps 'i,j' and 'k'
    sdfg.replace('lx',str(lx_val))           # Specify symbol to a constant
                                             # enables constant propagation
    entry = find_map_by_param(sdfg, 'e')
    exit  = find_map_by_param(sdfg, 'k')
    exit.schedule = dc.ScheduleType.GPU_ThreadBlock #Promotes thread-level

    for arr in ['ud', '', ..., '']:
       InLocalStorage.apply_to(sdfg,dict(array=arr),node_a=entry,node_b=exit)

def ax_3D_optimization_2(sdfg : dc.SDFG, lx_val : int)
    entry = find_map_by_param(sdfg, 'e2')
    MapExpansion.apply_to(sdfg, map_entry=entry)
    # Explicit Map collapse
    MapCollapse.apply_to(sdfg, outer_map_entry=find_map_by_param(sdfg, 'k2'),
                               inner_map_entry=find_map_by_param(sdfg, 'j2'))
    MapCollapse.apply_to(sdfg, outer_map_entry=find_map_by_param(sdfg, 'j2'),
                               inner_map_entry=find_map_by_param(sdfg, 'i2'))
    exit = find_map_by_param(sdfg,'k2')
    exit.schedule = dc.ScheduleType.GPU_ThreadBlock

    for arr in ['dxtd', '', ..]
       InLocalStorage.apply_to(sdfg,dict(array=arr),node_a=entry,node_b=exit)

    # Fuse the two outer Maps (over elements)
    sdfg.apply_transformations(MapFusion)
    sdfg.simplify()
```

Listing 1.3. The first function treats the first part of the code. It has two main tasks: to promote data containers to shared memory and to prepare for the fusion of the `e1` and `e2`. The second function treats the remaining code. It has two main tasks: to promote data containers to shared memory and to fuse the `e1` and `e2` maps.

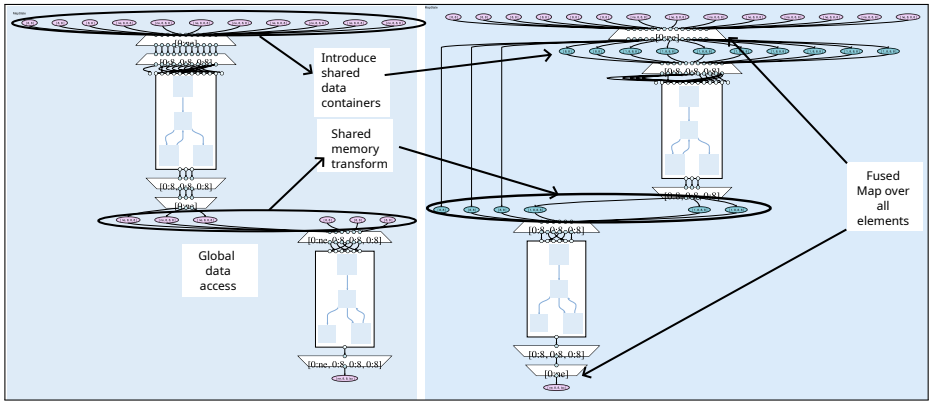

图 3. 一个从原始 SDFG 实现转换为 3D 并行化的可读 IR 示例。数据容器是椭圆形的。深青色 表示共享内存数据容器，而暗粉红色 表示全局 GPU 数据容器。由于所有数据移动都发生在相同的并行映射内，因此允许内存提升。

4.4 实验设置

本研究依赖于三个系统，其中包括 Nvidia 和 AMD 的 GPU。我们有两个 Nvidia 节点（NJ 和 Sleipner）在一个本地集群中，该集群之前被用于评估 Nvidia 硬件 A100 [16]，和 GH200 [15] 的性能。AMD 节点来自芬兰 Kajaani 的 CSC 数据中心的 LUMI-G 部分超级计算机 LUMI。硬件和软件设置总结在 Table 2中，仅考虑一个 GCD。Neko 在这些实验中的配置很简单，因为只需要几个依赖项。Neko 和 DaCe 内核使用双倍编译。我们在拥有

表 2. 测试系统和软件设置。

系统名称	图形处理单元	CUDA/ROCm	编译器	DaCe	猫
NJ	A100	CUDA 11.5	GCC 12	0.15.1*	0.9.99
Sleipner	GH200	CUDA 12.5	GCC 13	0.15.1*	0.9.99
LUMI-G	MI250X	ROCm 6.0.3	Cray 16	1.0.1**	0.9.99

* 我们发现在两套 NVIDIA 系统上，近期 DaCe 版本（0.16 和 1.0.1）的性能显著下降。

** 较旧的 DaCe 版本 0.15.1 和 0.16 在 MI250X 机器上没有生成正确的代码或正确的构建配置。

128、256、512、1024、2048、4096、8192、16384 和 32768 个元素 (N_E) 及每个问题 $N_E(lx-1)^3$ 个未知数的九种不同大小的立方网格上进行了评估。我们对 $3 \leq lx \leq 8$ 执行性能测量。

5 结果

本研究通过在三种不同类型 GPU 上的九种网格尺寸和六个基函数多项式阶次中测得的 Gflops/s 来评估这三种实现方式。

GH200 (figure 4): DaCe 实现的并行化策略在大多数超过 2048 个元素的网格尺寸上表现优于 `1D` 策略。`K` 步策略在大多数配置中表现出色，尽管 $lx = 6$ 值得注意，因为 DaCe 是在大网格尺寸和最高实现性能方面表现最好的策略。最高实现性能的是 `K` 步策略，这种情况发生在最大的网格尺寸上以及 $lx = 8$ 上，这是预期的结果。我们注意到 `K` 步法在使用较小且 lx 较高的网格时性能达到峰值，随后随着元素数量的增加性能急剧下降，然后逐渐上升。

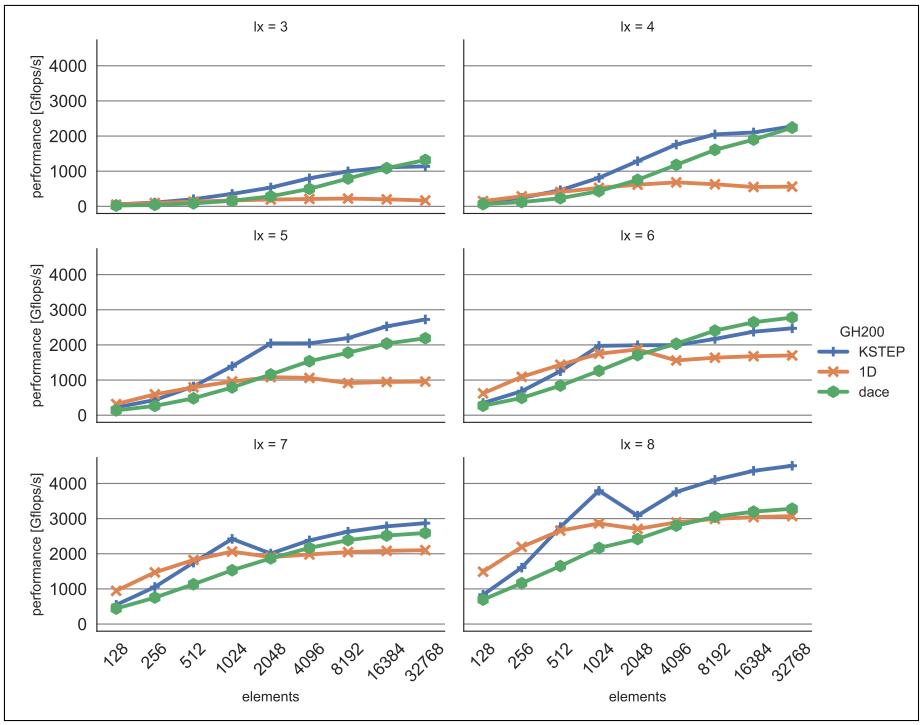

图 4. 不同网格大小和 $3 \leq lx \leq 8$ 在 GH200 节点上不同实现的性能（Gflops/s）。

A100 (figure 5): 来自 A100 机器的结果由于更多的后台进程而具有较大的标准偏差，因为它是一个共享节点。总体趋势与 GH200 相似；然而，DaCe 和 K 步在 $lx < 7$ 上表现更为相似。对于更高的 lx：步骤 K 表现更好。1D 实现对于大网格尺寸始终具有最低的性能，但对于非常少的元素则具有最佳性能。与 GH200 的结果相反，峰值性能通常在小网格尺寸下找到，除了 $lx = 3$。与 GH200 的情况相反，在下降之后性能的缓慢增加从未赶上峰值性能

。

MI250X (figure 6): 我们注意到 DaCe 在 MI250X 上的性能与基础实现相比明显不足，但值得注意的是，所有策略的性能都远不及 A100 的峰值性能。对于 $lx > 6$，1D 策略在大多数网格尺寸下表现最佳，这与 Nvidia 的结果形成了鲜明对比。对于小 lx，行为类似于之前的测试。峰值性能在中等大小网格上是 K 步法和 1D 的平局，以及 $lx = 8$

图 5. 不同网格尺寸和 $3 \leq lx \leq 8$ 在 A100 节点上使用不同实现的性能（Gflops/s）。

6 相关工作

性能移植框架随着 HPC 系统不断演化，出现多样化的硬件架构，实现跨不同处理设备的性能移植变得越来越重要。为应对这一挑战，人们开发了多种方法。Kokkos 是一个广泛采用的 C++库，它提供了一个抽象层，用于在多个硬件平台（包括 CPU 和 GPU）上进行并行执行和内存管理 [17]。类似地，RAJA 是一种并行编程框架，旨在提供抽象以实现跨异构硬件的性能移植 [4]。此外，羊驼 [19] 支持多种执行模型，如 CUDA、OpenMP 和 SYCL。**便携式高性能谱元核和无矩阵核**谱元素代码 NekRS [6] 依赖于 OCCA 内核 [1, 7] 来实现可移植性（CUDA、HIP、OpenCL）。类似的基准代码基于 `Nekbone`，有针对 FPGA [11] 和 OpenCL [18] 的实现。低阶模板基矩阵自由算子在便携式美洲驼基于泊松求解器 [13] 中进行了研究。

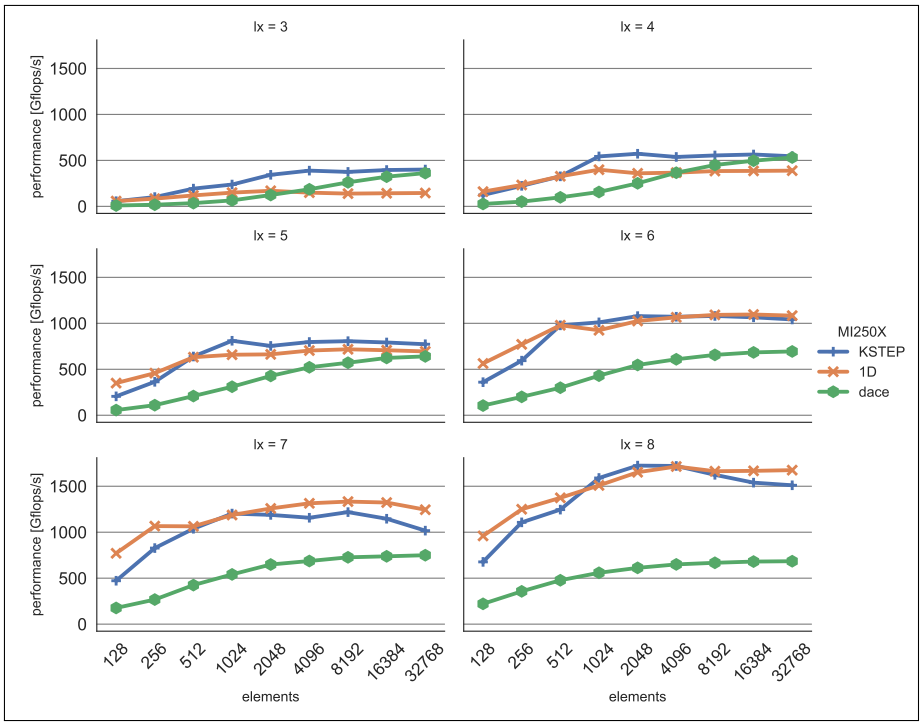

图 6. 不同网格尺寸和 $3 \leq lx \leq 8$ 的性能（Gflops/s）在使用一个 GCD 的 MI250X 节点上的不同实现。

7 讨论与结论

我们展示了 Neko 矩阵评估的主要计算内核在 DaCe 中的一组优化传递实现，生成的代码可与某些手调后端特定实现相竞争。我们将生成的代码集成到了基于现代 Fortran 的大规模求解器中。我们承认可以按照以往文献进行进一步的优化，例如其他并行化策略，比如更适合高于 lx 八阶的 2D 或 1D 并行化策略。我们在 AMD 系统上展示了成功的运行，但不幸的是性能有所损失。这可能是由于 DaCe 从 0.15.1 版本到 0.16 版本之间的变化导致了与 Nvidia 版本相比性能下降的程度类似于我们在 AMD 系统上看到的情况。总之，使用单源实现的性能可与高度调优的 Neko 内核相媲美，并且易于调整。

Acknowledgments. 本工作的计算由瑞典国家超级计算学术基础设施项目 NAISS 2024/22-531 和 NAISS 2024/9-20 以及 ScaLab 小组提供的 Sleipner 集群访问所支持。资金支持来自瑞典 e-Science 研究中心 (SeRC) 的百亿亿次仿真软件倡议 (SESSI)。

参考文献

1. Abdelfattah, A., Barra, V., Beams, N., Bleile, R., Brown, J., Camier, J.S., Carson, R., Chalmers, N., Dobrev, V., Dudouit, Y., et al.: Gpu algorithms for efficient exascale discretizations. Parallel Computing **108**, 102841 (2021)
2. Andersson, M.I., Markidis, S.: A case study on dace portability & performance for batched discrete fourier transforms. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. pp. 55–63 (2023)
3. Andersson, M.I., Karp, M., Markidis, S.: Towards performance portable kernels for computational fluid dynamics using dace. In: Extended Abstract In The 53rd International Conference on Parallel Processing Workshops (ICPP Workshops '24) (2024)
4. Beckingsale, D.A.e.A.: RAJA: Portable performance for large-scale scientific applications. In: 2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc). pp. 71–81. IEEE (2019)
5. Ben-Nun, T., de Fine Licht, J., Ziogas, A.N., Schneider, T., Hoefler, T.: Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19 (2019)
6. Chalmers, N., Karakus, A., Austin, A.P., Swirydowicz, K., Warburton, T.: libParanumal: a performance portable high-order finite element library (2022). `https://doi.org/10.5281/zenodo.4004744`, `https://github.com/paranumal/libparanumal`, release 0.5.0
7. Fischer, P., Min, M., Rathnayake, T., Dutta, S., Kolev, T., Dobrev, V., Camier, J.S., Kronbichler, M., Warburton, T., Świrydowicz, K., et al.: Scalability of high-performance pde solvers. The International Journal of High Performance Computing Applications **34**(5), 562–586 (2020)
8. Jansson, N., Karp, M., Perez, A., Mukha, T., Ju, Y., Liu, J., Páll, S., Laure, E., Weinkauf, T., Schumacher, J., Schlatter, P., Markidis, S.: Exploring the ultimate regime of turbulent rayleigh – bénard convection through unprecedented spectral-element simulations. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '23, Association for Computing Machinery, New York, NY, USA (2023). `https://doi.org/10.1145/3581784.3627039`, `https://doi.org/10.1145/3581784.3627039`
9. Jansson, N., Karp, M., Podobas, A., Markidis, S., Schlatter, P.: Neko: A modern, portable, and scalable framework for high-fidelity computational fluid dynamics. Computers & Fluids p. 106243 (2024)

10. Karp, M., Massaro, D., Jansson, N., Hart, A., Wahlgren, J., Schlatter, P., Markidis, S.: Large-scale direct numerical simulations of turbulence using gpus and modern fortran. The International Journal of High Performance Computing Applications p. 10943420231158616 (2023)

11. Karp, M., Podobas, A., Jansson, N., Kenter, T., Plessl, C., Schlatter, P., Markidis, S.: High-performance spectral element methods on field-programmable gate arrays : Implementation, evaluation, and future projection. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1077–1086 (2021). https://doi.org/10.1109/IPDPS49936.2021.00116

12. Karp, M., Suarez, E., Meinke, J.H., Andersson, M.I., Schlatter, P., Markidis, S., Jansson, N.: Experience and analysis of scalable high-fidelity computational fluid dynamics on modular supercomputing architectures. The international journal of high performance computing applications p. 10943420241303163 (2025)

13. Pennati, L., Andersson, M.I., Steiniger, K., Widera, R., Narwal, T., Bussmann, M., Markidis, S.: A parallel and highly-portable hpc poisson solver: Preconditioned bi-cgstab with alpaka (2025), https://arxiv.org/abs/2503.08935

14. Schaad, P., Schneider, T., Ben-Nun, T., Calotoiu, A., Ziogas, A.N., Hoefler, T.: Fuzzyflow: Leveraging dataflow to find and squash program optimization bugs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3581784.3613214, https://doi.org/10.1145/3581784.3613214

15. Schieffer, G., Wahlgren, J., Ren, J., Faj, J., Peng, I.: Harnessing integrated cpu-gpu system memory for hpc: a first look into grace hopper. In: Proceedings of the 53rd International Conference on Parallel Processing. p. 199 – 209. ICPP '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3673038.3673110, https://doi.org/10.1145/3673038.3673110

16. Svedin, M., Chien, S.W.D., Chikafa, G., Jansson, N., Podobas, A.: Benchmarking the nvidia gpu lineage: From early k80 to modern a100 with asynchronous memory transfers. In: Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies. HEART '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468044.3468053, https://doi.org/10.1145/3468044.3468053

17. Trott, C.R., et Al.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2021)

18. Vincent, J., Gong, J., Karp, M., Peplinski, A., Jansson, N., Podobas, A., Jocksch, A., Yao, J., Hussain, F., Markidis, S., Karlsson, M., Pleiter, D., Laure, E., Schlatter,

P.: Strong scaling of openacc enabled nek5000 on several gpu based hpc systems. In: International Conference on High Performance Computing in Asia-Pacific Region. p. 94 – 102. HPCAsia '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3492805.3492818, https://doi.org/10.1145/3492805.3492818

19. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka - an abstraction library for parallel kernel acceleration. IEEE Computer Society (May 2016)