arXiv:2512.17101v1 [cs.MS] 18 Dec 2025

MIRGE：一种面向科学计算的基于数组的计算框架

MATTHIAS DIENER, MATTHEW J. SMITH, MICHAEL T. CAMPBELL,KAUSHIK KULKA-RNI, MICHAEL J. ANDERSON, ANDREAS KLÖCKNER, WILLIAM GROPP, JONATHAN B. FREUND, 和 LUKE N. OLSON*,伊利诺伊大学厄巴纳-香槟分校，美国

MIRGE 是一种基于*NumPy* 类数组计算的科学计算方法，但它通过惰性求值将计算重新表示为数据流图，其中节点表示不可变的多维数组。数组表达式的求值会延迟到需要其值时才进行；此时会调用一个流水线，通过多阶段过程将高级数组表达式转换为较低级的中间表示（IR），最终生成可执行代码。领域特定变换（例如元数据驱动的优化、图形处理器并行化策略和循环融合技术）能够提升性能和内存效率。

MIRGE 采用“数组上下文”来抽象数组表达式与异构执行环境之间的接口（例如通过 OpenCL 进行惰性求值，或通过 *NumPy* 或 *CuPy*进行急切求值）。因此，该框架不仅实现了性能可移植性，还实现了应用逻辑、底层实现与优化之间的关注点分离。MIRGE 在提升科学表达能力的同时促进性能调优，为计算研究和科学应用开发提供了一个稳健且可扩展的平台。

本文概述了 MIRGE。我们进一步介绍了一个名为 *MIRGE-Com*的 MIRGE 应用，用于间断伽辽金有限元框架下的超音速燃烧流。我们展示了其作为求解器的能力，并重点介绍了它在大规模 GPU 硬件上的性能特征。

CCS 概念： • **计算数学 → 数学软件性能**； •**计算方法 → 大规模并行和高性能模拟**； •**软件及其工程 → 编译器**。

附加关键词和短语：基于数组、GPU、并行、数值偏微分方程、模拟

1 引言

科学模拟软件的设计通常需要在有助于理解的高层数学抽象与追求性能的底层实现细节之间做出艰难取舍。后者可能导致代码复杂，难以维护和移植。本文介绍 MIRGE（**M**ath–**I**ntermediate **R**epresentation–**G**eneration–**E**xecution，数学中间表示生成与执行），这是一种利用 Python 高层易用性，并面向现代及预期中的加速器架构实现性能可移植性的方法。

MIRGE 的核心理念是关注点分离：科学应用采用直观的数组表达式编写，这些表达式等价于 Python 中的 *NumPy* 表达式；同时，通过转换和优化来实现高性能。至关重要的是，这些转换针对特定应用（但包含可复用部分），并由用户通过数组上下文定制来驱动。这种设计有助于快速开发模拟，同时也不妨碍使用图形处理器并行化和循环融合等性能优化技术。

科学计算代码面临着诸多挑战，包括：(1) 既要便于领域专家阅读，也要便于开发人员阅读；(2) 需要在现代

*通讯作者。

作者联系方式：Matthias Diener，mdiener@illinois.edu；Matthew J. Smith，mjsmith6@illinois.edu；Michael T. Campbell，mtcampbe@illinois.edu；Kaushik Kulkarni，kgk2@illinois.edu；Michael J. Anderson，manders2@illinois.edu；Andreas Klöckner，andreask@illinois.edu，andreask@illinois.edu；William Gropp，wgropp@illinois.edu；Jonathan B. Freund，jbfreund@illinois.edu；Luke N. Olson，lukeo@illinois.edu，伊利诺伊大学厄巴纳-香槟分校，美国伊利诺伊州厄巴纳。

平台上实现（并保持）高性能；(3) 易于维护和扩展；(4) 通过建模识别性能提升机会与挑战。我们策略的一项原则是，物理表达式和模型表达式的清晰性不应受到实现细节、特定于硬件的考量或性能优化的影响。

MIRGE 是一种方法，构成了 *MIRGE-Com* 模拟库实现的基础；该库是一个用于高速燃烧的求解器。这种方法可以直接描述计算模型，使其与物理模型的数学形式高度相似，同时不预先规定为实现性能而采用的平台专属算法。

某种程度上，MIRGE 采用了与 *JAX* [15] 和 *torch.fx* [54]类似的方法。 *JAX*例如，使用其称为跟踪 的技术来记录函数的操作，然后重构一个可能更高效的函数调用，以便在设备上执行[1]。 *torch.fx* 专注于捕获 *PyTorch* 代码并进行面向设备执行的转换，但其明确聚焦于“深度学习程序的有向无环图表示”。许多代码都采用了类似的 图模式 执行方式；MIRGE 则专注于数组计算，因此也专注于面向偏微分方程的数值模拟。其原因之一见图1。左侧是一个三角形网格示例（也可以使用其他网格，例如三维四面体网格，或四边形网格和六面体网格；使用张量积基时，这些网格可能更具优势）。在节点间断伽辽金框架下 [31], , 自由度（DOFs）局部归属于单元（如中间图所示），因此可以自然地用数组格式表示（如右图所示）。因此，数值方法通常会批量处理数组中的数据，从而为提高整体计算效率提供途径。

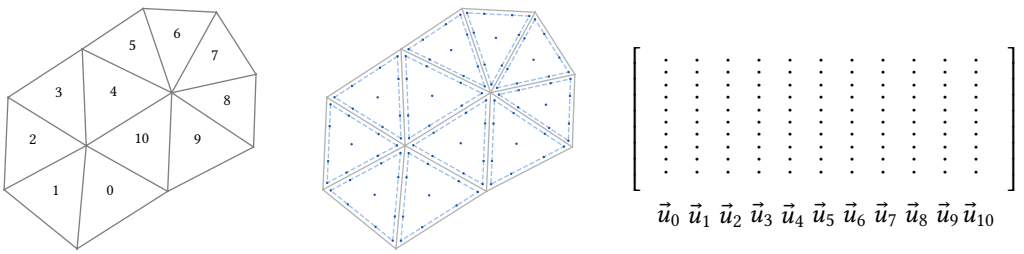


图1. 间断伽辽金布局。网格（左）中，每个网格单元都有局部节点自由度（中），由此形成基于数组的未知量布局（右）。

MIRGE 是众多方法之一，旨在让用户在 Python 中保持高效开发，同时在底层追求计算效率。例如，*CuPy*基于 Python，专注于在图形处理器上轻松执行类似*NumPy*的代码，既提供可直接替换*NumPy* 的实现，也提供直接编写内核的接口。同样，*PyOpenCL*和 *PyCUDA* [38] 可让用户通过 Python 直接访问图形处理器执行，但提供的即插即用功能较少。在这方面，MIRGE 与*CuPy*更为相似，但它的侧重点不同：MIRGE 致力于提供对中间表示的访问（部分由*loopy* [37] 包提供），以便进行代码变换。例如，*Hercules* [8],也采用了类似的、支持变换的方法，其重点是将代码变换为并行代码。Hercules 的方法着重于中间表示和变换，但有意避开 Python 等现有高级编程语言。相比之下，MIRGE 力求将这类优化工作变得更易于使用，方法是将其作为变换步骤融入基于 Python 的处理链中

[1]https://docs.jax.dev/en/latest/jit-compilation.html

。另一个区别在于，MIRGE 的设计不受特定执行目标（如 AMD、英伟达、CPU 等）的限制；可以使用不同的后端，从而在各种平台上执行。

下一节将深入探讨过去和现在的相关工作，对现有方法进行广泛分类，并介绍 MIRGE 在其中的定位。后续各节将概述 MIRGE，详细说明其针对通用基于数组计算的惰性评估和 *JAX* 风格评估，旨在实现总体目标：兼具 Python 级语义与低层级性能。第 3 节将对 MIRGE 进行高层次介绍，奠定基础并阐述更多动机。第 4 节将详述 MIRGE，深入探讨如何跟踪基于数组的代码、生成中间表示以便进一步优化，以及最终生成可在各种平台上执行的代码。第 5 节将介绍 *MIRGE-Com*，这是 MIRGE 在超音速燃烧流领域的一项应用。我们将提供基准应用的性能测量结果，与现有代码进行比较，并展示 *MIRGE-Com* 的实际应用。

2 相关工作

数组编程作为一种范式已得到广泛研究。早期系统（如 *Fortran-I* [10]）主要将多维数组引入为一种存储格式。艾弗森等人 [23] 设计了 *APL*，采用张量分析中的符号来表示 *n*维数组。*APL* 中的每个对象都是数组，且该语言支持重塑、秩查询以及内积或外积等高级例程。*APL* 的后继语言包括*J*、*K*、*MATLAB*、*NumPy* [30], 和*Julia* [14],；它们都推动了以数组为中心的计算在广泛科学应用中的普及。

有两个关键因素会影响数组编程系统在计算科学工作负载中的易用性。第一是系统的执行吞吐量，它决定了在给定计算时间内能够解决的问题规模。第二是系统的可调试性，它决定了计算科学家编写工作负载程序的速度。可调试性的提升使人们能够检查程序的中间状态，从而加快开发进程。这两项特性都直接取决于系统的执行模型。

诸如 *NumPy*、早期版本的 *Torch* [51],*CuPy* [50], *APL* 和 *ArrayFire* [46]等工具包采用急切求值 模型。在该模型中，每个操作一经遇到便立即执行，其结果会持久保存在内存中。这使程序能够进行渐进式求值 [28]，有助于快速原型开发，但由于操作数需要频繁往返内存，以及在图形处理器环境中反复进行内核启动，也会产生性能开销。另一种选择是延迟求值编程模型，在该模型中，程序会被构建为计算图，随后由编译器进行降低，并通过运行时层执行。这些框架通过内核融合以及使用经过调优的微内核库（如基础线性代数子程序库 [17], *cuDNN* [20]）等技术，解决了此前所述的大部分性能限制。此类系统的示例包括：*TensorFlow* [1], *JAX* [25], *Julia* [14], *torch.fx* [54], *Theano* [57], 以及*Legate* [13]。运行时层通过跟踪 [25], 或遍历抽象语法树（AST） [43]，或将两者结合起来 [58]，构建与用户程序相对应的中间表示。随后，优化编译器会利用与用户程序相对应的中间表示。

从设计上看，与急切求值相比，延迟求值更适合进行复杂的优化，因为编译器能获得关于中间数组操作使用者的额外信息。这些信息对于许多编译器技术至关重要，包括内核融合 [39],数组收缩 [26], 和数据布局变换。将数组图转换为能够实现接近屋顶线性能的计算内核颇具挑战性，主要是因为

庞大的变换空间，其中包括执行网格大小的选择、物化决策、地址空间的选择等。先前的研究表明，降级过程中的许多子问题都是 NP难问题。例如，Rosenkrantz等人 [56] 证明，确定哪些数组需要物化是 NP完全问题；Lam等人 [42] 证明，为爱因斯坦求和选择最优收缩路径是 NP难问题；Kennedy等人 [35]则表明，为最大限度地提高数据重用而进行循环融合同样是 NP难问题。因此，许多数组框架采取务实的做法，将编译器优化的范围限制在适用于特定领域的优化上。这一趋势在 *TensorFlow*、*JAX*和 *torch.fx*等框架中显而易见。这些框架主要用于机器学习应用，因为其内置编译器优化的效益并不能迁移到其他领域。

多维数组是科学计算各个领域中广泛使用的一种基础数据结构。然而，据我们所知，目前没有任何先进的仿真引擎原生使用上述任何一种数组框架。具有代表性的例子包括 *Nek5000* [49, 24], ，它是一款基于*Fortran*的有限元方法（FEM）求解器，提供用于定义弱形式的有限元专用高级例程。Abdelfattahetal. [2] 进一步针对高性能架构对其进行了优化。*MFEM* [7] 是一款 C++ 有限元库，它集成了来自*libCEED* [16] 的优化组装内核，并利用 *PETSc* [11]中的迭代求解器。统一形式语言（UFL） [3] 为 Ciarlet框架 [21], 中的弱形式提供了高级抽象，并作为端到端求解器的前端，例如*FEniCS* [4], *Firedrake* [53],和*Dune* [12]。这些系统通过有限元专用优化，将用户定义的变分形式编译为高效内核 [36, 33, 41]。

尽管取得了这些进展，但由于相对于 Roofline 而言性能显著偏低，将用户定义的数组表达式集成到此类求解器中仍然不切实际。此前的工作已部分解决了这些问题。Hirata [32] 提出了张量收缩引擎，为电子结构计算提供了高级操作，并允许用户使用 einsum表示法指定属于张量收缩操作子集的计算。类似地，*Comet* [48] 允许使用*NumPy*的一个子集来表达计算化学程序，通过*MLIR* [44] 的张量代数方言降低计算，并应用转置-转置-GEMM-转置 [32]等优化。

在本文中，我们提出了一个用于科学仿真开发的框架，并将其应用于一个间断伽辽金（DG）有限元求解器，其中数组是用户唯一可见的数据类型。用户可以完全通过高级数组操作来表达计算，这些操作涵盖*NumPy*的一个纯子集，即一组不允许原地修改的操作。我们通过*Pytato*/*Loopy* 中间表示降低生成的数组图，以解决以往工作中观察到的一项关键局限：无法注入领域专用代码变换。

3 MIRGE 方法

图 2 展示了 MIRGE 方法：从领域专用数学（左侧），到经过转换、针对主机的计算内核（中间），再到在目标平台（图形处理器或 CPU）上进行代码生成和执行。图中以灰色虚线（顶部）表示的非优化急切执行路径，是预期 惰性 执行处理路径之外的一种回退方案。这种方法在预测科学领域具有创新性，通过连接性能研究与科学模拟的双向通道，既能访问大规模预测科学应用，也能将性能优势直接带给应用。

我们采用一种使用惰性执行的流水线，以支持代码变换。该流水线依赖于面向用户的类似*NumPy*的数组构造（即数组上下文）——它在内部以符号方式捕获计算，将执行推迟，从而支持性能优化，并适配不同的执行环境，例如图形处理器。在此框架中，数组被视为不可变

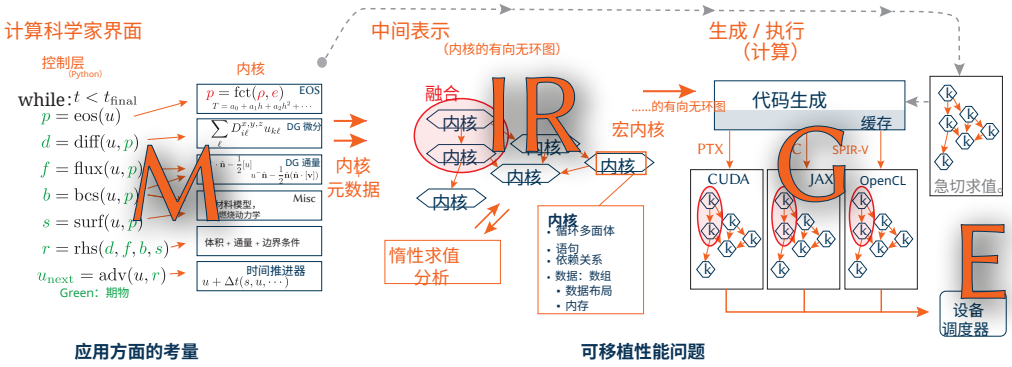


图 2. TheMIR GE 科学计算方法。（左）领域专用数学，（中）优化后的 ，特定于主机的内核，以及（右）特定平台代码的生成与执行。

表示计算基本构建块的对象。通过以这种方式评估程序并生成数组数据流图（ADFG），即可全面了解用户意图中的计算，并可应用诸如融合和并行化等变换。图 3 的左侧展示了面向用户部分的一个示例。在这里，3 中的数组操作 `axpy_max0()` 以符号方式进行评估，并根据生成的 ADFG 生成代码，随后执行该代码。

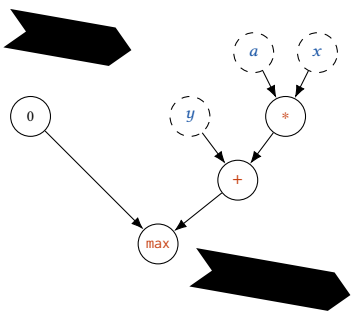
```
导入NumPy 作为np
从arraycontext 导入ArrayContext

actx = ArrayContext()

定义axpy_max0(x, y,    a):
    w = actx.np.multiply(x, a)
    w = actx.np.add(w, y)
    w = actx.np.maximum(w,    0)
    返回 w

x = np.linspace(0,    1, 10)
y = np.linspace(0,    1, 10)
a = 0.5

axpy_max0_compiled = actx.编译(
        axpy_max0)
axpy_max0_compiled(x, y,    a)
```




```
// OpenCL 内核版本

__内核void axpy_max0(
    __global const float *x,
    __global const float *y,
    常量浮点数 a,
    __global浮点数 *w,
    常量整数 n)
{

    int i = get_global_id(0);

    如果(i  < n) {
        浮点数 t  = a *x[i] +y[i];
        w[i] = t  > 0.0f ? t : 0.0f;
    }

}
```

图 3. MIRGE 方法的第 1 阶段和第 2–5 阶段：对两个数组 x 和 y（带有标量 a）执行示例操作。$\max(0,\ y + ax)$。（左）采用 MIRGE 方法的类*NumPy*应用程序代码。（中）数组数据流图（ADFG），其中节点表示数组值表达式，边表示它们之间的依赖关系。（右）经过前述转换后生成的 OpenCL 代码。

模拟是如何展开的？在多个阶段中，收集计算步骤，将其转换为目标语言（例如 OpenCL），然后执行：

阶段1：捕获数组数据流图 (ADFG) 流水线首先将计算的数学步骤表示为数组数据流图 (ADFG)，其中每个节点代表一个数组或数组操作。计算过程以类似 *NumPy* 的数组程序表示，并通过符号跟踪来构建 ADFG。跟踪会持续进行，直到执行到达用户定义的停止点（通过 actx.freeze()/ actx.compile()），此时

代码生成流水线的其余部分便会启动。图3 展示了中间的 ADFG；节点代表数组值表达式，边则代表它们之间的依赖关系。

阶段 2：转换 ADFG 接下来，对 ADFG 应用领域特定变换，例如代数化简和初步确定是否物化，也就是说，判断给定的中间值是否存储在内存中。此过程很大程度上由稀疏的描述性元数据驱动，这些元数据由用户代码或库代码在阶段1附加到 ADFG 节点上。

第 3 阶段：转换为标量中间表示（IR） 接下来，将经过变换的 ADFG 转换为标量（循环级）中间表示（IR），其中引入了循环结构和内存分配。在下一阶段，元数据也可能传播到 IR 中，例如作为循环变量的元数据。ADFG 表达的是应计算什么，而 IR 表达的是应如何 计算，包括执行顺序和内存顺序。

阶段4：转换标量中间表示 基于此 IR，应用并行化和内存访问变换。例如，通常将循环划分为块/线程结构会更有优势。此外，从先前阶段传播而来的元数据可用于确定要融合哪些循环，以及如何将数组轴映射到并行硬件上。本阶段涵盖了大量性能调优工作，也是与外部工具连接的契机。

阶段 5：生成目标代码 经过优化变换后，使用中间表示生成目标代码。我们的示例使用 OpenCL，不过*JAX* 或 CUDA 也是可选方案。此阶段还会将更高级别的构造转换为高效的低级代码，并可能依赖 OpenCL 专用代码来实现内存访问和并行执行。

阶段 6：程序执行 在最后阶段，已编译内核会在目标硬件上执行。数组上下文负责管理内存分配、OpenCL 执行上下文以及内核的启动。由于前面的阶段已经对程序进行了优化和编译，因此运行时开销极低。

我们在变换链中使用两种不同的中间表示，以分别关注不同方面：ADFG 在很大程度上并不表示“空间”（内存）和“时间”（求值顺序与并行化），因此在其层面应用的变换大多属于代数性质。与此同时，依赖这些具体特征的优化则在标量/循环级 IR 层面进行。

4 计算的阶段：更细致的要点

上一节概述了 MIRGE 的整体方法。接下来，我们将探讨具体细节，重点介绍实现通用性和大规模性能所必需的要素。

如前所述，MIRGE 中的数组操作通过类似 *NumPy* 的 数组上下文来表达。从名义上说，数组上下文负责控制用户提供的数组程序的执行。更具体地说，这包括选择执行环境，以及控制程序变换以实现高性能。为此，所提供的数组编程接口侧重于支持惰性求值，但也提供其他执行环境，参见表 1。

为使读者了解变换链各个阶段所执行处理的复杂程度，图 4 展示了一个示例问题在各阶段所耗时间的占比（这是一个可压缩多组分流动案例，与第 5.4 节介绍的案例类似，但网格较小，约有 20 万个自由度）。在我们的应用中，处理

表 1. 急切求值和惰性求值数组上下文概览。本文详细介绍了**基于惰性 PyOpenCL 的**数组上下文。

急切求值数组上下文	惰性求值数组上下文
基于 PyOpenCL 的急切求值	**基于 PyOpenCL 的惰性求值**
JAX-急切求值	*JAX*-惰性求值
NumPy	
CuPy	

时间几乎不受数组数据规模的影响。在分布式环境中，数据在处理器间的布局会产生一定影响，但这种影响有上限：它会随着相邻处理器数量的增加而增大，并最终趋于饱和。相反，扩展性与应用程序代码中指定的工作量及其性质有着复杂的关系。第 1 和第 2 阶段通常随 ADFG 中节点的数量扩展，而后续阶段则会引入额外的扩展依赖因素，例如生成的语句数量及其相互关系。实际上，图中所示的情况在定性上能代表我们在大多数问题中观察到的现象。因此，我们管理处理时间的主要方法，是采用各种策略尽量缩小初始 ADFG 的规模（示例见第 4.1.3 节）。

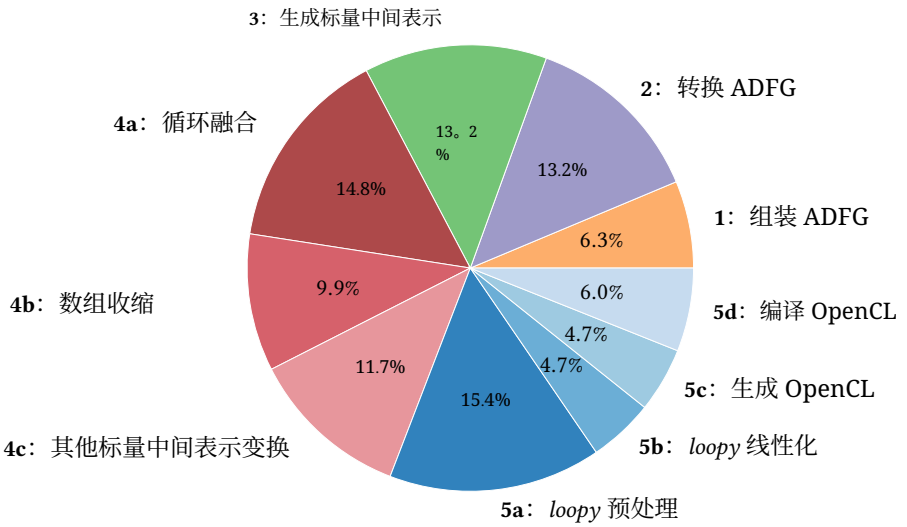


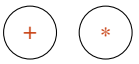
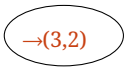
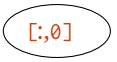
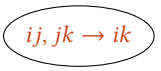
图4. 一个示例案例中执行前阶段耗时的细分。

4.1 阶段1：捕获数组数据流图（ADFG）

编译流水线的第一阶段从捕获计算的符号表示（即 ADFG）开始。系统使用多种节点类型来表示操作；表2总结了其中一些最常见的类型。

这种数组级中间表示的设计有意加以约束：它具有足够的表达能力，能够支持融合等转换，同时又受到足够的限制，以控制实现此类转换的

表2。ADFG 中常见节点类型示例。

节点类型	示例
Data 常量数值数据数组	
占位符 尚未提供的输入数组	
索引λ函数 逐元素代数运算	
重塑 逻辑数组形状的变化	
索引 数组切片和间接索引	
爱因斯坦求和运算 类似爱因斯坦求和的运算	
连接 串接、堆叠等。	
函数 “概述式”函数定义和调用	
通信 分布式发送与接收	

复杂度。与 LLVM IR 或 MLIR 等其他中间表示不同，MIRGE 的 ADFG 在引用节点时不使用间接引用（例如名称字符串），而是直接使用对象引用。这意味着修改 ADFG 的代价更高，因为每次转换都需要进行图遍历；但另一方面，它无需维护全局状态或名称映射。结合数组节点数据结构的不可变性，这一设计选择简化了图之间相等性的概念，也更容易找到冗余计算。

4.1.1 编译与冻结。 由于对惰性数组进行操作只会向 ADFG 追加节点，因此需要进行干预以触发代码生成和执行。针对这一问题的早期方法（例如 [40]）往往会隐式触发代码生成，例如在用户获取数组条目时触发。不幸的是，这样做会带来高昂且重复的成本，原因在于需要构建和处理 ADFG，至少要处理到可以从缓存中获取现有代码为止。在 MIRGE 中，我们因此采用了更明确的方法，为用户提供两个接口：如果代码只需调用一次，用户可以计算其数组代码，并将其传递给 `actx.freeze()`，以触发流水线的其余部分，生成并最终执行代码。如果代码需要执行多次，用户则可以将一个函数传递给 `actx.compile()`；该函数会使用占位输入对其进行计算，触发流水线的其余部分，并返回一个可用于重复执行的对象。这两种方式的示例见图 5。

需要注意的是，compile() 本身的实现也是惰性的。在调用 compile() 时，尚不知道将传递给已编译函数的参数；为了避免要求用户预先指定参数元数据，compile()

```
一个 =actx.from_numpy(np.random.rand(5,2))

a_sum_sq  =actx.np.求和(a**2)      # ADFG
a_sum_sq_frozen  = actx.冻结(a_sum_sq)          # 设备数组
a_sum_sq_np  = actx.to_numpy(a_sum_sq_frozen)        # NumPy 数组
```

（a）使用 freeze()。

```
def sum_sq(x):
    返回 actx.np.sum(x**2)

sum_sq_compiled = actx.编译(sum_sq)

a  = actx.from_numpy(np.random.rand(5,2))

a_sum_sq  = sum_sq_compiled(a)        # 设备数组
a_sum_sq_np = actx.to_numpy(a_sum_sq)        # NumPy 数组
```

（b）使用 compile()。

图5. 使用惰性求值数组时触发代码生成/执行的方法。

会将大部分处理推迟到首次调用已编译函数时进行。这样，同一个已编译函数也可以使用不同的可选参数配置进行调用；每种情况都会生成单独的代码，并在适用时复用代码。

4.1.2 分布式计算。 在 MIRGE 中，基于数组上下文的计算可以分布到多个计算单元上。因此，必须考虑此类环境中惰性求值所产生的 ADFG 的表示方式。一种方法与计算的单控制流 模型（PGAS、分布式数组）类似，即构建一个包含整个计算过程的共享全局数组数据流图。在小规模情况下，这种朴素的实现尚可行，但当常量数组数据（例如有限元网格中的连接性数据）过大，无法装入单个计算单元的内存时，便会失效。更复杂的变体或许能通过将常量数据与节点分开存储来避免这类问题，但在对 ADFG 应用变换时，这会增加所需的簿记工作量。因此，我们采用多控制流的方法，为每个计算单元分别构建局部数组数据流图，并在其中嵌入进程间通信的节点表示。从用户的角度来看，这相当于编写类似典型 MPI 代码的应用程序；但与直接调用 MPI 不同，用户会调用专用例程来创建发送节点和接收节点，并将其插入 ADFG。

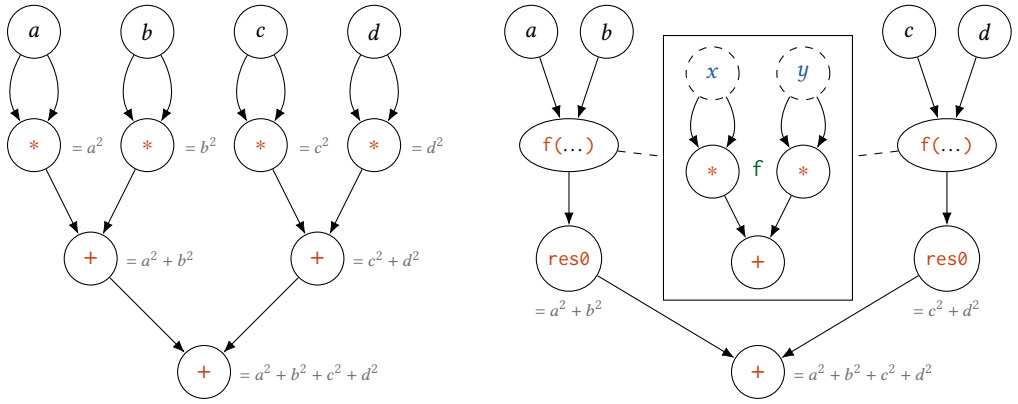
这种方法面临的一个挑战是，从数据流的角度来看，发送操作并非数组；它们是没有输出的“死端”节点，因此不适合直接表示在 ADFG 中。目前，MIRGE 基础设施通过引入一种节点类型来规避这一问题，该节点类型：(1) 充当 ADFG 中现有数组的直通节点；(2) 持有对待发送数据的引用（该数据由 ADFG 节点表示）。虽然这种方法可行，但也有缺点。具体而言，必须确保包含发送操作的子表达式作为依赖项保留在 ADFG 中（换言之，其结果必须用于整体计算）；否则，可能会导致通信不匹配（即，出现只有接收而没有发送，或只有发送而没有接收的情况）。目前正在研究其他方法，例如在用户 API 中显式跟踪发送操作。

4.1.3 ADFG 函数。由于数据流图是通过对数组代码进行符号执行生成的，因此函数调用不会自动显式表示出来。相反，每次函数调用都会为其操作创建一个新的子图，这类似于将每次调用的代码内联。因此，ADFG 的规模可能会增大，导致处理时间增加。因此，提供一种有选择地对函数进行“外提”的方式将很有帮助，即阻止函数调用被内联到 ADFG 中。

当函数被提取出来时，不会直接返回表示函数操作的内联节点，而是返回一个表示函数调用的节点。该节点包含函数节点的表示形式，并通过使用占位符将其与 ADFG 的其余部分隔离开来，用于表示

函数参数。相应的具体参数也单独存储在调用节点中。示例如图6所示。

在 ADFG 中显式表示函数调用，可以让下游代码生成流水线将其处理为内核函数；在某些情况下，还可以通过 ADFG 转换对其进行优化。示例见第4.2.4节。



(a) 内联　　(b) 提取为函数

图 6. 对包含两个函数调用的 ADFG 进行函数外提的示例。该函数 `f` 计算输入值的平方和 x和 y。注意：函数可以有多个结果，因此在 ADFG 中，调用节点（`f(...)`）与结果节点（`res0`）分别表示。

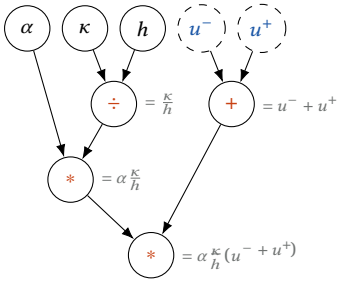
4.2 阶段 2：转换数组数据流图

初始数据流图构建完成后，会对该图进行解析和转换，为后续阶段做准备。这包括识别图中的数学简化、标记需要存储（或物化）的节点、组织图以支持分布式执行，以及为提高效率而合并某些函数调用。接下来将详细介绍这些概念；总体而言，它们构成了 ADFG 的“转换阶段”。

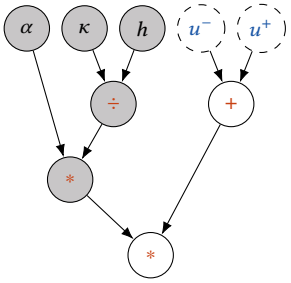
4.2.1 应用数学简化。 MIRGE中的转换工具用于对中间 ADFG 应用数学简化。其中一个示例是常量折叠：仅依赖常量数据的计算部分会提前求值，如图 7所示。

4.2.2 物化。计算创建的 ADFG 最初并不包含任何关于物化的概念，也就是说，图中的哪些数组需要显式存储在内存中。因此，由转换链负责做出这一决定。为了提高算术强度，图中被物化的数组所占比例应尽可能小。另一方面，物化不足可能导致大量重复工作，因为未物化的数组每次被引用时都会重新计算（示例见图8 ）。因此，物化算法需要在这两个极端之间取得平衡。

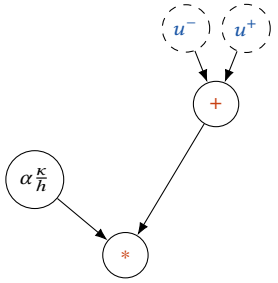
目前，我们采用一种基于简单启发式方法的贪心物化策略：物化那些拥有多个已物化前驱节点且拥有多个后继节点的节点。此外，由于归约（在我们的场景中通常是 einsum 节点）具有较高的算术强度，因此所有包含归约的节点都会被强制物化。



(a) 初始 ADFG。



(b) 第一轮处理后：可折叠部分已被识别-部分。



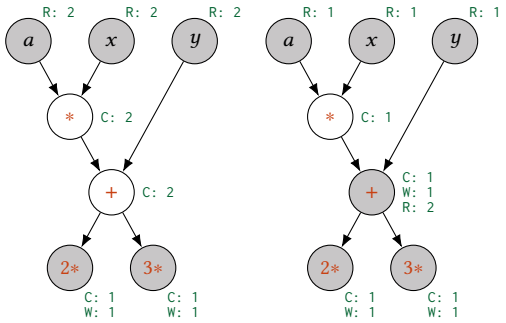
(c) 第二遍之后：包含新常量数据的 ADFG。

图 7. 对混合了常量数据（α、 κ和 h）与占位符（u^-、 u^+）的表达式进行常量折叠。

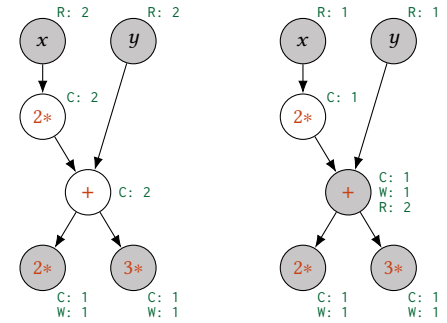
对于若干物理模型，我们发现这种简单方法效果良好：它易于应用，且能取得可接受的结果（将物化率限制在 5-15%，并将重复计算控制在 10% 以下）。然而，将这一策略应用于我们的一些更复杂的物理模型时，也暴露出了一些不足：（1）较小的数组，其元素在计算单个较大数组时会被反复使用，但由于较大数组被视为单个后继节点，导致不必要的重复计算；（2）简单的子表达式（已物化前驱节点少于两个）却拥有大量后继节点。在我们考察过的 ADFG 中，至少有一个因遗漏物化而导致重复计算率超过 75%。尽管如此，尝试扩展该启发式方法，以便在这些情况下直接物化，也未能奏效：计算时间没有变化，甚至略有增加。鉴于我们的应用大多受内存带宽限制，在这些情况下物化所减少的浮点运算，很可能会被增加的内存访问所抵消。图 8b展示了这种情况的一个示例。我们预计，未来对改进启发式方法的研究，尤其是那些与流水线后续阶段所应用的变换更紧密结合的方法，有望带来显著提升。

4.2.3 分布式划分。为便于进行分布式内存计算，会对包含通信节点的 ADFG 进行分析，以确定一个粗粒度通信图，其中每个节点表示一个通信操作（包括发送和接收），每条边表示进程局部数据流。各计算单元分别计算各自的段，然后将其发送到协调计算单元进行处理。该协调器会对通信操作进行拓扑排序，并采用贪心策略将其分组成批次（图 9a）。通信批次形成后，会将其分发回所有计算单元，随后将其视为待构建的节点局部计算子ADFG之间的边界。完整 ADFG 中的计算节点会插入各批次之间：(1) 在其所有前置接收操作完成之后；(2) 在需要其输出之前（图 9b）。

全局构建通信图可确保分区过程不会导致节点本地、逐批次的纯计算 ADFG 之间出现任何循环依赖。全局通信图的另一项优势是，它支持分析过程检查通信不匹配。尽管这种方案在渐近意义上确实无法扩展，但在实践中，对于相关问题规模，我们发现这一选择并未带来显著的运行时负担。

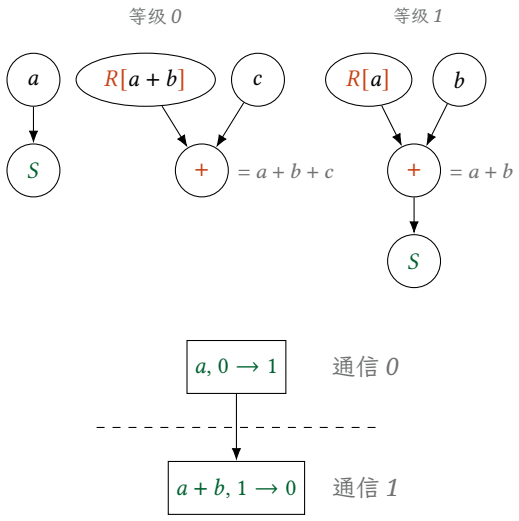


(a) 一种成功的物化选择（of $ax + y$），可在不增加内存访问的情况下减少计算量。

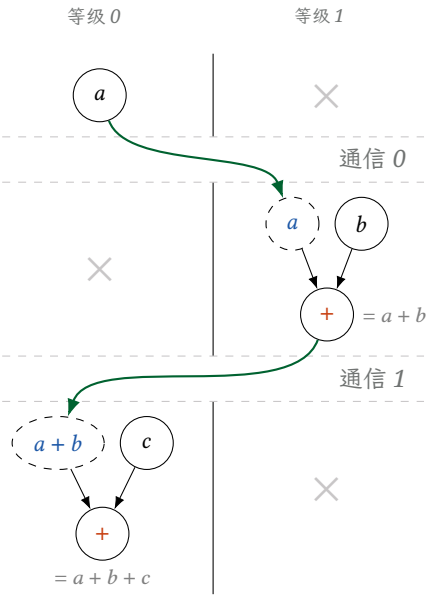


（b）一种效果较差的物化选择（of $2x + y$），它减少了计算量，却增加了内存访问次数。

图 8. 两个物化示例，展示其物化前后的成本，分别以数组读取（R）、写入（W）和计算（C）表示。未物化节点以白色显示；已物化节点以灰色显示。



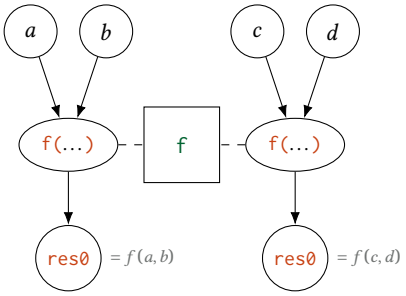
(a) 上：两个进程上的分布式 ADFG 示例。下：叠加了批次的全局通信图。



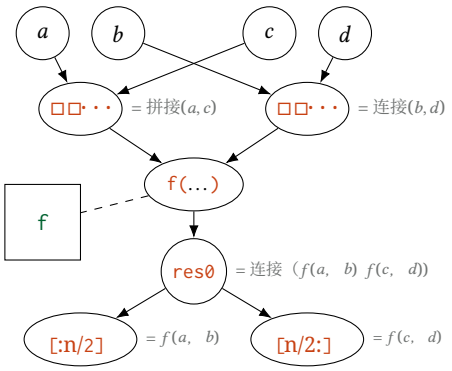
（b）分区后的 ADFG。原始 ADFG 的片段被插入通信批次之间（绿色），从其他进程接收的数据被视为输入（蓝色）。

图 9. 对分布式 ADFG 进行分区。注：对于这个双秩示例，发送操作的目标秩（S）和接收操作的源秩（$R[\ldots]$）都很简单，因此省略。

4.2.4 函数调用拼接。如果提取函数内部的操作（见第 4.1.3 节）足够“良好”（见下文），则可以沿适当的轴串接每个调用的输入、输出和函数体节点，将一组独立调用转换为单个调用，从而创建一个更大的函数调用（如图 10 所示）。这种转换通过减少内核调用次数并增大所操作数组的规模来降低计算成本，从而提高 GPU 利用率。



(a) 不进行串接



(b) 采用串接。 `f` 是原始 `f` 的修改版本，其数组形状经过调整，以适应串接后的输入。

图1 通过函数调用拼接转换 ADFG。为简单起见，此示例假设 a、 b、 c 和 d 是长度相同的一维数组 n。

“行为良好”的定义取决于许多具体细节。首先，所有对特定用户指定函数的调用都被视为可以进行串接。随后，将串接组拆分，以确保所有拼接调用彼此独立。最后，轴匹配过程会确定可以进行串接的轴，同时保持其余数组维度不变。我们探索了多种用于这种匹配的不同算法，它们对轴位置变化的容忍度各不相同。最终，一种相对简单的算法证明已足够：它根据节点输入端的轴与输出端的轴之间的连接关系进行匹配。

4.3 第 3 阶段：从数组级中间表示到标量 ScalarIR

ADFG 经转换和优化后，下一阶段会将图重写为一种更低层级的中间表示，通常以*Loopy*编程模型的形式表示（详情请参见 [37] ）。这一阶段会引入循环结构、临时变量和控制流，并首次明确执行顺序和内存使用。ADFG 处理的是整数组表达式和符号图，而*Loopy* 模型则基于标量多面体计算模型运行，其中计算按循环域索引，内存布局得到精确定义，执行顺序则可通过依赖关系和标签加以控制——示例见图11。这一降级阶段通过将每个节点转换为数组推导式形式来实现，其中每个数组元素的公式由其已物化前驱节点的元素表示。生成这些数组推导式后，即可轻松将它们转换为赋值语句，并将每条语句分别封装在一个循环嵌套中。

```
knl = lp.make_内核(
    "{ [i,j,k]: 0 <= i < 3 和 0 <=j  < 3和 0 <=k <n_点积 }",
    """
    点积 = sum(k,   w[k] * x[k])
    结果[i * 3 + j] =x[idx[i,j]] + 点积
    """
)
```

图11。一个 *Loopy* 内核示例。

4.4 阶段4：ScalarIR 转换

生成低层级中间表示后，还会应用其他转换，以利用并行化机会并优化存储层次结构。具体而言，循环融合对于减少上一阶段生成的、数量可能很大的独立循环至关重要。

为了执行循环融合，我们依赖附加在 ADFG 数组轴上的描述性元数据（稀疏分布）。在我们的场景中，轴元数据附加在不连续有限元基础设施中，而非应用级代码中。此元数据的主要用途是标识数组轴的类型，例如，某个轴是否由特定离散化的单元编号、物理维度，或特定局部单元类型的单元内自由度进行索引。随后，这些元数据会传播到 ADFG 中所有“相连”的轴，并以附加在循环上的元数据形式保留在标量中间表示中。

具有匹配元数据的循环将成为 融合候选项。对于每组候选项，文献中的贪心循环融合算法 [35], 经过少量修改后，会在考虑循环嵌套之间依赖关系性质的同时，选出一组合适的候选项。

接下来，我们执行数组收缩。例如，考虑图12中的循环嵌套。由于变量 `tmp` 周围的出向依赖关系是逐元素的，因此每次循环迭代中只需使用数组的一个元素，这使得数组引用 `tmp[i]` 可以替换为标量临时变量`tmp`。我们同样采用文献中的一种简单方法 [42]来实现这一转换。

需要注意的是，一旦主要通过循环融合发现相应机会的收益，这一步实际上会撤销流水线早期做出的一些物化决策（参见第 4.2.2 节）。在未来的工作中，我们希望采用一种更加集成化的方法，这很可能有助于降低变换成本。

```
tmp  = [无] * n
z = [None] * n
遍历 i 位于范围(n):
  tmp[i] =x[i] +y[i]
  z[i] = 2*tmp[i]
```

(a) 收缩前。

```
z = [None]* n
对于 i在范围内(n):
  tmp  =x[i] +y[i]
  z[i] = 2*tmp
```

(b) 收缩后。

图12. 使用 Python 伪代码说明数组收缩。数组收缩将中间数组 tmp 替换为标量临时变量，从而减少内存使用。

最后，我们利用传播得到的数组轴/循环变量元数据来确定并行化候选项，主要是将其映射到 GPU 编程模型提供的硬件并行轴。

4.5 阶段 5：生成目标代码/OpenCL

经过上一节所述的转换后，标量中间表示中的代码已达到与传统 OpenCL/CUDA 代码相同的抽象层次。在我们的方案中，会生成 OpenCL 代码，并通过各种供应商的实现进行即时编译，尤其是 *PoCL* [34]和 RoCM AMD GPU OpenCL ICD。图 3 展示了在简单场景下生成的代码示例。

这一生成过程大多较为直接，不过有若干方面使其在图4中约占 25%。影响因素包括：(1) 生成的代码量。如下文所述，典型工作负载会生成约 100 个图形处理器内核，代码量约为 10,000 行。(2) 值得注意的是，*loopy*的中间表示本身是无序的，因此需要执行排序遍历，并进行相应的有效性检查。（图中将其标为“线性化”。）(3) 由于 loopy 基于多面体模型，许多操作都涉及核心多面体原语。虽然单个原语的成本较低，但数量庞大，因而带来了相当大的开销。

对于英伟达设备，我们通常不采用英伟达的 OpenCL 厂商实现，而是倾向于使用 *PoCL*的 CUDA 目标。这有助于我们避免已观察到的各种性能限制，也能避开厂商实现中明显缺少性能分析功能的问题。*PoCL*的英伟达目标使用 LLVM 的 NVPTX 后端，生成英伟达专有的 PTX 中间语言。随后，GPU 驱动程序会将其即时编译为设备二进制文件，以供硬件使用。为提高这种方法的可行性，我们为*PoCL*的英伟达目标贡献了多项改进 [52]，尤其涉及暂存/共享内存的 32 位指针算术、统一/虚拟内存的使用，以及解决与暂存/共享内存相关的指针别名问题。

4.6 阶段 6：执行

对于单个计算单元，最终代码的执行十分直接：通过将 OpenCL 内核调用加入队列来完成，这些调用来自经过优化的生成式 Python 代码（在 *loopy* 中称为“调用器”）。我们采用 OpenCL，借助统一的运行时支持广泛的计算硬件；不过，实际的供应商实现种类繁多，包括 *PoCL*（CPU 和英伟达图形处理器）、AMD ROCm、NVIDIA CL、Intel CL 和 Apple CL。

4.6.1 分布式执行。 要执行分区后的计算图，执行引擎首先会针对所有通信批次中预期接收的数据，提交非阻塞的（MPI）接收操作。随后，它进入事件驱动循环，只要某个计算单元本地的 ADFG 部分满足其依赖条件（本地和远程），就会立即执行。图形处理器完成该 ADFG 部分的计算后，必要的发送操作也会随之以非阻塞的 MPI 方式提交。此过程持续进行，直至所有 ADFG 部分均执行完毕。资源管理通过对数组进行引用计数来实现；当剩余部分不再需要某个数组时，系统会自动回收其内存。

在进一步提升效率方面仍有空间，尤其是在重叠计算与通信方面。例如，数据就绪后并不会立即发送，而是要等当前部分结束后才发送。同样，只要仔细安排调度，很可能就能减少图形处理器的空闲时间。

4.6.2 内核性能分析支持。MIRGE 中的内核性能分析通过 OpenCL 内置的事件性能分析功能实现，包括使用专门的性能分析数组上下文，在计算过程中跟踪并测量内核的执行时间。性能分析通过存储与内核执行开始和结束时间相关的 OpenCL 事件来管理。目前，这

项统计以一次图形处理器内核启动为粒度进行。还需要进一步研究，以帮助用户将开销映射到 ADFG 的各个片段。

5 应用

该 MIRGE 方法的内部结构较为复杂，但这与其对外强调生产效率的目标相辅相成：我们的目标是在多种架构上提供经过优化的多物理场模拟，同时不让应用开发人员受硬件特性的束缚。本节将通过数值示例，展示 MIRGE 在实践中的应用。我们将通过*MIRGE-Com* [6], 这一由百亿亿次级超燃冲压发动机设计中心（CEESD）开发的模拟库 [18]来展示，该中心位于伊利诺伊大学厄巴纳-香槟分校。*MIRGE-Com* 是一个可移植、高度并行的 DG 库，能够模拟反应混合物黏性流和耦合材料响应，其主要目标是预测超燃冲压发动机燃烧室的内部行为。

我们考虑三个主要案例：
（1）无黏流中的等熵涡，（2）存在激波和边界层
的流动，以及（3）大规模超燃冲压发动机模拟。

5.1 *MIRGE-Com*

在*MIRGE-Com* 中，实现前几节所述各阶段的软件包栈包括*pytato*[2] 和*arraycontext*[3], ，它们有助于将代码描述为 ADFG；还包括*loopy*[4]，后者提供中间表示和转换工具。网格描述和 DG 组件借助*meshmode*[5], 、*modepy*[6],和*grudge*[7]实现。

对于跨多个处理单元的数据分发，*MIRGE-Com* 通过*mpi4py* Python 绑定使用基于主机的 MPI。借助轻量级安装工具*emirge* [19]实现了可移植性；该工具会安装一个 Conda 环境，其中包含实现MIRGE的组件，以及*MIRGE-Com* 接口。生成的环境支持在大多数系统上开箱即用地执行；更多详细信息请参见https://github.com/illinois-ceesd/mirgecom。

5.2 *MIRGE-Com* 与*MFEM*

在这些实验中，我们将*MIRGE-Com* 的运行时性能与开源有限元库*MFEM* [7]进行了比较。我们给出了两种代码的计时数据，但必须认识到，严格追求性能最大化并非唯一目标。这仅是一个总体性的比较，因为这两个应用在设计、实现、用户体验和可移植性方面存在显著差异。*MFEM* 是一个 C++ 库，针对 CPU 和图形处理器平台进行了调优。它可作为参照，用于广泛评估*MIRGE-Com* 相对于成熟软件包的效率。MIRGE 核心转换代码的性能分析已在其他地方报告（例如，参见 [37]）。

我们首先评估*MIRGE-Com* 在一个典型流动基准测试案例中的基准 CPU 性能：周期区域内无黏流中的等熵涡。该流动应用是一个打包好的*MFEM* 示例。在两种实现中，均使用张量积单元（TPE）上的 DG 离散来求解可压缩无黏流的欧拉方程，并通过显式四阶龙格–库塔方法推进

[2]https://github.com/inducer/pytato [3]
https://github.com/inducer/arraycontext [4]
https://github.com/inducer/loopy [5]
https://github.com/inducer/meshmode [6]
https://github.com/inducer/modepy [7]
https://github.com/inducer/grudge

时间。*MFEM*的无黏涡示例不直接支持图形处理器执行，因此本次比较是在 Apple M2 处理器上进行的。

值得注意的是，在图形处理器上运行 *MIRGE-Com* 也很容易，而图形处理器对于现代应用来说是更有吸引力的目标平台。为展示图形处理器上的执行情况，我们比较了 *MIRGE-Com* 与 *MFEM* 在周期区域中求解标量平流方程时的运行时性能。对于此示例应用，*MFEM* 通过部分组装 DG 算子支持图形处理器执行，大幅降低了内存占用，但代价是增加计算量。 *MIRGE-Com*则对所有 DG 算子进行完全组装，因此内存占用和流量都会增加。图 *13* 展示了 *MIRGE-Com* 和 MFEM在 CPU 和图形处理器上执行时的每个时间步耗时。对于较少的单元数，MIRGE 存在明显的性能下限，这与 *MIRGE-Com*中 OpenCL 内核调度层的开销有关；该层在 CPU 和图形处理器平台上均会使用。在 *MFEM*的图形处理器执行时间中，也能明显观察到性能下限。*MFEM* 使用 AMD 异构计算可移植性接口（ HIP ） API [5] 来实现图形处理器就绪内核，这些内核采用静态编译。

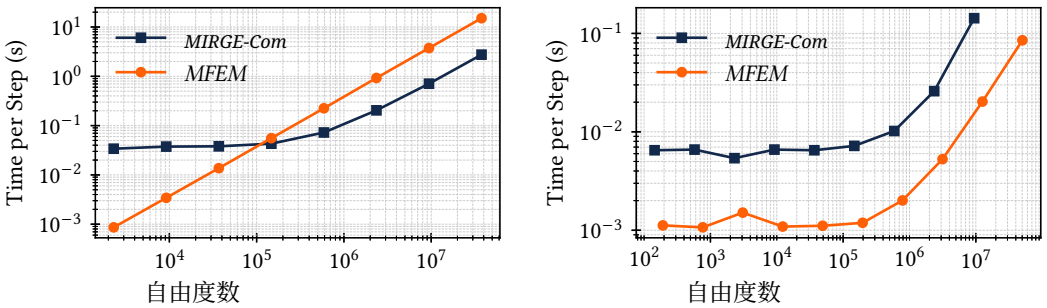


图1 3. MFEM 和 MIRGE-Com 的网格尺度扩展（*h*-细化）。左：在 Apple M2 处理器上模拟无黏涡流，使用 $p = 4$ 个 TPE，并采用固定步长的四阶龙格–库塔法（RK4）。右：在 AMD MI300A 上使用$p = 1$ 个 TPE 求解标量平流方程。

5.3 激波算例与*Ryujin*的比较

接下来，我们考察含激波的算例；为稳定流动，需要采用过积分、人工黏性和激波限制。这些更复杂的流动和离散化会产生相应复杂的 ADFG，适合用于测试 MIRGE 方法的稳健性。

作为参考，我们分别在*MIRGE-Com* 和*Ryujin* 软件包 [45, 29], 中开展实验；后者是面向可压缩流的 *deal.ii* [9]应用程序。

第一个例子是经典的无黏流绕过前向台阶的设置，其计算域如图 14所示。假设气体为完全气体（$\gamma = 1.4$），并在边界的顶部和底部施加绝热滑移条件。在流入处 Γ_{in}，设置马赫数为 $M = 3$ 的流入速度；超音速流出处 Γ_{out}不施加边界条件。速度以声速无量纲化后，初始流场的速度为$(u, v) = (3, 0)$。

该网格分别包含用于 *MIRGE-Com* 和*Ryujin* 的 ~1.6M 和 ~1M 个一阶TPE（每个标量对应的自由度总数分别为 ~6.6M 和 ~4.1M）。阶梯会迅速形成弓形激波，并逐渐扩展至与顶部边界发生反射，从而形成马赫杆和一系列反射激波。

图 14 展示了马赫杆形成后某一模拟时刻的密度梯度大小，并以其最大值进行归一化。MIRGE-Com 和以及*Ryujin* 均显示了台阶前方的强弓形激波，以及由顶部边界反射形成的马赫杆。

三重点以及顶部/底部壁面产生的激波反射也清晰可见。在三重点（即弓形激波与马赫杆的交点）脱落的涡旋存在细微差异。*MIRGE-Com* 和*Ryujin*产生的流场结果非常接近，差异可归因于底层离散化方法的不同，例如*Ryujin*采用连续伽辽金（CG）方法、使用了不同的限制器，以及采用了人工黏性。此外，*Ryujin* 求解可压缩欧拉方程，而*MIRGE-Com* 求解完整的可压缩纳维-斯托克斯方程。

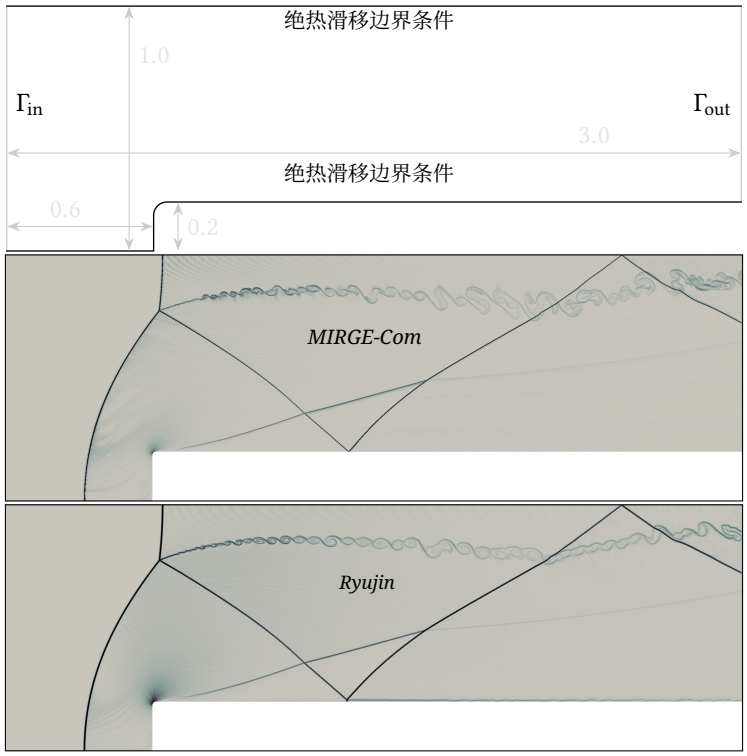


图14。流体绕过前向台阶的无黏流。顶部显示问题设置，中部显示 *MIRGE-Com* 的结果，底部显示 *Ryujin* 的结果，展示了模拟推进至较晚时刻时的密度梯度大小。

接下来，我们考虑激波与边界层的相互作用。计算域如图 15所示，并假设气体为完全气体（$\gamma = 1.4$）。流体初始处于静止状态，$(u,\ v) = (0,\ 0)$，密度存在不连续性，其中 $\rho_L = 120.0$ 位于 Ω_L，而 $\rho_R = 1.2$ 位于 Ω_R，压力则由等熵流动关系给出：$P = \rho/\gamma$。顶部和两侧施加绝热滑移条件，而计算域底部施加绝热无滑移条件。这些条件会产生一道从左向右传播、朝向反射壁运动的激波。当激波掠过底壁时，沿底壁形成边界层。激波从端壁反射后，会与边界层发生相互作用，并由于密度梯度与压力梯度未对齐而产生涡量。

该*MIRGE-Com* 网格包含 200 万个 TPE，采用 1 阶基函数（800 万个自由度），而 *Ryujin* 网格包含~2 万个单元（840 万个自由度）。图 15 展示了流体发展成熟后的某个模拟时刻，密度梯度的幅值（以最大值为基准进行缩放）。

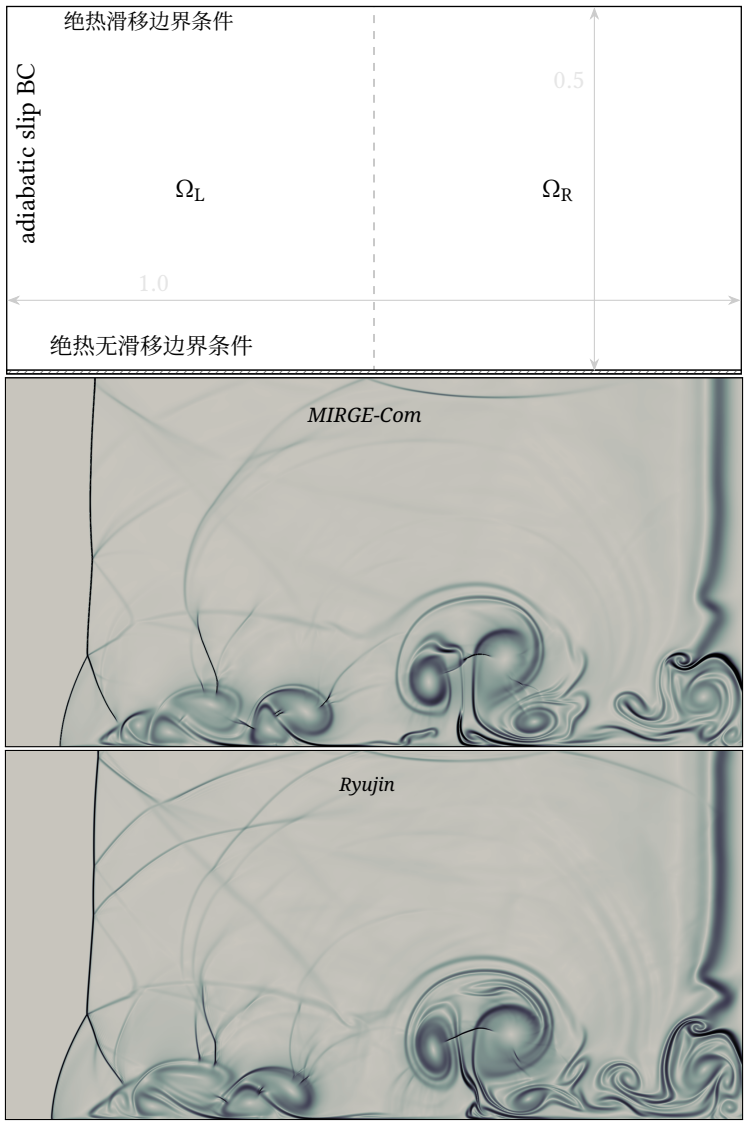


图15。激波与边界层的相互作用。顶部显示问题设置，中部显示 *MIRGE-Com* 的结果，底部显示 *Ryujin* 的结果；图中展示的是激波反射后、模拟时间较晚阶段的密度梯度大小。

由于采用了不同的模型和离散化，难以直接比较性能。尽管如此，原始数据仍有参考价值，因此我们报告了两种应用在 Apple M2 上的右端项（RHS）评估时间：*Ryujin* 的吞吐量约为每秒 380 万个自由度，而*MIRGE-Com* 的吞吐量约为每秒 160 万个自由度。*MIRGE-Com* 还得益于其可移植性和设备就绪性。例如，在 AMD MI300A 设备上运行这一基准测试时，*MIRGE-Com* 的 RHS 吞吐量为每秒 1.004 亿个自由度。

性能的自然对应指标是生产力，而*MIRGE-Com*的一项优势在于，得益于以数组为中心的设计以及 Python 的表现力，扩展和修改模型相对简单。衡量生产力的一项指标是代码总行数，例如 Ryujin 有 ~48.4k 行（不包括*deal.ii*），而*MIRGE-Com* 有 ~21.1k 行（不包括*pytato*等子包）。不过，归根结底，代码行数并不能充分体现用 Python 添加新功能的便捷程度，也无法体现面向设备的可移植性的价值。

5.4 复杂几何模拟

MIRGE-Com 被 CEESD 用于一项实际科学应用，以预测在伊利诺伊州的电弧加热燃烧风洞-II (ACTII)设施中开展的实验室规模超燃冲压发动机燃烧室实验的行为 [27, 55] 。ACTII 实验装置及其计算对应模型如图 16所示。在此模拟中，由 34M 个 TPE 构成的网格产生了超过 3B 个自由度，用于三维气体域模拟。计算和网格通过 Metis 分配到各处理单元 [47]。

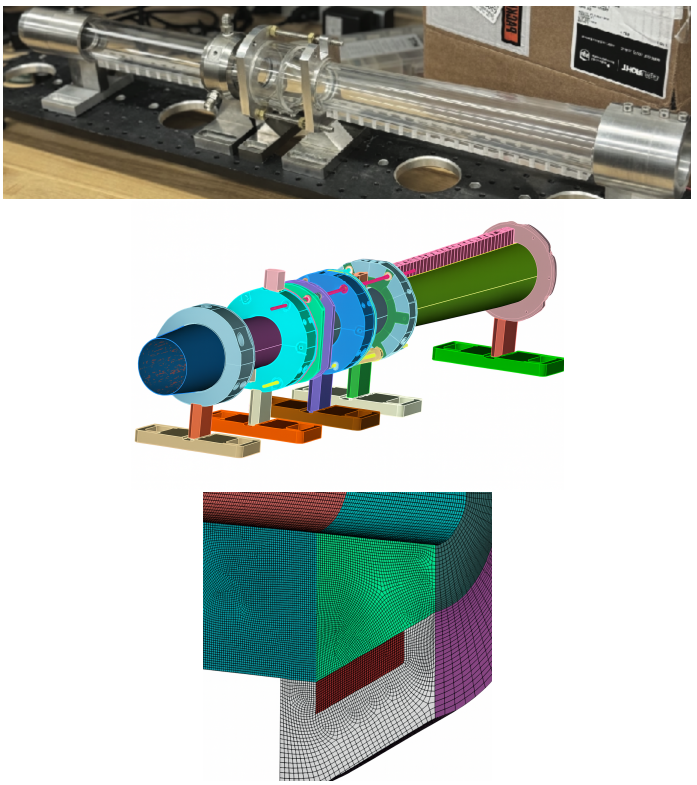


图16.（上）ACTII 超音速燃烧装置 [图片来源：Izzi Gessman，伊利诺伊州]（中）原始 CAD 模型 [CAD 模型鸣谢：Qili Liu、Izzi Gessman、Mike Anderson，伊利诺伊州]；（下）使用 Coreform 生成的计算网格局部区域 [22]。

模拟使用了多种计算平台（基于便利性和可用性）进行；许多范围探索模拟和能力开发则是在笔记本电脑上完成的（包括 Intel/Linux 和 Apple 设备）。这些开发和范围探索运行支持大规模模拟，此类模拟运行于美国能源部国家实验室的高性能计算资源上，尤其是劳伦斯利弗莫尔国家实验室的资源。

Tuolumne 系统配备了 AMD MI300A 加速器。综合模拟的快照如图 17 所示。

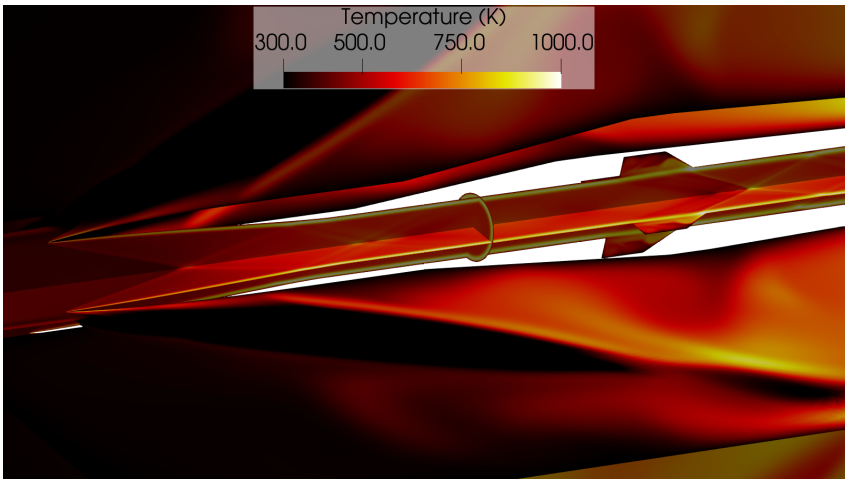


图17. 在劳伦斯利弗莫尔国家实验室（LLNL）的图奥勒米平台上，使用256个AMD MI300A 加速处理器进行的超燃冲压发动机燃烧室模拟片段。

模拟的并行可扩展性十分重要，因为这一趋势可用于估算给定模拟所需的计算资源以及预期求解时间（TTS）。在此，我们评估了劳伦斯利弗莫尔国家实验室（LLNL）图奥勒米平台上大规模超燃冲压发动机模拟的性能，并测量其弱扩展性和强扩展。

弱扩展性用于衡量问题规模和计算资源同时增大时的性能。图18 展示了三维区域中一维气体激波撞击气体-材料界面的弱扩展性结果，可为完整的超燃冲压发动机模拟提供基准。结果表明，*MIRGE-Com* 在这一领先级高性能计算平台上，扩展至整台机器规模时仍能实现近乎理想的扩展性。

强扩展用于衡量问题规模固定时，增加计算资源所带来的性能提升。本测试将模拟的网格规模固定为 ~3M个单元（即 ~10.26B个自由度），使每个进程的自由度从 ~80M降至 ~20.5M，同时将图形处理器数量从16个增至512个。图19 展示了强扩展结果。对于本次运行，我们观察到扩展至256个进程时仍具有良好的扩展性，这已足以在合理的周转时间内完成所需的模拟。

MIRGE 处理流程往往会模糊代码与内核的对应关系，因此很难将生成的内核与特定 DG 运算的预期性能联系起来。MIRGE 基础设施会在运行时动态转换应用程序代码，而这些转换取决于具体问题和配置。对于低阶 ($p \leq 2$) DG，我们预计工作主要由向量（BLAS-1）和矩阵-向量（BLAS-2） [17] 运算构成，从而生成算术强度较低、执行受内存带宽限制的内核。

为了评估 *MIRGE-Com* 的绝对性能，我们考察 [59]由 MIRGE 进程为正在运行的超燃冲压发动机模拟生成的、开销最高的内核（即应用程序耗时最多的内核） $p = 1$ 单纯形单元的 Roofline 性能。图 20展示了使用 AMD 屋顶线工具 *rocprof-compute*根据开销最高的十个内核构建的经验屋顶线模型；该工具通过经验评估算术强度，并

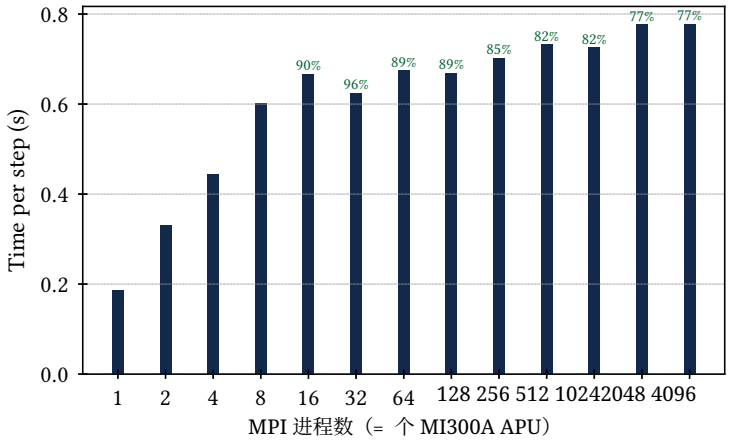


图18。Tuolumne（AMD MI300A）上超燃冲压发动机基准模拟的弱并行扩展。以百分比给出 n 个进程相对于 8 个进程的弱扩展并行效率（$100 \cdot t_8/t_n$）。

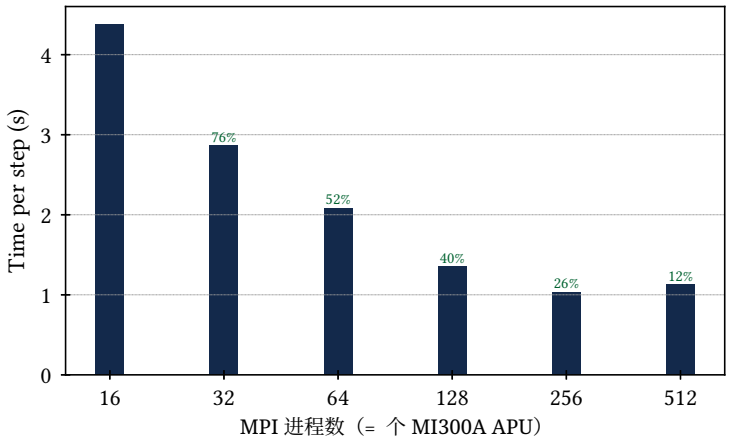


图19。Tuolumne（AMD MI300A）上超燃冲压发动机模拟的并行强扩展。以百分比给出 n 个进程相对于 16 个进程的强扩展并行效率（$100 \cdot 16 \cdot t_{16}/(n \cdot t_n)$）。

在运行时测量设备的运行带宽。屋顶线图表明，为该问题生成的内核运行性能接近主高带宽内存（高带宽内存）的预期带宽。超过高带宽内存性能的内核具有更高程度的数据重用（例如缓存一致性），而明显低于高带宽内存带宽的内核则表明，它们可能存在改进空间。

6 结论

面向基于数组的高性能计算的 MIRGE 方法建立在惰性求值 这一数组计算概念之上。通过将计算表示为数组数据流图（ADFG），它能够进行在纯急切执行模型中难以实现的变换与优化。这些图会被降级为标量中间表示，在其中可以进行传统的编译器式变换

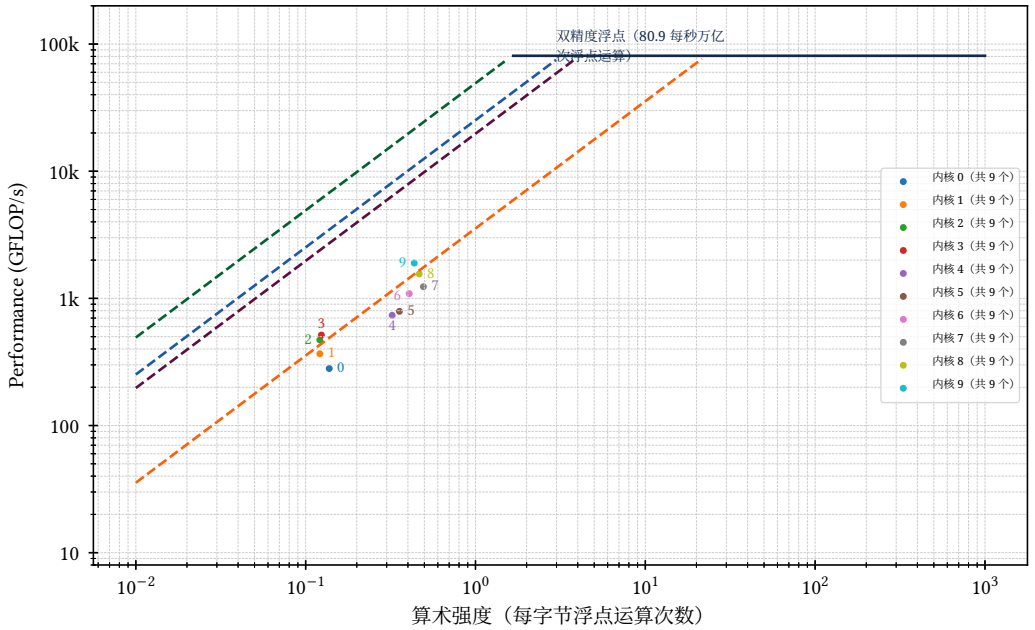


图 20. 在 Tuolumne（AMD MI300A）上运行超燃冲压发动机模拟时，前 10 个计算开销最高的 *MIRGE-Com* 内核的经验 Roofline，使用 $p = 1$ 单形。

例如循环融合。从此，系统会生成并执行针对特定目标的 OpenCL 内核。该框架还通过专用通信节点，将分布式执行直接集成到 ADFG 中，使大规模计算能够以统一的方式表达。模拟演示表明，该框架具有平台可移植性，性能与其他有限元流体模拟工具大致相当。

致谢

本材料部分基于能源部国家核安全管理局资助的工作，授予编号为 DE-NA0003963；同时也得到美国国家科学基金会资助，授予编号为 1931577。

参考文献

[1] Martín Abadi。2016。TensorFlow：大规模学习函数。载于第 *21* 届 *ACM SIGPLAN* 函数式编程国际会议论文集 (ICFP 2016)。计算机协会，奈良，日本，1。 ISBN：9781450342193。doi：10.1145/2951913.2976746。[2] Ahmad Abdelfattah等人。2021。用于高效百亿亿级离散化的 GPU 算法。 《并行计算》，108，102841。doi：10.1016/j.parco.2021.102841。 [3] Martin S. Alnæs、Anders Logg、Kristian B. Ølgaard、Marie E. Rognes 和 Garth N. Wells。2014。统一形式语言：一种用于偏微分方程弱形式的领域专用语言。 *ACM Trans. Math. Softw.*，40，2，文章 9，（2014 年 3 月），37 页。doi：10.1145/2566630。 [4] Martin S. Alnaes 等人。20151 00153 1011588201520553。《FEniCS 项目》版本 . .《数值软件档案》，。doi：。/ans。 。 。 。[5] [软件] AMD，HIP：异构计算可移植性接口 2025年。 网址： https://rocm.docs.amd.com/projects /HIP/en/latest/。

[6] [软件] Michael J. Anderson、Mike Campbell、Matthias Diener、Jonathan B. Freund、Kaushik Kulkarni、Andreas Klöckner、Luke N. Olson 和 Matthew J. Smith，MIRGE-Com：用于燃烧高速流动的数学中间表示-生成-执行 2025年。网址：https://github.com/illinois-ceesd/mirgecom。[7] Robert Anderson等人。2021。MFEM：一个模块化有限元方法库。《计算机与应用数学》，81，（2021 年 1 月），42–74。doi:10.1016/j.camwa.2020.06.009。[8] Russel Arbore、Aaron Councilman、Xavier Routh、Ryan Ziegler、Praneet Rathi 和 Vikram Adve。2025年。Hercules：一种用于异构系统高效编程的编译器。（2025年）。arXiv：2503.10855 [cs.PL]。doi:10.48550/ar Xiv.2503.10855。[9] Daniel Arndt等人。2021。deal.II 有限元库：设计、特性与见解。《计算机与应用数学》，81，（2021 年 1 月），407–422。doi:10.1016/j.camwa.2020.02.022。[10] John Backus。1978 9601 18 13, 8, 1978 165 180 10 1145 808380 。FORTRAN I、II 和 III 的历史。SIGPLAN Not.，（Aug. ），–。doi：。/。 。[11] Satish Balay等人。2025年。PETSc/TAO 用户手册。技术报告 ANL-21/39 -修订版 3.24。阿贡国家实验室。doi:10.2172/2998643。[12] Peter Bastian等人。2021。DUNE 框架：基本概念与最新进展。《计算机与应用数学》，81，75–112。doi:10.1016/j.camwa.2020.06.007。[13] Michael Bauer 和 Michael Garland。2019。Legate NumPy：加速的分布式数组计算。载于《国际高性能计算、网络、存储与分析会议论文集》（SC '19），文章 23。计算机协会，科罗拉多州丹佛，23 33561 75 9781450362290 10 1145 3295500 页。ISBN：。doi：。/。 。[14] Jeff Bezanson、Alan Edelman、Stefan Karpinski 和 Viral B Shah。2017。Julia：一种数值计算的新方法。《SIAM 综述》，59，1，65–98。doi:10.1137/141000671。[15] [软件] James Bradbury等人，JAX：Python 的可组合变换+NumPy 程序版本 0.3.13，2018年。网址：http://github.com/jax-ml/jax。[16] Jed Brown等人。2021。libCEED：用于高阶基于单元的离散化的快速代数。《开放源代码软件期刊》，6，63，2945。doi:10.21105/joss.02945。[17] C.L.Lawson、R.J.Hanson、D.R.Kincaid 和 F.T.Krogh。1979 5, 308 323, 3 10 1145 355841 355847 。适用于 Fortran 的基本线性代数子程序。–。doi：。/。 。[18] 百亿亿次级超燃冲压发动机设计中心（CEESD）。2025年。百亿亿次级超燃冲压发动机设计中心。https://ce esd.illinois.edu。[19] [软件] 百亿亿次级超燃冲压发动机设计中心（CEESD），emirge：MIRGE 和 MIRGE-Com 的可移植安装工具 2025年。网址：https://github.com/illinois-ceesd/emirge。[20] Sharan Chetlur、Cliff Woolley、Philippe Vandermersch、Jonathan Cohen、John Tran、Bryan Catanzaro 和 Evan Shelhamer。2014。cuDNN：用于深度学习的高效原语。（2014年）。arXiv：1410.0759 [cs.NE]。doi:10.48550/arXiv .1410.0759。[21] Philippe G. Ciarlet。2002。《椭圆问题的有限元方法》。应用数学经典丛书。第 40 卷。工业与应用数学学会，xxviii+530。ISBN：0-89871-514-8。doi:10.1137/1.9780898719208。[22] Coreform。2025年。Coreform：毫不妥协的模拟。https://coreform.com。[23] Adin D. Falkoff 和 Kenneth E. Iverson。1978。APL 的演进。SIGAPL APL Quote Quad，9，1，（1978 年 9 月），30–44。doi:10.1145/586040.586048。[24] Paul Fischer等人。2022 114, 102982 10 1016 2022 102982 。NekRS：一款由图形处理器加速的谱元纳维–斯托克斯求解器。《并行计算》，。doi：。/j.parco.。 。[25] Roy Frostig、Matthew James Johnson 和 Chris Leary。2018。通过高级追踪编译机器学习程序。载于《机器学习与系统会议论文集》。https://mlsys.org/Conferences/doc/2018/146.pdf。[26] G. Gao、R. Olsen、V. Sarkar 和 R. Thekkath。1993。用于数组收缩的集体循环融合。载于《并行计算语言与编译器》。Utpal Banerjee、David Gelernter、Alex Nicolau 和 David Padua（编）。施普林格·柏林·海德堡，柏林，海德堡，281–295。ISBN：978-3-540-48201-7。doi:10.1007/3-540-57502-2_53。[27] Isabella Gessman、Shruti Ghanekar、Gyu Sub Lee、Keunsoo Kim、Nozomu Kato、Jihyung Yoo 和 Tonghun Lee。2023。基于电弧加热器稳压腔热力学分析的超燃冲压发动机进气流表征。载于 AIAA SCITECH 2023 论坛。（2023 年 1 月）。doi:10.2514/6.2023-2332。[28] T.R.G. Green 和 M. Petre。1996。可视化编程环境的可用性分析："认知维度"框架。《可视化语言与计算期刊》，7，2，131–174。doi:10.1006/jvlc.1996.0009。[29] Jean-Luc Guermond、Martin Kronbichler、Matthias Maier、Bojan Popov 和 Ignacio Tomas。2022。关于一种用于可压缩纳维-斯托克斯方程的稳健高效基于有限元的并行求解器的实现。《计算机在应用力学与工程中的方法》，389，114250。doi:10.1016/j.cma.2021.114250。[30] Charles R. Harris等人。2020。使用 NumPy 进行数组编程。《自然》，585，7825，（2020 年 9 月），357–362。doi:10.1038/s 41586-020-2649-2。

[31] Jan S. Hesthaven 和 Tim Warburton。2007。《节点间断伽辽金方法：算法、分析与应用》。应用数学丛书，第 54 卷。Springer，xiv+500。ISBN：978-0-387-72065-4。doi:10.1007/978-0-387 -72067-8。[32] So Hirata。2003。《张量收缩引擎：组态相互作用、耦合簇和多体微扰理论的抽象与自动并行实现》。The Journal of Physical Chemistry A，107，46，9887 9897 10 1021 034596 – 。doi: . /jp z。[33] Miklós Homolya、Lawrence Mitchell、Fabio Luporini 和 David A. Ham。2018。TSFC：一种保结构形式编译器。SIAM Journal on Scientific Computing，40，3，C401–C428。doi:10.1137/17M1130642。[34] Pekka Jääskeläinen、Carlos Sánchez de La Lama、Erik Schnetter、Kalle Raiskila、Jarmo Takala 和 Heikki Berg。2015。pocl：一种性能可移植的 OpenCL 实现。International Journal of Parallel Programming，43，5，752–785。doi:10.1007/s10766-014-0320-y。[35] Ken Kennedy 和 Kathryn S. McKinley。1994。通过循环融合与分布最大化循环并行性并改善数据局部性。载于《并行计算语言与编译器》。Utpal Banerjee、David Gelernter、Alex Nicolau 和 David Padua（编）。施普林格·柏林·海德堡，柏林，海德堡，301–320。ISBN：978-3-540-48308-3。doi:10.1007/3-540-57659-2 18。_[36] Robert C. Kirby 和 Anders Logg。2006。一种变分形式编译器。ACM Trans. Math. Softw.，32，3，（2006 年 9 月），417–444。doi:10.1145/1163641.1163644。[37] Andreas Klöckner。2014。Loo.py：基于变换的图形处理器和 CPU 代码生成。载于 ACM SIGPLAN 数组编程国际研讨会论文集（ARRAY'14）。计算机协会，英国爱丁堡，82 262737 3 87 9781450329378 101145 2627387 – 。ISBN：。doi：。/ 。[38] Andreas Klöckner、Nicolas Pinto、Yunsup Lee、Bryan Catanzaro、Paul Ivanov 和 Ahmed Fasih。2012。PyCUDA 和 PyOpenCL：一种基于脚本的 GPU 运行时代码生成方法。《并行计算》，38，3，157–174。doi:https://doi.org/10.1016/j.parco.2011.09.001。[39] Mads R.B. Kristensen、Simon A.F. Lund、Troels Blum 和 James Avery。2016。并行数组操作的融合。载于《2016 年并行架构与编译国际会议论文集》（PACT '16）。计算机协会，以色列海法，71–85。ISBN：9781450341219。doi:10.1145/2967938.2967945。[40] Mads RB Kristensen、Simon AF Lund、Troels Blum、Kenneth Skovhede 和 Brian Vinter。2013。Bohrium：在 CPU、GPU 和集群上运行未经修改的 NumPy 代码。载于第四届 Python 高性能与科学计算研讨会（PyHPC'13）。[41] Kaushik Kulkarni 和 Andreas Klöckner。2025。基于符号变分形式，为图形处理器生成接近 Roofline 性能的有限元动作代码。（2025）。arXiv：2506。17471 [cs.DC]。doi:10.48550/arXiv.2506.17471。[42] Chi-Chung Lam、P. Sadayappan 和 Rephael Wenger。1997。一类嵌套循环的最优重排序与映射，以实现并行执行。载于《并行计算语言与编译器》。David Sehr、Utpal Banerjee、David Gelernter、Alex Nicolau 和 David Padua（编）。施普林格·柏林·海德堡，柏林，海德堡，315–329。ISBN：978-3-540-69128-0。doi:10.1007/BFb0017261。[43] Siu Kwan Lam、Antoine Pitrou 和 Stanley Seibert。2015。Numba：基于 LLVM 的 Python 即时编译器。载于第二届 LLVM 高性能计算编译器基础设施研讨会论文集（LLVM '15），文章 7。计算机协会，美国德克萨斯州奥斯汀，6 页。ISBN：9781450340052。doi:10.1145/2833157.2833162。[44] Chris Lattner 等。2021。MLIR：扩展领域特定计算的编译器基础设施。载于 2021 年 IEEE/ACM 代码生成与优化国际研讨会（CGO），2–14。doi:10.1109/CGO51591.2021.9370308。[45] Matthias Maier 和 Martin Kronbichler。2021。采用保持不变域的二阶有限元格式，高效并行计算可压缩欧拉方程的三维问题。ACM Trans. Parallel Comput.，8，3，文章 16，（2021 年 9 月），30 页。doi:10.1145/3470637。[46] James Malcolm、Pavan Yalamanchili、Chris McClanahan、Vishwanath Venugopalakrishnan、Krunal Patel 和 John Melonakos。2012。ArrayFire：一个 GPU 加速平台。载于《国防系统与应用建模和仿真 VII》。Eric J. Kelmelis（编），第 8403 卷。国际光学与光子学学会。SPIE，84030A。doi:10.1117/12.921122。[47] [SW], METIS：串行图划分与减少填充的矩阵排序，第 5.1.0 版。网址：https://github.com/Ka rypisLab/METIS。[48] Erdal Mutlu、Ruiqin Tian、Bin Ren、Sriram Krishnamoorthy、Roberto Gioiosa、Jacques Pienaar 和 Gokcen Kestor。2022。COMET：一种面向高性能计算化学的领域特定编译。载于《并行计算语言与编译器》。Barbara Chapman 和 José Moreira（编）。Springer International Publishing，Cham，87–103。ISBN：978-3-030-95953-1。doi:10.1007/978-3-030-95953-1_7。[49] [SW], Nek5000。阿贡国家实验室，数学与计算机科学部。网址：https://nek5000.m cs.anl.gov。

[50] Ryosuke Okuta、Yuya Unno、Daisuke Nishino、Shohei Hido 和 Crissman Loomis。2017。CuPy：一个兼容 NumPy、用于英伟达 GPU 计算的库。载于神经信息处理系统第三十一届年会（NIPS）机器学习系统研讨会（LearningSys）论文集。http://learningsys.org/nips17/assets /papers/paper_16.pdf。[51] Adam Paszke 等。2019。PyTorch：一种命令式风格的高性能深度学习库。载于《神经信息处理系统进展 32》。Curran Associates, Inc.，8024–8035。doi:10.48550/arXiv.1912.01703。[52] pocl。2020。1.6 版发行说明。（2020）。检索日期：2025-12-16，网址：https://portablecl.org/pocl-1.6.html。存档于：https://web.archive.org/web/20250513114526/https://portablecl.org/pocl-1.6.html。[53] Florian Rathgeber、David A. Ham、LawrenceMitchell、Michael Lange、Fabio Luporini、Andrew T. T. Mcrae、Gheorghe-Teodor Bercea、Graham R. Markall 和 Paul H. J. Kelly。2016。Firedrake：通过组合抽象实现有限元法自动化。ACM Trans. Math. Softw.，43，3，文章 24，（2016 年 12 月），27 页。doi:10.1145/2998441。[54] James Reed、Zachary DeVito、Horace He、Ansley Ussery 和 Jason Ansel。2022。torch.fx：用于 Python 深度学习的实用程序捕获与变换。载于《机器学习与系统论文集》。D. Marculescu、Y. Chi 和 C. Wu（编），第 4, 638 651 _ 2022 7 98 9 7 2 90 – 卷。911 23 9 72 6 https://proceedings.mlsys.org/paper files/paper/ /file/ c f c ab df da f ce ed e-Paper.pdf。[55] Samuel Richardson、Jack R. Edwards、Isabella Gessman、Jie Lim、Arthur Paganini 和 Tonghun Lee。2025。在电弧加热直连式超燃冲压发动机设施中进行富氧燃烧的数值模拟。载于 AIAA SCITECH 2025 论坛。（2025 年 1 月）。doi:10.2514/6.2025-2258。[56] Daniel J. Rosenkrantz、Lenore R. Mullin 和 Harry B. Hunt III。2006。关于最小化数组型临时变量的物化。ACM Trans. Program. Lang. Syst.，28，6，（2006 年 11 月），1145–1177。doi:10.1145/1186632.1186637。[57] Theano 开发团队等。2016。Theano：一种用于快速计算数学表达式的 Python 框架。（2016）。arXiv：1605。02688 [cs.SC]。doi:10.48550/arXiv.1605.02688。[58] [SW] Vijay Thakkar 等，CUTLASS 3.0.0 版，2023 年 1 月。网址：https://github.com/NVIDIA/cutlass。[59] Samuel Williams、Andrew Waterman 和 David Patterson。2009。Roofline：一种适用于多核架构、直观易懂的可视化性能模型。ACM 通讯，52，4，65–76。doi:10.1145/1498765.1498785。

收稿日期：X；修订日期：X；录用日期：X