格点QCD求解器中多个右端项的性能可移植优化与分析

Shiting Long[1], Gustavo Ramirez-Hidalgo[2],Stepan Nassyr[2],Jose Jimenez-Merchan[3],Andreas Frommer[3],Dirk Pleiter[1,4]1 瑞典皇家理工学院，斯德哥尔摩，瑞典2 于利希研究中心¨有限公司，于利希¨，德国3 伍珀塔尔大学，伍珀塔尔，德国4 格罗宁根大学，格罗宁根，荷兰电子邮箱： [1] shitingl@kth.se [2]g.ramirez.hidalgo@fz-juelich.de [2] s.nassyr@fz-juelich.de[3]jimenez@math.uni-wuppertal.de, [3] frommer@uni-wuppertal.de [1]pleiter@kth.se

arXiv:2601.05816v1 [cs.DC] 9 Jan 2026

摘要——**在科学计算中，如何降低稀疏线性系统迭代求解器的高昂计算成本一直是一项公认的挑战。此外，科学应用常受内存带宽限制，因此优化数据局部性并提高数据传输效率至关重要。我们扩展了格点QCD求解器 DD-αAMG，使其在采用或不采用奇偶预条件的情况下，均可在 Wilson-Dirac 算子求值和 GMRES 求解器中处理多个右端项（rhs）。为优化自动向量化，我们引入了支持多种数据布局的灵活接口，并实现了一种新的数据布局，以提高单指令多数据利用率。我们在 x86 和 Arm 集群上评估了这些优化，结果表明其具有性能可移植性，并取得了相近的加速比。本工作的一个重要贡献是对这些优化进行性能分析，揭示了架构约束和编译器行为所带来的复杂性。此外，我们还探索了利用 Arm 新型矩阵指令集 SME 的不同实现，并对其潜在优势进行了初步评估。**

索引词—**格点QCD、多重网格求解器、性能可移植性、可扩展矩阵扩展**

I. 引言

尽管多年来高性能计算系统的计算能力迅速提升，但由于数据集庞大且内存访问模式复杂，科学计算往往仍受内存限制。这是因为内存技术难以与计算能力的进步保持同样的扩展速度，从而成为高性能计算架构中的瓶颈。

格点量子色动力学（LQCD）是一种非微扰方法，它采用离散化的四维时空格点来研究强相互作用。强相互作用是一种基本力，支配着夸克与胶子之间的相互作用。格点QCD应用受内存限制，因为与必须在内存和处理器之间传输的数据量相比，其计算量相对较小，因此内存带宽和延迟成为限制因素。根据格点QCD应用和具体工作负载的不同，执行时间中有相当大一部分（通常为 $> 90\%$）花在迭代求解器上。

近年来，LQCD 求解器中出现了引入多个右端项（rhs）的趋势，以提升性能，因为：1）单个 rhs 的矩阵-向量运算被批量运算取代，从而提高算术强度；2）块 Krylov 子空间求解器可通过促进多个 rhs 之间的信息共享，减少达到收敛所需的迭代次数。然而，对这种 rhs 分块的探索尚缺乏详细且系统的性能评估。当这类分块技术与算法改进结合时，这一点尤为重要，因为目前尚不清楚二者各自对性能的影响。

随着高性能计算系统不断发展，传统的格点量子色动力学代码往往难以适应现代硬件的能力。现代 CPU 架构日益多样，因此，确保应用程序具备可移植性，对于适应可预见的硬件变化并保持高效性能至关重要。

本文重点优化开源 LQCD 求解器 DD-αAMG 的 C++ version [1]版本，但我们强调，该求解器仅作为一个示例，其中的见解也可广泛应用于其他科学计算应用。

本工作的贡献如下：首先，我们在 Wilson-Dirac 算子求值内核中实现右端项分块，并将其集成到 DD-α AMG 的批处理 GMRES 求解器中。此前版本的代码依赖 128 位 SSE 内部函数来实现高效执行，因此受到限制：不仅无法充分利用高性能计算系统中现已提供的 512 位单指令多数据指令，也难以移植到基于 Arm 的架构。因此，我们对代码进行重构，将自动向量化纳入考量，并引入一种专为循环向量化设计的新数据布局，同时保留兼容的数据布局，以便与旧版代码无缝集成。

其次，我们在三种截然不同的高性能计算架构上，对改进后的可移植求解器进行了全面的性能与架构分析：JUWELS（x86）、Ookami（Arm）和 HAICGU（Arm）集群。结果表明，与原始未分块版本相比，在内核中采用右端项分块可实现 10% 至 24% 的加速比。然而，其性能受有效内存带宽限制，而有效内存带宽对分块大小的选择较为敏感。我们还发现，尽管采用优化的数据布局进行自动向量化能够提高单指令多数据利用率，但性能提升往往受到架构约束的限制。此外，我们观察到，采用自动向量化的编译器可能会针对不同的分块大小生成显著不同的代码，从而使性能预测更加复杂。

求解器层面的算法变化可能会进一步降低内核级加速比。这些发现表明，右端项分块虽有益处，但也受到限制，因此应谨慎应用。

最后，为应对现代 CPU 架构的新硬件能力，我们研究了 Arm 的可扩展矩阵扩展（ SME ）在格点量子色动力学中的应用。我们在 Wilson-Dirac 算子求值中使用 SME 实现了一个格点量子色动力学例程，并通过仿真器分析其潜在性能。

本文结构如下：第二节回顾相关工作，第三节概述方法和实现，第四节定义评估配置，第五节和第六节呈现并讨论结果，最后总结全文并展望未来。

II. 相关工作

线性系统 $Ax_i = y_i$ （ $i = 1$、.....、b 个右端项）的求解可以通过扩展一个由所有右端向量 [3]共同生成的 Krylov 子空间来实现。在第 k 次迭代时，该方法可访问块 Krylov 子空间

$$K_k(A, Y) = \text{span}\{Y, AY, A^2Y, \ldots, A^{k-1}Y\},$$

其中， $Y = [y_1 \mid \ldots \mid y_b]$ 是所有右端项组成的矩阵。该组合子空间包含与每个 y_i 对应的单独 Krylov 子空间作为子集，因此能够同时降低所有系统的残差。

早期针对多个右端项方法的并行实现包括广义最小残量法的块版本 [4], [5], 以及 BiCGSTAB 的块版本 [6]。为解决后者的稳定性问题， [7]提出了块双共轭梯度法。作为这些算法的替代方案， [8] 开发了特征共轭梯度法。这是一种改进的共轭梯度法，通过在共轭梯度迭代过程中累积一段时间内的历史残差来扩展搜索空间。

块求解器已应用于物理领域，以应对特定的计算挑战。Forcrand等人 [9] 利用这些方法降低力项计算中的方差，而Boyle等人 [10]则将其应用于畴壁费米子系统。 [11]还成功地将这些方法用于格点量子色动力学中 Wilson-Dirac 算子的求解。

近期有关块求解器的研究着重于提升其在现代高性能计算系统上的效率。 [12]介绍了一种在图形处理器上高效实现块 Krylov 方法的方案。Birket等人 [13]在 DD-αAMG 中采用块 Krylov 求解器，同时求解多个右端项和移位系统。

格点量子色动力学始终是一个高度依赖高效利用高性能计算资源的研究领域。因此，人们开展了各种性能分析与建模工作，Bilardi 等人 [14]和 Bauer等人 [15] 的研究便是早期的例子。前者与本文尤为相关，因为它提出了一种系统分析数据传输的方法，而数据传输通常是限制性能的因素。对性能的追求也推动了人们探索新的处理器架构。本文也研究了 A64FX 处理器；由于其内存带宽高，该处理器很快便受到格点量子色动力学应用领域的广泛关注 [16]。Alappat 等人 [17]为 A64FX 上的一项应用提出了性能模型，该应用与本文所研究的应用有些相似之处。

Arm 最近推出的指令集架构（ISA）扩展——可扩展矩阵扩展（SME）——目前只有少数研究涉及。Remke 和 Beuer [18] 在近期的一款 Apple 处理器上探索了相对简单的内核。H¨ubner 等人 [19] 将研究扩展到了更复杂的高性能计算应用。Zhang 等人 [20] 介绍了使用 SME 指令实现快速傅里叶变换的优化方案。其他厂商也推出了支持矩阵或张量运算的指令。例如英特尔的高级矩阵扩展（AMX）（评估示例见 [21]）。由于它不支持双精度浮点运算，因此本文不予考虑。

III. DD-αAMG 的优化

本节介绍我们针对 DD-αAMG 代码进行优化的实现。首先，我们将在第 III-A 节介绍右端项分块，随后在第 III-B 节重新设计数据布局，以提高单指令多数据利用率。最后，我们将在第 III-C 节详细介绍针对矩阵运算的 SME 实现。

A. DD-αAMG 中的多个右端项

DD-αAMG 是一种基于聚合的自适应代数多重网格方法，使用区域分解来求解采用 Wilson-Dirac 算子的格点量子色动力学系统。除最粗层级外，所有层级均使用带多重网格预条件的柔性 GMRES，而最后一层则采用奇偶 GMRES。由于最细层级占据了主要的计算成本，本文重点优化该层级上的 GMRES 求解器，并省略多重网格预条件，以避免增加复杂性。

1) Wilson-Dirac 算子求值： 在 LQCD 求解器中，相当大一部分计算时间用于 Wilson-Dirac 算子的求值，因此该部分对于优化至关重要。我们将离散化 Wilson-Dirac 算子的公式表示如下：

$$D_W\psi(x) = \frac{4+m_0}{a}\psi(x) - \frac{1}{a}\sum_{\mu=0}^{3}\left(\pi_\mu^- \otimes U_\mu(x)\psi(x+\hat{\mu}) + \pi_\mu^+ \otimes U_\mu^\dagger(x-\hat{\mu})\psi(x-\hat{\mu})\right),$$

其中， m_0 表示质量参数， ψ 表示输入向量， x 表示一个格点。 $\mu = 0$、...、3 是维度索引，$\hat{}\mu$ 是移位向量。在本研究中，格距 a 为 1。 $\pi_\mu^\pm$ 是常量投影算子， U 表示规范场构型。该 formulation 包含两个部分：自耦合项（第一项），用于描述格点局域相互作用；以及跃迁项（第二项），用于描述相邻格点之间的相互作用。为降低离散化误差，引入克洛弗项 $C(x)$，改进后的 Wilson-Dirac 算子表示为

$$D\psi(x) = D_W\psi(x) - C(x)\psi(x).$$

算法1 Wilson-Dirac 算子求值

输入： ψ

输出： $\eta = D\psi$

1: $\forall x$，计算 $\eta(x) \leftarrow (4+m_0)\,\psi(x) - C(x)\psi(x)$

2: $\forall x, \forall \mu$，计算 $\lambda_\mu(x) \leftarrow \pi_\mu^- \otimes I_3 \psi(x)$

3: $\forall \mu, \forall -\hat{}\mu$ 光晕站点 x'，将 $\lambda_\mu(x')$ 发送给 $-\hat{}\mu$ 邻居 x'

4: $\forall x, \forall \mu$，计算 $\chi_\mu(x+\hat{}\mu) \leftarrow \pi_\mu^+ \otimes U_\mu^\dagger(x)\psi(x)$

5: $\forall \mu, \forall +\hat{}\mu$ 光晕站点 x''，将 $\chi_\mu(x'' + \hat{\mu})$ 至 $+\hat{\mu}$ 邻居的 x''

6: $\forall \mu$、$\forall +\hat{}\mu$ 光晕站点 x''，从 $+\hat{}\mu$ 接收 $\lambda_\mu(x''+\hat{}\mu)$ 邻居

7: $\forall x$, $\forall \mu$，计算 $\eta(x) \leftarrow \eta(x) - [I_4 \otimes U_\mu(x)]\lambda_\mu(x+\hat{}\mu)$

8: $\forall \mu$、$\forall -\hat{}\mu$ 光晕站点 x'，从 $-\hat{}\mu$ 邻居接收 $\chi_\mu(x')$

9: $\forall x$, $\forall \mu$，计算 $\eta(x) \leftarrow \eta(x) - \chi_\mu(x)$

我们在算法1中给出了 Wilson-Dirac 算子求值算法 [22]。

在 DD-αAMG 代码中，每个格点由四个坐标标识，分别对应晶格的四个维度。为了给每个格点分配唯一编号，我们使用位置编号系统将这些坐标映射为一个整数，其中每个维度的步长由该维度的大小决定。假设晶格的大小为 $N_0 \times N_1 \times N_2 \times N_3$，格点坐标 $(x_0,\ x_1,\ x_2,\ x_3)$ 可通过以下计算映射为格点编号：

$$x_0 \times N_1 N_2 N_3 + x_1 \times N_1 N_2 + x_2 \times N_1 + x_3.$$

数组 ψ 是一维的，并按照唯一格点编号的升序遍历晶格上的格点。我们记作 $\psi = [\psi(1)\ \psi(2)\ \dots\ \psi(N_L)]^T$，其中 N_L 表示格点总数。对于任意格点 x，$\psi(x) \in \mathbb{C}^{12}$，这表明

$$\psi = [\psi_1(1)\ \psi_2(1)\ \dots\ \psi_{12}(1)\ \psi_1(2)\ \dots \\ \dots\ \psi_{12}(2)\ \dots\ \psi_1(N_L)\ \dots\ \psi_{12}(N_L)]^T.$$

规范场构型 U 存储在一个二维数组中，其索引为格点和格点空间维度。请注意，对于表示格点的任意数组维度，其存储顺序遵循上文讨论的相同格点编号约定。因此，我们写作

$$U_\mu = [U_\mu(1)\ U_\mu(2)\ \dots\ U_\mu(N_L)]^T, \mu = 0 \dots 3.$$

每个 $U_\mu(x)$ 都是一个 3×3 矩阵，按行优先顺序存储，即 $U_\mu(x) \in \mathbb{C}^{3\times 3}$。算法1第 4 行和第 7 行中的操作可解释为同时将 $U_\mu(x)$ 应用于长度为 3 的向量。

克洛弗项为块对角结构，每个块均为厄米矩阵，因此，$C(x) \in \mathbb{C}^{42}$。投影算子 $\pi_\mu^\pm$ 为块对角结构中的反厄米矩阵，因此可以生成 λ_μ 和 χ_μ，使每个格点只需 6 个元素，即 $\lambda_\mu(x)\chi_\mu(x) \in \mathbb{C}^6$。在实践中，该方法应用于算法1的第 2 行和第 4 行。将数据写入 $\eta(x)$（第 7 行和第 9 行）时，还会进行额外的线性组合，以将数据恢复为长度 12。

将多个右端项引入 Wilson-Dirac 算子的求值中，即 $\psi = [\psi^{(1)}|\psi^{(2)}|\cdots|\psi^{(b)}]$，其中上标表示右端项编号。我们将 $v = [v(1)\ v(2)\ \dots\ v(N_L)]^T$ 形式的一组向量视为一个块向量（矩阵）：

$$V = \begin{bmatrix} v^{(1)}(1) & v^{(2)}(1) & \cdots & v^{(b)}(1) \\ v^{(1)}(2) & v^{(2)}(2) & \cdots & v^{(b)}(2) \\ \vdots & \vdots & & \vdots \\ v^{(1)}(N_L) & v^{(2)}(N_L) & \cdots & v^{(b)}(N_L) \end{bmatrix}. \quad (1)$$

要对以列优先顺序存储的 ψ 求解该系统，可以迭代应用算法1，并将每一列作为一个右端项。不过，这种方法无法改善数据局部性。更高效的方法是采用行主序存储，这样就能将遍历右端项的循环并入遍历格点的循环中x，例如，算法中的第 4 行可以改写为

$$\forall x, \forall \mu, \forall i = 1 \dots b, \text{Evaluate } \chi_\mu^{(i)}(x+\hat{\mu}) \leftarrow \pi_\mu^+ \otimes U_\mu^\dagger(x)\psi^{(i)}(x).$$

请注意，λ_μ、χ_μ 和 η 现在也多了一个大小为 b 的维度。这种方法通过减少内存访问总量，提高了算法1的算术强度，因为既不需要再次访问 clover 项，也不需要再次访问规范场构型。忽略缓存效应，算法1的算术强度为

$$\mathrm{AI}(b) = \frac{I_{\mathrm{fp}}(b)}{I_{\mathrm{mem}}(b)} = \frac{2574\,b\,\mathrm{Flop}}{(168\,b + 114)\,16\,\mathrm{Byte}}, \quad (2)$$

其中，b表示分块大小，16 表示双精度复数的大小。I_{fp} 和 I_{mem} 分别表示浮点运算次数以及与内存交换的数据量。

2) 批处理 *GMRES* 求解器：到目前为止，我们已在一个格点量子色动力学计算内核中引入了右端项分块，即 Wilson-Dirac 算子的求值。接下来，我们将详细介绍一种集成该内核的迭代求解器，以便研究完整格点量子色动力学求解过程中右端项分块的效果。

为支持 Wilson-Dirac 算子的分块求值，我们实现了一个批处理 GMRES 求解器（算法 2），该求解器在每次迭代中处理多个右端项。算法第 2 行调用 Wilson-Dirac 算子的分块求值，其中针对每个 $i = 1 \dots b$计算 $D\psi^{(i)}$。在完整求解过程中，为改善局部性而重新排列数据布局的开销，相较于数百次迭代而言微不足道。即使采用列主序存储，在求解前转换为行主序、求解后再转换回来仍然切实可行。

我们没有使用块 Krylov 子空间求解器，而是采用批处理版本，分别独立求解每个右端项。这种方法仅得益于机器层面的性能提升，因为它具有更有利的内存访问模式。相比之下，块 Krylov 子空间求解器还具有额外的算法性能优势，因此难以分离并评估各因素的实际影响。

奇偶预条件是格点量子色动力学中常用的一种技术，通常可将迭代次数减少约 2 倍至 3 [23]。它根据格点的位置将格点划分为奇格点和偶格点，并将含有 D 的线性系统转换为仅一半大小的 Schur 补系统

算法 2批处理 GMRES

输入： D，η，容差 ϵ
输出： $\forall i = 1 \ldots b,\ D\psi^{(i)} = \eta^{(i)}$

对于 $rn = 0;\ rn < restartNum$ && $!finish;\ rn++$ **执行**
  $\forall i$，计算 $r^{(i)} = \eta^{(i)} - D\psi^{(i)}$，
  $norm^{(i)} = \|r^{(i)}\|_2,\ V[0]^{(i)} = r^{(i)}/norm^{(i)}$
  对于 $rl = 0;\ rl < restartLen$ && $!finish;\ rl++$ **执行**
    $\forall i$，使用 $V[j]^{(i)}$ 执行一步阿诺尔迪迭代
    $\forall i$，更新 $H^{(i)},\ \gamma^{(i)}$，
    计算 $relnorm^{(i)} = \gamma[j+1]^{(i)}/\|\eta^{(i)}\|_2$
    如果 $max(relnorm) < \epsilon$ **则**
      $finished = 1$
    结束条件
  结束循环
  $\forall i$，计算 $\psi^{(i)} = \psi^{(i)} + V_{(i)} * y^{(i)}$，
  其中 $\|\gamma^{(i)} - H^{(i)}y^{(i)}\|_2 \rightarrow$ 最小值
结束循环

。我们将奇偶预条件作为批处理 GMRES 的可选功能，并将其应用扩展到 DD-αAMG 中的细网格层级。

B. 多右端项的数据布局

DD-αAMG 中用于分块的数据布局重排比上述讨论的情况更复杂。方程 1 中矩阵的一行可以展开为

$$\mathrm{Row}_x(V) = \begin{bmatrix} v_1^{(1)}(x) & v_1^{(2)}(x) & \cdots & v_1^{(b)}(x) \\ v_2^{(1)}(x) & v_2^{(2)}(x) & \cdots & v_2^{(b)}(x) \\ \vdots & \vdots & & \vdots \\ v_s^{(1)}(x) & v_s^{(2)}(x) & \cdots & v_s^{(b)}(x) \end{bmatrix}, \quad (3)$$

其中，s 表示与 $v(x)$关联的元素数量。该矩阵可以按列主序（布局 1）或行主序（布局 2）存储；我们的实现同时支持这两种布局，并可通过接口切换。

为了比较这两种布局，我们以矩阵向量乘法函数为例。我们以算法1第 7 行中的运算进行说明，其中一个 3×3矩阵（$U_\mu(x)$）作用于长度为 3 的向量（取自$\lambda_\mu(x + \hat{\mu})$）。为简便起见，我们将 3×3 矩阵记为 A，并用 M 表示公式 3 中的矩阵，其中 $s = 3$。

对于布局 1，M 按列优先顺序存储。这种布局只需对原始代码做最少的修改，因为可以复用将矩阵作用于向量的函数。我们可以在分块大小为 b的循环中执行 $A\,m^{(1)}$，直到 $A\,m^{(b)}$。然而，这种布局难以充分利用单指令多数据指令。如图 1(a) 所示，顺序加载 A会得到按行排列的数据，而加载 M 会得到按列排列的数据；两者相乘会产生标量数据。为了有效利用单指令多数据，可以对 A 或 M进行散布加载，但这会带来额外开销。

为克服这一问题，我们可以像布局 2 中那样，按行优先顺序存储 M。如图 1(b) 所示，可以使用

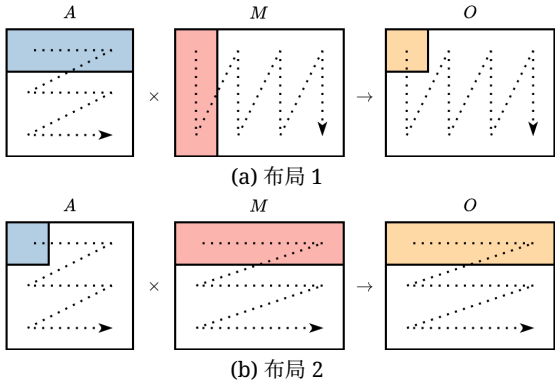


图 1. 使用两种数据布局进行矩阵运算的示例，其中 $b = 4$，虚线和箭头表示内存中的存储顺序。

掩码加载读取 A 中的标量，然后将其广播到 SIMD寄存器的所有通道。顺序加载 M 可获得逐行数据，乘法运算的输出也为逐行数据。与布局 1 相比，这使布局 2 更适合单指令多数据处理。

过去，对于支持 SIMD 指令的处理器架构上的复数运算，通常将实部和虚部存储在不同的数组中，以避免 SIMD 通道置换。现代 SIMD 指令集架构（如 AVX 和 SVE）对复数运算提供了良好的支持（示例见第 III-C 节），因此，使用复数数组更有利，因为这样可以改善数据局部性。因此，我们保留自然的复数存储格式，而不将其分离为各个分量。

C. 可扩展矩阵扩展实现

虽然格点量子色动力学应用涉及大规模线性系统，但所得矩阵大多是稀疏的。因此，在 DD-αAMG 中对整个线性系统使用可扩展矩阵扩展并不可取，因为计算效率低下。我们决定使用可扩展矩阵扩展实现一个矩阵-矩阵乘法函数，以实现 $AM = O$，如第三节-B 所述。尽管该函数开销较小，但在 Wilson-Dirac 算子求值过程中会占用大量计算时间，因此对整体格点量子色动力学求解而言不可忽略。

ZA 瓦片是可扩展矩阵扩展（SME）中的基本架构组件，它是一个大型二维数组寄存器，其维度取决于可扩展向量长度（SVL）。由于A 的大小为 3×3，而 M的大小为 $3 \times b$，代码会导致 ZA 瓦片未得到充分利用。例如，考虑 256 比特或 512 比特的 SVL，3 个复数双精度数的大小（3×128 比特）会使 ZA 瓦片中大小为 128 比特的通道闲置。

可扩展矩阵扩展（SME）中的关键指令是 fmopa，它会计算两个 SVL 向量的外积并进行累加。由于 O可以通过对 $i = 1 \ldots 3$重复执行 $A_{i^{th}col} \otimes M_{i^{th}row}$的加法来计算，因此可以直接使用 fmopa 指令实现操作 $AM = O$。

然而，由于复数运算，实际计算比简单地应用 fmopa 更为复杂。

算术，实际计算比简单地应用 fmopa 更复杂。两个复数相乘需要进行四次实数乘法。由于实部和虚部以交错格式存储，这些乘法需要 SVE/SME 中一种特殊的向量加载机制。SVE/SME 指令集架构提供了解交错结构化加载指令 ld2d，可将实部和虚部分别放入两个向量寄存器中。此外，还提供了交错存储指令 st2d，可将两个独立向量寄存器中的数据存入内存。对于足够大的向量或矩阵，这两条指令可实现高效的工作流程：1) 使用 ld2d 加载操作数；2) 分别计算实部和虚部的部分乘积；然后 3) 使用 st2d 存储结果。

鉴于 A 和 M 都至少包含一个较小维度，我们通过仅对其中一个操作数进行解交错来实现额外的优化。这种方法能让向量寄存器和 ZA 瓦片中有更多通道处于活动状态，从而更有效地利用可用的硬件资源。不过，这种方法需要存储一份交错操作数的副本，其中实部和虚部互换，且虚部取反，因此会增加两条指令：revd 和 fneg。

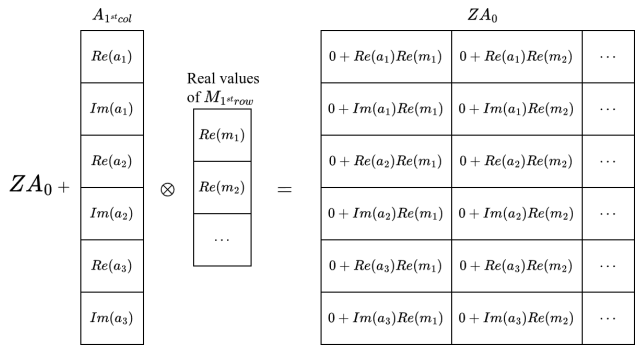
虽然布局 2 与 M 的内存访问模式自然匹配，但我们通过转置 A 进一步优化性能，以确保乘法运算中的访问完全连续。因此，我们对 A 采用列主序存储，对 M 采用行主序存储。图 2 展示了 $A_{1^{st}col} \otimes M_{1^{st}row}$ 运算，该运算需要在顺序步骤中执行两条 fmopa 指令。我们将 ZA_0 使用的 ZA 瓦片记为 ZA 瓦片，并将其初始化为零。图中省略了 ZA 瓦片上未使用的通道。在步骤 1 中，可以连续加载 $A_{1^{st}col}$，但必须使用解交错指令 ld2d 将 $M_{1^{st}row}$ 的实部和虚部分别加载到两个独立的向量寄存器中。随后，在步骤 2 的运算中会使用存放 $M_{1^{st}row}$ 虚部的向量寄存器。为了在步骤 2 中计算互换并取反后的 $A_{1^{st}col}$，我们使用上述 revd 和 fneg 指令。

经过两个步骤后，存储在 ZA_0 中的数据为

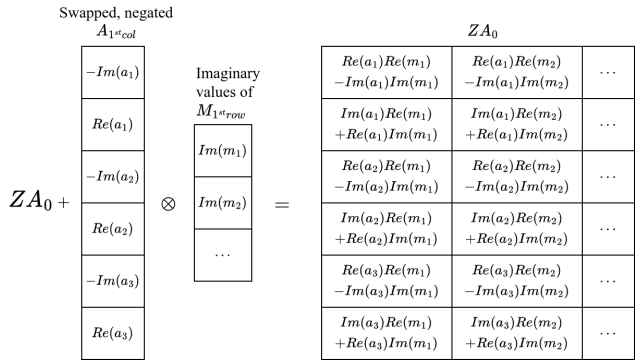
$$\begin{bmatrix} Re(a_1m_1) & Re(a_1m_2) & \dots \\ Im(a_1m_1) & Im(a_1m_2) & \dots \\ Re(a_2m_1) & Re(a_2m_2) & \dots \\ Im(a_2m_1) & Im(a_2m_2) & \dots \\ Re(a_3m_1) & Re(a_3m_2) & \dots \\ Im(a_3m_1) & Im(a_3m_2) & \dots \end{bmatrix},$$

这些正是 $A_{1^{st}col} \otimes M_{1^{st}row}$ 在复数中的完整结果。$AM = O$ 操作需要对所有 $i = 1 \dots 3$迭代执行这两个步骤。

我们实现了两种 SME 变体来执行 $AM = O$，两者的区别在于哪个输入操作数采用交错排列，并需要执行取反和交换操作。第一种变体（neg-A）对 A 的元素执行取反和交换操作，如图 2 所示。第二种变体（neg-M）则对 M 的元素执行这些操作。在此变体中，我们依次加载 $M_{i^{th}row}$，但在外积计算中分别加载 $A_{i^{th}col}$ 的实部和虚部。对于较大的



(a) 步骤 1



(b) 步骤 2

图 2. 使用可扩展矩阵扩展演示 $A_{1^{st}col} \otimes M_{1^{st}row}$。

分块大小 b，预计此变体的效率较低，因为所需的取反和交换操作数量会随 M 的行维度增加。同时，A 的列也会进一步降低 ZA 瓦片的利用率，因为一个维度中仅使用 3×64 比特。

IV. 测试规范

A. 测试平台

为评估我们在 DD-αAMG 中的实现，我们在三个不同的平台上运行该应用：JUWELS（x86）、Ookami [24]（Arm）和 HAICGU（Arm）。这些实现包括在 Wilson-Dirac 算子计算（内核）和批处理 GMRES（求解器）中采用两种数据布局的右端项分块。

表I列出了第 V 节评估中使用的硬件和软件。除了理论内存带宽外，我们还采用 STREAM 基准测试 [25] v5.10 来测量峰值可达内存带宽以及达到这些带宽所需的核心数。采用 STREAM 基准测试是为了建立一个经验性性能上限，相比理论内存总线极限，它更为准确。在许多架构上，STREAM Triad 基准测试可实现最高内存带宽。然而，对于写密集型工作负载，Triad带宽仍可能偏乐观，尤其是在读写带宽不对称的架构上。为考虑此类情况，我们还在表I中列出了 STREAM Copy 带宽。

由于目前没有可用的 HPC 级硬件 SME实现，我们使用仿真器进行 SME 评估。

表I 针对优化后的 DD-αAMG 评估的基准测试平台的节点级硬件和软件配置。

	JUWELS	Ookami	HAICGU
架构	x86	Arm	Arm
处理器	2× 英特尔至强铂金 8168	A64FX	2× 鲲鹏 920
核心数	2×24	48	2×64
CPU 频率	2.7 吉赫兹	1。8 GHz	2.6 GHz
L1 数据缓存	每个核心 32 KiB	每个核心 64 KiB	每个核心 64 KiB
L2 缓存	每个核心 1 兆二进制字节	4×8 兆二进制字节	每个核心 512 KiB
L3 缓存	每个中央处理器 33 兆二进制字节	none	每个中央处理器 32 兆二进制字节
SIMD 宽度	512 比特	512 比特	128 比特
内存	96 吉比字节	32 吉比字节	128 吉比字节
理论内存带宽	256 GB s^{-1}	922 GB s^{-1}	341 GB s^{-1}
STREAM Triad 带宽	155GB s^{-1} (48 个核心)	619GB s^{-1} (24 个核心)	218GB s^{-1} (128 个核心)
STREAM Copy 带宽	133GB s^{-1} （48 个核心）	540GB s^{-1} （24 个核心）	214GB s^{-1} （128 个核心）
网络带宽	100 Gbit s^{-1}	100 Gbit s^{-1}	100 吉比特/秒$^{-1}$
所用编译器	GCC v13.3.0	GCC v13.2.0	GCC v14.1.0
单指令多数据的编译器标志	-mavx512f	-march=armv8.2-a+sve	-march=armv8.2-a+simd

我们曾考虑使用周期级模拟器gem5 [26] ，但截至本文撰写时，gem5 中的 SME实现尚不够成熟，无法进行性能分析。因此，在第 VI 节中，我们使用仿真器 QEMU [27] v9.2.50及其插件 *insn*来评估 SME。

B. 测试配置

1) 实证性能评估：我们评估了两种格点配置，其规模分别为：1) 128×64^3，这是 LQCD 模拟中常用的配置 [28], 以及 2) 64×16^3，适用于单节点运行。这两种格点均用于第 V-A 节中的内核评估，而只有 128×64^3 格点用于第 V-B 节中求解器评估的示例。

我们使用 GCC 自动向量化（通过 -O3 启用），并采用表I中列出的其他特定于平台的编译器标志进行单指令多数据优化。并行性通过现有 DD-αAMG 代码中的 MPI 实现。此外，我们还为 Wilson-Dirac 算子求值实现了 OpenMP 支持。这对于增加内存级并行并充分利用 Ookami 上所采用的高带宽内存（HBM2）架构是必要的。

鉴于 A64FX 节点具有更高的内存带宽，且 JUWELS 节点采用双路架构，为确保多节点评估中的性能比较公平，我们使用的 JUWELS 节点数量（256）是 Ookami 节点数量（128）的两倍。受硬件限制（28 个节点），HAICGU 无法容纳 128×64^3 晶格配置。因此，本文不纳入 HAICGU 的多节点结果。

对于多节点和单节点评估，JUWELS 和 Ookami 使用相同的节点级并行设置。在 JUWELS 上，每个节点配置 16 个 MPI 进程。晶格分解要求 MPI 进程数为 2 的幂，因此继续增加进程数（例如增至 32 个）会妨碍高效的多线程。虽然减少每个节点的 MPI 进程数可以让每个进程使用更多线程，但我们发现实际性能并不理想。因此，我们将每个节点配置为 16 个 MPI 进程，每个进程使用 2 个线程，因为在 JUWELS 上增加线程数的收益会逐渐降低。Ookami 也采用类似配置，我们将每个节点配置为 16 个 MPI 进程，但每个进程使用 3 个线程，以最大限度地提高核心利用率。HAICGU 的核心数为 2 的幂，这带来了架构上的优势。通过实测，我们确定 HAICGU 上的最佳配置为 32 个 MPI 进程，每个进程使用 4 个线程。我们对 MPI 和 OpenMP 的亲和性进行了细致配置，以确保在所有平台上，每个节点内的进程和线程都能得到最优分布。

2) SME 的模拟性能评估：我们实现了一个基准测试驱动程序，用于执行第三节-C 中讨论的矩阵-矩阵乘法函数。该驱动程序支持四种变体：BLIS [29]v1.1 实现的 ZGEMM 例程（函数 cblas_zgemm）、使用 GCC v14.2.0 自动向量化的 SVE 代码，以及我们实现的两种 SME 版本（neg-A 和 neg-M）。为确保指令计数准确，驱动程序会多次遍历感兴趣区域（ROI）。

我们还编写了自定义性能建模脚本：1）通过 QEMU 执行驱动程序，分别采用两种不同的分块大小（b_1、b_2）；2）计算两次运行的指令计数差异；3）应用针对特定指令的成本权重，得出总计算成本的差异。

该方法有意排除程序开销，专注于隔离并量化分块大小区间 $|b_1-b_2|$ 变化所导致的计算成本差异。也就是说，为了评估模拟性能，我们运行建模脚本来统计与计算相关的指令，并应用相应的成本。该方法使我们能够比较不同变体，并估算无停顿执行时的潜在计算加速比，而不考虑任何与内存相关的瓶颈。

V. 实证性能评估

A. Wilson-Dirac 算子求值

1) 多节点性能：我们使用屋顶线模型展示图 3 中的多节点性能，其中 AI(b) 表示分块大小b对应的算术强度；参见公式 2。由于该应用受内存限制，

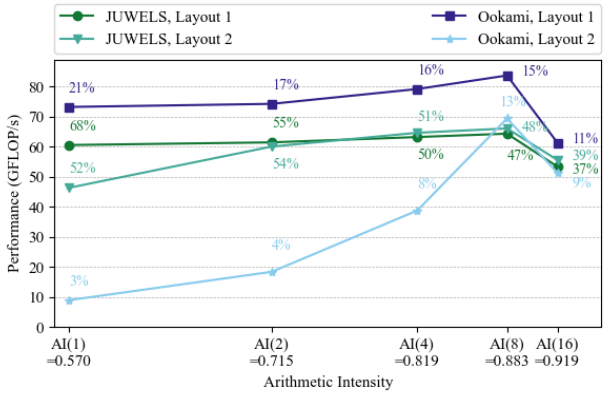


Fig. 3. Wilson-Dirac 算子求值中每节点性能随算术强度的变化 - 在 JUWELS 和 Ookami 上对 128×64^3 晶格进行 Dirac 算子求值，百分比标签表示架构效率。

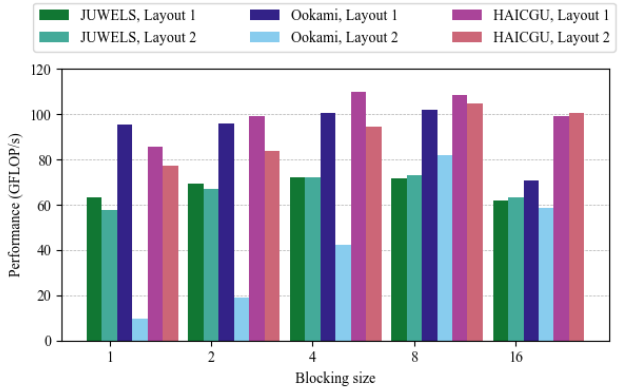


Fig. 4. 在 JUWELS、Ookami 和 HAICGU 单节点上 Wilson 算子的性能比较 - 128×16^3 晶格进行狄拉克算子求值。

表II JUWELS、Ookami 和 HAICGU 上 Wilson-Dirac 算子求值的架构效率。

$b =$	1		2		4		8		16	
布局	1	2	1	2	1	2	1	2	1	2
JUWELS	72%	65%	62%	60%	57%	57%	52%	54%	43%	44%
Ookami	27%	3%	22%	4%	20%	8%	19%	15%	12%	10%
HAICGU	69%	62%	64%	54%	62%	53%	56%	54%	50%	50%

其性能上限取决于内存带宽和算术强度。该上限还受到一些因素的约束，这些因素会影响基于制造商提供的硬件信息计算出的理论性能。通过 STREAM Triad 基准测试测得的内存带宽，为可达到的内存带宽提供了更现实的上限。因此，我们将上限定义为

$$\text{Theor. HW. Perf.}(b) = \text{STREAM Triad Bandwidth} \times \text{AI}(b). \tag{4}$$

我们进一步使用架构效率 [30] 这一指标，将测得的性能量化为性能上限的百分比。该指标定义为

$$\text{Arch. eff.} = \frac{\text{Meas. Perf.}}{\text{Theor. HW. Perf.}}, \tag{5}$$

所有数据点对应的数值均报告于图3中。

由于 DD-αAMG 中原始的、未分块且未经优化的实现等同于分块大小为 $b = 1$ 的布局 1，因此从图 3 可以看出，性能最佳的右端项分块变体（布局 2）在 JUWELS 上相比原始实现最高可提升 10% 的性能。在 Ookami 上，性能提升幅度增至 15%，不过该平台上性能最佳的右端项分块变体是布局 1。

2）单节点性能：图4展示了单节点性能对比，表II列出了相应的架构效率，揭示了不同测试平台之间显著的性能差异。

JUWELS 和 HAICGU 上的性能差异大致与表I中的 STREAM Triad 带宽差异一致。尽管 Ookami 节点的内存带宽分别比 JUWELS 和 HAICGU 节点高出4×和 2.8×，但这一优势未能得到利用。原因之一是，HAICGU 的核心数远多于 Ookami，因此可通过提高线程利用率实现更高的内存级并行度。另一个原因是，与 Ookami 上的 A64FX 相比，JUWELS 和 HAICGU 上的中央处理器缓存容量显著更大。这些架构差异反映在性能上：采用右端项分块时，HAICGU 的性能超过 Ookami（与不分块的实现相比，加速比最高可达 24%）；而 JUWELS 的性能约为它们的 70%。

3) 性能分析：鉴于详细的性能结果，我们针对优化的有效性以及观察到的异常提出了几个关键问题。

不同的数据布局如何影响单指令多数据利用率，这又会如何影响整体性能?

如第三节 III-B 所述，我们采用布局 2 以有效利用单指令多数据，并保留原始布局 1 作为传统方案和比较基准。为了量化右端项分块中的单指令多数据利用率，我们使用 PAPI [31] 读取与已执行的单指令多数据指令相关的性能计数器。

对于 JUWELS，我们通过读取 FP_ARITH:512B_PACKED_DOUBLE 事件来测量执行的 512 位 AVX 指令。表 III 显示，除 $b = 2$ 外，布局 2 执行的此类指令比布局 1 更多；对于 $b = 2$，由于向量长度为 2 且元素大小为 128 位，使用 512 位寄存器对标量-向量运算进行向量化的效率较低。图 3 和 4 表明，在 JUWELS 上，布局 2 在 $b > 2$ 方面优于布局 1，其性能优势得益于更高效的单指令多数据利用率。

尽管两种布局都在 JUWELS 上使用 AVX512，但我们发现，尽管 Ookami 上两种布局均启用了 SVE，只有布局 2 使用了 SVE。因此，我们测量 SVE（FP_DP_SCALE_OPS_SPEC）和 NEON（FP_DP_FIXED_OPS_SPEC）指令，以分析单指令多数据的利用情况。表 IV 显示，SVE 指令生成于

表 III 浮点算术：JUWELS 上多个右端项的 Wilson-Dirac 算子求值的 512 位打包双精度读数。

$b=$	2	4	8	16
布局 1	22675456	45350912	90701824	181403648
布局 2	11075584	59768832	160890880	305463296

表 IV浮点_双精度_缩放_运算_规格 （缩放）和浮点_双精度_定点_运算_规格 （定点）读数：Ookami 上多个右端项的 Wilson-Dirac 算子求值。

$b=$		2	4	8	16
缩放	布局 1	0	0	0	0
	布局 2	39322800	39322800	41420016	82840032
固定	布局 1	45090149	90180149	180360293	360720437
	布局 2	5	5	5	5

布局 2，而布局 1 仅生成 128 位单指令多数据指令。然而，尽管执行的指令更少，布局 2 在 Ookami 上的表现通常不如布局 1（图 3 和 4）。这可能表明，在 Ookami 上使用 SVE 指令的成本更高。

由于缺乏有关鲲鹏 920 处理器性能计数器的文档，我们无法对 HAICGU 进行定量的 SIMD 分析。不过，我们注意到，HAICGU 上的每个核心仅包含一条 128 位双精度 SIMD 流水线，这为 SIMD 运算造成了性能瓶颈。因此，在 HAICGU 上的大多数情况下，布局 1 的性能略优于布局 2（图4）。

关于性能可移植性，我们能得出什么结论?

如图 3 和表II所示，我们的 Wilson-Dirac 算子求值实现，在 JUWELS 和 HAICGU 上均实现了较高的架构效率，表明 DD-αAMG 已成功移植到 Arm。值得注意的是，尽管 Ookami 的理论内存带宽最高，其效率却最低，这表明该架构给代码实现充分的带宽饱和带来了更大的挑战。

为什么性能没有随着算术强度的增加而同比提升?

对于内存受限的应用，屋顶线模型表明，性能上限应随算术强度成比例提升。假设内存带宽持续达到饱和，测得的性能也应相应提升。然而，图 3 和 4 中的性能趋势表明，屋顶线模型未涵盖的其他因素也会影响结果，导致曲线的上升斜率与预期不同，并在分块大小较大时向下弯曲 b。这一现象与观察到的架构效率下降相符。

我们研究了性能下降的几个原因。首先，对于较大的 b，随着需要传输的数据增多，网络带宽可能成为瓶颈。其次，较大的 b 可能会引发缓存效应，并在数据访问中导致跨步。第三，有效内存带宽可能会随 b 而变化，例如读写比的变化可能会造成这种情况。

第一个原因可以排除，因为算法1有效地重叠了计算与通信。第 3 行和第 5 行的通信分别与第 6 行和第 8 行配对。第 4 行和第 7 行的充足计算最大限度地缩短了等待时间。在实践中，我们观察到，MPI Wait() 时间约为计算时间的 10%，且与 b 无关。图 3 和图 4 中多节点运行与单节点运行的趋势几乎相同，也佐证了这一点。

鉴于上述分析，我们接下来将分析重点放在单节点性能上。为了进行定量分析，我们使用 Ookami 上可用的性能计数器，评估仅启用 MPI 的 64×16^4 晶格（见表 V），从而消除线程干扰对计数器读数的影响。

与大多数模板算法不同，算法1并不依赖缓存重用。它针对每个 μ 遍历晶格格点；由于这些格点在同一方向上没有共同的邻居，因此在固定的 μ 内不会重复使用数据。在 Ookami 上表现最佳的变体（布局 1）中，缓存再填充和回写随 b 线性增长，这表明时间局部性的利用几乎没有变化。

排除两种可能性后，我们转而分析读写比。使用算法1描述的 Wilson-Dirac 算子求值，其读写比为 $204b+330$： $204b$，未考虑数据局部性或缓存写分配行为。鉴于 A64FX 的峰值写吞吐量为 461 GB s^{-1}（为峰值读吞吐量的一半），该比值表明，随着 b 增大，代码可能无法充分利用读吞吐量，并且会变得更加偏向写操作。例如，如表 V 所示，布局 1 的缓存再填充与回写之比从 $b=1$ 时的约 2.4 降至 $b=16$ 时的 2。作为参考，STREAM Copy基准测试测得的比值为 2.43，带宽为 540 GB s^{-1}，比 Ookami 上的 Triad带宽低 13%。

我们还使用 PAPI 在 Ookami 上测量了 L2 缓存与内存之间的双向有效带宽（列于表 V），测量时采用

$$\frac{(\texttt{L2D_CACHE_REFILL}+\texttt{L2D_CACHE_WB})\times 256\,\mathrm{B}\times f}{\texttt{CPU_CYCLES}}, \tag{6}$$

其中，256B 是缓存行大小，f 是处理器频率。我们观察到，布局 1 的有效内存带宽持续下降，从 $b=1$降至 16，降幅达 40%。尽管屋顶线模型预测性能将提升 60%，但这些因素综合作用的结果是，右端项分块几乎没有带来净收益。

尽管 JUWELS 缺少用于跟踪硬件预取器行为的性能计数器，因此无法获取精确的内存加载和存储信息，但从 STREAM 基准测试结果可以推断，随着内存存储与加载之比上升，内存带宽会降低。例如，STREAM Copy 和 Scale 基准测试在 JUWELS 上达到约 133 GBs^{-1}，比 Add 和 Triad 带宽低 14%。也就是说，读写比的变化可能会影响 JUWELS 上的性能，这解释了性能变化为何不成比例。

表 VL2D_缓存_重新填充、L2D_缓存_回写、中央处理器_周期读取数据及在 Ookami 上对多个右端项进行 Wilson-Dirac 算子求值时的派生带宽。

$b=$	1		2		4		8		16	
布局	1	2	1	2	1	2	1	2	1	2
重新填充	392270	400478	581620	677120	979685	1238074	1745437	2365484	3272209	3422734
回写	165508	164374	238375	325464	422127	652722	856033	1303890	1617233	1516062
周期（10^6）	32	357	63	361	125	311	244	305	490	619
带宽（GBs^{-1}）	127	12	97	21	83	45	79	89	74	59

由于缺少性能计数器文档，无法在 HAICGU 上实现全面的内存加载和存储监控。不过，与其他平台相比，分块大小增加对 HAICGU 性能的影响较小。作为参考，STREAM Copy 和 Scale 基准测试在 HAICGU 上达到约 214GBs^{-1} ，与其 Add 和 Triad 带宽几乎相同。这表明，与其他平台相比，读写比对 HAICGU 性能的影响要小得多。

是否还有其他影响性能的因素? 我们观察到，GCC 通过自动向量化生成的汇编代码表明，不同的分块大小选择 b 可能会生成差异很大的代码。这些差异可能源于循环展开、单指令多数据指令，甚至复数运算方面的决策。以布局 2 中一个简单的向量-标量运算例程（参见算法1中的第 2 行）为例，Ookami 上的编译器会将循环展开为 b，即在一次迭代中对所有右端项执行向量-标量乘法。与 $b=8$, 16 相比，这一决策会为 $b=1$, 2 生成显著更多的数据加载和存储指令。这表明，机器级别的实际算术强度可能不会像根据代码分析所预期的那样扩展。因此，使用公式 2 的屋顶线模型进行性能预测可能过于乐观。这也有助于解释相对于公式 2 给出的算术强度，所观察到的非成比例性能变化。

B. 批处理 *GMRES* 求解器

1) 多节点性能：为全面评估我们对 Wilson-Dirac 内核所做优化的影响，我们现在分析其在包含迭代求解器的 LQCD 模拟框架中的性能。为此，我们评估算法 2 中描述的批处理 GMRES 求解器，并将迭代次数固定为 100（重启 10 次，每次长度为 10）。

如图 5 所示，两种布局的性能趋势比图 3 中更平滑，这是因为在批处理 GMRES 中，随着每个右端项存储测试向量 V 和Hessenberg 矩阵 H 等额外数据，算术强度会降低。与 JUWELS 上的非分块实现相比，较低的算术强度使峰值性能加速比降至 7%。

奇偶预条件情况下的性能进一步受到计算舒尔补所产生的更多非顺序内存访问模式的限制

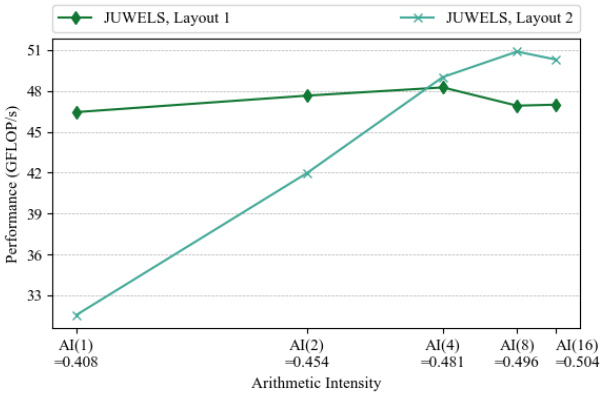


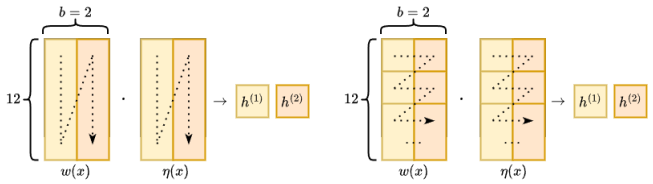
图 5. JUWELS 上 128×64^3 晶格的批处理 GMRES 执行 100 次迭代时，单节点性能随算术强度的变化。

。我们观察到，采用奇偶预条件的批处理 GMRES 性能约为未采用预条件时的 70%。结果揭示了算法效率与机器性能之间的权衡。尽管机器性能较低，但奇偶预条件线性系统的总收敛迭代次数减少了一半，因此总体上仍更节省时间。

批处理 GMRES 的结构继承自旧版 DD-αAMG 代码，采用单线程运行，在 Ookami 上的内存带宽利用率不佳，且相比 JUWELS 并无优势。因此，此处不讨论批处理 GMRES 在 Ookami 上的表现。

2) 性能分析： 由于 Wilson-Dirac 算子求值内核在批处理 GMRES 中占据了超过 90% 的计算时间，因此批处理 GMRES 的性能趋势与该内核非常相似。具体而言，关于数据布局和屋顶线模型所导致的性能差异，其原因已在第 V-A3 节中解释。图 5 中布局 2 的 $b \leq 4$ 批处理 GMRES 性能还出现了另一处异常：其性能提升幅度超过了图 3 中内核的增长幅度。这种现象至少部分是由 Arnoldi 方法中的点积运算造成的；当分块大小较小时，这些运算在布局 2 下的效率会显著降低。

点积是在 Arnoldi 向量w 与 Wilson-Dirac 算子求值的输出向量 η 之间计算的；二者均具有公式 1 所示的形式。需要执行的操作是 $h^{(i)} = w^{(i)} \cdot \eta^{(i)}$，针对 $i = 1 \ldots b$。



(a) 布局 1 (b) 布局 2

图 6. 使用 $b = 2$ 的两种数据布局下点积运算示例，颜色用于区分不同的右端向量。

表 VI： $b = 8$，16 和迭代次数为 100 时，512 比特可扩展向量长度下的指令开销。

方法	BLIS	SVE	Neg-M	neg-A
覆盖关闭	31704	7104	5004	2604
覆盖开启	29122	8022	6222	2922

为展示点积计算，我们在图 6 中演示$h^{(i)} = w^{(i)}(x)\cdot\eta^{(i)}(x)$。假设使用 512 位寄存器，对于布局 1，加载 $w(x)$ 和 $\eta(x)$ 后，得到按顺序排列的数据，分别对应第一个或第二个右端项；对它们进行点积计算，分别会对标量 $h^{(1)}$ 或 $h^{(2)}$作出贡献。然而，布局 2 中的数据是交错排列的，对 $w(x)$ 中的 4 个元素和 $\eta(x)$ 中的 4 个元素进行点积计算，会同时对 $h^{(1)}$ 或 $h^{(2)}$作出贡献，因此需要执行额外操作，将数据从 512 位寄存器中分离出来以进行归约。

我们引入了一项修正，将数据分离延迟到最后一步，并使用缓冲区为较小的 b 存储额外数据。此方法使 $b = 1, 2$ 的性能提升了 4–5%，这往往使布局 2 在批处理 GMRES 中的性能增长与内核保持一致。

VI. 可扩展矩阵扩展的模拟性能评估

按照第 IV-B2 节中指定的指令成本评估方法，我们统计实验中的浮点、NEON、SVE 和可扩展矩阵扩展指令（包括加载和存储指令），并假设整数指令的执行成本可以忽略不计。实验分别采用 8 和 16 的分块大小，两种情况下结果相同。详细数据见表 VI。

在不覆盖成本、将每条指令的权重设为 1 时，表现最佳的 SME 变体 neg-A 比 SVE 快 2.73× 倍，比 BLIS 快 12.2× 。neg-M 变体虽然如第三节所述，会因处理较大的 b 时需要额外的 revd 和 fneg 指令而变慢，但相较于 SVE 仍有 42% 的优势，较 BLIS 则提升 6.3× 。

我们还研究了采用以下覆盖设置时的开销：mov 指令的开销为 0（假设其只是寄存器重命名），存储操作的开销为 2，SME 外积指令的开销为 2。将存储操作的开销设为加载操作的两倍，符合 A64FX 的架构特点，因为其加载端口数量是存储端口的两倍。同样，将 SME 外积指令的开销设为两倍，反映了它们的吞吐量低于较简单的算术运算。采用这些覆盖设置后，SME 变体 neg-A 相比 SVE 的加速比为 2.75× 相比 BLIS 的加速比为 9.97× 。neg-M 变体相比 SVE 性能提升 30%，相比 BLIS 提升 4.7×。

VII. 结论

我们为格点量子色动力学求解器 DD-αAMG 提出了多项优化。首先，我们讨论了 Wilson-Dirac 算子求值和批处理 GMRES 求解器中的右端项分块实现。其次，我们引入了一种数据布局，旨在提升 Wilson-Dirac 内核中右端项分块的单指令多数据利用率。最后，我们使用可扩展矩阵扩展实现了一个格点量子色动力学例程。本文的相关成果，包括复现结果所需的全部源代码、脚本和数据集，均可在线获取 [32]。

我们在 JUWELS、Ookami 和 HAICGU 上对内核和求解器的优化进行了实证评估，并分析了这些平台间的性能可移植性。性能分析表明，增大分块大小通常能够提升性能；然而，这也会提高写入强度，从而抵消收益，甚至可能导致性能下降。此外，编译器行为也可能以难以预测的方式影响性能。我们得出结论：尽管使用多个右端项是加速内存受限科学应用的常见策略，但要实现最佳性能，必须仔细考虑硬件和编译器的约束。算法变更还会带来权衡，例如批处理 GMRES 求解器的评估所示，此类变更可能导致机器性能下降。

我们使用指令成本模型评估了采用 SME 指令实现的格点量子色动力学例程的模拟性能，结果显示其相较于其他实现具有显著优势。然而，要发挥这一性能优势，需要足够高的内存带宽。

未来的工作包括优化威尔逊–狄拉克算子求值中的数据局部性。通过修改循环结构，可以立即重用中间值。此类修改也能解决分块大小较大时观察到的写入密集型行为。此外，使用内在函数实现内核的关键部分，并将结果与本文所述结果进行比较，也很有价值。可扩展性与性能之间的相互影响也是一个值得研究的重要方面。如果未来能获得面向高性能计算的 SME 硬件，我们计划在真实场景中测试实现，以进行验证。

致谢

作者谨此感谢石溪大学研究计算与网络基础设施以及石溪大学先进计算科学研究所，使我们得以使用创新型高性能Ookami计算系统；该系统得益于美国国家科学基金会提供的500万美元资助（编号：#1927880）。作者衷心感谢高斯超算中心通过约翰·冯·诺依曼计算研究所（NIC）在J¨lich超级计算中心的GCS超级计算机JUWELS上提供计算时间，为本项目提供资助。

（JSC），项目编号为MUL-TRA。此外，我们还要感谢开放边缘与高性能计算倡议，使我们能够通过法兰克福歌德大学的HAICGU 集群使用基于Arm的开发环境。

本工作的部分经费来自欧盟 HORIZON MSCA 博士网络计划 AQTIVATE，资助协议编号为 101072344。G.R-H. 感谢 EoCoE-III 项目提供的经费支持；该项目获得了欧洲高性能计算联合事业的资助，资助协议编号为 101144014。

参考文献

[1] M. R. Artur Strebel，“DDalphaAMG-Cpp”，https://git.uni-wuppertal.de/ strebel/DDalphaAMG-Cpp，2022年。[2] A. Frommer 等人，“用于格点 Wilson–Dirac 算子的自适应聚合式域分解多重网格”，SIAM 科学计算期刊，第36卷，第4期，A1581–A1608页，2014年。[3] M. H. Gutknecht，《用于具有多个右端项的线性系统的块 Krylov 子空间方法：导论》，I. S. Duff、A. H. Siddiqi 和 O. Christensen 编。印度新德里：Anamaya 出版社，2006年。[4] B. Vital，“在多处理器上求解大型线性问题的若干方法研究”，博士论文，雷恩大学，法国，1990年。[5] H. Calandra 等人，“一种修正的块柔性 GMRES 方法：每次迭代均进行消去，用于求解具有多个右端项的非厄米线性系统”，SIAM Journal on Scientific Computing，第35卷，第5期，S345–S367页，2013年。[6] A. El Guennouni、K. Jbilou 和 H. Sadok，“用于具有多个右端项的线性系统的块 BiCGSTAB 版本”，Electron. Trans. Numer. Anal.，第16卷，129–142页，2003年。[7] H. Tadano、T. Sakurai 和 Y. Kuramashi，“块 BiCGGR：一种用于计算高精度解的新型块 Krylov 子空间方法”，JSIAM Letters，第1卷，44–47页，2009年。[8] A. Stathopoulos、A. M. Abdel-Rehim 和 K. Orginos，“QCD 中具有多个右端项的求逆消去法”，J. Phys. Conf. Ser.，第180卷，012073页，2009年。[9] P. de Forcrand 和 L. Keegan，“使用块求解器和多个赝费米子的有理混合蒙特卡罗”，Phys. Rev. E，第98卷，043306页，2018年10月。[10] P. A. Boyle，“使用多重网格预条件块共轭梯度算法，为畴壁费米子实现多重网格多右端项求解”，2024年9月。[11] T. Sakurai、H. Tadano 和 Y. Kuramashi，“块 Krylov 子空间算法在格点 QCD 中求解具有多个右端项的 Wilson–Dirac 方程中的应用”，《计算物理通讯》，第181卷，第1期，113–117页，2010年。[12] M. A. Clark 等人，“通过在图形处理器上高效实现块 Krylov 子空间求解器，突破内存带宽限制”，《计算物理通讯》，第233卷，29–40页，2018年。[13] S. Birk 和 A. Frommer，“用于格点 QCD 计算中多个右端项和多个位移的共轭梯度方法”，PoS，第 Lattice 2011 卷，027页，2012年。[14] G. Bilardi 等人，“片上多处理在 QCD 计算机中的潜力”，载于 HiPC 2005，《计算机科学讲义》系列。Springer Berlin Heidelberg，2005年，第3769卷，386–397页。[15] G. Bauer、S. Gottlieb 和 T. Hoefler，“MILC 格点 QCD 应用 su3 rmd 的性能建模与比较分析”，载于 CCGrid，2012年5月，652–659页。[16] N. Meyer、P. Georg、D. Pleiter、S. Solbrig 和 T. Wettig，“支持 SVE 的格点 QCD 代码”，载于 CLUSTER 会议，2018年，623–628页。[17] C. Alappat 等人，“A64FX 上流式内核与稀疏矩阵向量乘法的性能建模”，载于 PMBS 研讨会，2020年，1–7页。[18] S. Remke 和 A. Breuer，“你好，SME！使用可扩展矩阵扩展生成快速矩阵乘法内核”，载于 SC24 研讨会，2024年，1443–1454页。[19] P. H¨ubner、A. Hu、I. Peng 和 S. Markidis，“苹果与橘子：评估 Apple Silicon M 系列片上系统在高性能计算性能与能效方面的表现”，2025年。

[20] R. Zhang、H. Jia、Y. Zhang、B. Yan、P. Ma、L. Wang 和 W. Zhao，“OpenFFT-SME：面向 ARM SME CPU 的高效外积模式 FFT 库”，载于 *IPDPS*，2024年，938–949页。[21] C. S. Mummidi、V. C. Ferreira、S. Srinivasan 和 S. Kundu，“分块 AMX 加速器上的高效自检矩阵乘法”，*ACM*《体系结构与代码优化汇刊》，第21卷，第2期，2024年2月。[22] M. Rottmann，“格点 QCD 的自适应区域分解多重网格”，博士论文，伍珀塔尔大学，博士论文，2016年，2016年。[23] J. C. Osborn、R. Babich、J. Brannick、R. C. Brower、M. Clark、S. Cohen 和 C. Rebbi，“用于 clover 费米子的多重网格求解器”，*arXiv* 预印本 *arXiv:1011.2775*，2010年。[24] A. C. Calder 等人，“Ookami：A64FX 计算资源”，第2742卷，第1期，012019页，2024年。[25] J. D. McCalpin，“当前高性能计算机的内存带宽与机器平衡”，*IEEE TCCA* 通讯，第2卷，25，1995 19no. - 。[26] N. Binkert 等人，“gem5 模拟器”，*ACM SIGARCH* 计算机体系结构新闻，第39卷，第2期，1–7页，2011年。[27] F. Bellard，“QEMU：一种快速且可移植的动态翻译器”，载于 *USENIX* 年度技术会议，*FREENIX* 专题，第41卷，第46期。美国加利福尼亚州，2005年，10–5555页。[28] G. Bali 等人，“格点规范场系综与数据管理”，*PoS*，第 LATTICE2022 卷，203页，2023年。[29] F. G. Van Zee 等人，“BLIS 框架：可移植性实验”，*ACM*《数学软件汇刊》*(TOMS)*，第42卷，第2期，1–19页，2016年。[30] S. J. Pennycook、J. D. Sewall 和 V. W. Lee，“性能可移植性度量的影响”，《未来一代计算机系统》，第92卷，947–958页，2019年。[31] S. Browne 等人，“用于现代处理器性能评估的可移植编程接口”，《国际高性能计算与应用杂志》，第14卷，第3期，189–204页，2000年。[32] S. Long，“HiPC-2025-Paper253-Artifacts”，2025年10月。 [在线]。可访问：https://doi.org/10.5281/zenodo.17495916