arXiv:2304.04876v1 [math.NA] 10 Apr 2023

GPU 上两级Schwarz域分解预条件子的实验研究

Ichitaro Yamazaki†、Alexander Heinlein*、Sivasankaran Rajamanickam†
†美国新墨西哥州阿尔伯克基，桑迪亚国家实验室
*荷兰代尔夫特，代尔夫特理工大学

摘要——**广义 Dryja–Smith–Widlund (GDSW) 预条件子是一种两级重叠 Schwarz 域分解（DD）预条件子，它将经典的单级重叠 Schwarz 预条件子与能量最小化粗空间相结合。GDSW 预条件子用于加速 Krylov 子空间迭代方法的收敛时，可为求解由各类偏微分方程离散化产生的稀疏线性系统提供稳健性和可扩展性。本文介绍 FROSch（Fast and Robust Schwarz），这是一个域分解求解器软件包，可为 CPU 和 GPU 集群实现 GDSW 类型的预条件子。为提升求解器在 GPU 上的性能，我们采用一种新颖的分解策略，在每个 GPU 上运行多个消息传递接口进程，从而降低求解器的计算成本和存储成本，并有望改善收敛速度。借助这一策略，与仅使用 CPU 相比，我们使用 GPU 获得了具有竞争力甚至更快的性能。我们在配备 NVIDIA V100 GPU 的 Summit 超级计算机上展示了 FROSch 的性能，并使用 NVIDIA 多进程服务（ MPS ）实现了这一分解策略。**

该求解器具有多种算法和实现选择，这既为其 GPU 实现带来了机遇，也带来了挑战。我们针对不同的求解器选项开展了全面的实验研究，其中包括在GPU 上精确或非精确地求解局部重叠子域问题。我们还讨论了采用不完全LU分解的迭代变体和稀疏三角求解作为近似局部求解器，以及使用较低精度计算整个 FROSch 预条件子的影响。总体而言，使用 GPU 后，求解时间缩短了约 $2\times$ 倍，而数值设置时间的 GPU 加速效果取决于求解器选项和局部矩阵大小。

I. 引言

区域分解方法（DDM） [13], [26] 可用于构建一类高效的并行求解器，用于求解由偏微分方程离散化产生的稀疏线性方程组。在 DDM 中，全局问题被分解为较小的局部子问题，可并行处理。因此，DDM 预条件子非常适合在分布式内存计算机上求解大规模线性系统。然而，对于单层 DDM，求解收敛所需的迭代次数通常会随着子域数量的增加而增加。为此，引入了由精心设计的粗空间基函数确定的第二层粗网格系统。这样一来，预条件矩阵的条件数，以及所需的迭代次数，渐近地不再依赖于子域数量。本文研究了广义 Dryja–Smith–Widlund（GDSW） [11] 双层 Schwarz DDM 将经典的单层重叠加性 Schwarz 预条件子与能量最小化粗空间基函数相结合，已被证明在求解许多具有挑战性的问题时具有稳健性和可扩展性。

我们的重点是 GDSW 预条件子在 GPU 上的性能。GDSW 算法有多种算法和软件选项。每种选项都有多个可调参数，参数的良好选择可能取决于架构和问题。其中一些选项除了涉及实现选择外，还会对算法产生影响。例如，局部稀疏求解器的计算复杂度随局部矩阵大小的增长速度超过线性。由于通常 CPU 核心的数量多于 GPU，因此将全局域划分为每个 GPU 一个子域，而不是每个 CPU 核心一个子域，可能会增加局部子域求解器的复杂度。此外，较少的子域会形成较小的粗空间，这可能导致收敛性能下降。最后，DDM 求解器所依赖的许多内核，例如稀疏直接求解器、不完全分解、稀疏三角求解器和稀疏矩阵-矩阵乘法，都难以在 GPU 上优化。所有这些特性都给求解器的实现以及针对 GPU 架构的性能调优带来了挑战。因此，据我们所知，目前仍缺乏对两级 DDM 求解器进行全面的 GPU 性能研究，尤其是在大规模场景下。

为填补这一空白，我们研究了求解器软件包 FROSch（Fast and Robust Schwarz）在 GPU 上的性能。该软件包在 Trilinos 软件框架中实现了 GDSW 预条件子；参见 [18]。我们的实现还可通过同一套代码移植到不同的硬件架构。鉴于架构快速演变，软件栈的设计至关重要，必须确保求解器能够移植到不同的硬件架构（例如，将特定于硬件的代码和优化与高层求解器设计和实现隔离开来）。虽然这样可以避免针对每种新架构都重新编写求解器，但若要在特定架构（包括 GPU 集群）上获得高性能，就必须精心设计软件栈，并且可能需要采用新的算法变体。我们评估了对 GDSW 预条件子的 GPU 性能至关重要的多种算法和软件选项，其中包括为此新增的求解器功能，例如，

- 单次归约变体 [30] Krylov 求解器，每次迭代仅执行一次全局归约，

	used	效应	其他可用选项
Krylov 求解器	单次归约 GMRES	改进数据访问	标准型、流水线型、通信规避型
GDSW	双层 rGDSW	减少粗空间大小	标准型，多层
直接求解器	GPU 加速的 Tacho	允许使用 GPU	仅使用中央处理器，例如 SuperLU
稀疏三角求解	基于超节点的 Kokkos-Kernels	提高 GPU 利用率	基于元素的，分块逆
不精确求解器	迭代变体 FastILU/FastSpTRSV	展现更多并行性	CPU/GPU 上的标准型
精度	单精度 HalfPrecisionOp	减少数据量	统一双精度
子域数量	CPU 核心数量	降低求解器开销 并改善收敛性	GPU 数量

表 I：本文使用的求解器选项：每种求解器选项都有多个可调参数，例如 GMRES 的重启长度和正交化方案、rGDSW 的重叠大小、Tacho 的矩阵排序方案、FastILU 的层数，以及 Jacobi 的迭代次数和阻尼因子等主要参数。

- 基于 GPU 上直接稀疏 LU 分解求解局部重叠子域和粗空间问题的精确解 [21],
- 一种基于超节点的稀疏三角求解器 [28],它减少了内核启动次数，并利用 GPU 上的层次并行性，而且
- ILU 分解和稀疏三角求解器的迭代变体 [8]，其计算成本要高得多，但相比标准的基于代入的算法能展现出更高的并行性。

表 I 列出了本文将评估的一些主要求解器选项，以及我们根据经验和以往结果选定的其他参数。

在配备 NVIDIA V100 GPU 的 Summit 超级计算机上进行的实验结果表明，两级域分解求解器在 GPU 集群上具有应用潜力。我们将 GPU 性能与每个节点使用全部 CPU 核心时的 CPU 性能进行了比较。我们认为，这种比较既保守又公平，能够反映应用实际预期的性能。使用 GPU 后，三维弹性问题的求解时间缩短了约 $2\times$ 倍，而对数值求解器设置时间的影响则取决于求解器选项和局部矩阵大小。对于使用局部直接求解器的情况，单个线性方程组的总求解时间（数值设置时间与求解时间之和）缩短至原来的约 $1.1\times$ 至 $1.7\times$。如果应用需要对具有不同右端项的一系列线性方程组进行求解，则数值设置成本可以分摊到多次求解中，从而获得更接近 $2\times$ 的加速比。

本工作的主要贡献如下：

- 实现了双层 DDM 求解器的 GPU 版本，并对其进行了大规模 GPU 性能研究；
- 提出了一种新颖的分解策略，使得可以利用 NVIDIA 多进程服务（MPS）在每个 GPU 上运行多个 MPI 进程，并显著降低 DDM 求解器的计算成本和存储成本，还可能改善收敛性（第 VI 节）。据我们所知，尚无其他研究将 MPS 与可用于生产环境的线性求解器结合使用。在 Summit 上使用 MPS 的性能研究中，FROSch 的数值设置时间和求解时间最多可缩短 $3\times$ 倍。
- 对几种求解器选项的详细实验研究

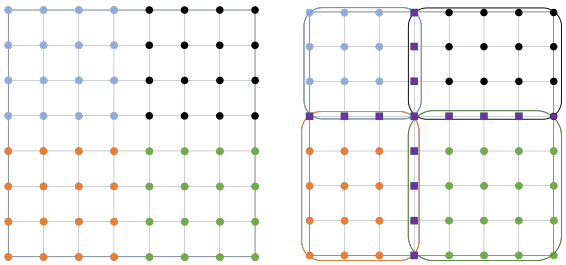
  对于包含直接、不完全和近似分解的两层 DDM，每种分解都采用了多种参数设置。
- 针对使用不精确或近似局部线性求解器的情况，或使用较低精度 DDM 预条件子的情况，开展了数值性能和 GPU 性能研究，从而能够求解规模更大的线性系统；典型 DDM 求解器通常只能求解较小规模的线性系统（其精确解采用双精度局部直接求解器，而典型的 GDSW 在实践及其理论中均以此为基础）。

II. 相关工作

随着 GPU 成为科学计算中的关键组成部分，已有多项工作致力于优化计算内核，而这些内核同样是 DDM 求解器所需的。另一方面，尚未有研究在大规模场景下展示使用这些内核以及其他内核的双层 DDM 求解器 GPU 实现。Luo 等 [22] 研究了单层 DDM 的 GPU 性能，其中 MPI 进程数受 GPU 数量限制。我们将表明，这种做法在性能和收敛性方面并非最优。通过采用一种分解方式，使每个 CPU 核心映射一个 MPI 进程，并使每个 GPU 映射多个 MPI 进程，可以提升求解器性能。此外，他们仅使用 GPU 加速局部子域求解器，该求解器基于平滑聚合多重网格，并采用稠密粗网格求解器。因此，GPU 仅执行稀疏矩阵-向量乘法、稠密向量更新和稠密三角求解，这些操作相对容易在 GPU 上并行化（该论文避免使用不完全LU分解，因为它难以在 GPU 上并行化）。

ˇ 与我们的工作同期，S´ıstek 和 Oberhuber 在 [31] 中使用 GPU 加速约束平衡域分解（BDDC）方法中的局部求解；具体而言，他们在 GPU 上对稠密局部舒尔补矩阵进行分解，并执行前代和回代。对于时变模拟，由于不同时间步之间可以重复使用分解结果，他们发现，仅将稠密矩阵存储在 GPU 上，相比仅使用 CPU，可获得高达 $5\times$ 的加速比。

本文将使用 DDM 求解器中更常用的核函数——稀疏直接分解和不完全分解，研究两级DDM 求解器的 GPU 性能。



(a) 原始问题。 (b) 非重叠分区。

图 1：局部 4×4 问题的区域分解。在图 1(b) 中，内部和界面分别用圆形和方形标记表示。

III. GDSW 算法

我们考虑线性方程组的求解，

$$Ax = b.$$

DDM 求解器已在理论和实践中得到广泛研究，研究对象是由椭圆型偏微分方程离散化产生的矩阵，但它们也已成功应用于许多其他问题 [13], [26]。

两级重叠加法 Schwarz 预条件子基于将全局区域 Ω 分解为 n_p 个不重叠子域 $\Omega_1, \dots, \Omega_{n_p}$。随后，将这些子域扩展 ℓ 层网格单元（或者网格节点），得到相应的重叠子域 $\Omega'_1, \dots, \Omega'_{n_p}$。预条件子定义为

$$M^{-1} = \Phi A_0^{-1} \Phi^T + \sum_{i=1}^{n_p} R_i^T A_i^{-1} R_i, \tag{1}$$

其中，R_i 是从全局域 Ω 到第 i 个重叠子域 Ω'_i 的限制算子，且 $A_i = R_i A R_i^T$。要构造稳健且高效的预条件子，关键在于粗基函数，即矩阵 Φ 的列，它们生成粗矩阵 $A_0 = \Phi A \Phi^T$。

GDSW [11] 型预条件子的粗基函数 Φ 是通过能量最小化扩展构造的，从非重叠 DD 的界面 Γ 延伸至子域的内部 I；分解示意见图 1，FROSch 中实现方式的讨论见 [18]。在本文讨论中，我们对全局矩阵 A 重新排序并划分为 2×2 分块结构

$$\begin{bmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma I} & A_{\Gamma\Gamma} \end{bmatrix},$$

使得索引 I 和 Γ 分别对应内部和界面自由度（dofs）。

设 R_Γ 为从全局自由度到界面自由度的限制算子，使得 $A_{\Gamma\Gamma} = R_\Gamma A R_\Gamma^T$，且 n_I 和 n_Γ 分别表示内部自由度和界面自由度的数量。然后，GDSW 粗基函数定义如下：

1) 界面 Γ 被划分为 n_c 个连通分量，$\Gamma_1, \dots, \Gamma_{n_c}$，这些分量可能存在重叠，并且 R_{Γ_i} 是从全局界面 Γ 到 Γ_i 的限制算子。

2）为了在界面上获得单位分解，同时考虑界面分解中各重叠部分的影响，我们引入对角缩放矩阵 D_{Γ_i}，

$$\sum_{i=1}^{n_c} R_{\Gamma_i}^T D_{\Gamma_i} R_{\Gamma_i} = I_\Gamma,$$

其中 I_Γ 是 Γ 上的单位矩阵。

3) 现在，为了获得稳健且高效的预条件子 M^{-1}，GDSW 类型预条件子的关键组成部分是 n-by-n_n 矩阵 Z，其列包含与 A 对应的全局 Neumann 矩阵的零空间。对于某些情况（例如，Laplace 问题只有一个常数列），该矩阵可以通过“代数”方式计算；而在某些应用中，零空间可能是显式可得的。

在第VIII 节中，我们给出了一个三维线性弹性问题的性能结果。对于该问题，零空间由（线性化的）刚体运动组成，即平移和线性化旋转。如 [16], 所述，线性化旋转无法通过代数方式直接获得；然而，当仅使用平移时，该方法仍可能表现良好。

4) 最后，给定零空间矩阵 Z，能量最小化粗基函数计算如下

$$\Phi = \begin{bmatrix} -A_{II}^{-1} A_{I\Gamma} \\ I \end{bmatrix} \Phi_\Gamma, \tag{2}$$

其中 Φ_Γ 是一个 n_I×($n_c n_n$) 矩阵，给出为

$$\Phi_\Gamma = [R_{\Gamma_1}^T \Phi_{\Gamma_1}, \dots, R_{\Gamma_{n_c}}^T \Phi_{\Gamma_{n_c}}]$$

并且，每个 n_I×n_n 矩阵 Φ_{Γ_i} 张成限制在第 i 个界面上的零空间，$D_{\Gamma_i} R_{\Gamma_i} (R_\Gamma Z)$。因此，$\Phi_\Gamma$ 的维度为 n_I×($n_n n_c$)，而 Φ 的维度为 n ×($n_n n_c$)。

(1) 中子域问题的计算易于并行化，因为在采用消息传递接口的分布式内存实现中，第 i 个子域 A_i 被分配给第 i 个消息传递接口进程，并可并行处理。扩展 (2) 也可以类似地并行化，因为 A_{II} 具有块对角结构，diag($A_{I_1 I_1}$, ..., $A_{I_{n_p} I_{n_p}}$)，其中 $A_{I_i I_i}$ 对应于第 i 个非重叠子域的内部部分。随着子域数量增加，GDSW 粗空间可以使预条件矩阵 AM^{-1} 的条件数以及迭代次数渐近地保持不变；例如，参见 [11]。

GDSW 有几个变体，包括：

- 随着子域和消息传递接口进程数量的增加，粗问题 A_0^{-1}最终会成为并行性能瓶颈。为缓解这一瓶颈，GDSW的一种“简化”变体（rGDSW）仅使用与顶点对应的粗基函数，而不使用与面或边对应的粗基函数；参见 [12], [15]。此外，还提出了多层方法，以递归方式将 GDSW 应用于粗问题；参见 [19]。

- 为了增强具有高度异质系数（可能存在较大跳跃）的问题的粗空间，“自适应”GDSW（AGDSW）通过增加由求解局部广义特征值问题所计算的附加分量来丰富粗空间；例如参见 [17]。

本文未使用自适应或多层变体，不过本研究的若干结果同样适用于这些变体。此外，GDSW 的行为已与其他两层 DDM 进行了广泛比较，而且许多 DDM 求解器使用相似的底层内核。因此，我们的实验研究或可为其他方法提供参考。

IV. FROSCH 软件

FROSch [18] 在 Trilinos 软件框架 [2], 中实现了 GDSW 类型的预条件子。Trilinos 是一组开源软件包，可用作构建模块，用于开发大规模科学应用。图 2 展示了用于求解线性方程组的 Trilinos 核心软件包。这些软件包可以组合起来，开发出灵活且可适应的大规模科学应用求解器。例如，FROSch 提供了与这些求解器软件包的接口，用于求解其局部重叠子域问题和粗问题：直接求解器（Amesos2 [5]）、非精确和预条件 Krylov 求解器（Ifpack2 [23]和 Belos [5]），以及局部代数多重网格求解器或 Schwarz 方法（MueLu [6]和 FROSch [18]）。此外，FROSch 还可用作 Belos 的预条件子。Belos 实现了 Krylov 求解器，其中包括可针对 GPU 架构进行优化的变体，例如单次归约、避免通信和流水线变体 [29]。

此外，FROSch 构建于Trilinos 软件栈之上，具体而言，构建于提供跨不同硬件架构的可移植性能的软件包之上：特别是，

- Kokkos [27] 是一个 C++性能可移植编程生态系统。它提供内存抽象，并支持将特定函数分派到 CPU 或 GPU 上特定执行空间，以执行并行操作。这使得使用单一代码库便能在不同众核架构上实现可移植的线程性能（前提是算法具有性能可移植性）。

- Kokkos-Kernels [24]是一个基于 Kokkos 的内核集合，用于在中央处理器和 GPU 上执行节点内稀疏或稠密矩阵以及图操作。

- Tpetra [1] 为中央处理器和 GPU 集群实现分布式图、矩阵和向量运算。

尽管我们聚焦于 FROSch 软件栈，但这并不是可移植性能软件栈的唯一选择。例如，先前的研究已经比较了节点内可移植层和单个内核的性能 [27]。

V. GPU 加速

当前许多高性能计算机由异构的计算节点架构组成，即每个节点由多核CPU 和多个 GPU加速器构成。具有大量计算核心和高内存带宽的 GPU 适用于高度并行的具有规则内存访问模式的计算，而某些操作更适合 CPU。

这为设计高性能稀疏线性求解器（包括 FROSch）带来了挑战与机遇。这些求解器所需的大多数计算内核都具有不规则的内存访问模式，且计算量与数据访问量之比较低。因此，其性能瓶颈往往不在计算，而在内存带宽，甚至可能受限于内存延迟。为了充分利用 GPU，必须精心设计求解器及其底层计算内核，并且可能需要开发新的算法变体。本节将讨论在 GPU 集群上提升 FROSch 性能所采用的一些具体方法。

A. 软件方面的考虑

1) 软件结构：在许多科学与工程模拟中，对于具有相同稀疏结构的给定网格，我们往往需要多次进行数值分解；或者需要求解一系列具有不同右端向量的线性方程组。因此，Trilinos 中的所有线性求解器都包含三个不同的阶段：

(a)符号分解，给定稀疏结构或图后，执行所有符号分析和分解，并分配所需的 GPU 内存。此处会执行 LU 分解的符号分析、三角求解的层级集计算等操作。通常在 CPU 上完成。

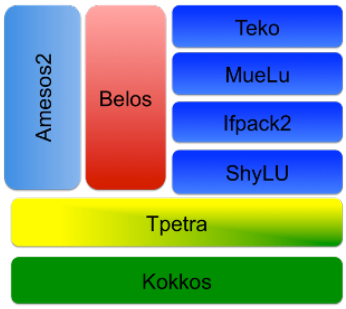
(b)数值分解，给定输入矩阵的数值后，执行数值分解；在 FROSch中，这一部分包括粗网格基函数的计算、粗空间矩阵的计算，以及重叠的局部子域矩阵和粗网格矩阵的分解。用于计算 A_0的稀疏矩阵-稀疏矩阵乘法、LU 分解或不完全分解的数值分解等步骤也属于此步骤。适当时，我们在 GPU 上执行这些计算。

(c)求解阶段，给定右端向量，计算线性方程组的解。作为该阶段的一部分，局部矩阵的直接分解或不完全分解的稀疏三角求解在 GPU 上完成。

这些不同阶段至关重要，尤其是对于 GPU 而言，因为符号分析的大部分工作难以并行化，而且 GPU 内存分配可能需要相当长的时间。

2）低精度预条件： 在 Trilinos 中，软件包 Belos 实现了 Krylov求解器。它使用 Operator类来应用预条件子，包括代数多重网格（AMG）和区域分解（DD）预条件子，分别由 MueLu 和 FROSch 软件包实现。这些预条件子通常由一种称为 CrsMatrix 的稀疏矩阵类构建。

开发了一种新的实用函数，可将 CrsMatrix 对象转换为精度减半的新对象



(a) Trilinos 求解器软件栈。

灵活求解器接口	
ShyLU	分布式DD预条件器（FROSch）和 基于节点内分解的局部求解器（Basker、Tacho）
MuLue	代数多重网格求解器
Amesos2	直接求解器接口（例如 KLU、PaRDISO、SuperLU、Tacho）
Belos	Krylov求解器（例如共轭梯度法、GMRES、双共轭梯度法，以及 其避免通信型或流水线型变体）
Ifpack2	代数预条件子（不完全LU分解、松弛法、单层施瓦茨法）
可移植性能	
Tpetra	分布式稀疏/稠密矩阵-向量运算
Kokkos-Kernels	具备性能可移植性的节点内图和稀疏/稠密矩阵运算
Kokkos	面向性能可移植应用的 C++ 编程模型 适用于不同的节点架构（例如，CPU、NVIDIA/AMD GPU）

（b）线性求解器软件包描述。

图 2：Trilinos 线性求解器软件包

（例如，如果原始矩阵采用双精度，那么新矩阵将采用单精度），从而使用户也能以半精度构造预条件子。此外，还实现了新的 HalfPrecisionOperator 类。该类继承工作精度下的基类 Operator 类，并将半精度下的算子作为成员变量在内部保存。将此新算子应用于向量时，它会在内部先将输入向量类型转换为半精度，再应用半精度下的算子（例如预条件子），然后将生成的输出向量类型转换回工作精度。尽管类型转换会带来开销，这些新功能仍使用户能够在当前的 Trilinos框架中以半精度应用许多已有的 Trilinos 预条件子。目前，MueLu 和 FROSch（其应用成本通常远高于向量类型转换的成本）已扩展为支持使用这些新功能

B. 算法考量

1) 稀疏直接矩阵分解：域分解求解器的大多数理论结果（包括预条件矩阵的条件数估计）都假设重叠子域和粗问题能够得到精确求解。因此，域分解求解器在实践中通常使用稀疏直接求解器。

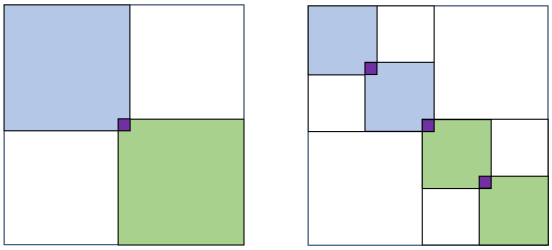
稀疏直接求解器是许多科学应用中的关键组件，人们已投入大量精力开发高性能稀疏直接求解器 [9]。在我们的实验中，使用了 SuperLU 和 Tacho 软件包，它们提供了两种不同的稀疏直接求解方法：

- SuperLU [10] 实现了采用部分选主元的左看型稀疏 LU 分解。它主要面向单个 CPU 核心，不过也可以链接到多线程 BLAS 或 LAPACK，以便在多核 CPU 上运行。它利用 LU 因子的超节点块结构来充分利用存储层次结构。
- Tacho [21] 基于仅在波前矩阵内部进行主元选择的多波前分解。原版 Tacho 使用 Kokkos 的 任务 编程模型。尽管当前实现仍使用 Kokkos，但它通过结合层集调度和团队级 BLAS/LAPACK 类内核，充分利用 GPU 上可用的层次并行性。它还提供了调用供应商优化内核的接口（即 NVIDIA 的 CuBLAS/cuSolver 和 AMD 的 rocBLAS/rocSolver），以通过 GPU 流对大型波前矩阵进行分解。目前，Tacho 分别支持对对称正定矩阵、对称不定矩阵以及具有对称模式的数值非对称矩阵进行 Cholesky、LDL^T 或 LU 分解。

2) 稀疏三角求解：使用稀疏直接矩阵分解后，得到的稀疏三角矩阵通常包含称为超节点的稠密块。利用这种超节点结构可以加速三角求解。例如，Kokkos-Kernels 实现了一种基于超节点块层集调度的稀疏三角求解器 [28]。在 GPU 上，与矩阵元素相比，以块为单位进行处理可能带来多项性能优势。例如，这可以减少层集树的高度，以及稀疏三角求解并行执行中的关键路径长度（例如内核启动次数）。此外，这种方法还支持层次并行化，与 GPU 上可用的层次并行性相契合，并可通过 Kokkos-Kernels 中基于团队的内核加以利用。

Kokkos-Kernels 中的稀疏三角求解器还提供了执行分块逆的选项 [3]，可将稀疏三角求解转换为一系列稀疏矩阵-向量乘法，从而比标准型的基于代入的算法提供更多并行性 [28]。

3）不完全局部求解器：尽管 DD 理论基于对局部子域和粗问题的精确求解，但非精确局部求解器在实践中也可能表现良好，尤其是在其应用效果与使用精确求解器的效果达到一定程度的谱等价时。本文探讨了基于层的稀疏不完全 LU 分解的非精确局部求解器。尽管已有多种并行 ILU 实现，但 ILU 和稀疏三角求解的标准并行化方案基于层集调度 [4]。在 Trilinos 中，这些方案由 Kokkos Kernels 中的 SpILU 和 SpTRSV 实现 [24]。



(a) 两个子域。 (b) 四个子域。

图 3：将域重叠分解为两个或四个子域（例如，在两个GPU 上每个 GPU 分配一个或两个 MPI进程）。我们不再将两个大型子域分配给两个 GPU（左图），而是将四个小型子域分配给两个 GPU（右图）。

尽管不完全分解可以减少填充并展现更多并行性，但仍可能无法提供足够的并行性来充分利用 GPU。为展现更多并行性，研究者提出了稀疏近似分解和稀疏三角求解器的迭代变体 [8]。这些方法使用 Jacobi 迭代来近似 LU 因子的各个元素，或使用稀疏三角矩阵近似求解线性方程组。尽管每次迭代所需的浮点运算数量与标准算法大致相同，但这种变体的并行性显著更高。因此，当求解器只需少量迭代即可达到所需的近似精度（我们的默认设置是两种求解器均迭代五次）时，它在 GPU 上可能显著缩短求解时间。在 Trilinos 中，这些方法由 FastILU 和 FastSpTRSV 实现 [7]。

VI. 讨论

当前的异构节点架构通常在每个节点上配备的 CPU 核心多于 GPU（例如，Summit 超级计算机的每个节点都配备了 42 个 IBM Power9 CPU 核心和 6 个 NVIDIA V100 GPU）。这种异构节点架构给在节点间比较 DD 求解器在 CPU 上和 GPU 上的性能带来了挑战。这是因为在大多数情况下，局部稀疏求解器的计算复杂度随局部矩阵大小的增长速度超过线性。举例来说，对于一个三维问题，当使用嵌套剖分 [14]对维度为 n_i的局部矩阵进行置换时，局部问题的稀疏直接分解及其对应的稀疏三角求解的计算复杂度通常分别为 $\mathcal{O}(n_i^2)$ 和 $\mathcal{O}(n_i^{4/3})$。因此，在我们的强并行扩展性研究中，DD 求解器的计算成本会随着 MPI 进程数的增加而超线性下降。由于每个节点上的 GPU 通常少于 CPU 核心，如果 CPU 运行时在每个 CPU 核心上启动一个 MPI 进程，而 GPU 运行时在每个 GPU 上启动一个 MPI 进程（这是实践中最常见的设置），那么与 GPU 运行相比，CPU 运行中的每个进程所承担的计算成本要小得多。

此外，使用 GDSW 预条件子进行预条件的矩阵，其条件数满足以下有界性：

$$\kappa\left(M_{\mathrm{GDSW}}^{-1}A\right) \leq C\left(1+\frac{H}{\delta}\right)\left(1+\log\left(\frac{H}{h}\right)\right)^2;$$

其中， H 是子域的最大直径， δ 是重叠宽度，参见 [11]。因此，子域尺寸（H/h）越小，条件数越低；例如，在问题规模固定时，子域数量增加时便是如此。

在我们的实验中，我们使用 NVIDIA多进程服务（MPS）在每个 GPU 上运行多个消息传递接口进程（参见图 3）。与每个GPU 上仅运行一个进程相比，这不仅降低了域分解求解器的计算和存储成本，还可能改善预条件矩阵的条件数，从而提高 Krylov 求解器的收敛速度。通过在每个消息传递接口中使用多个子域并采用GPU 流，也可以获得相同的分解。然而，对于两层求解器而言，这将需要重大的算法创新和软件投入。尽管这种方式可能不是最优的，MPS使我们能够在无需进行这些代码更改的情况下运行现有代码，并且可以在每个 GPU 上运行多个子域时显著提升性能。

VII. 实验设置

对于本文中प्रस्तुत的所有性能结果，我们使用了具有一个代数重叠的“缩减”GDSW粗空间。然后，作为我们的Krylov求解器，使用了广义最小残量（GMRES）方法的单次归约变体 [30] [25]，该方法是用于求解非对称线性方程组的常用 Krylov 方法。我们将重启长度设置为 30，并在残差范数降低10^{-7}倍时认为 GMRES 已收敛。最后，本文重点研究了三维弹性问题。尽管 FROSch 中还有其他几种预条件选项，但由于我们的重点是性能比较，而不是对 GDSW 进行数值研究，因此这些设置能够代表 FROSch 的性能。

我们在橡树岭领导力计算设施的 Summit 超级计算机上展示性能结果。Summit 的每个节点配备 42 个 IBM Power9 CPU 核心和 6 个 NVIDIA V100 GPU。除非另有说明，CPU 测试中，我们在每个节点上启动 42 个消息传递接口进程（每个 CPU 核心对应一个消息传递接口进程）；GPU 测试中，我们使用 NVIDIA 多进程服务（MPS），在每个 GPU 上最多运行 7 个消息传递接口进程（每个节点最多运行 42 个消息传递接口进程）。代码使用 CUDA 10.2.89 和 GCC 7.5.0 编译，并链接到供应商优化的库：NVIDIA CUBLAS、GPU 上的 CuSparse，以及 IBM 工程与科学子程序库（ESSL）6.3。

VIII. 性能结果

A. 精确局部求解器

在本节中，我们研究了 FROSch 在局部重叠子域和粗空间问题中使用精确解时的性能。我们使用 Metis [20] 提供的嵌套剖分排序来减少 LU 中的填充数

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU		**2.03 (75)**	**2.07 (69)**	**1.87 (61)**	**1.95 (58)**	**2.48 (69)**
GPU	n_p/GPU = 1	1.43 (47)	1.52 (53)	2.82 (77)	2.44 (68)	2.61 (75)
	2	1.03 (46)	1.36 (65)	1.37 (60)	1.52 (65)	1.98 (86)
	4	0.93 (59)	0.91 (53)	0.98 (59)	1.33 (77)	1.21 (66)
	6	0.67 (46)	0.99 (65)	0.92 (57)	0.91 (57)	0.95 (57)
	7	**1.03 (75)**	**1.04 (69)**	**0.90 (61)**	**0.97 (58)**	**1.18 (69)**
加速比		**2.0×**	**2.0×**	**2.1×**	**2.0×**	**2.1×**

(a) SuperLU。

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU		**1.60 (75)**	**1.63 (69)**	**1.49 (61)**	**1.51 (58)**	**1.90 (69)**
GPU	n_p/GPU = 1	1.17 (47)	1.37 (53)	1.92 (77)	1.78 (68)	2.21 (75)
	2	0.79 (46)	1.14 (65)	1.05 (60)	1.18 (65)	1.70 (86)
	4	0.85 (59)	0.81 (53)	0.78 (59)	1.22 (77)	1.19 (66)
	6	0.60 (46)	0.86 (65)	0.75 (57)	0.84 (57)	0.91 (57)
	7	**0.99 (75)**	**0.93 (69)**	**0.82 (61)**	**0.93 (58)**	**1.22 (69)**
加速比		**1.6×**	**1.8×**	**1.8×**	**1.6×**	**1.6×**

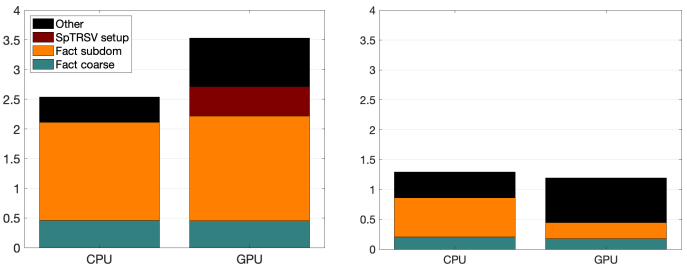
(b) Tacho。

表 II：Summit 上弱尺度三维弹性问题的总迭代时间（秒）和迭代次数。

因子，同时也可以暴露出更多并行性。我们分别使用 SuperLU 或 Tacho，在 CPU 或 GPU 上对局部矩阵和粗矩阵进行分解。为了在 GPU 上使用由 SuperLU 计算得到的 LU 因子来应用预条件子，我们采用了 Kokkos-Kernels 中的超节点稀疏三角求解器 [28]；而在 CPU 上，我们使用 SuperLU 的内部三角求解器，因为 Kokkos-Kernels 求解器旨在利用众核架构，并不适用于单个 CPU 核心。对于 Tacho，我们在 CPU 和 GPU 运行中均使用其内部稀疏三角求解器。我们没有使用分块逆，并且所有在 CPU 或 GPU 上运行的稀疏三角求解器在数值上都是等价的。

表 II 展示了使解收敛所需总迭代时间的弱扩展并行性，其中每个计算节点上的局部问题规模保持不变，而全局矩阵规模随计算节点数量线性增长。我们使用 MPS 在每个 GPU 上运行多个消息传递接口进程。正如我们在第 VII 节中讨论的那样，对于局部重叠子域矩阵进行直接分解后，所得的稀疏三角求解的计算成本相对于局部矩阵的大小呈超线性扩展。因此，当我们使用 *MPS* 将更多消息传递接口进程映射到每个 *GPU* 上时，局部子域会变得更小，迭代求解时间也会显著缩短（加速比为 $1.3 \sim 2.7\times$）。总体而言，与 CPU 运行相比，使用 GPU 可将求解时间缩短约 $2\times$ 倍。

在某些应用中，设置时间也可能占据总模拟时间的很大一部分。因此，我们现在研究 FROSch 的数值设置时间。图 4 展示了 Summit 的单个计算节点上数值设置时间的细分情况。正如预期的那样，尤其是在 CPU 上，数值设置时间的很大一部分由稀疏直接求解器耗费。对于使用 SuperLU 的 CPU 和 GPU 运行，局部重叠矩阵和粗网格矩阵均在 CPU 上进行分解，并且在 CPU 上和使用 GPU 时的分解时间相同。



(a) SuperLU (b) Tacho

图 4：Summit 单个节点上的数值设置时间细分（42 个消息传递接口进程上的 $n = 375$K）。

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU		**2.5**	**3.0**	**3.3**	**3.8**	**3.6**
GPU	n_p/GPU = 1	60.0	71.5	70.9	85.6	85.1
	2	22.9	22.3	26.1	26.9	25.8
	4	8.4	9.4	9.1	9.8	10.2
	6	5.5	5.2	5.2	6.2	6.3
	7	**3.5**	**4.2**	**4.8**	**5.4**	**5.4**
性能下降		**1.4×**	**1.4×**	**1.5×**	**1.4×**	**1.5×**

(a) SuperLU。

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU		**1.3**	**1.6**	**1.7**	**1.8**	**1.9**
GPU	n_p/GPU = 1	3.2	3.5	3.9	5.4	5.6
	2	2.1	2.3	2.9	3.3	3.4
	4	1.4	1.8	2.1	2.2	2.3
	6	1.5	1.6	1.7	2.0	2.3
	7	**1.2**	**1.6**	**1.7**	**2.0**	**2.2**
减速		**0.9×**	**1.0×**	**1.0×**	**1.1×**	**1.1×**

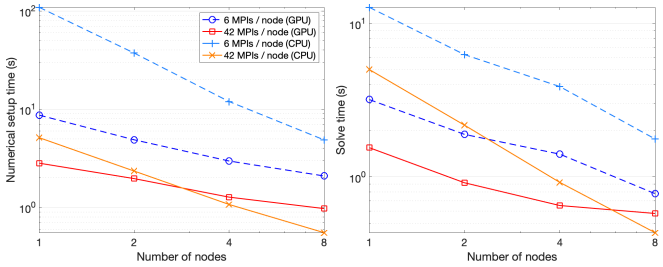
(b) Tacho。

表 III：Summit 上弱尺度三维弹性问题的数值设置时间（秒）。每个 GPU 行之间的 GPU 消息传递接口进程数从 1 变为 7。这使得 GPU 运行时 SuperLU 和 Tacho 的数值设置时间分别最多缩短 17 倍和 3 倍。在此阶段，GPU 运行速度略慢于中央处理器运行。

GPU。另一方面，Tacho 能够利用 *GPU*，因此 *GPU* 运行时的局部分解时间缩短了$2.4\times$。这是使用 GPU 的第一个优势。遗憾的是，我们还发现，除稀疏直接求解器之外的一些设置时间在 GPU 上运行得更慢（柱状图的“黑色”部分）[1]，，而且使用 SuperLU 设置 Kokkos-Kernels 稀疏三角求解也耗费了大量时间：

- SuperLU 在数值分解过程中执行部分选主元。这保证了求解器的数值稳定性，但 LU 因子的稀疏结构取决于数值。因此，符号分解中几乎没有可复用的工作。例如，使用 SuperLU 时，Kokkos-Kernels

[1]这主要是由于形成粗网格矩阵时需要进行稀疏矩阵乘法，以及形成局部重叠子域矩阵时需要进行通信。



(a) 数值设置时间。 (b) 求解时间。

图 5：三维弹性问题（$n = 1$M）的强并行扩展性。

稀疏三角求解器的符号设置和数值设置都需要在每次数值分解后执行，这构成了图中 CPU 与 GPU 设置时间差异的重要部分。

- 另一方面，Tacho 仅在其波前矩阵内进行主元选择。对于具有相同稀疏结构的输入矩阵，LU 因子的稀疏结构保持不变，因此我们可以在对不同矩阵进行数值分解时重用符号设置。此外，Tacho 可以利用 GPU。总体而言，我们看到 Tacho 在 CPU 和 GPU 上的数值设置时间相近。

表 III 比较了使用最多 672 个 CPU 核心和 96 个 GPU 时 FROSch 的弱扩展数值设置时间。正如我们在第 VII 节中讨论的，分解局部重叠子域的计算成本会随局部矩阵的规模呈超线性增长。因此，与迭代求解时间类似，使用 MPS 可显著缩短数值设置时间（使用 SuperLU 时加速比为 $15 \sim 17\times$，使用 Tacho 时加速比为 $2 \sim 3\times$）。在每个 GPU 上运行多个消息传递接口进程也减少了存储 LU 因子所需的内存，从而能够求解更大的线性方程组。

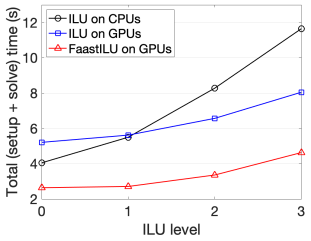
总体而言，使用Tacho 时，总求解时间（设置和求解一个单精度线性系统所需时间之和）在*GPU*上$1.1 \sim 1.8\times$ 更快。如果应用程序需要连续求解一系列线性系统（相同的矩阵 A，但按顺序使用不同的右端项 b），那么数值分解的成本可以在多次求解之间摊销，从而可以获得更接近 $2\times$ 的加速比。

总结我们对精确求解器的研究，图5展示了强并行扩展性结果，其中每个节点使用了6个或42个消息传递接口进程。对于每个节点使用6个消息传递接口进程的中央处理器运行，我们将Tacho与线程版ESSL链接，并为每个消息传递接口进程使用7个线程。我们清楚地看到了，在中央处理器和GPU运行中，每个节点使用42个消息传递接口进程的优势。总的来说，只要局部矩阵大小足够大，*GPU*就能在设置和求解时间方面提供加速比。

B. 近似局部求解器

1）不完全 *LU* 分解：现在，我们研究使用不完全 LU（ILU）分解作为局部求解器对 FROSch 性能的影响。在这些实验中，

ILU层级		0	1	2	3
CPU	No	**1.5**	**1.9**	**3.0**	**4.8**
	ND	1.6	2.6	4.4	7.4
GPU	KK(No)	1.4	1.5	1.8	2.4
	KK(ND)	1.7	2.0	2.9	5.2
	Fast（No）	**1.5**	**1.6**	**2.1**	**3.2**
	Fast（ND）	1.5	1.7	2.5	4.5
加速比		**1.0×**	**1.2×**	**1.4×**	**1.5×**



(a) 设置（左侧表格）和总时间（右侧图）。

ILU层级		0	1	2	3
CPU	No	**2.55 (158)**	**3.60 (112)**	**5.28 (99)**	**6.85 (88)**
	ND	4.17 (227)	5.36 (134)	6.61 (105)	7.68 (88)
GPU	KK(No)	3.81 (158)	4.12 (112)	4.77 (99)	5.65 (88)
	KK(ND)	2.89 (227)	4.27 (134)	5.57 (105)	6.36 (88)
	Fast（No）	**1.14 (173)**	**1.11 (141)**	**1.26 (134)**	**1.43 (126)**
	Fast（ND）	1.49 (227)	1.15 (137)	1.10 (109)	1.22 (100)
加速比		**2.2×**	**3.2×**	**4.3×**	**4.8×**

(b) 求解时间（迭代次数）。

表 IV：在一个 Summit 节点上（$n = 648$K，使用 42 个消息传递接口进程），采用局部 Kokkos-Kernels 不完全 LU 分解（KK）或 FastILU（Fast），以及不重排序（No）或嵌套剖分（ND）时，FROSch 在三维弹性问题上的性能。

我们仅将基于层次的不完全LU分解(k)用作求解局部重叠子域问题的局部求解器，而 Tacho 则用于计算基函数和求解粗问题。非精确求解器降低了所需的存储成本，因此，与第VIII-A节求解的线性系统相比，本节求解的线性系统规模更大。

表IVa展示了不完全LU分解层级数量k对数值设置时间的影响。除了SpILU和SpTRSV（基于层集调度）之外，我们还展示了它们的迭代变体FastILU和FastSpTRSV的性能，其中我们分别进行了三次和五次雅可比迭代。随着层级的增加（以及计算不完全LU分解因子所需的计算量增加），使用GPU进行数值设置时间所获得的加速比也随之提高。

表IVb展示了随着不完全LU分解层级增加而变化的总迭代时间。FastILU计算不完全LU分解因子的近似值，而FastSpTRSV近似地求解三角系统。因此，与SpILU相比，使用FastILU时，GMRES需要更多次迭代才能收敛。然而，它们提供了更多的并行性，GPU可以加以利用。总体而言，使用迭代变体时，*GMRES*具有最快的求解时间，加速比为 $2.8 \sim 4.4\times$。

最后，表V展示了使用非精确 ILU(1) 局部求解器、在最多 672 个 CPU 核心和 96 个 GPU 上得到的弱扩展结果。在所有这些实验中，我们使用了原始的矩阵排序方式，因为矩阵重排序并未显著提升性能，反而可能增加迭代次数。即使使用非精确局部求解器，迭代次数也几乎与子域数量无关。从表V可以看出，即使迭代次数更高，非精确（Fast）选项仍比精确三角求解（KK）更快。

# 计算节点		1	2	4	8	16
矩阵大小		648K	1.2M	2.6M	5.2M	10.3M
CPU		**1.9**	**2.2**	**2.4**	**2.4**	**2.6**
GPU	KK	1.4	2.0	2.2	2.4	2.8
	Fast	**1.5**	**2.2**	**2.3**	**2.5**	**2.8**
加速比		**1.3×**	**1.0×**	**1.0×**	**1.0×**	**0.9×**

（a）设置时间（秒）。

# 计算节点		1	2	4	8	16
矩阵大小		648K	1.2M	2.6M	5.2M	10.3M
CPU		**4.0 (119)**	**3.8 (110)**	**3.7 (105)**	**3.3 (97)**	**4.1 (109)**
GPU	KK	4.3 (119)	3.9 (110)	4.8 (105)	4.3 (97)	4.9 (109)
	Fast	**1.2 (154)**	**1.0 (133)**	**1.1 (130)**	**1.3 (117)**	1.6 (131)
加速比		**3.3×**	**3.8×**	**3.4×**	**2.5×**	**2.6×**

(b) 求解时间（秒）（迭代次数）。

表V：使用 ILU(1) 作为局部子域求解器时，Summit 上 3D 弹性问题的弱扩展并行性能（每个节点 42 个 MPI 进程）。

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU	双精度	2.5	3.0	3.3	3.8	3.6
	单精度	1.8	2.2	2.3	2.6	2.6
	加速比	1.4×	1.4×	1.4×	1.5×	1.4×
GPU	双精度	3.5	4.2	4.8	5.6	5.4
	单精度	2.6	3.2	3.4	4.0	4.0
	加速比	1.3×	1.3×	1.4×	1.4×	1.4×

(a) SuperLU。

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU	双精度	1.3	1.6	1.7	1.8	1.9
	单精度	1.0	1.2	1.3	1.4	1.4
	加速比	1.3×	1.3×	1.3×	1.3×	1.4×
GPU	双精度	1.2	1.6	1.7	2.0	2.2
	单精度	1.0	1.3	1.4	1.7	2.0
	加速比	1.2×	1.2×	1.2×	1.2×	1.1×

(b) Tacho.

表VI：使用单精度或双精度FROSch 的数值设置时间（以秒为单位）：Summit 上的弱尺度三维弹性问题。

使用 *GPU* 上的迭代变体，我们获得了 $3.1 \sim 4.4\times$ 加速比。我们还观察到，使用 *GPU* 可获得 $2.5 \sim 3.8\times$ 加速比。在多个节点上使用 *ILU(1)* 时，*CPU* 和 *GPU* 的设置时间几乎相同。

2) 单精度 *FROSch*：尽管典型的科学应用要求达到双精度精度，但一些新兴硬件能够以更高性能执行低精度算术。还有一些机器在双精度和单精度算术下提供相同的性能。即便如此，使用低精度算术也能减少所需的数据传输量。由于稀疏求解器的性能往往受内存带宽限制，仅减少所需的通信量就可能缩短求解器运行时间。

表VII展示了性能结果，其中使用单精度 FROSch 为双精度 GMRES 进行预条件。对于这些特定问题，设置时间

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU	双精度	2.03 (75)	2.07 (69)	1.87 (61)	1.95 (58)	2.48 (69)
	单精度	1.89 (76)	1.60 (69)	1.71 (62)	1.75 (58)	2.37 (69)
	加速比	1.0×	1.3×	1.1×	1.1×	1.0×
GPU	双精度	1.03 (75)	1.04 (69)	0.90 (61)	0.97 (58)	1.18 (69)
	单精度	1.01 (75)	1.03 (69)	0.98 (62)	1.10 (58)	1.28 (69)
	加速比	1.0×	1.0×	0.9×	0.9×	0.9×

(a) SuperLU。

# 计算节点		1	2	4	8	16
矩阵大小		375K	750K	1.5M	3M	6M
CPU	双精度	1.60 (75)	1.63 (69)	1.49 (61)	1.51 (58)	1.90 (69)
	单精度	1.11 (76)	1.13 (69)	1.02 (62)	1.04 (58)	1.30 (69)
	加速比	1.4×	1.4×	1.4×	1.4×	1.5×
GPU	双精度	0.99 (75)	0.93 (69)	0.82 (61)	0.93 (58)	1.22 (69)
	单精度	1.00 (75)	0.92 (69)	0.84 (62)	0.93 (58)	1.21 (69)
	加速比	1.0×	1.0×	1.0×	1.0×	1.0×

(b) Tacho。

表VII：在 Summit 上针对弱扩展三维弹性问题，使用双精度或单精度 FROSch 时的总迭代时间和迭代次数。

通过使用单精度FROSch而缩短，同时收敛所需的迭代次数与双精度用例达到相同精度时保持不变。具体而言，在 *672* 个 *CPU* 核心和 *96* 个 *GPU* 上使用单精度时，我们观察到基于 SuperLU 的设置加速比为 $1.3 \sim 1.5\times$，而基于 Tacho 的设置加速比为 $1.1 \sim 1.4\times$。使用单精度并未带来求解时间方面的收益（表VII）。然而，使用单精度或双精度预条件子时，*GMRES* 收敛所需的迭代次数相近。

C. 关键结果总结

每个 GPU 使用多个子域可显著提升性能。就使用直接分解作为子域求解器时的设置时间而言，Tacho 中基于 GPU 的分解相较于基于中央处理器的 SuperLU 具有明显优势。使用 SuperLU 与 Kokkos Kernels 的组合或使用 Tacho 时，GPU 上的求解时间并无明显优势。不完全分解使我们能够求解更大规模的问题。尽管迭代次数与设置时间和求解时间之间存在权衡，因此相较于直接分解，性能并无显著差异，但使用 GPU 缩短了求解时间。即使迭代次数增加，迭代不完全分解和三角求解带来的加速比也并不显著，优于标准型不完全分解。采用较低精度计算使我们能够求解规模更大的线性方程组，并缩短了设置时间，但未缩短求解时间。

IX. 结论

我们介绍了 FROSch，它在 Trilinos 软件框架中实现了适用于 GPU 集群的 GDSW 算法。我们在 Summit 超级计算机上使用 NVIDIA V100 GPU 获得的性能结果展示了 FROSch 的潜力：使用 GPU 时，数值设置时间与

使用 CPU 时大致相同，而求解时间可缩短约 $2\times$ 倍。我们进行了全面的实验研究，考察了多种求解器选项，包括两种直接求解器、采用不同填充级别和不同排序方式的不完全分解技术、非精确分解，以及使用低精度算术。虽然我们只展示了使用 NVIDIA GPU 的性能结果，但通过使用 Kokkos，我们的实现也可移植到其他 GPU。我们计划研究 FROSch 在 AMD GPU 上的性能。

致谢

作者谨此感谢 Christian Glusa 协助将 HalfPrecisionOperator 集成到 FROSch中。本工作部分得到了美国能源部科学办公室高级科学计算研究办公室的支持；该支持通过 FASTMath研究所的科学计算高级计算发现（SciDAC）计划提供，合同编号为 DE-AC02-05CH11231，项目实施地点为桑迪亚国家实验室。桑迪亚国家实验室是一所多任务实验室，由Sandia National Laboratories 的全资子公司 Honeywell International, Inc. 旗下的 National Technology andEngineeringSolutions of Sandia,LLC 管理和运营；该实验室依据合同 DE-NA-0003525 为美国能源部国家核安全管理局服务。本文介绍的是客观的技术成果与分析。文中可能表达的任何主观看法或意见并不一定代表美国能源部或美国政府的观点。

参考文献

[1] *Tpetra* 并行线性代数。https://trilinos.github.io/tpetra.html。[2] *Trilinos* 项目网站。https://trilinos.github.io。[3] F. L. Alvarado、A. Pothen 和 R. Schreiber。高度并行的稀疏三角求解。载于图论与稀疏矩阵计算：*IMA* 数学及其应用丛书，第 56 章，第 141–157 页。Springer，纽约州纽约市，1993 年。[4] E. Anderson 和 Y. Saad。在并行计算机上求解稀疏三角线性系统。国际高速计算期刊，1：73–95，1989 年。[5] E. Bavier、M. Hoemmen、S. Rajamanickam 和 H. Thornquist。Amesos2 和 Belos：大型稀疏线性系统的直接与迭代求解器。科学编程，第 20 卷，2012 年。[6] L. Berger-Vergiat、C. A. Glusa、J. J. Hu、M. Mayr、A. Prokopenko、C. M. Siefert、R. S. Tuminaro 和 T. A. Wiesner。MueLu 用户指南。技术报告 SAND2019-0537，桑迪亚国家实验室，2019 年。[7] E. G. Boman、A. Patel、E. Chow 和 S. Rajamanickam。FastILU：细粒度异步迭代不完全LU分解。技术报告 2016-12066C 649521，桑迪亚国家实验室，2016 年。在 2016 年 SIAM 应用线性代数会议上作报告。[8] E. Chow 和 A. Patel。细粒度并行不完全 LU 分解。*SIAM* 科学计算期刊，37(2)：C169–C193，2015 年。[9] T. A. Davis、S. Rajamanickam 和 W. M. Sid-Lakhdar。稀疏线性系统直接法综述。数值学报，25 566，2016 383：–。[10] J. W. Demmel、S. C. Eisenstat、J. R. Gilbert、X. S. Li 和 J. W. H. Liu。稀疏部分选主元的超节点方法。*SIAM* 矩阵分析与应用期刊，20(3)：720–755，1999 年。[11] C. R. Dohrmann、A. Klawonn 和 O. B. Widlund。用于重叠 Schwarz 预条件子的能量最小化粗空间族。载于科学与工程中的区域分解方法 *XVII*，第 247–254 页。Springer Berlin Heidelberg，2008 年。[12] C. R. Dohrmann 和 O. B. Widlund。关于区域分解算法小型粗空间的设计。*SIAM* 科学计算期刊，39(4)：A1466–A1488，2017 年。

[13] V. Dolean、P. Jolivet 和 F. Nataf。《区域分解方法导论：算法、理论与并行实现》。工业与应用数学学会，2015 年。[14] A. George。规则有限元网格的嵌套剖分。SIAM 数值分析期刊，10(2)：345–363，1973 年。[15] A. Heinlein、C. Hochmuth 和 A. Klawonn。针对不可压缩流体流动问题的整体 Schwarz 区域分解方法的降维 GDSW 粗空间。国际数值方法工程期刊，121(6)：1101–1119，2020 年。[16] A. Heinlein、C. Hochmuth 和 A. Klawonn。用于弹性问题的全代数两级重叠 Schwarz 预条件子。载于《数值数学与高级应用 ENUMATH 2019》，第 531–539 页。Springer International Publishing，2021 年。[17] A. Heinlein、A. Klawonn、J. Knepper 和 O. Rheinbach。用于三维重叠 Schwarz 方法的自适应 GDSW 粗空间。SIAM 科学计算期刊，41(5)：A3045–A3072，2019 年。[18] A. Heinlein、A. Klawonn、S. Rajamanickam 和 O. Rheinbach。FROSch：一种基于 Xpetra、集成于 Trilinos 的快速稳健重叠 Schwarz 区域分解预条件子。载于《科学与工程中的区域分解方法 XXV》，第 176–184 页，2020 年。[19] A. Heinlein、O. Rheinbach 和 F. R¨over。利用 Theta 超级计算机，将三级 FROSch 预条件子的并行可扩展性提升至 220000 个核心。SIAM 科学计算期刊（Copper Mountain 迭代法与多重网格方法会议），第 S173–S198 页，2021 年。[20] G. Karypis。METIS：用于非结构化图划分、网格划分及稀疏矩阵减少填充排序计算的软件包。技术报告，2013 年。[21] K. Kim、H. C. Edwards 和 S. Rajamanickam。Tacho：内存可扩展的任务并行稀疏 Cholesky 分解。载于 IEEE 国际并行与分布式处理研讨会专题讨论会，2018 年。[22] L. Luo、C. Yang、Y. Zhao 和 X. C. Cai。一种基于区域分解与代数多重网格的可扩展混合算法，用于在 CPU/GPU 集群上求解偏微分方程。载于 GPU 与科学应用国际研讨会，2011 年。[23] A. Prokopenko、C. M. Siefert、J. J. Hu、M. Hoemmen 和 A. Klinvex。Ifpack2 用户指南 1.0。技术报告 SAND2016-5338，桑迪亚国家实验室，2016 年。[24] S. Rajamanickam、S. Acer、L. Berger-Vergiat、V. Q. Dang、N. D. Ellingwood、E. Harvey、B. Kelley、C. R. Trott、J. J. Wilke 和 I. Yamazaki。Kokkos Kernels：性能可移植的稀疏/稠密线性代数与图计算内核。CoRR，abs/2103.11991，2021 年。[25] Y. Saad 和 M. Schultz。GMRES：用于求解非对称线性系统的广义最小残差算法。SIAM 科学计算期刊，7：856–869，1986 年。[26] B. F. Smith、P. E. Bjørstad 和 W. D. Gropp。《区域分解：椭圆偏微分方程的并行多层方法》。剑桥大学出版社，1996 年。[27] C. Trott、L. B.-Vergiat、D. Poliakoff、S. Rajamanickam、D. L.-Grandie、J. Madsen、N. Awar、M. Gligoric、G. Shipman 和 G. Womeldorff。Kokkos 生态系统：面向高性能计算的全面性能可移植性。《计算机科学与工程》，23(5)：10–18，2021 年。[28] I. Yamazaki、S. Rajamanickam 和 N. Ellingwood。面向众核架构、性能可移植的超节点稀疏三角求解器。载于并行处理国际会议，2020 年。[29] I. Yamazaki、S. Thomas、M. Hoemmen、E. Boman、K. ´Swirydowicz 和 J. Elliott。用于 Trilinos 中 s 步和流水线型 Krylov 求解器的低同步正交化方案。载于 SIAM 科学计算并行处理会议论文集，第 118–128 页，2020 年。[30] K. ´Swirydowicz、J. Langou、S. Ananthan、U. Yang 和 S. Thomas。低同步 Gram–Schmidt 与广义最小残差算法。数值线性代数及其应用，28(2)：e2343，2021 年。[31] Jakub ˇS´ıstek 和 Tom´aˇs Oberhuber。利用子域上的图形处理器加速并行 BDDC 求解器。《高性能计算应用国际期刊》，第 10943420221136873 页，2022 年 11 月。出版方：SAGE Publications Ltd STM。