https://crad.ict.ac.cn
62(2): 517−531, 2025

小规模非规则 TRSM 实现与优化

郭容园[1,2] 贾海鹏[2] 张云泉[2] 韦存阳[2] 邓明森[1] 陈婧蕊[3] 周振亚[3]
[1]（贵州财经大学信息学院 贵阳 550025）
[2]（中国科学院计算技术研究所 北京 100190）
[3]（北京华大九天科技股份有限公司 北京 100102）
（guoryuangradu@163.com）

Small-Scale Irregular TRSM Implementation and Optimization

Guo Rongyuan[1,2], Jia Haipeng[2], Zhang Yunquan[2], Wei Cunyang[2], Deng Mingsen[1], Chen Jingrui[3], and Zhou Zhenya[3]
[1]（*School of Information*, *Guizhou University of Finance and Economics*, *Guiyang* 550025）
[2]（*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190）
[3]（*Empyrean Technology Co.*, *Ltd.*, *Beijing* 100102）

Abstract TRSM (triangular matrix equation solver) is a commonly used algorithm for solving systems of linear equations, and is the core algorithm of various scientific computing libraries and mathematical software, which is widely used in the fields of scientific computing, engineering computing and machine learning. The small-scale irregular TRSM algorithm limits the scope of problem-solving and is an algorithm for efficiently handling smaller-scale, irregular data inputs. With the development of personalization and refinement in the field of high-performance computing, the demand for small-scale irregular TRSM computation in the scientific and industrial communities is becoming more and more obvious. While traditional algorithms are better suited for large-scale and regular TRSM computation, there is still room for improvement in the computational efficiency of small-scale and irregular TRSM. In this paper, we propose a small-scale irregular TRSM optimization scheme by combining hardware architecture and application scenario characteristics, designing a high-performance kernel from the perspectives of register chunking, boundary processing, and vectorization computation, and constructing an algorithmic library of small-scale irregular SI_TRSM (small-scale irregular TRSM) covering double-precision real numbers and double-precision complex numbers based on which the performance of this algorithm is greatly improved. Based on experimental results, the double-precision small-scale irregular TRSM algorithm library developed in this paper has shown to enhance the average performance of double-precision small-scale irregular real numbers by 29.4 times, and double-precision small-scale irregular complex numbers by 24.6 times in comparison with similar algorithms available in the MKL (Intel math kernel library).

Key words TRSM algorithm; BLAS; small-scale irregular; SIMD; assembly optimization

摘 要 TRSM（triangular matrix equation solver）是线性方程组求解的常用算法，是各种科学计算库和数学软件的核心算法，广泛应用于科学计算、工程计算、机器学习等领域. 小规模非规则 TRSM 算法限定解决问题范围，是高效处理较小规模、非规则数据输入的算法. 随着高性能计算领域个性化、精细化发展，科学

收稿日期：2023-10-31；**修回日期**：2024-02-27
基金项目：国家重点研发计划项目（2023YFB3001701）；山西省科技重大专项（202201010101004）；国家自然科学基金项目（61972376, 62372432, 62072431）
This work was supported by the National Key Research and Development Program of China (2023YFB3001701), the Shanxi Provincial Science and Technology Major Project (202201010101004), and the National Natural Science Foundation of China (61972376, 62372432, 62072431).
通信作者：贾海鹏（jiahaipeng@ict.ac.cn）

界、工业界对小规模非规则 TRSM 计算的需求愈加明显. 传统算法更偏重于大规模、规则 TRSM 的计算, 在小规模非规则 TRSM 计算上效率不佳. 结合硬件体系结构、应用场景特征提出小规模非规则 TRSM 优化方案, 从寄存器分块、边界处理、向量化计算角度设计高性能内核, 在此基础上构建覆盖双精度实数、双精度复数的小规模非规则算法库 SI_TRSM (small-scale irregular TRSM), 大幅度提升该算法性能. 实验结果表明, 构建的双精度小规模非规则 TRSM 算法库, 较 MKL (Intel math kernel library) 同类算法, 在双精度小规模非规则实数上平均性能提高 29.4 倍, 在双精度小规模非规则复数上平均性能提高 24.6 倍.

关键词 TRSM 算法; BLAS; 小规模非规则; SIMD; 汇编优化

中图法分类号 TP301

DOI: 10.7544/issn1000-1239.202330864 **CSTR:** 32373.14.issn1000-1239.202330864

TRSM(triangular matrix equation solver)算法[1-2]是矩阵计算核心操作之一, 也是基本线性代数子程序(BLAS)的重要组成部分. TRSM 是针对 $\boldsymbol{A}\times\boldsymbol{X}=\alpha\times\boldsymbol{B}$ 或 $\boldsymbol{X}\times\boldsymbol{A}=\alpha\times\boldsymbol{B}$ 进行求解的算法, 其中 $\boldsymbol{A}$ 是一个上三角或下三角矩阵, $\boldsymbol{X}$ 和 $\boldsymbol{B}$ 是矩阵, α 是一个标量. 该算法应用广泛, 是高斯消元、QR 分解和 LU 分解等算法的基础操作, 对电磁场分析、声学模拟、计算流体力学有支撑作用, 也可被用于实现矩阵求逆操作和解线性方程组问题.

TRSM 算法在许多软件的底层具有丰富的应用. 例如材料科学和生物分子模拟 DFT 软件包、芯片电路设计 EDA 软件等在底层对 TRSM 算法需求颇丰. 图 1 展示某 EDA 软件部分模块在底层对 TRSM 算法 7 194 396 次调用需求的规模分布, 占比 86.3% 的调用规模分布在图 1 中 SI 区域($m\leqslant100$, $n\leqslant100$). SI 区域以外的规模, 业界标准库 MKL[3], OpenBLAS[4], AMDLIB (AMD math library)[5] 等可取得良好性能效果, 甚至可达到硬件峰值的 90% 以上, 但在 SI 区域上, 目前已有算法的性能表现并不理想.

SI 区域 TRSM 计算具备“小规模非规则”特征. 该区域包含计算规模“$m\leqslant100$, $n\leqslant100$”, 定义“$q=m/n$”, 根据 q 值的变化, 矩阵会呈现长条($q<1$)、瘦高($q>1$)、小方块($q=1$)等小规模、非规则情况. 另外, SI 区域 TRSM 计算包含的数据量基本可一次性放入一级缓存(主流处理器 32~256 KB), 符合“规模较小”特征.

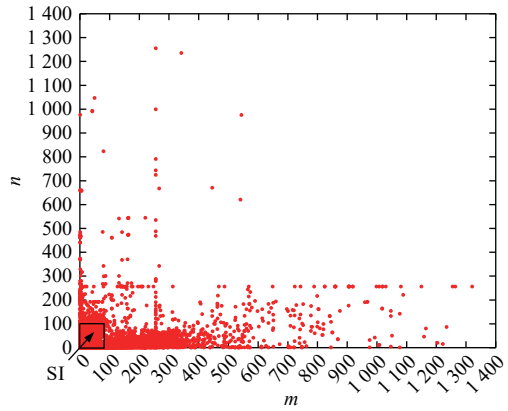


Fig. 1 TRSM demand hotspot size

图 1 TRSM 需求热点规模

传统 TRSM 算法在 SI 区域性能略显不足. 传统 TRSM 算法通常采用分块、Pack 的方法来优化性能. Pack 操作保证计算的连续性, 分块操作充分利用缓存, 这些方法在较大规模、方阵或趋近方阵的数据上将获得较好的性能优势. 但 SI 区域的数据规模较小, 基本可一次性放入一级缓存, 已具备缓存优势, 继续采用基于缓存的分块、Pack 操作对性能提升没有意义; 另外, “边界处理”在 SI 小规模非规则区域 TRSM 计算时不可忽视. 传统 TRSM 算法在内核设计时, 对边界数据并不关注, 较大规模 TRSM 计算中边界数据计算量占比较小, 忽视边界处理对整体性能的影响较小. 但在 SI 区域的 TRSM 计算中, 边界数据计算量占比较高, 忽视边界处理严重影响整体性能.

本文研究的小规模非规则 TRSM 算法充分考虑 SI 区域 TRSM 计算特征, 摒弃传统算法的优化思路, 结合硬件体系结构和应用场景, 提出小规模非规则 TRSM 优化方案, 从寄存器分块、边界处理、向量化计算角度, 重新设计了高性能内核, 最后遵循 BLAS 标准完成算法库, 定义为 SI_TRSM(small-scale irregular TRSM)库. 实验表明, 在 x86 平台上, 本文 TRSM 算法较业界领先且在 x86 平台适配性最佳的 MKL 同类算法, 有大幅度的性能提升.

本文的贡献有 3 点:

1)提出了基于应用场景、关注小规模非规则数据特征的优化方案. 该方案为 BLAS 相关算法的个性化需求提供参考.

2)设计了有效利用硬件特征进行分块、关注双精度小规模非规则数据边界处理问题的高性能内核. 与传统内核思路不同, 本文内核设计大幅度提高了

运行效率.

3)构建了覆盖双精度实数与复数的 SI_TRSM 算法库. 该算法库接口符合 BLAS 标准, 与 x86 平台适配性最佳的 MKL 同类算法相比, 表现出较好的性能优势.

1 相关工作

TRSM 算法是建立在广泛研究和深入理论探索基础上的经典算法, 但针对本文所述小规模非规则 TRSM 的研究缺乏相关的成果. 早期的 TRSM 算法基于高斯消元法, 难以处理大规模矩阵. 1971 年, McNamee[6] 提出用于求解稀疏上三角线性方程组的算法, 这是率先用于解决 TRSM 问题的子程序. 1979 年 BLAS 作为标准正式发布, TRSM 算法随之得到规范, 逐步发展出高效的双精度 DTRSM 算法, 可处理一般密集矩阵. 1990 年, Dongarra 等人[7] 提出 level-3 BLAS 的模型实现与测试方法, 单精度 STRSM 得到详细测试, 为该算法的优化奠定基础. 1992 年, Hanson[8] 发布 PC-BLAS, 在精度、实数、复数方面对 BLAS 进行高度优化, 推动 TRSM 算法在内的 BLAS 库等走向应用. 1994 年, Choi 等人[9] 在分布式系统上研究 BLAS 计算, BLAS 算法的应用场景进一步扩大. 2002 年, Dongarra[10] 主导推进 BLAS 标准更新, 对密集和稀疏矩阵的计算都提出并行要求, TRSM 算法也受到规范. 2008 年, Chan 等人[11] 对多线程 BLAS 深入研究, 其中 Cholesky 因式分解过程时通过分块大量运用 TRSM 算法, TRSM 算法在多线程的计算中得到发展. 2010 年, GPU 硬件广泛应用, Volkov 等人[12] 提出一种基于 GPU 的相关算法, 该算法利用 GPU 的高并行性来加速矩阵计算. 近年来随着深度学习、机器学习的发展, 研究人员开始研究将 TRSM 算法与其相结合[13], 用于直接并行化加速训练学习.

学者们对 TRSM 算法的其他研究[14-16] 在不断深入, 以满足越来越复杂的矩阵计算需求. 但针对本文所述小规模非规则 TRSM 实现与优化的研究, 目前没有针对性较强的成果.

另外, TRSM 算法也在众多公司和组织的 BLAS 商业库及解决方案中得到了发展. MKL, OpenBLAS, ATLAS[17], BLIS 等算法库在业界处于领先地位, 但也存在缺陷. 针对本文研究的小规模非规则 TRSM 算法, MKL 对于自身的体系架构充分优化适配, 但缺乏小规模非规则 TRSM 的专门优化; OpenBLAS 会因默认使用标准分块而对小规模非规则 TRSM 数据过度分块; ATLAS 自动参数优化在小规模计算时管理开销太大; BLIS 模块化的设计带来的函数调用开销影响较大.

综上, 传统算法对小规模非规则 TRSM 问题尚未进行较好解决, 因此本文算法具有一定的研究价值和改进空间, 有较强的工业实践意义和学术理论意义.

2 小规模非规则 TRSM 优化方案

2.1 小规模非规则 TRSM 优化方案概述

本文提出适合 SI_TRSM 的优化方案, 该方案充分适应双精度小规模非规则数据. 如图 2 所示, 我们梳理 SI_TRSM 遇到的基本情形, 包括“长条”“瘦高”“小方块”等特征的数据, 图 2 中三角形代表矩阵 $\boldsymbol{A}$, 矩形代表矩阵 $\boldsymbol{B}$.

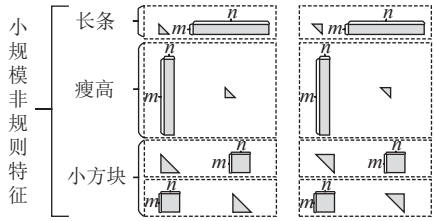


Fig. 2 Small-scale irregular features

图 2 小规模非规则特征

对于三角矩阵位于左侧的 TRSM 计算, 需要将更多注意力放在“长条”数据特征上. 在 TRSM 计算中, 规模参数 m 和 n 决定矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的形状. 当三角矩阵 $\boldsymbol{A}$ 在左侧时, 矩阵 $\boldsymbol{B}$ 的列长度受限于参数 m, 需同列长度、行长度皆为 m 的矩阵 $\boldsymbol{A}$ 保持一致. 此时矩阵 $\boldsymbol{B}$ 的行长度 n 不受限制, 在小规模的限定下, n 值或大或小, 本质是“长条”数据特征问题, 故对于左侧为三角时 TRSM 计算, 需更注重“长条”形状数据的处理.

对于三角矩阵位于右侧的 TRSM 计算, 需将更多注意力放在“瘦高”数据特征上. 当三角矩阵 $\boldsymbol{A}$ 位于右侧时, 矩阵 $\boldsymbol{B}$ 的行长度受限于参数 n, 需同列长度、行长度皆为 n 的矩阵 $\boldsymbol{A}$ 保持一致. 此时矩阵 $\boldsymbol{B}$ 的列长度 m 不受限制, 在小规模的限定下, m 值或大或小, 本质上是“瘦高”数据特征问题. 故对于右侧三角的 TRSM 计算, 需更注重“瘦高”形状数据的处理.

对于“小方块”数据特征, 其本质考验 SI_TRSM 的边界处理能力. 该特征包含于 n 较小的“长条”问题中, 或包含于 m 较小的“瘦高”问题中. 区别是相对于“长条”“瘦高”特征, “小方块”数据的边界计算占

比升高. 本文 SI_TRSM 关注“边界处理”工作, 故“小方块”无需单独进行考虑.

综上, SI_TRSM 需做到对“长条”“瘦高”特征自适应选择特定处理内核. 这 2 类特征的共同点是在规模较小时, 矩阵 $\boldsymbol{A}$ 可完全以广播形式放入向量寄存器, 此时“长条”或“瘦高”形状的矩阵 $\boldsymbol{B}$ 可在计算上采用分块的方式复用矩阵 $\boldsymbol{A}$, 这样既降低矩阵 $\boldsymbol{A}$ 在整个计算中的访存量, 又能极大降低 TRSM 边界处理在整个计算中的占比. 基于这个发现, 我们给出了 SI_TRSM 覆盖双精度实数、双精度复数的优化方案.

为方便下文阐述, 这里定义参数 *side*, *transa*, *uplo*, *diag* 依次为“R(或 L)、N(或 T, C)、L(或 U)、N(或 U)”, 例如“LNLN”表示“$\boldsymbol{A}$ 为左侧下三角非单位矩阵且无需转置操作”, “LCLN”表示“$\boldsymbol{A}$ 为左侧下三角非单位矩阵且需进行共轭转置操作”.

2.2 双精度实数优化方案

图 3 左侧对双精度实数的 8 种类型进行分类, LNLN, LTLN, LNUN, LTUN 这 4 种类型属于“长条”问题, RNLN, RTLN, RNUN, RTUN 这 4 种类型属于“瘦高”问题.

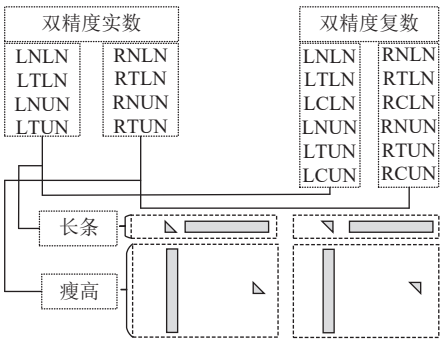


Fig. 3 Classification of small-scale irregular problems

图 3 小规模非规则问题分类

矩阵 $\boldsymbol{A}$ 位于左侧时, 双精度实数包含 LNLN, LTLN, LNUN, LTUN 这 4 种类型, 此时关注于解决“长条”问题. 于是, 每种类型下参数 m=1, 2, 3, 4, 5, 6, 7, 8 和 m>8 这 9 种情况, SI_TRSM 自适应选择特定内核, 共计 36 个. 对于参数 m=1, 2, 3, 4, 5, 6, 7, 8 时的特定内核, 我们已知矩阵 $\boldsymbol{A}$ 大小, 其不需要分块. 矩阵 $\boldsymbol{B}$ 的参数 n 是不固定的, 会呈现“长条”特征, 这种情况下, 需处理 n 维度上矩阵 $\boldsymbol{B}$ 分块、边界处理问题; 对于参数 m>8 时的特定内核, 参数 m, n 皆未知, 故矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 都需要进行分块, 此时涉及矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 分块和 $\boldsymbol{A}$, $\boldsymbol{B}$ 边界处理问题.

矩阵 $\boldsymbol{A}$ 位于右侧时, 双精度实数包含 RNLN, RTLN, RNUN, RTUN 这 4 种类型, 此时关注于解决“瘦高”问题. 于是, 每种类型下参数 n=1, 2, 3, 4, 5, 6, 7, 8 和 n>8 这 9 种情况, SI_TRSM 自适应选择特定内核, 共计 36 个. 对于参数 n=1, 2, 3, 4, 5, 6, 7, 8 时的特定内核, 我们已知矩阵 $\boldsymbol{A}$ 大小, 其不需要分块. 矩阵 $\boldsymbol{B}$ 的参数 m 是未知的, 会呈现“瘦高”特征, 这种情况下, 需处理 m 维度上矩阵 $\boldsymbol{B}$ 分块、边界处理问题; 对于参数 n>8 时的特定内核, 参数 m, n 皆未知, 故矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 都需要进行分块, 此时涉及 $\boldsymbol{A}$, $\boldsymbol{B}$ 分块和 $\boldsymbol{A}$, $\boldsymbol{B}$ 边界处理问题.

2.3 双精度复数优化方案

图 3 中右侧对双精度复数的 12 种类型进行分类, LNLN, LTLN, LCLN, LNUN, LTUN, LCUN 这 6 种类型属于“长条”问题, RNLN, RTLN, RCLN, RNUN, RTUN, RCUN 这 6 种类型属于“瘦高”问题.

矩阵 $\boldsymbol{A}$ 位于左侧时, 双精度复数包含 LNLN, LTLN, LCLN, LNUN, LTUN, LCUN 这 6 种类型, 此时面临的是“长条”问题, 于是, 每种类型下参数 m=1, 2, 3 和 m>3 这 4 种情况, SI_TRSM 自适应选择特定内核, 共计 24 个. 对于参数 m=1, 2, 3 时的特定内核, 我们已知矩阵 $\boldsymbol{A}$ 大小, 其不需要分块. 矩阵 $\boldsymbol{B}$ 的参数 n 是不固定的, 所以呈现“长条”特征, 这种情况下, 需处理 n 维度上矩阵 $\boldsymbol{B}$ 分块、边界处理问题; 对于参数 m>3 时的特定内核, 参数 m, n 皆未知, 故矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 都需要进行分块, 此时涉及 $\boldsymbol{A}$, $\boldsymbol{B}$ 分块和 $\boldsymbol{A}$, $\boldsymbol{B}$ 边界处理问题.

矩阵 $\boldsymbol{A}$ 位于右侧时, 双精度复数包含 RNLN, RTLN, RCLN, RNUN, RTUN, RCUN 这 6 种类型, 此时面临的是“瘦高”问题, 于是, 每种类型下参数 m=1, 2, 3 和 m>3 这 4 种情况, SI_TRSM 自适应选择特定内核, 共计 24 个. 对于参数 n=1, 2, 3 时的特定内核, 我们已知矩阵 $\boldsymbol{A}$ 大小, 其不需要分块. 矩阵 $\boldsymbol{B}$ 的参数 m 是未知的, 所以呈现“瘦高”特征, 这种情况下, 需要处理 m 维度上矩阵 $\boldsymbol{B}$ 分块、边界处理问题; 对于参数 n>3 时的特定内核, 参数 m, n 皆未知, 故矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 都需要进行分块, 此时涉及 $\boldsymbol{A}$, $\boldsymbol{B}$ 分块及 $\boldsymbol{A}$, $\boldsymbol{B}$ 边界处理问题.

在本节中给出了小规模非规则 TRSM 优化方案, 下文介绍高性能内核实现, 重点讲解分块和边界处理.

3 高性能内核设计

3.1 高性能内核概述

SI_TRSM 优化方案中 20 种内核扩展为 120 个实际内核, 第 2 节介绍了这些内核如何自适应“长条”

“瘦高”等特征. 本节将具体介绍内核实现部分的寄存器分块与边界处理策略.

表 1 中给出所有内核的名称. 本文 SI_TRSM 算法去掉 Pack 步骤, 内核设计需考虑转置、实数、复数等问题, 其处理双精度实数和双精度复数需 20 种不同类型内核, 在第 3 节优化方案里拓展为 120 个实际内核. 如表 1 所示, 对于双精度实数 DTRSM, 每种类型下包含“参数 m=1, 2, …, 8, n”, 或“参数 n=1, 2, …, 8, n”分别 9 个内核. 对于双精度复数 ZTRSM, 每种类型下包含“参数 m=1, 2, 3, n”, 或“参数 n=1, 2, 3, n”分别 4 个内核.

Table 1 Kernel Names

表 1 内核名称

计算类型	DTRSM	ZTRSM
LNLN	Kernel=D_m_LNLN m=1, 2, …, 8, n	Kernel=Z_m_LNLN m=1, 2, 3, n
LTLN	Kernel=D_m_LTLN m=1, 2, …, 8, n	Kernel=Z_m_LTLN m=1, 2, 3, n
LCLN		Kernel=Z_m_LCLN m=1, 2, 3, n
LNUN	Kernel=D_m_LNUN m=1, 2, …, 8, n	Kernel=Z_m_LNUN m=1, 2, 3, n
LTUN	Kernel=D_m_LTUN m=1, 2, …, 8, n	Kernel=Z_m_LTUN m=1, 2, 3, n
LCUN		Kernel=Z_m_LCUN m=1, 2, 3, n
RNLN	Kernel=D_n_RNLN n=1, 2, …, 8, n	Kernel=Z_n_RNLN n=1, 2, 3, n
RTLN	Kernel=D_n_RTLN n=1, 2, …, 8, n	Kernel=Z_n_RTLN n=1, 2, 3, n
RCLN		Kernel=Z_n_RCLN n=1, 2, 3, n
RNUN	Kernel=D_n_RNUN n=1, 2, …, 8, n	Kernel=Z_n_RNUN n=1, 2, 3, n
RTUN	Kernel=D_n_RTUN n=1, 2, …, 8, n	Kernel=Z_n_RTUN n=1, 2, 3, n
RCUN		Kernel=Z_n_RCUN n=1, 2, 3, n

本节重点介绍基于寄存器的分块及边界处理. 边界处理也是一种特殊分块, 本文关注边界处理, 故将其从整体分块中取出单独命名为“边界处理”. 根据第 3 节优化方案, 当 *side*=L 时, 以参数 m 作为参照数. 在双精度实数 *side*=L 时 m≤8 的内核、双精度复数 *side*=L 时 m≤3 的内核中, 对矩阵 $\boldsymbol{A}$ 尽可能放入向量寄存器复用, 对矩阵 $\boldsymbol{B}$ 进行“长条”或“瘦高”方向上分块和边界处理. 在双精度实数中 *side*=L 时 m>8 的内核、双精度复数中 *side*=L 时 m>3 的内核中, 需对矩阵 $\boldsymbol{A}$、矩阵 $\boldsymbol{B}$ 分别进行分块和边界处理. 同理, 当 *side*=R 时, 以参数 n 作为参照数. 本文分块和边界处理归纳如下:

1)分块. 作为参照数的 m 或 n 为已知常数时, 此时矩阵 $\boldsymbol{A}$ 尽可能被放入寄存器进行复用, 涉及到矩阵 $\boldsymbol{B}$ 的分块. 作为参照数的 m 或 n 为未知数(整型正数)时, 此时涉及矩阵 $\boldsymbol{A}$、矩阵 $\boldsymbol{B}$ 分块.

2)边界处理. 作为参照数的 m 或 n 为已知常数时, 涉及矩阵 $\boldsymbol{B}$ 的边界处理. 作为参照数的 m 或 n 为未知数(整型正数)时, 此时涉及矩阵 $\boldsymbol{A}$、矩阵 $\boldsymbol{B}$ 两者的边界处理.

存储、计算、通信是高性能计算系统的 3 个基本组成部分, 本文不涉及分布式系统和多核处理, 故本文为内核设计只确立关于存储和计算的 2 条原则, 这是本文高性能内核设计的基石. 存储层面, 充分发挥向量寄存器在小规模非规则数据特征上的存取优势. 本文去掉传统算法中基于缓存的 Pack 操作和分块方法, 由于 SI_TRSM 规模较小, 本身已经具备在缓存方面的优势, 现需将注意力放在主动利用向量化寄存器的优势上, 获得在寄存器级别的优势. 计算层面, 充分关注小规模非规则数据向量化并行计算. 无论非边界处理部分和边界处理部分, 我们都将尽可能使用向量化并行计算方式, 关注边界处理本质上是关注边界处理上的并行计算.

本文研究中内核设计与传统算法的内核设计有较大区别, 在实现中比传统算法的内核数量更多、复杂度也更高, 但在小规模非规则 TRSM 各类型求解上获得了较好的计算效率.

3.2 寄存器分块

本文分块方式紧密联系硬件体系结构特征, 传统分块在这方面更关注缓存层面, 本文分块更关注寄存器层面. 除去边界块处理, 寄存器分块的 2 种情况如下:

1)作为参照数的 m 或 n 为已知常数(实数类型下参照数为 1, 2, …, 8; 复数类型下参照数为 1, 2, 3)时, 此时矩阵 $\boldsymbol{A}$ 尽可能放入寄存器进行复用, 涉及到矩阵 $\boldsymbol{B}$ 的分块.

2)作为参照数的 m 或 n 为未知数(实数类型下参照数大于 8, 复数类型下参照数大于 3)时, 此时涉及矩阵 $\boldsymbol{A}$、矩阵 $\boldsymbol{B}$ 分块.

分块大小和原理与计算机体系结构和应用场景密切相关.

在计算机体系结构上, 由于面向支持 512 b 向量运算的处理器进行优化, 其典型特点之一是具有位宽达到 512 b 的向量寄存器, 支持 AVX-512 向量化指令. 这意味着 1 个 ZMM 寄存器可同时处理 8 个双精度浮点数, 可极大限度地利用 SIMD 技术. 故分块的某一维度应该以 8 为参照, 这样才能充分利用 512 b 向量寄存器, 避免向量寄存器资源浪费.

在应用场景上, 本文将目前小规模非规则 TRSM 需求引申为 SI_TRSM 问题, 优化与实现重点集中在“参数 $m\leqslant 100$, $n\leqslant 100$”的“长条”“瘦高”问题上. “长条”的长度方向和“瘦高”的高度方向显然是本文向量化的方向, 这 2 个方向上的分块以填满 512 b 向量寄存器为准, 即 8 个双精度浮点数据. “长条”的宽度方向或“瘦高”的瘦度方向等于矩阵 $\boldsymbol{A}$ 的边长, 即决定矩阵 $\boldsymbol{A}$ 的大小, 在规模较小时我们尽可能要求矩阵 $\boldsymbol{A}$ 以广播形式放入向量寄存器复用, 限于硬件体系结构 32 个向量寄存器数量, 设矩阵 $\boldsymbol{A}$ 边长为 K, 2 条边的边长均至少应该满足公式 $(K\times(K+1))/2\leqslant 32$, 故对于双精度实数, 其中一个块边长不应该超过 8, 对于双精度复数, 边长均不应该超过 3.

基于以上对计算机体系结构和应用场景的分析研究, 本文对于分块大小概括如下:

对于情况 1, 矩阵 $\boldsymbol{B}$ 分块大小是“已知常数×8”. 这样便于在“长条”“瘦高”上对矩阵 $\boldsymbol{A}$ 进行复用, 在“长条”的长度方向、“瘦高”的高度方向做充分的向量化并行计算, 这 2 个方向上的剩余计算交给高效的边界处理予以解决.

对于情况 2, 矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 分块大小依次是“$K\times K$”和“$K\times 8$”(实数时 $K=8$, 复数时 $K=3$). 这种情况下, 矩阵 $\boldsymbol{A}$ 因为元素较多, 已经无法全部保留在寄存器中进行复用. 此时我们可以通过分块, 使得当前计算的矩阵 $\boldsymbol{A}$ 的分块一直保留在向量寄存器中, 然后在矩阵 $\boldsymbol{B}$ 分块后进行复用. 概括来说, 就是把当前规模稍大(实数时 $K>8$, 复数时 $K>3$)的计算转化为高效的“长条计算”或“高瘦计算”, 最后剩下的部分交给边界处理.

图 4 展示情况 2 中双精度实数 LNLN 类型下的内核分块, 当 $K=m=n=8$ 时, 矩阵 $\boldsymbol{A}$ 包含 36 个双精度浮点数, 矩阵 $\boldsymbol{B}$ 包含 8 行 8 列双精度浮点数据, 即内核实现上对 $\boldsymbol{A}$, $\boldsymbol{B}$ 均采用 8×8 分块. 这里需要说明的是, 矩阵 $\boldsymbol{A}$ 元素是通过广播的形式在向量寄存器中存放, 当 $K=6, 7, 8$ 时, 将依次占用 21, 28, 36 个向量寄存器, $K=6, 7$ 时显然满足硬件特征, $K=8$ 显然不满足硬件特征. 通过反复实验, 对 8×8 的矩阵 $\boldsymbol{A}$ 采取个别元素嵌入式广播, 依然比对矩阵 $\boldsymbol{A}$ 采用 6×6, 7×7 分块高效.

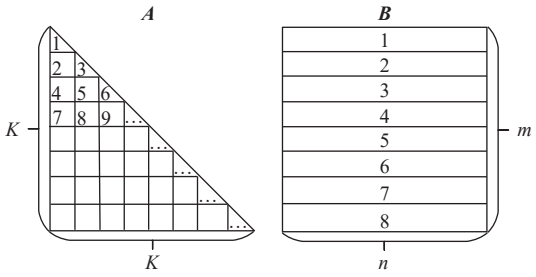


Fig. 4 Double-precision real LNLN kernel chunking

图 4 双精度实数 LNLN 内核分块

图 5 展示了情况 2 下双精度复数 LNLN 的内核分块. 双精度复数原始数据的实部、虚部紧邻排列, 在实际运算中实现向量化并行, 需继续借助向量寄存器将实部和虚部分离, 为了留出足够的向量寄存器进行辅助计算, 情况 2 下双精度复数的分块对矩阵 $\boldsymbol{A}$ 按照 3×3 分块, 对矩阵 $\boldsymbol{B}$ 按照 3×8 分块. 另外, 图 5 也展示了双精度复数 LNLN 分块的数据分布, 在列长度上 1 个实部和 1 个虚部构成 1 个复数, 对于 3×3 复数矩阵 $\boldsymbol{A}$, 其实际包含 12 个元素; 对于 3×8 复数矩阵 $\boldsymbol{B}$, 其实际包含 6 行 48 个元素.

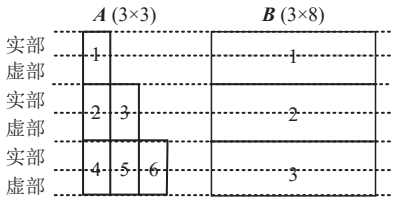


Fig. 5 Double-precision complex LNLN kernel chunking

图 5 双精度复数 LNLN 内核分块

3.3 边界处理

边界处理在 SI_TRSM 内核设计中不可忽视. SI_TRSM 规模较小, 边界处理在整个计算中的占比较高, SI_TRSM 形状非规则, 边界处理难以向量化, 需单独考虑. 涉及边界处理的情况有 2 个:

1)作为参照数的 m 或 n 为已知常数(实数类型下参照数为 1, 2, …, 8; 复数类型下参照数为 1, 2, 3)时, 此时涉及矩阵 $\boldsymbol{B}$ 的边界处理.

2)作为参照数的 m, n 为未知数(实数类型下参照数大于 8, 复数类型下参照数大于 3)时, 此时涉及矩阵 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{B}$ 的边界处理.

对于边界处理情况 1, 只涉及矩阵 $\boldsymbol{B}$ 的边界处理. TRSM 求解时, 矩阵 $\boldsymbol{A}$ 规模大小已知且理论上尽可能保留在向量寄存器中, 在矩阵 $\boldsymbol{B}$“长条”或“瘦高”

方向分块计算时进行复用，以及在矩阵 ***B*** “长条”或“瘦高”方向上最后不满足完整分块的地方进行边界处理. 这种情况下边界部分的特点是，其中一个维度的大小等于前面完整分块的一个维度，另一个维度取决于“长条”或“瘦高”方向上分块的剩余. 此时采用寄存器掩码的方式解决，例如“长条”或“瘦高”方向上余量为 7 时，使用掩码方式只启用向量寄存器前 448 位，存放 7 个双精度浮点数，然后在向量化计算上保持和前文所述完整分块相同.

对于边界处理情况 2，涉及矩阵 ***A*** 和矩阵 ***B*** 的边界处理. 本文定义一系列边界块，有可变边界块 ***B***、受限边界块 ***A***、边界三角块. 图 6 展示 LNLN 边界部分的分块，其中矩阵 ***A*** 规模是 27×27，矩阵 ***B*** 规模是 27×23，矩阵 ***A*** 和 ***B*** 分别进行 8×8 分块后，出现 Ⅶ, Ⅷ, Ⅸ, Ⅹ和 ***c***, ***f***, ***i***, ***j***, ***k***, ***l*** 这些边界块.

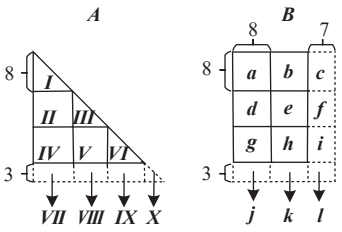


Fig. 6 LNLN boundary processing

图 6 LNLN 边界处理

本文对边界块进行分类，其中 ***c***, ***f***, ***i***, ***l*** 的维度不受矩阵计算限制，在 LNLN 类型下，***B*** 列长度需与 ***A*** 列长度保持一致，在行长度上是不受限制的，本文把这种边界形成的块定义为 ***B*** 可变块；Ⅶ, Ⅷ, Ⅸ与 ***j***, ***k*** 边界块大小受限于矩阵 ***A*** 列长度上分块剩余，定义为 ***A*** 受限边界块，规模为 n×8(n<8)；Ⅹ自成一类，即边界三角块，规模包含于 n×n(n<8).

对于 ***B*** 可变块和 ***A*** 受限边界块有 4 种情况：

1)***B*** 可变块需同类似图 6 中 Ⅰ这样的三角块进行矩阵计算. Ⅰ块采用广播形式取入 ZMM 寄存器，矩阵 ***B*** 采用的按行分散取值(gather 指令)的方式取入 ZMM 寄存器，此时的边界块采用寄存器掩码操作即可，本质上是行长度上取几个数据的问题，对取几个问题进行判别，给予自适应的掩码解决边界问题.

2)***B*** 可变块需同类似图 6 中 Ⅱ这样的 8×8 常规分块进行矩阵计算. 在介绍这种情况如何处理之前，先探讨图 6 中矩阵 ***B*** 的 ***f*** 块结果如何得到，首先需要完成“***f***←Ⅱ×***c***”来更新 ***f*** 值，然后求解 ***f*** 就变成“Ⅲ×***X***=***new_f***”问题(这里 ***X*** 表示解矩阵)，而“Ⅲ×***X***=***new_f***”是情况 1 中所讲的可变块遇到三角块的处理方式(这里 ***X*** 表示解矩阵)，在情况 1 中已经给出处理办法.

因此 ***B*** 可变块主要是解决“***f***←Ⅱ×***c***”边界问题，针对此边界问题的处理，我们设计了微内核 B_8x7_kernel, B_8x6_kernel, B_8x5_kernel, B_8x4_kernel, …, B_8x1_kernel, 依次处理“***f***←Ⅱ×***c***”中 ***f*** 规模为 8×7, 8×6, 8×5, 8×4, 8×3, 8×2, 8×1 的情况，具体的实现是采用带减法的矩阵乘. 这里给出 B_8x7_kernel 的设计思想如下：第 1 步，加载矩阵 ***A*** 中 Ⅱ块数据到 ZMM2, ZMM3, …, ZMM9 中；第 2 步，加载矩阵 ***B*** 中 ***f*** 块数据到 ZMM10, ZMM11, …, ZMM16 等 7 个向量寄存器；第 3 步，采用融合的乘减指令完成 ZMM10, ZMM11, …, ZMM16 数据减去“ZMM2, ZMM3, …, ZMM9 数据与矩阵 ***B*** 中 ***c*** 块数据相乘结果”. 具体实现可参考 GEMM 算法.

3)***B*** 可变块需同 ***A*** 受限边界块进行矩阵计算. 针对这种情况，我们根据 ***A*** 受限程度(可能有 7, 6, 5, 4, 3, 2, 1 等列长度剩余)，仿照 B_8x7_kernel 设计了 B_Nx7_kernel, B_Nx6_kernel, B_Nx5_kernel, B_Nx4_kernel, …, B_Nx1_kernel(N=7, 6, 5, 4, 3, 2, 1)等 49 种微内核.

4)***A*** 受限边界块需同 ***B*** 中 a 这样 8×8 正常分块进行矩阵计算，我们设计了 KERNEL_Nx8_M(N=7, 6, 5, 4, 3, 2, 1)微内核，实现上是矩阵乘的变体.

对于图 6 中 Ⅹ边界三角块，本文设计覆盖所有情况的微内核. 我们按照图 4 中分块 8×8 思想，完成了 7×7, 6×6, 5×5, 4×4, 3×3, 2×2, 1×1 等规模的边界三角块微内核.

双精度复数边界处理原理与双精度实数边界处理原理相同，只是规模大小发生了变化. 对双精度复数 LNLN 类型下矩阵 ***A***，按照 3×3(复数包含 12 个双精度元素)分块，对双精度复数矩阵 ***B*** 按照 3×8(复数是 6 行 8 列共 48 个双精度元素)分块. 对应图 6 中 ***B*** 可变边界块、***A*** 受限边界块、边界三角块依旧存在，只是规模发生相应变化，例如对于边界三角，双精度实数是 N×N(N=1, 2, …, 7)规模，双精度复数是 N×N(N=1, 2)规模，故这里不再详述.

3.4 内核关联

内核之间存在高度关联性，下面通过分类描述其关联，打通前文分块、边界处理等策略在各种类型内核上的应用阻碍.

处理双精度实数及复数共需 20 种不同类型内核，本文将 LNLN, LNUN, RNLN, RNUN 类型归为普通内核，特点是不涉及转置，双精度实数、复数都有这些类型，所以共 8 种；将 LTLN, LTUN, RTLN, RTUN 类型归为转置内核设计，特点是需要对三角矩阵 ***A*** 进

行转置操作, 双精度实数、复数都有这些类型, 共 8 种; 将 LCLN, LCUN, RCLN, RCUN 类型归为共轭转置内核设计, 特点是需要完成数据的共轭转置操作, 只有双精度复数有这些类型, 共 4 种; 以上包含 SI_TRSM 所有不同类型的内核.

对于转置内核设计, 其相比于普通内核设计, 需要对矩阵 $\boldsymbol{A}$ 进行转置操作. 例如对于 LTUN 类型的数据进行操作时, 上三角矩阵转置后是下三角矩阵, 可以采用 LNLN 计算形式, 但需要计算的数据分布于上三角部分, 如图 7 所示, 需先按照转置的方式取值, 再按照 LNLN 形式计算.

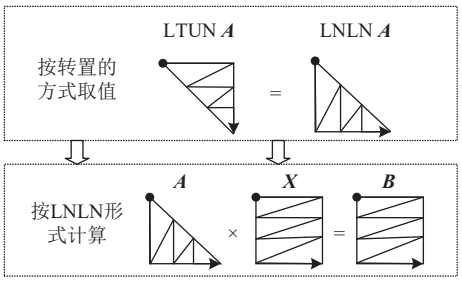


Fig. 7 Handling method of transposed kernels

图 7 转置内核的处理方法

共轭转置内核设计是在转置内核的基础上增加复数矩阵的共轭操作. 复数可以表示为 $a+bi$ 的形式, 其中 a 和 b 分别是实部和虚部, 共轭转置操作将复数的虚部取负数, 即 b 变为 "$-b$". 共轭转置内核设计需在转置内核的基础上对复数矩阵 $\boldsymbol{A}$ 增加取共轭步骤.

4 内核实现与优化方法

以上第 2 和 3 节依次介绍小规模非规则 TRSM 优化方案、高性能内核的设计, 注重从思路、研究的角度讲解本文的工作, 并从实现与优化方法的角度对本文工作进行介绍.

4.1 汇编实现

本文 SI_TRSM 在汇编级别实现 TRSM 任务的存取、计算等全部任务, 汇编实现提供高效、精确的硬件控制能力, 同时也为优化奠定了基础.

汇编实现为优化奠定基础. 首先, 按照 System V AMD64[18] 调用约定, 我们在汇编层得到 TRSM 全参数, 从而获得对数据操作较为直接的权限, 可以直接访问和控制内存, 精确地管理数据的存储和访问方式, 减少内存访问的延迟和带宽消耗. 其次, 汇编实现不会引入高级语言中可能存在的不必要抽象和额外代码, 我们可以直接控制生成的指令, 减少代码的冗余和不必要的计算, 从而提高执行效率. 最后, 汇编实现支持数据重排、指令融合、循环展开、手动分配寄存器、优化数据结构的存储布局、使用相关 SIMD 指令集等, 为性能优化提供了更多思路.

4.2 SIMD 优化

SIMD(single instruction multiple data)[19] 是一种计算机指令级并行处理的技术, 旨在同时处理多个数据元素, 提高计算性能和效率.

AVX-512 指令集[20] 的使用极大增强并行化程度. 该指令集是 Intel 公司推出的一套扩展指令集, 是 SIMD 技术的一种具体实现, 它引入了 32 个 512 b 的向量寄存器(ZMM0-ZMM31), 用于存储和处理更多的数据元素, 执行并行程度更高的向量化操作. 其次提供掩码操作, 这为边界数据的向量化操作提供了帮助. SI_TRSM 在计算时, 可充分采用 SIMD 技术进行优化, 本文研究的 TRSM 内核计算时涉及到的基本数据类型都是双精度浮点数, 1 个 ZMM 寄存器可容纳 8 个双精度浮点数, 意味着我们可以同时对 8 个双精度浮点数据执行 SIMD 操作, 在边界处理时不能填满向量寄存器时, 可采用掩码操作尽最大程度实现向量化.

如图 4 所示, 对于双精度实数 TRSM, 记 $\boldsymbol{X}1$ 为矩阵 $\boldsymbol{X}$ 第 1 行的结果, 记 $A1$ 为矩阵 $\boldsymbol{A}$ 中标号 1 的元素, 记 $\boldsymbol{B}1$ 为矩阵 $\boldsymbol{B}$ 标号 1 的行元素. 在不考虑标量 α 时, 图 8 中展示简单的 $\boldsymbol{X}1=\boldsymbol{B}1/A1$ 计算, 使用 ZMM 寄存器执行 SIMD 的乘法操作. 对于数据较多的公式 $\boldsymbol{X}8=(\boldsymbol{B}8-A29\times\boldsymbol{X}1-A30\times\boldsymbol{X}2-A31\times\boldsymbol{X}3-A32\times\boldsymbol{X}4-A33\times\boldsymbol{X}5-A34\times\boldsymbol{X}6-A35\times\boldsymbol{X}7)/A36$, 包含乘法和减法操作, 依旧可采用图 8 的思路, 可使用 "vfnmadd231pd" 指令实现减法和乘法的指令融合.

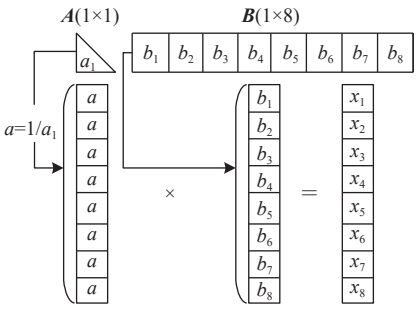


Fig. 8 SIMD multiplication

图 8 SIMD 乘法

4.3 向量寄存器使用

寄存器是位于处理器内部的存储器单元, 相比硬盘、主存、缓存, 其具有最快的存取速度. 向量寄

存器是一种特殊类型的寄存器，用于并行处理数据，以提高数据处理的并行性和效率.

本文追求对向量寄存器的“贪心”使用，这体现在对全部向量寄存器的分配和单个寄存器的使用上.

向量寄存器的分配使用与 SI_TRSM 的特征密切相关. SI_TRSM“小规模非规则”的特征让我们摒弃传统算法对于缓存命中、数据局部性的追求，本文工作更加追求对于向量寄存器的合理分配与使用. “$\boldsymbol{AX}=\alpha\boldsymbol{B}$ 或 $\boldsymbol{XA}=\alpha\boldsymbol{B}$”中 $\boldsymbol{A}$ 为三角方阵，在小规模的限定下，“$\boldsymbol{AX}=\alpha\boldsymbol{B}$”容易呈现“长条”特征，“$\boldsymbol{XA}=\alpha\boldsymbol{B}$”容易呈现“瘦高”特征，“长条”形状的宽度和“瘦高”形状的瘦度本质上是矩阵 $\boldsymbol{A}$ 的边长，故“长条”形状的宽度或“瘦高”形状的瘦度决定矩阵 $\boldsymbol{A}$ 元素数量. 增强矩阵 $\boldsymbol{A}$ 在“长条”“瘦高”特征方向的复用性对于处理非规则问题是关键，我们期望矩阵 $\boldsymbol{A}$(或 $\boldsymbol{A}$ 的分块)能尽可能保留在向量寄存器，这样可在“长条”“瘦高”方向上分块计算时进行复用，因此矩阵 $\boldsymbol{A}$ 元素数量是本文分配向量寄存器的首要标准，在向量寄存器分配上较为“贪心”，矩阵 $\boldsymbol{A}$ 能放则放，不能存放时，分块存放.

单个向量寄存器的充分利用需尽可能填满数据. 从单个向量寄存器的角度看，我们期望在“长条”长度方向、“瘦高”高度方向进行向量化操作，这样除边界处理外，向量寄存器的使用基本可被填满，进而执行并行计算操作.

反复实验对比，以上对于全部向量寄存器、单个向量寄存器的“贪心”使用方法，对于性能的提高有显著作用.

4.4 向量化计算

结合汇编实现、SIMD 向量化、向量寄存器使用等方法，本文工作在计算上尽最大可能实现向量化并行计算.

对于双精度实数 TRSM，在不考虑标量 α 时，获得解矩阵 $\boldsymbol{X}$ 第 1 行的结果(记为 $\boldsymbol{X}1$)，需图 4 中 $A1$(矩阵 $\boldsymbol{A}$ 中标号 1 的元素)取倒数后与 $\boldsymbol{B}1$ (矩阵 $\boldsymbol{B}$ 标号 1 的行)元素对应相乘，公式为 $\boldsymbol{X}1=\boldsymbol{B}1/A1$. 同理，$\boldsymbol{X}$ 第 2 行的结果 $\boldsymbol{X}2=(\boldsymbol{B}2-A2\times\boldsymbol{X}1)/A3$，$\boldsymbol{X}$ 第 3 行的结果 $\boldsymbol{X}3=(\boldsymbol{B}3-A4\times\boldsymbol{X}1-A5\times\boldsymbol{X}2)/A6$，…，$\boldsymbol{X}$ 第 8 行的结果 $\boldsymbol{X}8=(\boldsymbol{B}8-A29\times\boldsymbol{X}1-A30\times\boldsymbol{X}2-A31\times\boldsymbol{X}3-A32\times\boldsymbol{X}4-A33\times\boldsymbol{X}5-A34\times\boldsymbol{X}6-A35\times\boldsymbol{X}7)/A36$.

根据上述公式，我们在算法 1 中给出矩阵 $\boldsymbol{X}1$，$\boldsymbol{X}2$，$\boldsymbol{X}3$的求解伪代码. 行①~③对 3×3 三角矩阵主对角线元素进行取倒数操作，行④~⑨将矩阵 $\boldsymbol{A}$ 广播进向量寄存器，行⑩~⑫将规模 3×8 的矩阵 $\boldsymbol{B}$ 分散取值到向量化寄存器，行⑬~⑱是求解三角矩阵的四则运算，行⑲⑳将计算结果存入矩阵 $\boldsymbol{B}$，其完成了一个简易的 TRSM 计算(这里以 LNLN 模式下 3×3 的矩阵 $\boldsymbol{A}$ 和 3×8 的矩阵 $\boldsymbol{B}$ 中 TRSM 求解问题为例).

算法 1. 双精度实数 LNLN 类型向量化计算.

输入：标量 $A1$, $A2$, $A3$, $A4$, $A5$, $A6$, 向量 $\boldsymbol{B}1$, $\boldsymbol{B}2$, $\boldsymbol{B}3$;

输出：结果向量 $\boldsymbol{B}1(\boldsymbol{X}1)$, $\boldsymbol{B}2(\boldsymbol{X}2)$, $\boldsymbol{B}3(\boldsymbol{X}3)$.

① XMM1 ← 1.0/$A1$;

② XMM3 ← 1.0/$A3$;

③ XMM6 ← 1.0/$A6$;

④ ZMM1 ← broadcast(XMM1);

⑤ ZMM3 ← broadcast(XMM3);

⑥ ZMM6 ← broadcast(XMM6);

⑦ ZMM2 ← broadcast($A2$);

⑧ ZMM4 ← broadcast($A4$);

⑨ ZMM5 ← broadcast($A5$);

⑩ ZMM22 ← gather($\boldsymbol{B}1$);

⑪ ZMM23 ← gather($\boldsymbol{B}2$);

⑫ ZMM24 ← gather($\boldsymbol{B}3$);

⑬ ZMM22 ← ZMM22 × ZMM1;

⑭ ZMM23 ← ZMM23–ZMM2 × ZMM22;

⑮ ZMM23 ← ZMM23 × ZMM3;

⑯ ZMM24 ← ZMM24–ZMM4 × ZMM22;

⑰ ZMM24 ← ZMM24–ZMM5 × ZMM23;

⑱ ZMM24 ← ZMM24 × ZMM6;

⑲ scatter($\boldsymbol{B}1$, ZMM22);

⑳ scatter($\boldsymbol{B}2$, ZMM23);

㉑ scatter($\boldsymbol{B}3$, ZMM24).

双精度复数与双精度实数差异不大. 在图 5 中，记 $A1$ 为 1 个实部和 1 个虚部构成的复数，记 $\boldsymbol{B}1$是 1 行实部和 1 行虚部组成的 1 行复数，定义解矩阵 $\boldsymbol{X}1$ 的数据分布形式与 $\boldsymbol{B}1$相同. 在不考虑 α 时，$\boldsymbol{X}1=\boldsymbol{B}1/A1$，以此类推，$\boldsymbol{X}2=(\boldsymbol{B}2-A2\times\boldsymbol{X}1)/A3$，$\boldsymbol{X}3=(\boldsymbol{B}3-A4\times\boldsymbol{X}1-A5\times\boldsymbol{X}2)/A6$. 以上是复数矩阵计算涉及到的公式，其形式、原理与双精度实数相同.

双精度复数计算通过“实部和虚部分离”最大程度对数据进行向量化操作. 算法 2 给出了复数矩阵 $\boldsymbol{A}$ 规模“3×3”、矩阵 $\boldsymbol{B}$ 规模“3×8”的主要计算步骤. 输入 $A1$, $A2$, …, $A6$ 的实部、虚部(其中 $A1$, $A3$, $A6$ 等主对角线元素已完成取倒数操作)，依次按照双精度 64 位广播到 ZMM2, ZMM3, ZMM4, ZMM5, ZMM8, ZMM9, ZMM6, ZMM7, ZMM10, ZMM11, ZMM12, ZMM13 中. $\boldsymbol{B}1$，$\boldsymbol{B}2$，$\boldsymbol{B}3$的实部、虚部依次取入 ZMM22, ZMM23, ZMM24, ZMM25, ZMM26, ZMM27 中，其他

计算步骤如算法 2 所示, 行①②计算出 $\boldsymbol{X}1$, 行⑤~⑩计算出 $\boldsymbol{X}2$, 行⑬~㉒计算出 $\boldsymbol{X}3$, 其余伪代码负责存储.

算法 2. 双精度复数 LNLN 类型向量化计算.

输入: 复数标量 $A1$, $A2$, $A3$, $A4$, $A5$, $A6$, 向量 $\boldsymbol{B}1$, $\boldsymbol{B}2$, $\boldsymbol{B}3$, $\boldsymbol{B}4$, $\boldsymbol{B}5$, $\boldsymbol{B}6$;

输出: 向量 $\boldsymbol{B}1$, $\boldsymbol{B}2$, $\boldsymbol{B}3$, $\boldsymbol{B}4$, $\boldsymbol{B}5$, $\boldsymbol{B}6$(依次存储 $\boldsymbol{X}1$, $\boldsymbol{X}2$, $\boldsymbol{X}3$, $\boldsymbol{X}4$, $\boldsymbol{X}5$, $\boldsymbol{X}6$).

① ZMM14=ZMM2×ZMM22+ZMM3×ZMM23;
② ZMM15=ZMM3×ZMM22+ZMM2×ZMM23;
③ scatter([rdx], ZMM22 ← ZMM14);
④ scatter([rdx+0x8], ZMM23 ← ZMM15);
⑤ZMM14=ZMM4×ZMM22–ZMM5×ZMM23;
⑥ ZMM15=ZMM5×ZMM22+ZMM4×ZMM23;
⑦ ZMM24=ZMM24–ZMM14;
⑧ ZMM25=ZMM25–ZMM15;
⑨ ZMM14=ZMM8×ZMM24–ZMM9×ZMM25;
⑩ ZMM15=ZMM9×ZMM24+ZMM8×ZMM25;
⑪ scatter ([rdx+0x10], ZMM24 ← ZMM14);
⑫ scatter ([rdx+0x18], ZMM25 ← ZMM15);
⑬ ZMM14=ZMM6×ZMM22–ZMM7×ZMM23;
⑭ ZMM15=ZMM7×ZMM22+ZMM6×ZMM23;
⑮ ZMM26=ZMM26–ZMM14;
⑯ ZMM27=ZMM27–ZMM15;
⑰ ZMM14=ZMM10×ZMM2–ZMM11×ZMM25;
⑱ ZMM15=ZMM11×ZMM24+ZMM10×ZMM25;
⑲ ZMM26=ZMM26–ZMM14;
⑳ ZMM27=ZMM27–ZMM15;
㉑ZMM14=ZMM12×ZMM26–ZMM13×ZMM27;
㉒ZMM15=ZMM13×ZMM26+ZMM12×ZMM27;
㉓ scatter ([rdx+0x20], ZMM26 ← ZMM14);
㉔ scatter ([rdx+0x28], ZMM27 ← ZMM15).

向量化计算方式极大限度地发挥硬件体系结构优势. 算法 1 伪代码展示了矩阵 $\boldsymbol{B}$ 规模为“3×8”, 倘或矩阵 $\boldsymbol{B}$ 是规模“3×96”的“长条”矩阵, 矩阵 $\boldsymbol{B}$ 第 1 个分块后面 11 个“3×8”分块计算时, 矩阵 $\boldsymbol{A}$ 无需再次加载. 这种硬件优势在算法 2 上表现也是一致的.

4.5 指令操作

除 4.4 节所述向量化计算指令外, 本文 TRSM 还包含取值、流控制、存储等指令操作.

取值阶段负责将数据加载至特定向量寄存器. 对于三角矩阵 $\boldsymbol{A}$, 其取值方式为广播操作, 每一个数据填充 1 个寄存器, 在寄存器不够用时, 少许数据使用嵌入式广播操作. 对于矩阵 $\boldsymbol{B}$, 如图 9 所示, 若计算机系统是列主序存储, 涉及转置、非转置问题, 至少有 1 个方向是不连续的, 对于图 9 中非连续方向上的数据, 我们采用分散取值的指令“vgatherqpd”收集到寄存器中, 对于连续方向上的数据, 可以采用“vmovupd/vmovapd”等指令连续取值到寄存器. 对于连续、非连续边界, 都有特定的微内核, 其取值可采用 K 寄存器掩码取值方式或者直接改变计算方式适应边界. 存储阶段是取值阶段的反向操作, 使用指令的思路是一致的, 存取方向是相反的.

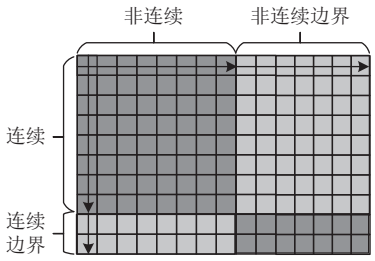


Fig. 9 Data distribution

图 9 数据分布

流控制主要完成条件判断、跳转和循环控制. 由上文可知, 我们设计了仅需矩阵 $\boldsymbol{B}$ 分块的内核和矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 皆需分块的内核. 如图 10 中①所示 LNLN 类型下, 对于仅需矩阵 $\boldsymbol{B}$ 分块的内核, 其流控制只涉及规模参数 n 维度, m 维度对于我们来说是已知的, 此时仅需要条件判断和跳转指令完成规模 n 维度循环, 在循环过程中, 矩阵 $\boldsymbol{A}$ 保留在向量寄存器中进行复用. 图 10 中②展示了矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$ 皆需分块的内核, 其规模 m, n 对于内核都是未知的, 其流控制需要负责 2 个维度上的条件判断和跳转, 指令的使用变得更加精细和复杂, 但本质并没有改变, 只是将其通过分块转化成“长条”(或“瘦高”)形状计算, 此时矩阵 $\boldsymbol{A}$ 已无法一次性放入向量寄存器, 但其三角分块可保留在寄存器中, 依然可在矩阵 $\boldsymbol{B}$ 通过分块形成“长条”(或“瘦高”)形状计算时进行复用.

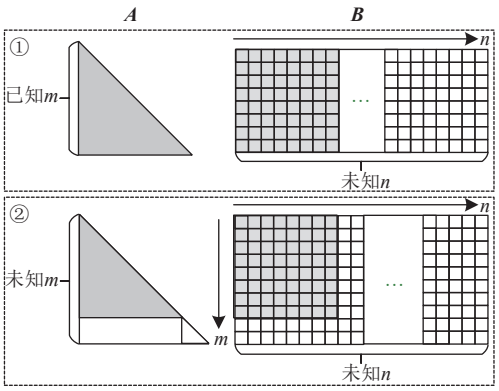


Fig. 10 Real LNLN flow control

图 10 实数 LNLN 流控制

5 实验评估

5.1 实验环境与评估方案

第 2~4 节对小规模非规则 TRSM 的工作已进行介绍，经过参考其他的评估方法[21-23]，下面我们给出针对本文工作的评估方案.

本文在 x86 架构平台测试，软硬件具体指标如表 2 所示，另外，选取了 x86 架构上适配性和运行效果最佳的 MKL 商业库进行对比.

Table 2 Experimental Environments

表 2 实验环境

不同环境	环境参数	参数指标
硬件环境	处理器	Intel Xeon Gold 6240
	架构	x86_64
	主频/GHz	2.6
	SIMD/b	512
	L1d cache/KB	32
	L1i cache/KB	32
	L2 cache/KB	1 024
软件环境	编译器	GCC7.5
	MKL	2023.0.0

本文对双精度实数 TRSM 的评估思路如下：对 TRSM 规模参数 $m\leqslant 100$(n=96)的样本点测试作为“小规模非规则长条数据特征”的评估；对 TRSM 规模参数 $n\leqslant 100$(m=96)的样本点测试，作为“小规模非规则瘦高数据特征”的评估；对 TRSM 规模参数 $m=n\leqslant 100$ 的样本点测试，作为“小规模数据特征”的评估.

本文对双精度复数 TRSM 的评估思路如下：复数包含实部和虚部(等同 2 个实数)，因此本文在测试规模上进行收缩. 对 TRSM 规模参数 $m\leqslant 50$(n=48)的样本点进行测试，作为“小规模非规则长条数据特征”的评估；对 TRSM 规模参数 $n\leqslant 50$(m=48)的样本点进行测试，作为“小规模非规则瘦高数据特征”的评估；对 TRSM 规模参数 $m=n\leqslant 50$ 的样本点进行测试，作为“小规模数据特征”的评估.

本文处理的数据都是双精度浮点数，采用每秒执行 10 亿次浮点运算数(GFLOPS)衡量性能. 为避免偶然性，本文实验数据随机生成，衡量时采用均值.

5.2 结果与性能分析

按照 5.1 节中的评估方案，实验结果如图 11、图 12 所示. 图 11 包含双精度实数小规模非规则 TRSM 和 MKL 各类型性能，图 12 包含双精度复数小规模非规则 TRSM 和 MKL 的各类型性能.

图 11 展示双精度实数小规模非规则 TRSM 和 MKL 的各类型性能. 图 11(a)点线图中，规模参数 n=96，规模参数 m 是自变量，该图展示 LNLN, LTLN, LNUN, LTUN 类型在“小规模非规则长条形状”数据上的浮点性能；图 11(b)点线图中，规模参数 $m=n\leqslant 100$，规模参数 $m=n$ 是自变量，该图展示 LNLN, LTLN, LNUN, LTUN 类型在“小规模方块形状”数据上的浮点性能；图 11(c)点线图中，规模参数 m=96，规模参数 n 是自变量，该图展示 RNLN, RTLN, RNUN, RTUN 类型在“小规模非规则高瘦形状”数据上的浮点性能；图 11(d)点线图中，规模参数 $m=n\leqslant 100$，规模参数 $m=n$ 是自变量，该图展示 RNLN, RTLN, RNUN, RTUN 类型在“小规模方块形状”数据上的浮点性能.

图 12 展示双精度复数小规模非规则 TRSM 和 MKL 的各类型性能. 图 12(a)点线图中，规模参数 n=48，规模参数 m 是自变量，该图展示 LNLN, LTLN, LCLN, LNUN, LTUN, LCUN 类型在“小规模非规则长条形状”数据上的浮点性能；图 12(b)点线图中，规模参数 $m=n\leqslant 50$，规模参数 $m=n$ 是自变量，该图展示 LNLN, LTLN, LCLN, LNUN, LTUN, LCUN 类型在“小规模方块形状”数据上的浮点性能；图 12(c)点线图中，规模参数 m=48，规模参数 n 是自变量，该图展示 RNLN, RTLN, RCLN, RNUN, RTUN, RCUN 类型在“小规模非规则高瘦形状”数据上的浮点性能；图 12(d)点线图中，规模参数 $m=n\leqslant 50$，规模参数 $m=n$ 是自变量，该图展示 RNLN, RTLN, RCLN, RNUN, RTUN, RCUN 类型在“小规模方块形状”数据上的浮点性能.

图 11(a)点线图中，相应类型 TRSM 在“小规模非规则长条形状”数据上相对 MKL 平均每个样本点加速 13.5 倍. 图 11(b)点线图中，相应类型 TRSM 在“小规模方块形状”数据上相对 MKL 平均每个样本点加速 46.6 倍. 图 11(c)点线图中，相应类型 TRSM 在“小规模非规则高瘦形状”数据上相对 MKL 平均每个样本点加速 16.1 倍. 图 11(d)点线图中，相应类型 TRSM 在“小规模方块形状”数据上相对 MKL 平均每个样本点加速 33.4 倍.

图 12(a)点线图中，相应类型 TRSM 在“小规模非规则长条形状”数据上相对 MKL 平均每个样本点加速 19.9 倍. 图 12(b)点线图中，相应类型 TRSM 在“小规模方块形状”数据上相对 MKL 平均每个样本点加速 50.9 倍. 图 12(c)点线图中，相应类型 TRSM 在“小规模非规则高瘦形状”数据上相对 MKL 平均

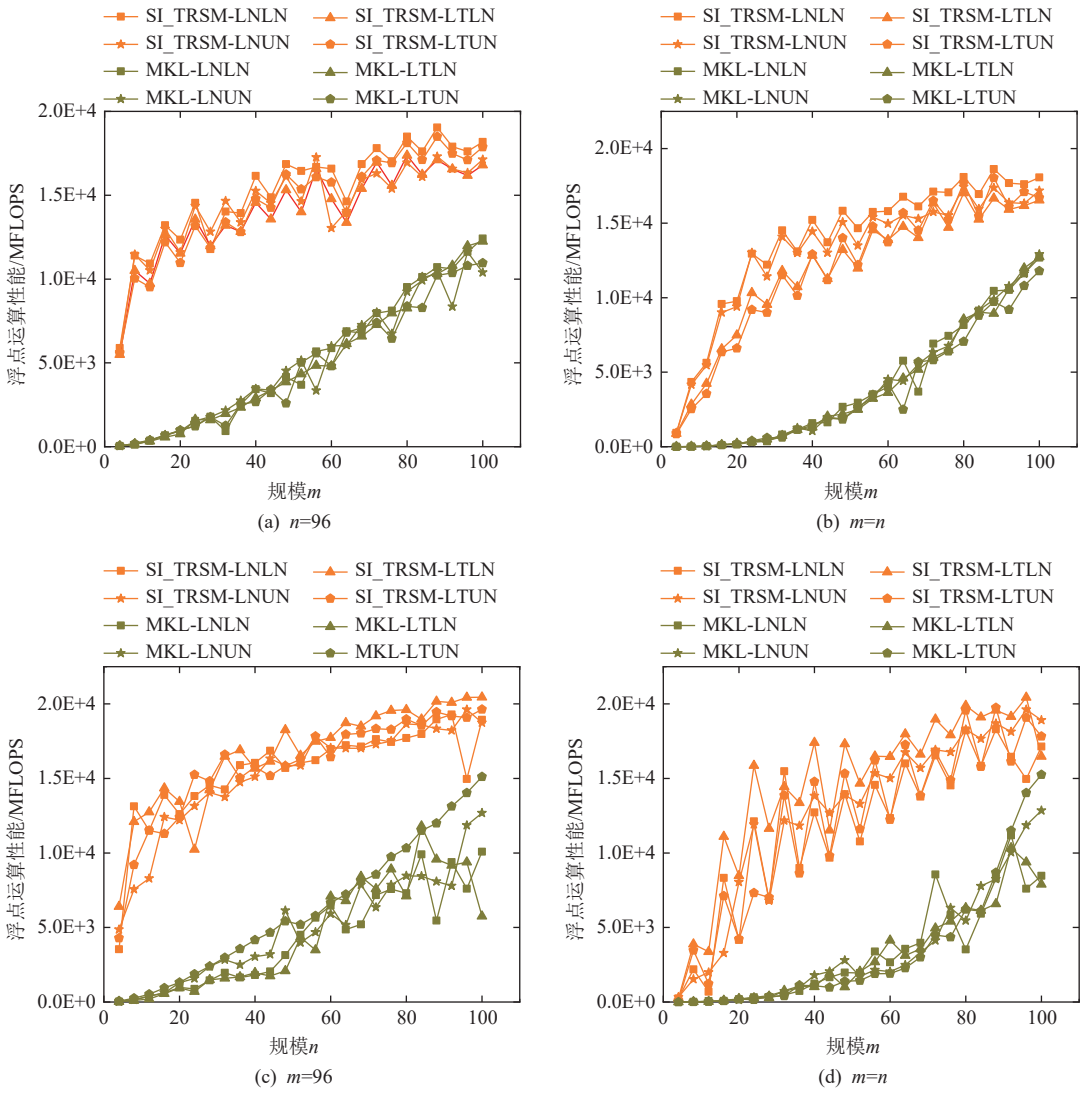


Fig. 11 Comparison of the performance of each type of double-precision real SI_TRSM and MKL

图 11 双精度实数 SI_TRSM 与 MKL 各类型性能对比

每个样本点加速 33.6 倍. 图 12(d)点线图中，相应类型 TRSM 在“小规模方块形状”数据上相对 MKL 平均每个样本点加速 74.5 倍.

根据以上评估数据，本文 TRSM 工作在 SI 区域(具备小规模非规则特征)相对于 x86 平台适配性最佳的 MKL 各类型算法取得了较大性能提升. 另外，根据图 11、图 12 可知，MKL 各类型算法性能在 SI 区域表现不佳，但在规模 m 或 n 趋近 SI 区域边界时逐渐获得性能优势，根据其上升趋势，随着规模继续增大取得较好效果. 故本文工作旨在解决小规模非规则 TRSM 问题，对于规模较大的 TRSM 计算，本文对业界已有 BLAS 库的卓越成果表示肯定.

5.3 通用性能分析

上面已经针对双精度实数、复数 TRSM 各类型适配小规模非规则特征进行样本评估. 为保证我们工作成果的通用性，在各类型下，让规模参数 m 取值 1~100，参数 n 取值 1~100，每种类型 m，n 皆排列组合 10 000 个样本，基本完全覆盖各种可能的小规模非规则情况，相对于 MKL 同类算法性能，我们完成平均加速统计如表 3 所示. 在双精度小规模非规则实数上平均性能提高 29.4 倍，在双精度小规模非规则复数上平均性能提高 24.6 倍.

6 结束语

本文旨在解决传统 TRSM 算法在处理小规模非规则数据时性能不佳的问题. 我们结合硬件特征、应用场景，分析研究了传统算法在该问题上的不足，提出了适合小规模非规则 TRSM 的自适应优化方案，从寄存器分块、边界处理、向量化计算角度实现了汇编

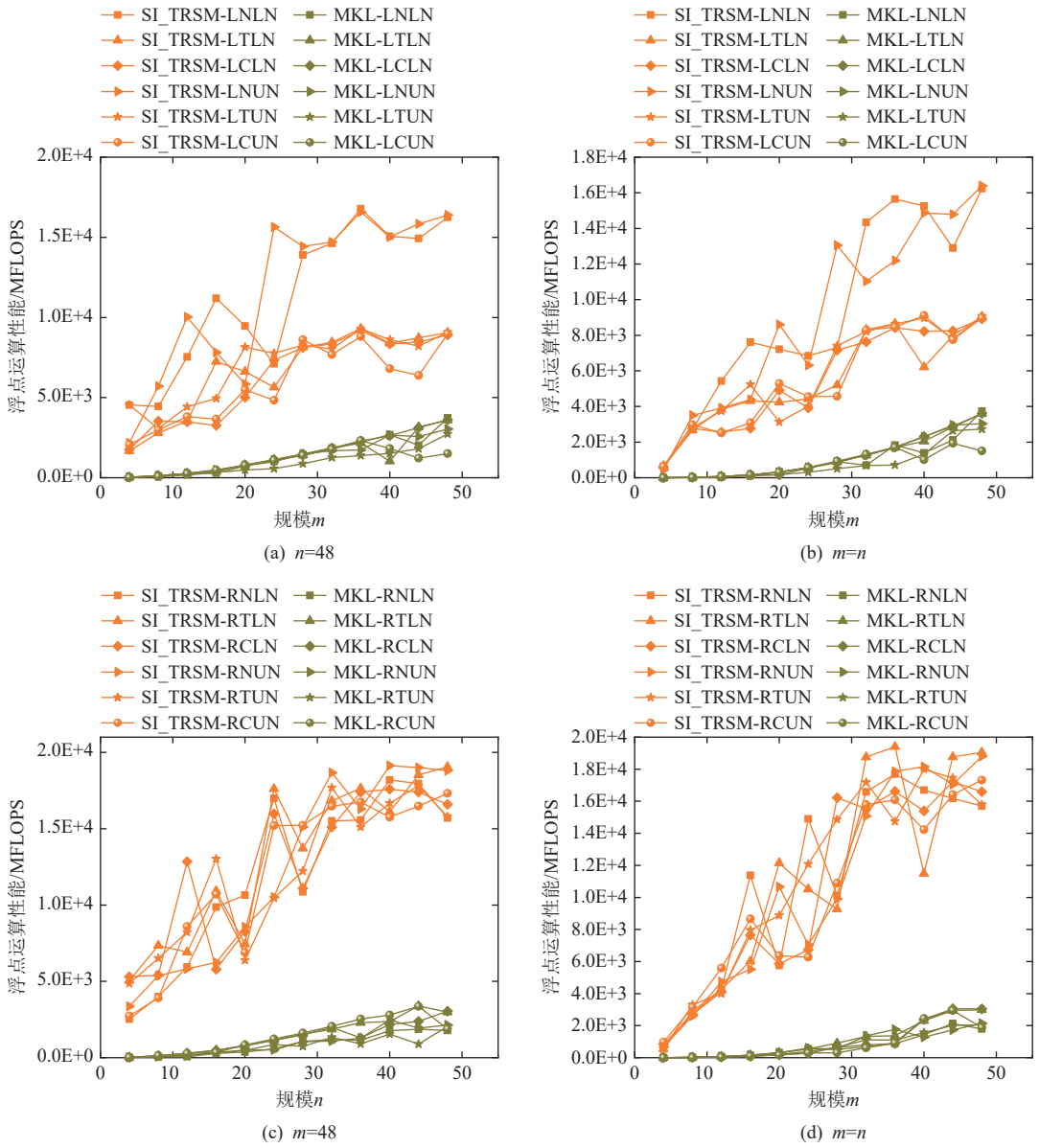


Fig. 12 Comparison of the performance of each type of double-precision complex SI_TRSM and MKL

图 12 双精度复数 SI_TRSM 与 MKL 各类型性能对比

Table 3 Average Acceleration Statistics Relative to MKL

表 3 相对于 MKL 的平均加速统计

类型	平均加速	类型	平均加速
DTRSMLNLN	34.7 倍	ZTRSMLCLN	16.8 倍
DTRSMLTLN	30.1 倍	ZTRSMLNUN	22.1 倍
DTRSMLNUN	32.8 倍	ZTRSMLTUN	28.9 倍
DTRSMLTUN	26.1 倍	ZTRSMLCUN	18.3 倍
DTRSMRNLN	19.8 倍	ZTRSMRNLN	26.5 倍
DTRSMRTLN	37.5 倍	ZTRSMRTLN	25.1 倍
DTRSMRNUN	31.6 倍	ZTRSMRCLN	32.1 倍
DTRSMRTUN	23.2 倍	ZTRSMRNUN	30.1 倍
ZTRSMLNLN	24.7 倍	ZTRSMRTUN	27.1 倍
ZTRSMLTLN	17.1 倍	ZTRSMRCUN	26.5 倍

级别的高性能内核，介绍了实现与优化方法，最终构建了完整规范的双精度小规模非规则 SI_TRSM 算法库.

未来，我们将研究实现单精度小规模非规则 TRSM 算法，进一步研究和优化 TRSM 算法在不同体系结构、应用场景和数据特征下的性能. 同时，我们还将关注 BLAS 领域中其他算法的个性化需求，以实现算法思路的迁移、扩展和创新.

作者贡献声明：郭容园完成代码实现、开展实验和撰写论文；贾海鹏指导研究方法，确立论文结构和协助论文修改；韦存阳指导代码的实现及评估；张云泉、邓明森审阅和指导论文；陈婧蕊、周振亚参与界定研究问题及审核实验结果.

参 考 文 献

[1] BLAS Forum. BLAS: Basic linear algebra subprograms[EB/OL]. [2023-05-28]. http://netlib.org/blas/

[2] Choi J, Dongarra J J, Walker D W. PB-BLAS: A set of parallel block basic linear algebra subprograms[J]. Concurrency: Practice and Experience, 1996, 8(7): 517−535

[3] Intel. Intel oneAPI math kernel library[EB/OL]. [2023-05-21]. http://www.intel.com/content/www/us/en/developer/tools/oneapi/one mkl.html

[4] Zhang Xianyi. OpenBLAS: An optimized BLAS library[EB/OL]. [2023-06-13]. http://www.openblas.net/

[5] AMD. AMDLIB (AMD math library): A high-performance mathematical function library[EB/OL]. [2023-06-29]. https://www.amd.com/en/developer/aocl/libm.html

[6] McNamee J M. Algorithm 408: A sparse matrix package (part I)[J]. Communications of the ACM, 1971, 14(4): 265−273

[7] Dongarra J J, Du C J, Hammarling S, et al. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs[J]. ACM Transactions on Mathematical Software, 1990, 16(1): 18−28

[8] Hanson R. PC basic linear algebra subroutines[EB/OL]. [2023-06-15]. https://www.osti.gov/biblio/1230151

[9] Choi J, Walker D W, Dongarra J J. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers.[J]. Concurrency: Practice and Experience, 1994, 6(7): 543−570

[10] Dongarra J J. Preface: Basic linear algebra subprograms technical (blast) forum standard[J]. The International Journal of High Performance Computing Applications, 2002, 16(1): 1−1

[11] Chan E, Van Zee F G, Bientinesi P, et al. Supermatrix: A multithreaded runtime scheduling system for algorithms-by-blocks[C]//Proc of the 13th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2008: 123−132

[12] Volkov V, Demmel J W. Benchmarking GPUs to tune dense linear algebra[C/OL]//Proc of the 2008 ACM/IEEE Conf on Supercomputing(SC'08). Piscataway, NJ: IEEE, 2008[2023-06-20]. https://ieeexplore.ieee.org/document/5214359

[13] Soorishetty A. Accelerating Linear Algebra and Machine Learning Kernels on a Massively Parallel Reconfigurable Architecture[M]. Arizona: Arizona State University, 2019

[14] Wei Cunyang, Jia Haipeng, Zhang Yunquan, et al. IATF: An input-aware tuning framework for compact BLAS based on ARMv8 CPUs[C/OL]//Proc of the 51st Int Conf on Parallel Processing. New York: ACM, 2022[2023-06-25].https://dl.acm.org/doi/10.1145/3545008.3545032

[15] Valero-Lara P, Greenwalt C, Vetter J S. SparseLU, a novel algorithm and math library for sparse LU factorization[C]//Proc of the 2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3). Piscataway, NJ: IEEE, 2022: 25−31

[16] Hu Yi, Chen Daokun, Yang Chao, et al. Many-core parallel optimization of level–3 BLAS function on domestic SW26010-Pro processor[J]. Journal of Software, 2023, 35(3): 1569−1584 (in Chinese)
(胡怡, 陈道琨, 杨超, 等. 国产 SW26010-Pro 处理器上 3 级 BLAS 函数众核并行优化[J]. 软件学报, 2023, 35(3): 1569−1584)

[17] Whaley R C, Petitet A, Dongarra J J. Automated empirical optimizations of software and the ATLAS project[J]. Parallel Computing, 2001, 27(1/2): 3−25

[18] Lu H, Matz M, Girkar M, et al. System V application binary interface AMD64 architecture processor supplement (with LP64 and ILP32 programming models) version 1.0[EB/OL]. [2023-06-28]. https://cs61.seas.harvard.edu/site/pdf/X86-64-abi-20210928.pdf

[19] Amiri H, Shahbahrami A. SIMD programming using Intel vector extensions[J]. Journal of Parallel and Distributed Computing, 2020, 135: 83−100

[20] Kusswurm D. Modern X86 Assembly Language Programming: Covers X86 64-bit, AVX, AVX2, and AVX−512[M]. Berkeley, CA: Apress, 2018: 421−432

[21] Li Chendi, Jia Haipeng, Cao hang, et al. Autotsmm: An auto-tuning framework for building high-performance tall-and-skinny matrix-matrix multiplication on CPUs[C]//Proc of the 2021 IEEE Intel Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). Piscataway, NJ: IEEE, 2021: 159−166

[22] Yang Weiling, Fang Jianbin, Dong Dezun, et al. LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores[C/OL]//Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2021[2023-06-26].https://dl.acm.org/doi/10.1145/3458817.3476217

[23] Sorokin A, Malkovsky S, Tsoy G. Comparing the performance of general matrix multiplication routine on heterogeneous computing systems[J]. Journal of Parallel and Distributed Computing, 2022, 160: 39−48



Guo Rongyuan, born in 1999. Master candidate. His main research interests include parallel computing and high performance computing.
郭容园, 1999 年生. 硕士研究生. 主要研究方向为并行计算、高性能计算.



Jia Haipeng, born in 1983. PhD, senior engineer. His main research interests include parallel computing and heterogeneous computing.
贾海鹏, 1983 年生. 博士, 高级工程师. 主要研究方向为并行计算、异构计算.



Zhang Yunquan, born in 1973. PhD, professor. His main research interests include parallel algorithm and parallel software.
张云泉, 1973 年生. 博士, 研究员. 主要研究方向为并行算法、并行软件.



Wei Cunyang, born in 1999. Master. His main research interests include parallel computing and high performance computing.

韦存阳, 1999 年生. 硕士. 主要研究方向为并行计算、高性能计算.



Deng Mingsen, born in 1981. PhD, professor. His main research interests include distributed parallel computing and feature extraction of big data.

邓明森, 1981 年生. 博士, 教授. 主要研究方向为分布式并行计算、大数据特征提取.



Chen Jingrui, born in 1987. Master, R&D manager. Her main research interests include circuit simulation and linear equation solvers.

陈婧蕊, 1987 年生. 硕士, 研发经理. 主要研究方向为电路仿真、线性方程求解器.



Zhou Zhenya, born in 1980. Master, senior engineer. His main research interest includes algorithms and software for transistor-level circuit simulation.

周振亚, 1980 年生. 硕士, 高级工程师. 主要研究方向为晶体管级电路仿真的算法和软件.