arXiv:2601.03754v1 [math.OC] 7 Jan 2026

图形处理器加速的块三对角矩阵 Cholesky 分解

RolandSchwan DanielKuhn Colin N. Jones

摘要

本文提出了一种图形处理器加速框架，用于求解工程和科学计算中众多实时应用自然产生的块三对角线性系统。通过采用基于嵌套剖分的多阶段置换策略，我们将计算复杂度从顺序 Cholesky 分解的 $\mathcal{O}(Nn^3)$ 降至并行资源充足时的 $\mathcal{O}(\log_2(N)n^3)$，其中 n 为块大小， N 为块数。该算法采用英伟达的 Warp 库和 CUDA 实现，以利用因式分解算法多个层级的并行性。与稀疏求解器 QDLDL 相比，我们的实现加速比超过 100 倍；与使用 BLASFEO 的高度优化 CPU 实现相比，加速比达到 25 倍；与英伟达的 CUDSS 库相比，加速比则超过 2 倍。随时域长度呈对数级增长的特性，使该方法尤其适用于实时应用中的长时域问题。在 NVIDIA GPU 上进行的全面数值实验表明，该方法在不同问题规模和精度下均具有实际效用。该框架为机器人学、自主系统及其他需要反复求解结构化线性系统的领域中的图形处理器加速优化求解器奠定了基础。该实现采用开源方式，发布于 `https://github.com/PREDICT-EPFL/socu`。

1 引言

块三对角线性系统自然地出现在工程和科学计算领域的众多实时应用中。在模型预测控制（MPC）、轨迹优化和卡尔曼滤波中，底层动力学的时间结构会形成块三对角系统，必须以高频率反复求解。应用场景涵盖多个领域，包括足式机器人 [1], 一级方程式赛车控制 [2], 列车轨迹优化 [3], 和航天器轨迹规划 [4]。为了提升性能并处理复杂约束，这些应用日益需要更长的预测时域；然而，尽管这类系统具有固有的稀疏性，顺序求解器仍然是计算瓶颈，限制了实时应用中可实现的采样率和时域长度。

本文提出了一种专为块三对角矩阵设计的 GPU 加速 Cholesky 分解框架。与修改优化算法本身的现有方法不同，我们直接在线性代数层面引入并行化。对于块大小为 n 、时域为 N的问题，通过嵌套剖分置换策略，在并行资源充足时，我们将计算复杂度从顺序分解的 $\mathcal{O}(Nn^3)$ 降至 $\mathcal{O}(\log_2(N)n^3)$。这种对数级扩展特性对于长时域问题尤为有利，因为随着时域增长，该方法与顺序方法之间的性能差距会显著扩大。该方法保留了底层求解器的数值性质和鲁棒性，因此可广泛应用于各种优化框架。

现有的块三对角求解器并行化方法存在显著局限。利用结构的顺序求解器虽然实现了最优的浮点运算次数，但由于固有的数据依赖 [5]，其复杂度从根本上受限于$\mathcal{O}(Nn^3)$。通用并行稀疏分解方法 [6] 未能充分利用规则的块结构，而基于分区的并行化策略[7] 由于顺序开销较大，只能取得有限的加速比。

此前已有研究探索了模型预测控制的并行化技术，实现了类似的$\mathcal{O}(\log_2(N)n^3)$复杂度 [8, 9, 10], ，并应用于机器人学 [11, 1]。然而，这些方法高度针对模型预测控制，无法适应更一般的问题结构。它们对优化问题本身进行分解，使该方法与模型预测控制的特定结构紧密绑定。相比之下，我们的方法

本文使用大语言模型提升行文质量，包括语法、清晰度和流畅度。本研究作为 NCCR Automation（国家研究能力中心）的一部分获得支持，该中心由瑞士国家科学基金会资助（资助编号：51NF40_225155）。Roland Schwan 和 Colin N. Jones 任职于瑞士洛桑联邦理工学院自动控制实验室。Roland Schwan 和 Daniel Kuhn 任职于瑞士洛桑联邦理工学院风险分析与优化讲席。

{roland.schwan, daniel.kuhn,colin.jones}@epfl.ch

沿用了我们先前工作的思路 [5]，直接在线性代数层面实现并行化。这样可以减少开销，并为未来扩展到更一般的结构提供支持。

本文提出的方法可直接应用于日益受到关注的机器人学优化算法部署。近年来，人们对基于 GPU 的求解器兴趣渐增，这类求解器采用预条件共轭梯度法（PCG）求解底层线性系统 [12, 13, 14]。尽管这些迭代方法具有出色的并行能力，但其收敛性在很大程度上取决于系统的条件性。因此，精度通常受到限制，且所需迭代次数会随问题维度增加而增多。

近年来，将模型预测控制与强化学习（RL）相结合的方式在机器人学应用中日益普及 [15, 16]；详细综述请见 [17]。尽管这种结合能够大幅减少学习所需的样本数量，但代价是每个样本的计算开销增加。这主要是因为底层优化问题的求解效率低下，往往仍然十分缓慢。相比之下，纯粹基于神经网络的策略可完全采用批处理方式，并在图形处理器上执行。本文旨在通过开发利用 GPU 加速的工具，缩小这一差距，使模型预测控制在强化学习应用中具备竞争力。

本文的主要贡献如下：（1）提出一种基于嵌套剖分的多阶段置换策略，可在图形处理器上实现对数复杂度；（2）通过 CUDA 流和原子操作实现操作级并行，进一步缩短关键路径长度；（3）针对不同的问题规模和内存限制，设计了融合式和分块式内核实现；（4）开展了全面的数值实验，结果表明，与稀疏求解器 QDLDL 相比，加速比达到 100 倍至 500 倍；与采用 BLASFEO 库、经过高度优化的 CPU 实现相比，加速比达到 25 倍至 40 倍；与英伟达的闭源 CUDSS 库相比，加速比超过 2 倍。

本文其余部分安排如下。第 2 节回顾分块 Cholesky 分解的三种变体及其内存访问模式。第 3 节介绍顺序基准算法并分析其计算复杂度。第 4 节阐述我们的并行化策略，从基于分区的方法逐步过渡到单阶段置换，最后介绍多阶段置换。第 5 节详述图形处理器实现，包括 CUDA 编程模型、内核融合策略和内存优化技术。第 6 节通过全面的数值实验，在各种问题规模和精度下，将我们的方法与最先进的 CPU 和基于图形处理器的求解器进行比较。最后，第 7 节总结全文并讨论未来的研究方向。

2 分块 Cholesky 分解

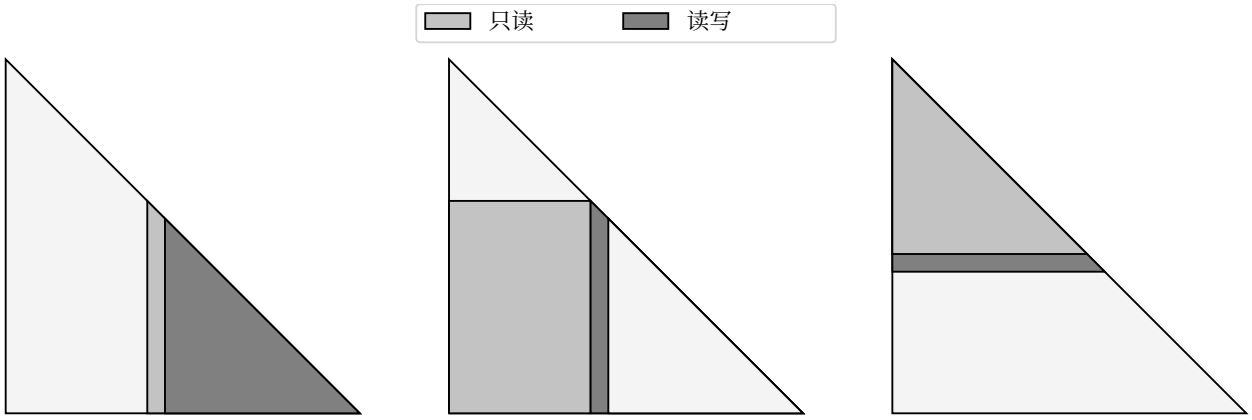
考虑一个正定分块矩阵 $A \in \mathbb{R}^{Nn \times Nn}$，它包含 $N \times N$ 个大小为 $n \times n$ 的块，其中每个块的索引表示为 A_{ij}。稠密 Cholesky 分解本质上由一系列基于以下基本运算的高斯消元步骤构成

$$A_{ij} \leftarrow A_{ij} - \left(A_{ik} A_{jk}^{\top}\right) A_{kk}^{-\top}$$

循环索引 i、j 和 k 可以按任意顺序嵌套，从而产生不同的内存访问模式，并形成图 1 所示的三种不同 Cholesky 分解算法:

1. **子矩阵 Cholesky** （向右看）：在外层循环中依次计算各列，并在内层循环中更新右侧子矩阵。这种方法也称为列立即更新 或扇出。2. **列 Cholesky** （向左看）：依次计算各列，但每列会利用左侧已计算的列，以延迟方式进行更新。这种方法也称为列延迟更新 或 扇入。3. **行 Cholesky** （向上看）：依次计算各行，并根据上方已计算的行进行更新。这种算法也称为行延迟更新 或扇入。

每种算法在内存访问模式及相应的优化数据结构方面各有优缺点，尤其是在稀疏矩阵场景中。更多详情请参见 [6]。后续章节中提出的算法结合了子矩阵和列 Cholesky 方法，在实现并行化的同时避免竞态条件。



子矩阵 Cholesky

对于 $k = 1$ 至 n **执行**
  $A_{kk} \leftarrow$Cholesky 分解(A_{kk})
  对于 $i = k+1$ 至 n **执行**
    $A_{ik} \leftarrow A_{ik} A_{kk}^{-T}$
  循环结束
  对于 $j = k+1$ 至 n，**执行**
    对于 $i = j$ 至 n，**执行**
      $A_{ij} \mathrel{-}= A_{ik} A_{jk}^{T}$
    循环结束
  循环结束
循环结束

列 Cholesky

对 $k = 1$ 至 n **执行**
  对 $j = 1$ 至 $k-1$ **执行**
    对 $i = k$ 至 n **执行**
      $A_{ik} \mathrel{-}= A_{ij} A_{kj}^{T}$
    循环结束
  循环结束
  $A_{kk} \leftarrow$Cholesky 分解(A_{kk})
  对于 $i = k+1$ 至 n **执行**
    $A_{ik} \leftarrow A_{ik} A_{kk}^{-T}$
  结束循环
结束循环

行 Cholesky

对于 $i = 1$ 至 n **执行**
  对于 $j = 1$ 至 i **执行**
    对于 k=1至 $j-1$ **执行**
      $A_{ij} \mathrel{-}= A_{ik} A_{jk}^{T}$
    结束循环
    如果 $i = j$ **则**
      $A_{jj} \leftarrow$Cholesky 分解(A_{jj})
    else
      $A_{ij} \leftarrow A_{ij} A_{jj}^{-T}$
    结束条件判断
  结束循环
结束循环

图 1：三种 Cholesky 分解变体及其内存访问模式。

3 顺序 Cholesky 分解

我们考虑如下形式的对称正定块三对角矩阵

$$\Psi = \begin{bmatrix} D_1 & E_1^\top & & & \\ E_1 & D_2 & E_2^\top & & \\ & E_2 & \ddots & \ddots & \\ & & \ddots & D_{N-1} & E_{N-1}^\top \\ & & & E_{N-1} & D_N \end{bmatrix}$$

其中 $D_i \in \mathbb{R}^{n \times n}$ 为对称对角块，$E_i \in \mathbb{R}^{n \times n}$ 为非对角块。Cholesky 因式分解 $\Psi = LL^\top$ 保留了块三对角结构，从而得到

$$L = \begin{bmatrix} \hat{D}_1 & & & & \\ \hat{E}_1 & \hat{D}_2 & & & \\ & \hat{E}_2 & \ddots & & \\ & & \ddots & \hat{D}_{N-1} & \\ & & & \hat{E}_{N-1} & \hat{D}_N \end{bmatrix}$$

^^其中，D_i 是下三角对角块，E_i 是相应的分解后的非对角块。

算法 1 展示了标准的左看顺序分解。该算法首先对第一个对角块进行因式分解，然后通过三角求解（`trsm`）求解非对角块，处理每个后续块行：使用对称秩-k 更新（`syrk`）更新对角块，并计算其 Cholesky 分解（`potrf`）。

该算法通过保留稀疏结构且不引入填充元素，实现了最优的浮点运算次数。然而，它存在相邻

算法 1 Ψ的顺序分解

1: $\hat{D}_1 \leftarrow$Cholesky 分解(D_1) `potrf` $\frac{n^3}{3}$
2: **对于** $i = 2, \ldots, N$ **do**
3: $\hat{E}_{i-1} \leftarrow E_{i-1}\hat{D}_{i-1}^{-\top}$ `trsm` n^3
4: $\hat{D}_i \leftarrow D_i - \hat{E}_{i-1}\hat{E}_{i-1}^{\top}$ `syrk` n^3
5: $\hat{D}_i \leftarrow$Cholesky 分解($\hat{D}_i$) `potrf` $\frac{n^3}{3}$
6: **结束循环**

表 1：基本矩阵运算的代价。

A：$m \times n$，B：$n \times p$，D，L：$n \times n$，L 下三角。

运算	内核	代价（浮点运算次数）
矩阵乘法 $A \cdot B$	`gemm`	$2mnp$
对称矩阵乘法 $A \cdot A^T$	`syrk`	m^2n
Cholesky 分解，使得 $LL^T = D$	`potrf`	$1/3n^3$
求解三角矩阵 $A \cdot L^{-T}$	`trsm`	mn^2

线程块之间固有的顺序依赖。线程块 i 必须等到线程块 $i-1$ 完全分解后才能开始处理，因为操作 `trsm` $\hat{}$ 需要使用前一次迭代中已完成分解的对角块 D_{i-1}。

3.1 计算复杂度

使用表 1 中的运算成本，顺序算法的每次迭代需要在 $n \times n$ 个线程块上执行一次 `trsm`、`syrk` 和 `potrf` 运算，每个线程块总计需要 $\frac{7n^3}{3}$ 次浮点运算。由于第一个线程块只需执行初始 Cholesky 分解，成本为 $\frac{n^3}{3}$ 次浮点运算，因此总计算成本为 —

$$\left(\frac{7}{3}N - 2\right) n^3$$

浮点运算次数。这使得复杂度为$\mathcal{O}(Nn^3)$，并随线程块数量 N 线性增长。然而，顺序依赖排除了跨线程块并行化的可能，因此我们在后续章节中提出矩阵重排序策略，以揭示因式分解过程中的并行性。

4 并行化策略

算法 1 中的顺序依赖源于线程块的自然顺序排列，每个线程块都必须等待前一个线程块完成。然而，通过精心选择置换来重新排列矩阵线程块，可以放宽这一依赖。应用置换矩阵 P 后，我们可以将原问题 $\Psi = LL^{\top}$ 转换为 $P\Psi P^{\top} = \hat{L}\hat{L}^{\top}$，其中置换后的矩阵揭示了并行计算的机会。

关键思路是将线程块划分为若干独立集合，使其能够并发处理，同时保证因式分解在数学上的正确性。我们提出了两种不同的置换方法，各自具有不同的计算权衡。第一种方法面向并行单元有限且同步开销可能较高的场景，这类场景常见于经典 CPU 和分布式计算集群。随后，我们将这一框架扩展至并行处理能力强大、同步成本低至中等的设备，这类设备以现代图形处理器为代表。

4.1 分区置换

第一种方法改编自 [7], 将矩阵划分为 矩阵 Ψ，如下所示:

$$
\Psi = \left[\begin{array}{ccc|cc|ccc|c|cc}
D_{11} & \star & & & & & & & & & \\
E_{11} & \ddots & \ddots & & & & & & & & \\
 & \ddots & D_{N_1 1} & \star & & & & & & & \\
\hline
 & & F_1 & A_2 & \star & & & & & & \\
\hline
 & & & B_2 & D_{12} & \star & & & & & \\
 & & & & E_{12} & \ddots & \ddots & & & & \\
 & & & & & \ddots & D_{N_2 2} & \star & & & \\
\hline
 & & & & & & F_2 & A_3 & \star & & \\
\hline
 & & & & & & & B_3 & & & \\
 & & & & & & & & \ddots & & \\
 & & & & & & & & & \star & \\
\hline
 & & & & & & & & & B_p & D_{1p} \quad \star \\
 & & & & & & & & & & E_{1p} \quad \ddots \quad \ddots \\
 & & & & & & & & & & \ddots \quad D_{N_p p}
\end{array}\right],
$$

其中， $p-1$ 主元将矩阵划分为 p 块，可由 p 线程并行处理。置换 P_p 将主元移至末尾，从而得到置换后的矩阵

$$
\Psi_p = \left[\begin{array}{ccc|ccc|c|c||ccc|c}
D_{11} & \star & & & & & & & & & & \\
E_{11} & \ddots & \ddots & & & & & & & & & \\
 & \ddots & D_{N_1 1} & & & & & & & & & \\
\hline
 & & & D_{12} & \star & & & & & & & \\
 & & & E_{12} & \ddots & \ddots & & & & & & \\
 & & & & \ddots & D_{N_2 2} & & & & & & \\
\hline
 & & & & & & \ddots & & & & & \\
\hline
 & & & & & & & D_{1p} & \star & & & \\
 & & & & & & & E_{1p} & \ddots & \ddots & & \\
 & & & & & & & & \ddots & D_{N_p p} & & \\
\hline\hline
 & & F_1 & B_2^\top & & & & & & A_2 & & & \\
\hline
 & & & & & F_2 & B_3^\top & & & & A_3 & & \\
\hline
 & & & & & & \ddots & & & & & \ddots & \\
\hline
 & & & & & & F_{p-1} & B_p^\top & & & & & A_p
\end{array}\right].
$$

通过主元重定位解耦各个分区的列，从而打破 Cholesky 分解中的顺序依赖。由此得到的乔列斯基因子 L_p 为

$$
L_p = \left[\begin{array}{cccc|ccc|ccc|c}
\hat{D}_{11} & & & & & & & & & & \\
\hat{E}_{11} & \ddots & & & & & & & & & \\
 & \ddots & \hat{D}_{N_1 1} & & & & & & & & \\
\hline
 & & & \hat{D}_{12} & & & & & & & \\
 & & & \hat{E}_{12} & \ddots & & & & & & \\
 & & & & \ddots & \hat{D}_{N_2 2} & & & & & \\
\hline
 & & & & & \ddots & & & & & \\
\hline
 & & & & & & \hat{D}_{1p} & & & & \\
 & & & & & & \hat{E}_{1p} & \ddots & & & \\
 & & & & & & & \ddots & \hat{D}_{N_p p} & & \\
\hline
 & & \hat{F}_1 & \hat{B}_{12}^\top & \cdots & \hat{B}_{N_2 2}^\top & & & & \hat{A}_2 & & & \\
 & & & & \hat{F}_2 & \hat{B}_{13}^\top & \cdots & \hat{B}_{N_3 3}^\top & & \hat{H}_2 & \hat{A}_3 & & \\
 & & & & & \ddots & & & & & \ddots & \ddots & \\
 & & & & & & \hat{F}_{p-1} & \hat{B}_{1p}^\top & \cdots & \hat{B}_{N_p p}^\top & & \hat{H}_{p-1} & \hat{A}_p
\end{array}\right].
$$

^^ 注意，置换会产生额外的填充项 B_{ik}和 H_k，分别对应 $i = 2, \ldots, N_k$和 $k = 2, \ldots, p$。虽然填充会增加总计算成本，但这种方法实现的并行化所带来的收益可能超过额外运算的代价。

算法 2 给出了针对 Ψ_p的详细分解过程，该过程分为两个不同阶段：先是并行阶段，随后是顺序阶段。通过将主元移至矩阵末尾，我们消除了依赖关系，并实现了各分区的并行化；之后进入顺序阶段，其过程与算法 1 非常相似。

为说明这一方法，考虑对 Ψ_p 的第二个分区进行因式分解。包含元素 $D_{12,\ldots}$和 $E_{12,\ldots}$的块三对角子矩阵的因式分解遵循顺序算法 1，对应算法 2 中第 5 至第 7 行。与顺序算法的关键区别在于，我们还必须处理由主元产生的填充，如第 8 至第 12 行所示。具体而言，引入置换$B_2^\top$和 E_{12} 会产生填充$B_{22}^\top$，随后该填充递归传播，最终遍及整行，即 $B_{12,\ldots}^\top, B_{N_2 2}^\top$。

现在讨论主元 A_i 及其相关填充 H_i的更新。算法 2 经过精心设计，可避免这些更新中的竞态条件。例如， F_i 和 $B_{i,k+1}$ 由不同线程处理，因此无法同时更新 A_{i+1}（针对 $i = 1,\ldots,p-1$）。为解决这一问题，我们将F_i 的更新延迟到第 23 行的顺序阶段执行，这属于一种左看操作。因此，算法 2 采用了混合左看/右看方法。

详细分析算法 2 可得出各阶段的计算成本如下：

	并行阶段（$k=1$）	并行阶段（$k>1$）	顺序阶段
成本（浮点运算次数）	$\left(\frac{7}{3}N_1 - 1\right) n^3$	$\left(\frac{19}{3}N_k - 1\right) n^3$	$\left(\frac{10}{3}p - \frac{16}{3}\right) n^3$

第一个分区的每个时域元素 N_1 成本明显低于其他分区，因此采用以下最优分配策略：

命题1 。对于一个共有 N 个对角元素的问题， N_1 与 N_k 之间的最优比率为 $rk > 1$ 给出y

$$\frac{N_1^\star}{N_k^\star} = \frac{19}{7} \approx 2.71,$$

with

$$N_1^\star = \frac{19N - 19p + 19}{7p + 12} \; and \; N_k^\star = \frac{7N - 7p + 7}{7p + 12}.$$

证明。 由于并行阶段中的并行线程彼此独立，因此总成本应在 $k = 1$ 和 $k > 1$ 之间平均分配，即 $\left(\frac{7}{3}N_1 - 1\right) n^3 = \left(\frac{19}{3}N_k - 1\right) n^3$。此外，总节点数必须保持一致，即 $N - (p-1) = N_1 + (p-1)N_k$。求解该线性方程组即可得到上述结果。 □

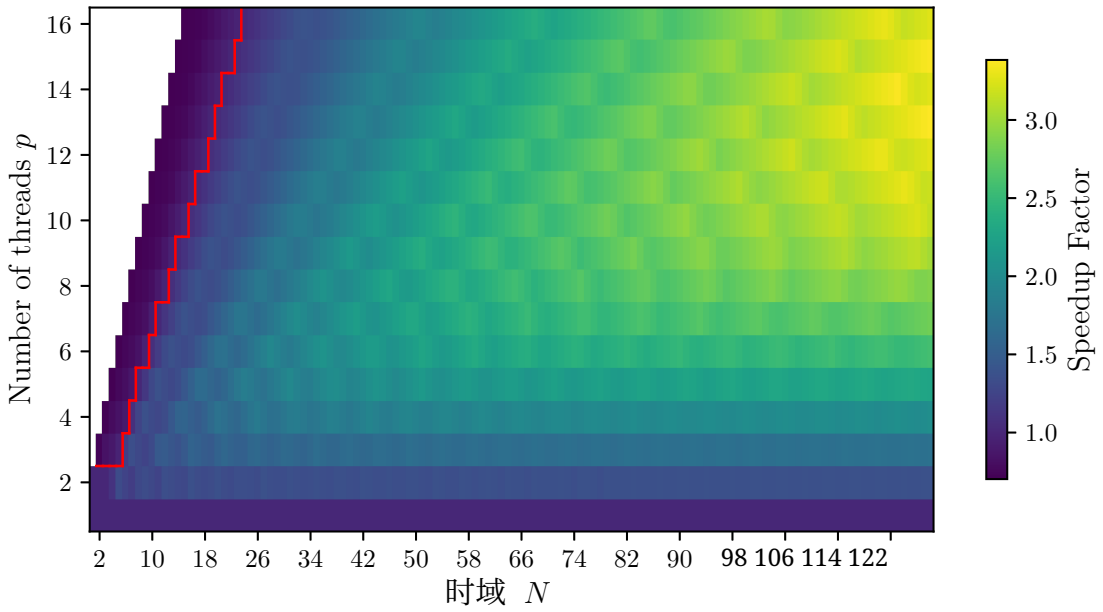


图 2：使用 p 个线程时，并行分解 Ψ_p 的理论加速与顺序分解 Ψ相比的结果。红线边界将实际加速与减速分隔开来。

请注意，最优比率与块大小 n 和分区数 p 均无关。由于 $N_1^\star$ 和 $N_k^\star$ 通常不是整数，我们采用一种实用的舍入策略：将 $N_k^\star$ 分别向下和向上取整，计算对应的 $N_1^\star = N-(p-1)N_k^\star$，评估相应的成本，并选择最大成本最低的配置。

图 2 展示了与顺序因式分解相比，我们的方法所实现的理论加速比。加速比随着线程数 p 和时域长度 N 的增加而提高。然而，当线程数较高时，顺序部分的固定开销会变得显著；当时域较短时，这可能导致速度下降。这表明，对于较短的时域，限制线程数可能更有利。此外，对于任意给定的线程数，加速比都会饱和于一个最大值。如$N \to \infty$所示，理论最大加速比趋近于

$$\frac{7p}{19} + \frac{12}{19}$$

对于 p 个线程。这一基本限制意味着，即使使用两个线程，最大可实现的加速比也只有 36.8%，而四个线程的加速比仅略高于 2.1 倍。

我们实现了 这种并行化策略，用于我们的求解器 PIQP [19]，作为利用结构利用回代 端 [5]。

4.2 单阶段置换

本节中，我们关注具有强大并行处理能力和高效同步机制的设备，例如图形处理器。我们将先前的方法扩展到极端情况，即有 $p \geq \lceil N/2 \rceil$ 个并行处理单元可用。我们定义置换 P_1，使得

$$\Phi_1 = P_1 \Psi P_1^\top = \left[\begin{array}{ccccccc|ccccccc} D_1 & & & & & & & & & & & & & \\ & D_3 & & & & & & & & & & & & \\ & & D_5 & & & & & & & \star & & & & \\ & & & D_7 & & & & & & & & & & \\ & & & & \ddots & & & & & & & & & \\ \hline E_1 & E_2^\top & & & & & & D_2 & & & & & & \\ & E_3 & E_4^\top & & & & & & D_4 & & & & & \\ & & E_5 & E_6^\top & & & & & & D_6 & & & & \\ & & & \ddots & \ddots & & & & & & \ddots & & & \end{array}\right].$$

算法 2 Ψ_p的并行分解

并行阶段

对 $k = 1, \ldots, p$ **并行执行**

$\hat{A}_k \leftarrow A_k$ **如果** $k > 1$

$\hat{B}_{1,k} \leftarrow B_k$ **如果** $k > 1$

对于 $i = 1, \ldots, N_p - 1$ **执行**

$\hat{D}_{ik} \leftarrow$ Cholesky 分解(D_{ik}) potrf $\frac{n^3}{3}$

$\hat{E}_{ik} \leftarrow E_{ik} \hat{D}_{ik}^{-\top}$ trsm n^3

$\hat{D}_{i+1,k} \leftarrow D_{i+1,k} - \hat{E}_{ik}\hat{E}_{ik}^{\top}$ syrk n^3

如果 $k > 1$ **那么**

$\hat{B}_{i,k}^{\top} \leftarrow \hat{B}_{i,k}^{\top} \hat{D}_{ik}^{-\top}$ trsm n^3

$\hat{A}_k \leftarrow \hat{A}_k - \hat{B}_{ik}^{\top}\hat{B}_{ik}$ syrk n^3

$\hat{B}_{i+1,\ k}^{\top} \leftarrow -\hat{B}_{ik}^{\top}\hat{E}_{ik}^{\top}$ gemm $2n^3$

结束 if

结束 for

$\hat{D}_{N_k k} \leftarrow$ Cholesky 分解$(D_{N_k k})$ potrf $\frac{n^3}{3}$

如果 $k > 1$**，则**

$\hat{B}_{N_k k}^{\top} \leftarrow \hat{B}_{N_k k}^{\top} \hat{D}_{N_k k}^{-\top}$ trsm n^3

$\hat{A}_k \leftarrow \hat{A}_k - \hat{B}_{N_k k}^{\top}\hat{B}_{N_k k}$ syrk n^3

结束 if

$\hat{F}_k \leftarrow F_k \hat{D}_{N_k k}^{-\top}$ **如果** $k < p$ trsm n^3

$\hat{H}_k \leftarrow -\hat{F}_k \hat{B}_{N_k k}$ **如果** $1 < k < p$ gemm $2n^3$

结束 for

顺序阶段

对于 $k = 2, \ldots, p-1$**，执行**

$\hat{A}_k \leftarrow \hat{A}_k - \hat{F}_{k-1}\hat{F}_{k-1}^{\top}$ syrk n^3

$\hat{A}_k \leftarrow$ Cholesky 分解$(\hat{A}_k)$ potrf $\frac{n^3}{3}$

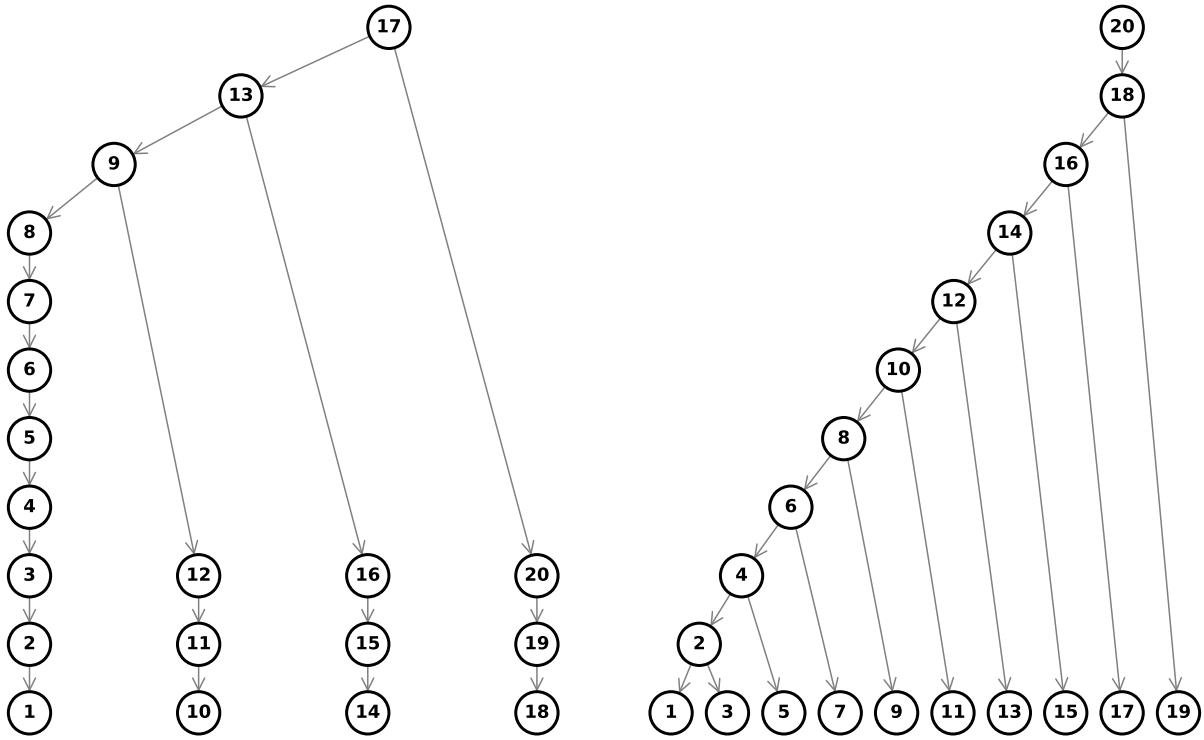
$\hat{H}_k \leftarrow \hat{H}_k \hat{A}_k^{-\top}$ trsm n^3

$\hat{A}_{k+1} \leftarrow \hat{A}_{k+1} - \hat{H}_k\hat{H}_k^{\top}$ syrk n^3

结束循环

$\hat{A}_p \leftarrow \hat{A}_p - \hat{F}_{p-1}\hat{F}_{p-1}^{\top}$ syrk n^3

$\hat{A}_p \leftarrow$ Cholesky 分解$(\hat{A}_p)$ potrf $\frac{n^3}{3}$



(a) Ψ_4 for $N = 20$的消去树。

(b) Φ_1 for $N = 20$的消去树。

图 3： A for $N = 20$不同置换方式下的消去树。

这种重新排序形成了一种块结构，其中所有奇数索引对角块（D_1， D_3、.....）都排在前面，后面是偶数索引块（D_2， D_4、.....）。关键观察是，所有奇数索引块彼此独立，可以并行分解，因为它们只依赖于原始矩阵数据。Cholesky 分解 $\Phi_1 = \hat{L}_1\hat{L}_1^\top$因此表示为

$$\hat{L}_1 = \left[\begin{array}{ccccc|ccccc} \hat{D}_1 & & & & & & & & & \\ & \hat{D}_3 & & & & & & & & \\ & & \hat{D}_5 & & & & & & & \\ & & & \hat{D}_7 & & & & & & \\ & & & & \ddots & & & & & \\ \hline \hat{E}_1 & \hat{E}_2^\top & & & & \hat{D}_2 & & & & \\ & \hat{E}_3 & \hat{E}_4^\top & & & \hat{H}_1 & \hat{D}_4 & & & \\ & & \hat{E}_5 & \hat{E}_6^\top & & & \hat{H}_2 & \hat{D}_6 & & \\ & & & \ddots & \ddots & & & \ddots & \ddots & \end{array}\right]$$

ˆ 其中，置换会产生新的填充块 H_i 。

为了理解为何能够实现这种并行化，以 Φ_1 的第二列为例。元素 D_3 和 $E_2^\top$会更新 ˆ D_2，而 D_3 和 E_3 会更新 ˆ D_4， $E_2^\top$和 E_3 则会产生填充 ˆ H_1。对于 Φ_1 中其他所有包含奇数索引对角元素的列，这种模式也同样成立。关键是，在此过程中，我们不会更新其他包含奇数索引对角元素的列中的任何值。因此，这些列的因式分解可以完全并行执行，彼此之间不存在依赖关系。

可通过消去树 [6] 来直观表示 Cholesky 分解中的计算依赖关系。消去树是理解计算依赖关系并预测所得因式分解中填充模式的强大工具。父节点必须等其所有子节点都完成处理后才能处理。此外，如果在消去节点 i的过程中，存在一条从节点 i 到节点 k 且经过节点 j 的路径，则位置（j， k）处会产生填充。

检查图 3b 中所示的 Φ_1 消去树，我们发现奇数索引对角块彼此独立。最低层不存在依赖关系，因此可以并行处理。

算法 3 Φ_1的并行分解

并行执行 for $i = 1, 3, \ldots, N$ **do**			
$\hat{D}_i \leftarrow$Cholesky 分解(D_i)		potrf	$\frac{n^3}{3}$
$\hat{E}_i \leftarrow E_i \hat{D}_i^{-\top}$	如果 $i+1 \le N$	trsm	n^3
$\hat{D}_{i+1} \leftarrow D_{i+1} - \hat{E}_i \hat{E}_i^\top$	如果 $i+1 \le N$	syrk	n^3
$\hat{E}_{i-1} \leftarrow \hat{D}_i^{-1} E_{i-1}$	如果 $i > 1$	trsm	n^3
$H_{(i-1)/2} \leftarrow -\hat{E}_i \hat{E}_{i-1}$	如果 $i > 1$ 且 $i+1 \le N$	gemm	$2n^3$
结束循环			
对 $i = 2,\ 4,\ \ldots,\ N-1$ **并行执行**			
$\hat{D}_i \leftarrow \hat{D}_i - \hat{E}_i^\top \hat{E}_i$		syrk	n^3
结束循环			
$\hat{D}_2 \leftarrow$Cholesky 分解$(\hat{D}_2)$		potrf	$\frac{n^3}{3}$
对于 $i = 4, 6, \ldots, N$ **do**			
$H_{i/2-1} \leftarrow H_{i/2-1} \hat{D}_{i-2}^{-\top}$		trsm	n^3
$\hat{D}_i \leftarrow \hat{D}_i - H_{i/2-1} H_{i/2-1}^\top$		syrk	n^3
$\hat{D}_i \leftarrow$Cholesky 分解$(\hat{D}_i)$		potrf	$\frac{n^3}{3}$
结束循环			

Ψ_4 消去树同样揭示了四个分区的并行处理结构。然而，它也暴露了置换 Φ_1 的局限性：虽然第一层可以完全并行化，但其余节点必须按顺序处理。这一观察表明，降低树高是可取的，因为它决定了所需顺序操作的最少数量。我们将在下一节讨论这一问题。

算法 3 展示了对 Φ_1 进行因式分解的优化并行版本，这是算法 2 的一个特例。使用 p 个并行线程时，计算成本为

$$\left(\left\lceil \frac{\lceil N/2 \rceil}{p} \right\rceil \frac{19}{3} + \left\lfloor \frac{N}{2} \right\rfloor \frac{7}{3} - 2 \right) n^3$$

$N \ge 3$ 的浮点运算次数。图4展示了相对于顺序算法的加速比。尽管实现了显著的并行性，第二阶段的顺序特性仍限制了整体性能。此外，若不并行执行，填充会带来显著开销。如图4所示，由于这一开销，只有在线程数超过 3–4 时才能实现加速。

4.3 多阶段置换

单阶段置换的主要局限在于其漫长的顺序尾部无法并行化。解决方法是递归置换顺序部分，直至其中不再有元素。这种方法的结果与应用嵌套剖分置换相同 [20], ，而嵌套剖分置换已被证明在并行稀疏分解方法中极为有效 [6]。将这种递归策略应用于 Ψ ，可得到置换后的矩阵

$$\Phi_\infty = P_\infty \Psi P_\infty^\top = \left[\begin{array}{ccccccc|ccc|ccc|c}
D_1 & & & & & & & & & & & & & \\
 & D_3 & & & & & & & & & & & & \\
 & & D_5 & & & & & & \star & & & \star & & \star \\
 & & & D_7 & & & & & & & & & & \\
 & & & & \ddots & & & & & & & & & \\
\hline
E_1 & E_2^\top & & & & & & D_2 & & & & & & \\
 & & E_5 & E_6^\top & & & & & D_6 & & & & & \\
 & & & & E_9 & & & & & D_{10} & & & & \\
 & & & & \ddots & & & & & & \ddots & & & \\
\hline
 & E_3 & E_4^\top & & & & & & & & & D_4 & & \\
 & & & & & & & & & & & & D_{12} & \\
 & & & & \ddots & & & & & & & & & \ddots \\
\hline
 & & & & \ddots & & & & & & & & & \ddots
\end{array}\right],$$

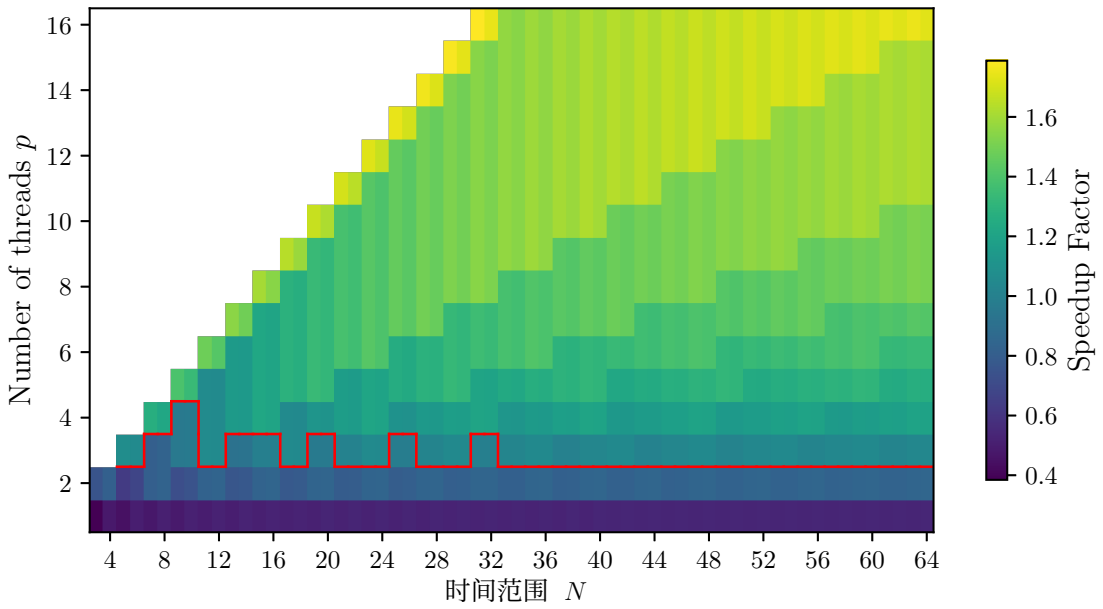


图4：与对 Ψ 进行顺序分解相比，对 Φ_1 进行并行分解的加速比。红线边界将实际加速与减速区分开来。

及其对应的 Cholesky因子

$$\hat{L}_\infty = \left[\begin{array}{cccccc|cccc|ccc|c} \hat{D}_1 & & & & & & & & & & & & & \\ & \hat{D}_3 & & & & & & & & & & & & \\ & & \hat{D}_5 & & & & & & & & & & & \\ & & & \hat{D}_7 & & & & & & & & & & \\ & & & & \ddots & & & & & & & & & \\ \hline \hat{E}_{1,1} & \hat{E}_{1,2}^\top & & & & & \hat{D}_2 & & & & & & & \\ & & \hat{E}_{1,5} & \hat{E}_{1,6}^\top & & & & \hat{D}_6 & & & & & & \\ & & & & \hat{E}_{1,9} & & & & \hat{D}_{10} & & & & & \\ & & & & \ddots & & & & & \ddots & & & & \\ \hline & \hat{E}_{1,3} & \hat{E}_{1,4}^\top & & & & \hat{E}_{2,1} & \hat{E}_{2,2}^\top & & & \hat{D}_4 & & & \\ & & & & & & & & \hat{E}_{2,5} & \hat{E}_{2,6} & & \hat{D}_{12} & & \\ & & & & \ddots & & & & & \ddots & & & \ddots & \\ \hline & & & & \ddots & & & & & \ddots & & & \ddots & \ddots \end{array}\right].$$

虽然得到的置换相同，但我们能够预先了解其结构，因此可以优化内存布局和访问模式。正如后文所示，这能显著提升性能。

请注意，对原始块三对角矩阵应用（块）循环约简 [21, 22] 时，也会隐式地产生相同的置换。尽管两者等价，但从计算置换后的矩阵的 Cholesky 分解这一角度来看，方法似乎更为通用，并且能够自然地扩展到箭形结构，类似于 [18]。

算法 4 详细说明了 Φ_∞ 的分解过程，其可视化表示见图 5。每次迭代中，多个并行线程会处理一个层级。

为说明该算法，我们将逐步讲解图 5 中概述的示例，其中处理的是第二次迭代的第二列。执行 Cholesky 分解 $\hat{D}_6 \leftarrow \text{chol}(\hat{D}_6)$ 的操作，随后进行三角求解 $\hat{E}_{2,3} \leftarrow \hat{E}_{2,3}\hat{D}_6^{-\top}$ and $\hat{E}_{2,2} \leftarrow \hat{D}_6^{-1}\hat{E}_{2,2}$ (cdiv)，以及产生更新项 $\hat{E}_{3,1} \leftarrow -\hat{E}_{2,3}\hat{E}_2^\top{}_{,2}$ 的填充操作，都与单阶段基础算法中的操作相同。但是，我们必须谨慎避免引入竞态条件。在纯右看算法中，我们会同时更新 $\hat{D}_4 \leftarrow \hat{D}_4 - \hat{E}_2^\top{}_{,2}\hat{E}_{2,2}$ 和 $\hat{D}_8 \leftarrow \hat{D}_8 - \hat{E}_{2,3}\hat{E}_2^\top{}_{,3}$。遗憾的是，处理第一列的线程也会更新 $\hat{D}_4 \leftarrow \hat{D}_4 - \hat{E}_{2,1}\hat{E}_2^\top{}_{,1}$，从而导致竞态条件。为避免这种情况，我们将操作 $\hat{D}_4 \leftarrow \hat{D}_4 - \hat{E}_2^\top{}_{,2}\hat{E}_{2,2}$ 延迟到下一次迭代，正如图 5 中的虚线箭头所示。在随后

算法 4 针对并行优化的 Φ_∞ 因式分解

并行执行 $i = 1, \ldots, N$ **中的循环**
  $\hat{D}_i \leftarrow D_i$
  $\hat{E}_{1,i} \leftarrow E_i$ 如果 $i < N$
结束循环
循环 $s = 1, 2, 4, \ldots, 2^{\lfloor \log_2(N) \rfloor}$ **操作**
  对 $i = s,\ 3s,\ 5s,\ \ldots\ldots,\ N$**执行并行循环**
    $\hat{D}_i \leftarrow \hat{D}_i - \hat{E}_{s-1,2i/s}^\top \hat{E}_{s-1,2i/s}$ 如果 $s > 1$ 且 $i \leq N - s/2$ `syrk` n^3
    $\hat{D}_i \leftarrow$ Cholesky 分解$(\hat{D}_i)$ `potrf` $\frac{n^3}{3}$
    $\hat{D}_{i+s} \leftarrow \hat{D}_{i+s} - \hat{E}_{s-1,2i/s+2}^\top \hat{E}_{s-1,2i/s+2}$ 如果 $s > 1$ 和 $i + s \leq N - s/2$ `syrk` n^3
    $\hat{E}_{s,i/s} \leftarrow \hat{E}_{s,i/s} \hat{D}_i^{-\top}$ 如果 $i + s \leq N$ `trsm` n^3
    $\hat{D}_{i+s} \leftarrow \hat{D}_{i+s} - \hat{E}_{s,i/s} \hat{E}_{s,\ i/s}^\top$ 如果 $i + s \leq N$ `syrk` n^3
    $\hat{E}_{s,\ i/s-1} \leftarrow \hat{D}_i^{-1} \hat{E}_{s,\ i/s-1}$ 如果 $i > s$ `trsm` n^3
    $\hat{E}_{s+1,\ (i-s)/(2s)} \leftarrow -\hat{E}_{s,\ i/s} \hat{E}_{s,\ i/s-1}$ 如果 $i > s$ 和 $i + s \leq N$ `gemm` $2n^3$
  结束循环
结束循环

的迭代中，这就成为一个左看操作。在这个具体示例中，这些操作对应于 $\hat{D}_6 \leftarrow \hat{E}_{1,\ 6}^\top \hat{E}_{1,\ 6}$ 和 $\hat{D}_8 \leftarrow \hat{E}_{1,\ 8}^\top \hat{E}_{1,\ 8}$。因此，该算法成为一种混合左看/右看算法。

该算法需要 $\lfloor \log_2(N) \rfloor + 1$ 次迭代，每次迭代都可以并行执行。在有 p 个线程可用的情况下，计算成本变为

$$\left(\left\lceil \frac{N/2}{p} \right\rceil \frac{16}{3} + \sum_{i=1}^{\lfloor \log_2(N) \rfloor - 1} \left\lceil \frac{\lceil N/2^{i+1} \rceil}{p} \right\rceil \frac{22}{3} + \frac{4}{3} \right) n^3 \tag{1}$$

浮点运算次数，适用于 $N \geq 2$。第一项对应初始迭代，其中不包含左看操作。最后一项表示最后一次迭代，由一次左看操作和一次 Cholesky 分解组成。当 $p < N/2$时，第一项满足 $\left\lceil \frac{N/2}{p} \right\rceil > 1$，这表明可用线程不足以将第一级完全并行化。这会导致初始迭代的执行时间增加，进而提高总体计算成本。尽管现代图形处理器提供了大量并行单元，但对于足够大的 N，我们最终会使这些并行单元达到饱和，从而使该项变得显著。

这种方法将顺序情形下的复杂度从 $\mathcal{O}(Nn^3)$ 降低到并行多阶段分解的 $\mathcal{O}(\log_2(N)n^3)$。图 6 展示了相对于顺序分解的预期加速比。图中在 $\lfloor \log_2(N) \rfloor$ 处出现了离散断点，断点之间的加速比线性增长。这一现象源于消去树的高度对于 $N \in [2^i, 2^{i+1})$ 保持不变，而在并行核心数量充足时，每次迭代的成本也保持不变。

图 7 比较了迄今讨论的所有并行分解方法相对于 Ψ 的顺序分解的加速比表现。我们还给出了两阶段变体的预期加速比；该变体采用了多阶段置换的基本思路，但在推导过程中于第二次递归迭代后终止。虽然这里不详细讨论这一变体，但它提供了一个有用的中间基准。我们观察到，不同方法的加速比会随时域长度 N 而变化，其中一些方法的表现甚至不如顺序分解方案。这种差异源于各方法产生的开销和填充量不同。因此，可以设想一种调度机制，根据给定的问题实例选择能够实现最佳加速比的方法。本章余下部分将重点讨论多阶段变体，因为它在较长时域 N 下可提供最大的加速比，而自适应方法选择则留待未来研究。

4.3.1 Cholesky 求解

到目前为止，我们只研究了因式分解 $P_\infty \Psi P_\infty^\top = \hat{L}_\infty \hat{L}_\infty^\top$。为了求解右端项，我们将 $\Psi x = b$重写为 $\Psi x = P_\infty^\top \hat{L}_\infty \hat{L}_\infty^\top P_\infty x = b$，其中 $x,\ b \in \mathbb{R}^{Nn \times m}$。按照标准的 Cholesky 方法，我们通过两个阶段求得 x：先进行前代，再进行回代。在前代阶段，我们求解 $P_\infty^\top \hat{L}_\infty y = b$，得到辅助变量 $y \in \mathbb{R}^{Nn \times m}$；随后在回代阶段求解 $\hat{L}_\infty^\top P_\infty x = y$。算法 5 给出了这一过程。求解阶段需要 $2\lfloor \log_2(N) \rfloor + 2$ 次迭代

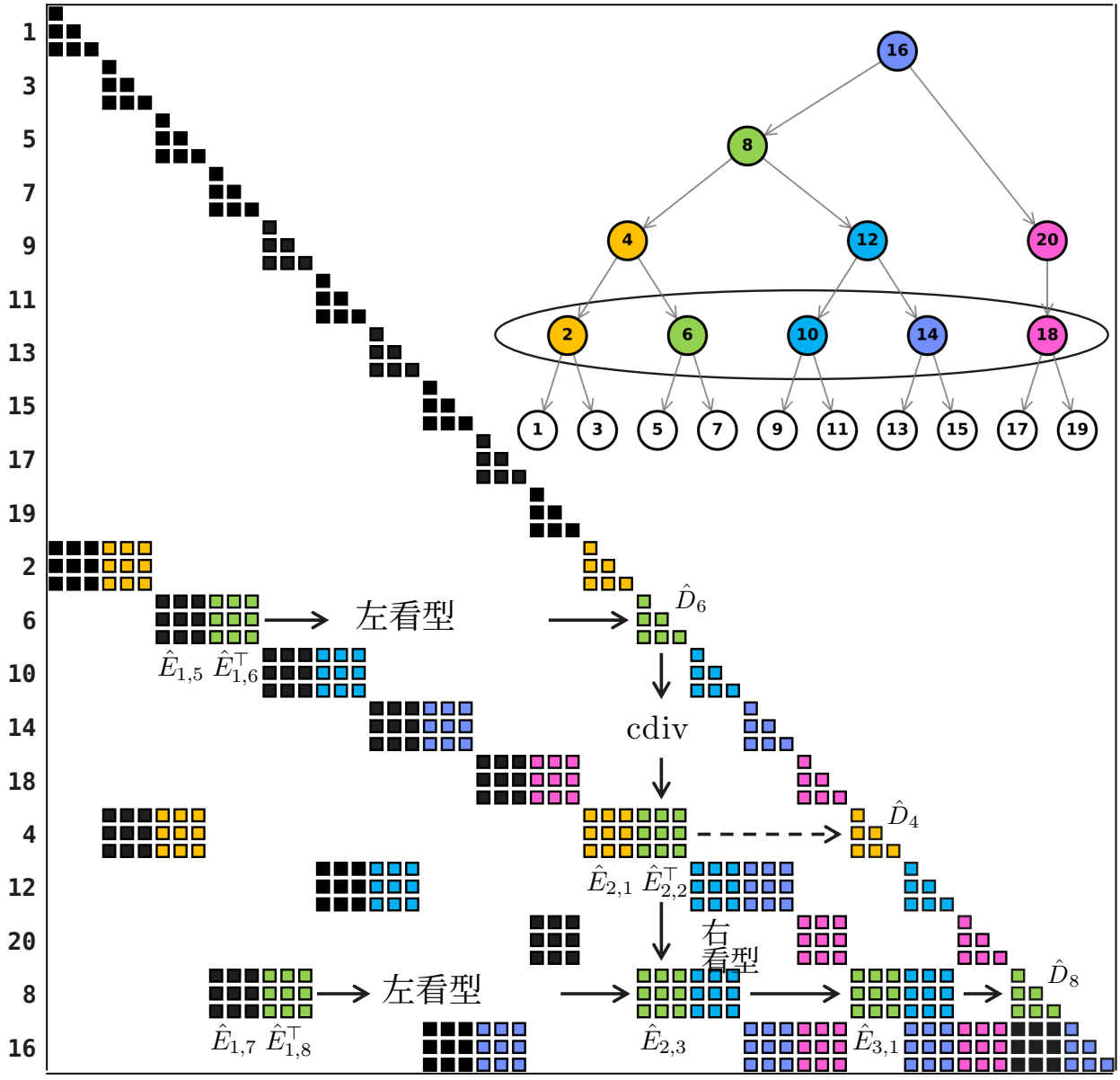


图 5：使用算法 4 对 Φ_∞ 进行 $N = 20$ 的 Cholesky 分解的可视化。图中展示的是 $s = 2$ 的并行迭代，即第二次迭代。实线箭头表示更新数据的操作，而虚线箭头表示延迟操作。

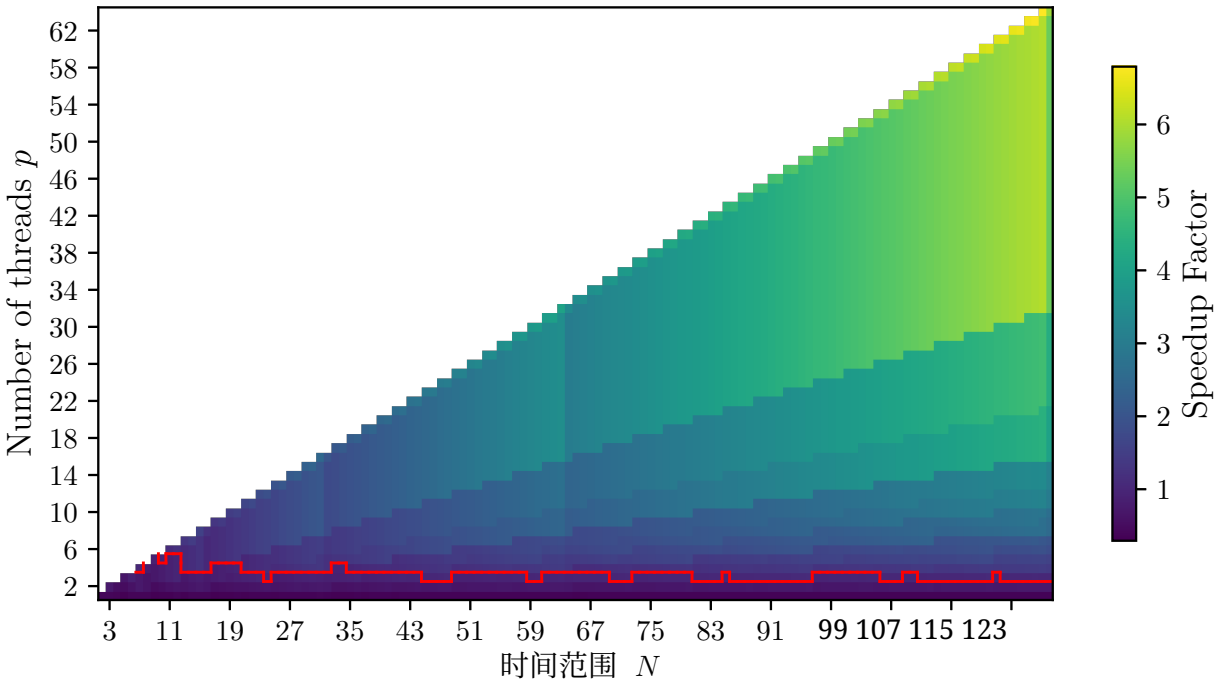


图 6：与 Ψ 的顺序分解相比，Φ_∞ 的并行分解的理论加速比。红线边界将实际加速与减速区分开来。

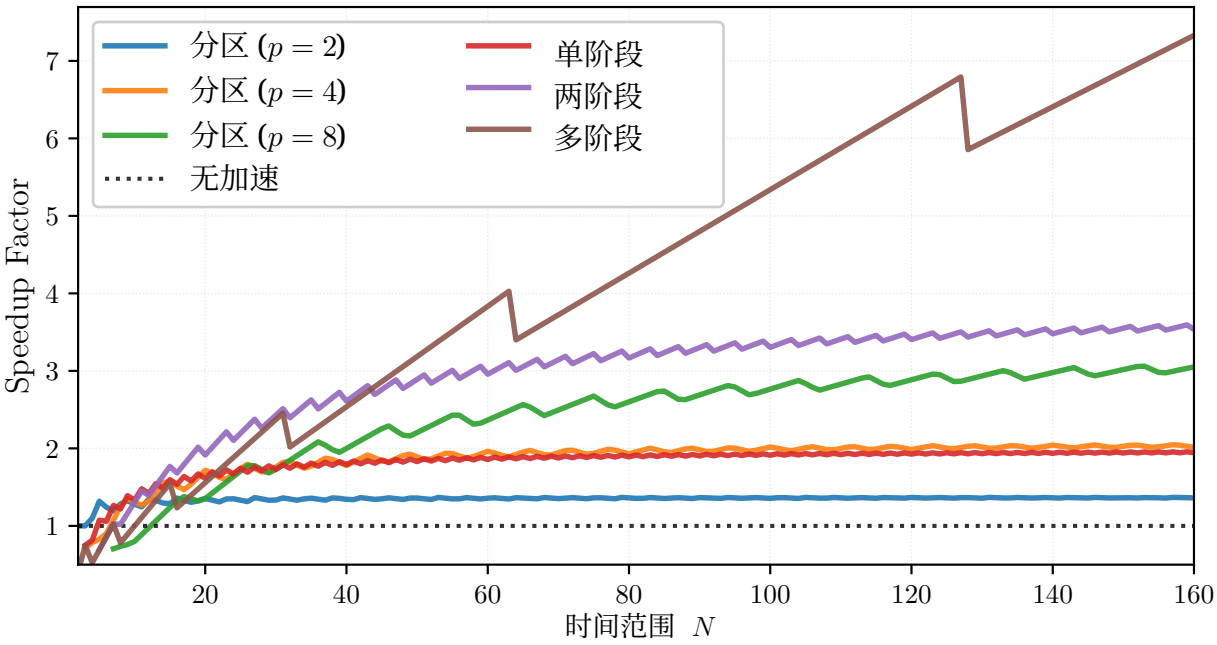


图 7：与 Ψ 的顺序分解相比，所有讨论过的并行分解方法的并行分解加速比。单阶段、两阶段和多阶段方法假设有 $p = \lceil N/2 \rceil$ 个并行线程。

算法 5 Cholesky 求解 $\Psi x = P_\infty^\top \hat{L}_\infty \hat{L}_\infty^\top P_\infty x = b$ 并行优化

前代： $P_\infty^\top \quad \hat{L}_\infty y = b$

1: $y \leftarrow b$
2: **对于** $s = 1, 2, 4, \ldots, 2^{\lfloor \log_2(N) \rfloor}$ **执行**
3: **对** $i = s、3s、5s，\ldots\ldots，N$ **并行执行**
4: $y_i \leftarrow \hat{D}_i^{-1} y_i$ `trsm` $n^2 m$
5: $y_{i+s} \leftarrow y_{i+s} - \hat{E}_{s,\ i/s} y_i$ 如果 $i + s \le N$ `gemm` $2n^2 m$
6: $y_{i-s} \leftarrow y_{i-s} - \hat{E}_{s,\ i/s-1}^\top y_i$ 如果 $i > s$ `gemm` $2n^2 m$
7: **结束循环**
8: **结束循环**

回代： $\hat{L}_\infty^\top P_\infty x = y$

9: $x \leftarrow y$
10: **对于** $s = 2^{\lfloor \log_2(N) \rfloor}, 2^{\lfloor \log_2(N) \rfloor - 1}, \ldots, 2, 1$ **do**
11: **对** $i = s, 3s, 5s, \ldots, N$ **并行执行**
12: $x_i \leftarrow x_i - \hat{E}_{s,i/s}^\top x_{i+s}$ 如果 $i + s \le N$ `gemm` $2n^2 m$
13: $x_i \leftarrow x_i - \hat{E}_{s,i/s-1} x_{i-s}$ 如果 $i > s$ `gemm` $2n^2 m$
14: $x_i \leftarrow \hat{D}_i^{-\top} x_i$ `trsm` $n^2 m$
15: **结束循环**
16: **结束循环**

算法 6 Φ_∞ 的右看分解，针对并行进行了优化

1: **对** $i = 1, \ldots, N$ **并行执行**
2: $\hat{D}_i \leftarrow D_i$
3: $\hat{E}_{1,i} \leftarrow E_i$ 如果 $i < N$
4: **结束循环**
5: **对于** $s = 1, 2, 4, \ldots, 2^{\lfloor \log_2(N) \rfloor}$ **执行**
6: **并行执行** $i = s, 3s, 5s, \ldots, N$
7: $\hat{D}_i \leftarrow \text{Cholesky 分解}(\hat{D}_i)$ `potrf` $\frac{n^3}{3}$
8: $\hat{E}_{s,i/s-1} \leftarrow \hat{D}_i^{-1} \hat{E}_{s,i/s-1}$ 如果 $i > s$ `trsm` n^3
9: $\hat{E}_{s,i/s} \leftarrow \hat{E}_{s,i/s} \hat{D}_i^{-\top}$ 如果 $i + s \le N$ `trsm` n^3
10: $\hat{D}_{i-s} \xleftarrow{\odot} \hat{D}_{i-s} - \hat{E}_{s,\ i/s-1}^\top \hat{E}_{s,\ i/s-1}$ 如果 $i > s$ `syrk` n^3
11: $\hat{D}_{i+s} \xleftarrow{\odot} \hat{D}_{i+s} - \hat{E}_{s,\ i/s} \hat{E}_{s,\ i/s}^\top$ 如果 $i + s \le N$ `syrk` n^3
12: $\hat{E}_{s+1,\ (i-s)/(2s)} \leftarrow -\hat{E}_{s,\ i/s} \hat{E}_{s,\ i/s-1}$ 如果 $i > s$ 和 $i + s \le N$ `gemm` $2n^3$
13: **结束循环**
14: **结束循环**

并且使用 p 个线程时，计算成本为

$$\left(\sum_{i=0}^{\lfloor \log_2(N) \rfloor} \left\lceil \frac{\lceil N/2^{i+1} \rceil}{p} \right\rceil 10 - 8 \left(\left\lceil \frac{N/2}{p} \right\rceil + 1 \right) \right) n^2 m \tag{2}$$

浮点运算次数。最后一项说明了当 $s = 1$ 和 $s = 2^{\lfloor \log_2(N) \rfloor}$ 时矩阵运算次数的减少。与顺序变体的 $\mathcal{O}(Nn^2m)$ 相比，其复杂度为 $\mathcal{O}(\log_2(N) n^2 m)$。

4.4 带操作并行性的多阶段置换

迄今为止，我们的方法主要关注列级并行化，如图 5 所示。不过，我们还可以利用操作级的额外并行性。例如，在算法 4 中，第 7 行和第 9 行的左看操作可以并发执行。追踪操作依赖关系后可以发现，关键路径包含四个顺序操作：左看秩-k 更新（第 7 行和第 9 行）、Cholesky 分解（第 8 行）、三对角求解（第 10 行和第 12 行），以及右看更新（第 11 行和第 13 行）。通过引入原子操作，我们可以使用完全右看分解例程将关键路径缩短至仅三个操作，如算法 6 所示，其中原子操作用 $\odot$ 符号表示。

在低竞争场景下，图形处理器上原子操作的性能开销很小，其延迟与标准内存操作相当 [23]。竞争是指写入的线程数量

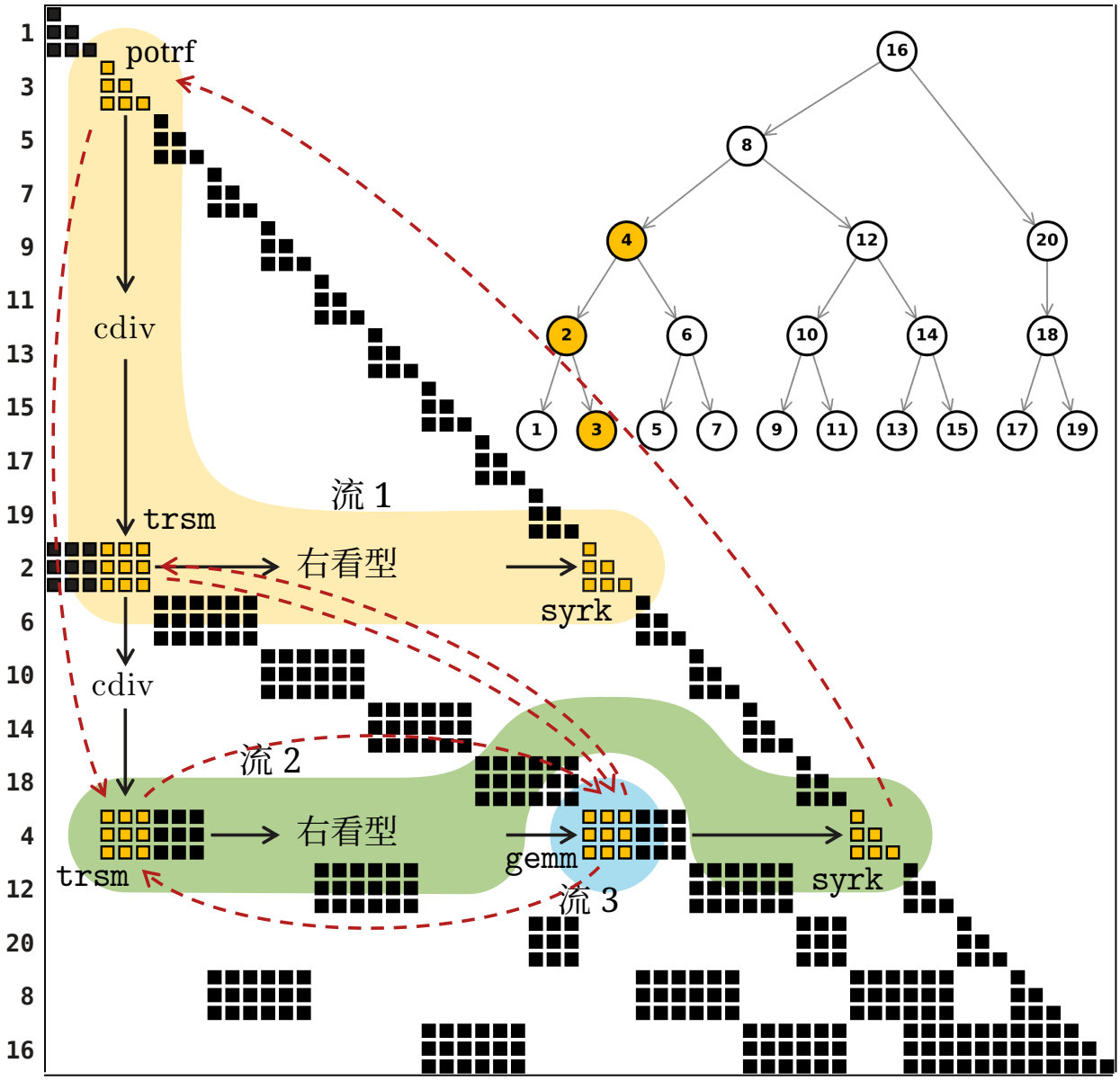


图8：使用算法 6 对 Φ_∞ 进行 $N = 20$ 的 Cholesky 分解可视化。图中展示了分配到不同 CUDA 流上并行运行的操作。红色虚线箭头表示流之间的同步，例如，流 3 中的 `gemm`操作必须等待流 1 和流 2 中的 `trsm` 操作先完成。

同时访问同一内存地址。由于我们的实现中最大竞争程度仅为 2，与将依赖链长度从 4 个操作缩短至 3 个操作所带来的收益相比，原子操作的开销可以忽略不计。我们使用 CUDA 流实现这种操作级并行，详情见第 5 节。图8展示了算法 4 如何映射到 CUDA 流，其中每个高亮区域都对应一个特定的流。同一流中的操作按顺序执行，而不同流中的操作则并行运行。流间同步可确保满足依赖约束。

现在，我们详细考察图8所示的示例。计算从 Cholesky 分解（`potrf`）开始，随后进行三角求解（`trsm`）；三角求解必须在 Cholesky 分解完成后执行。在流 1 中，通过顺序执行即可自动满足这一顺序要求。然而，流 2 中的三角求解需要显式同步；否则， `trsm` 操作可能与流 1 中的 `potrf` 操作并发执行。图8中的红色虚线箭头表示这一依赖关系及其对应的同步事件。类似地， `gemm` 操作依赖于前序 `trsm` 操作的完成，但可以与秩-k更新（`syrk`）操作并行执行。调度算法 6 的下一次迭代时，流 2 中的 `syrk` 操作可能仍在执行。因此，后续迭代中的 `potrf` 操作必须进行相应的同步。流 2 中的 `trsm` 操作也需要遵循相同的同步要求，等待前一个 `gemm` 操作完成。

图 9 展示了算法 6 应用于一个 $n = 64$ 和 $N =$128 的问题时的内核启动轨迹。上方轨迹清楚地展示了不同操作如何重叠并行执行。特别要注意，第一次 `gemm` 操作与第二次迭代的 `potrf` 重叠，从而最大限度地利用可用的图形处理器资源。此外，一旦相关数据可用，我们便可以在因式分解的同时并行执行算法 5 中的前代过程。图 9 的下方轨迹展示了这一点。这种重叠策略将总计算时间缩短了约 12%，从而更有效地利用图形处理器资源。

更多实现细节将在第 5.2 节中讨论。

5 数值实现

所有实验均在英伟达硬件上进行，具体使用了 RTX 3080 和 RTX 5090 图形处理器；基于中央处理器的实验则使用 AMD Ryzen 9 3900X 3.80 GHz 处理器。图形处理器实现使用 Python 和 Warp 库 [24], 开发，而中央处理器实现则使用 BLASFEO 库 [25] 和自定义 Python 接口，以 C++编写。Warp 库会生成 CUDA 和 C++ 代码，这些代码经即时（JIT）编译后通过 Python 接口调用。生成的代码可作为基础，用于在未来工作中开发经过手动调优、优化程度更高的内核。特别是，更谨慎的内存管理可以减少共享内存的使用，从而提高内核占用率。

在进一步讨论数值结果之前，我们先简要介绍 CUDA 编程模型 - 。 NVIDIA GPU。

5.1 CUDA 编程模型

NVIDIA GPU 采用一种为大规模并行性设计的分层并行执行模型。如图 10 所示，CUDA 编程模型将计算组织成三级层次：网格、线程块和线程。

在硬件层面，图形处理器由多个流式多处理器（SM）组成，每个流式多处理器都能独立执行指令（图 10 左侧）。启动 CUDA 内核时，会创建一个线程块网格，并将这些线程块分配到可用的流式多处理器上执行。每个线程块只分配给一个流式多处理器，不能拆分到多个流式多处理器上；不过，根据资源可用情况，一个流式多处理器可以同时执行多个线程块。只要有足够的空闲流式多处理器，也可以并行运行不同的内核，如图 10 中的线程块 4–7 所示。

每个线程块中的线程会被组织为每组 32 个线程的线程束，它们是 NVIDIA GPU 上执行的基本单位（图 10 右侧）。同一线程束中的所有线程都会以单指令多线程（SIMT）的方式同时执行相同的指令。当线程束中的线程遵循相同的控制流路径时，这种执行模型能实现高吞吐量；但当线程发生分歧时，性能就会下降，因为分歧分支必须串行执行。

线程块通过共享内存为线程之间的协作与同步提供了一种机制。共享内存是一种快速的片上存储空间，同一线程块中的所有线程都可以访问。共享内存层次结构与线程块内线程同步的能力相结合，可实现高效的

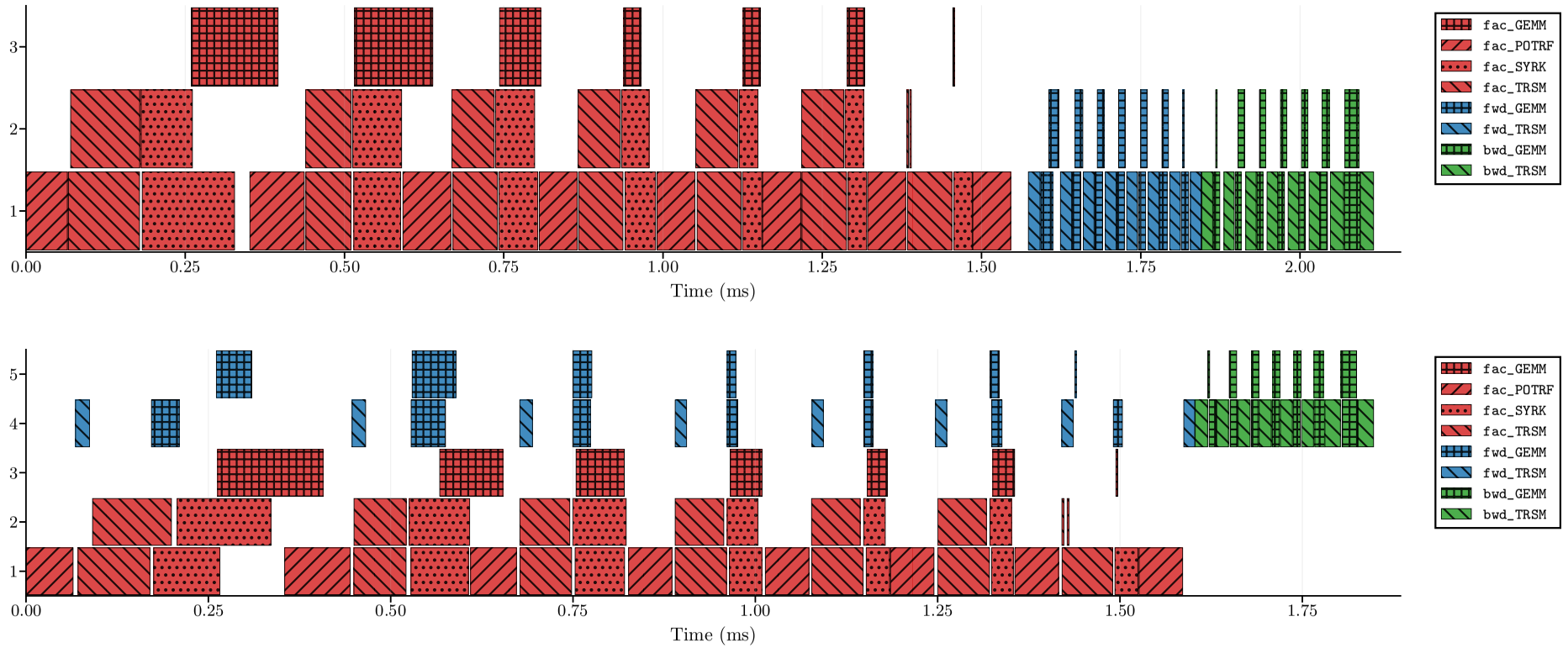


Figure 9: Traces of kernel launches of Algorithm 6 (`fac`: factorization) and Algorithm 5 (`fwd`: forward substitution, `bwd`: backward substitution) for $n = 64$ and $N = 128$. In the upper trace, the factorization and solve are run in sequential and in the lower trace, the forward substitution is interlaced with the factorization in parallel.

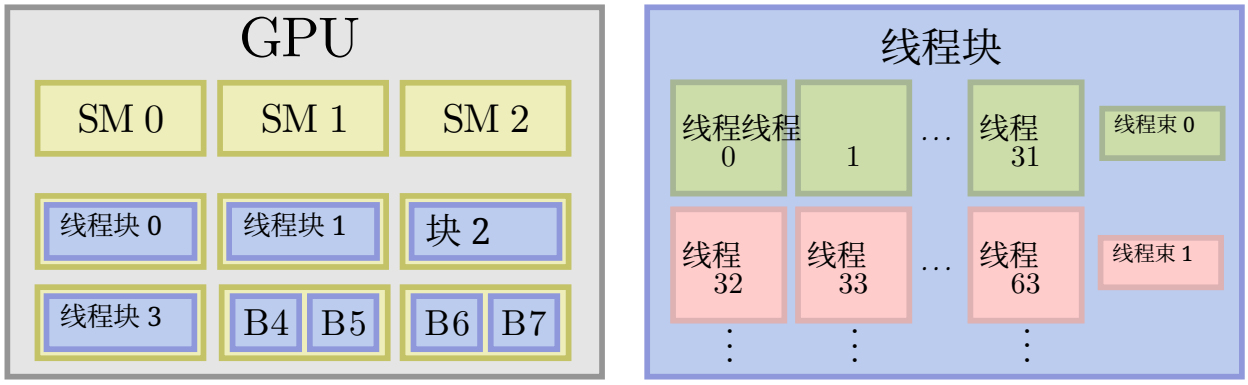


图 10：CUDA 编程模型

表 2：近期英伟达消费级 GPU 世代的硬件规格。

规格	RTX 3080	RTX 4090	RTX 5090
架构	Ampere	阿达·洛芙莱斯	Blackwell
流式多处理器	68	128	170
CUDA 核心	8,704	16,384	21,760
张量核心	272	512	680
基础频率	1.44 GHz	2.23 GHz	2.01 GHz
加速频率	1.71 GHz	2.52 GHz	2.41 GHz
显存容量	10GB GDDR6X	24 GB GDDR6X	32 GB GDDR7
显存带宽	760 GB/s	1,008 GB/s	1,792 GB/s
每个 SM 的共享内存	128 KB	128 KB	128 KB
每个线程块的最大线程数	1,024	1,024	1,024

实现需要本地数据共享和协调的算法。不过，通常不支持在单次内核执行期间跨不同线程块进行同步，因此必须谨慎设计算法，以遵守这一限制。

表 2 总结了英伟达最新三代消费级图形处理器的硬件规格。从 RTX 30 系列到 RTX 50 系列的发展，展现了多个维度上的显著架构改进。SM 数量大幅增加，从 RTX 3080 的 68 个增至 RTX 5090 的 170 个，直接带来了更高的并行吞吐量能力。流式多处理器数量的增长使图形处理器能够执行更多并发线程块，让新一代产品尤其适用于能够充分利用大规模并行性的吞吐量导向型工作负载。与此同时，加速时钟频率也有所提升，从 RTX 3080 的 1.71 GHz 增至 RTX 5090 的 2.41 GHz。更高的频率可降低单次操作的延迟，使具有顺序依赖或并行性有限的算法受益。SM 数量增加与频率提升相结合，大幅改善了吞吐量和延迟特性。此外，显存带宽从 760 GB/s 增至 1,792 GB/s，增幅超过一倍，缓解了常常限制数据密集型应用性能的内存瓶颈。

5.2 融合与分块内核

Warp 库 [24] 提供了一种瓦片编程模型，作为 NVIDIA mathdx 库的封装层，其中包括 cuBLASDx 和 cuSolverDx。这些库对应于 NVIDIA 久负盛名的 cuBLAS 和 cuSolver 库；后者在主机 API 层面为 NVIDIA GPU 提供经过优化的 BLAS 和 LAPACK 例程。相比之下，mathdx 库可以直接作为 CUDA 内核中的构建模块使用。

我们实现了两个不同版本的分解与求解内核：融合变体和分块变体。融合内核通过将单次迭代中的所有操作（第 6 至 14 行）融合到一个内核中，实现了算法 4。它会将数据加载到共享内存中，处理所有操作，然后将结果写回全局内存，从而最大限度地减少内存传输和内核调用。不过，这种方法受可用共享内存的限制。对于过大的块大小 n，所需的共享内存会超出流式多处理器（SM）的容量。

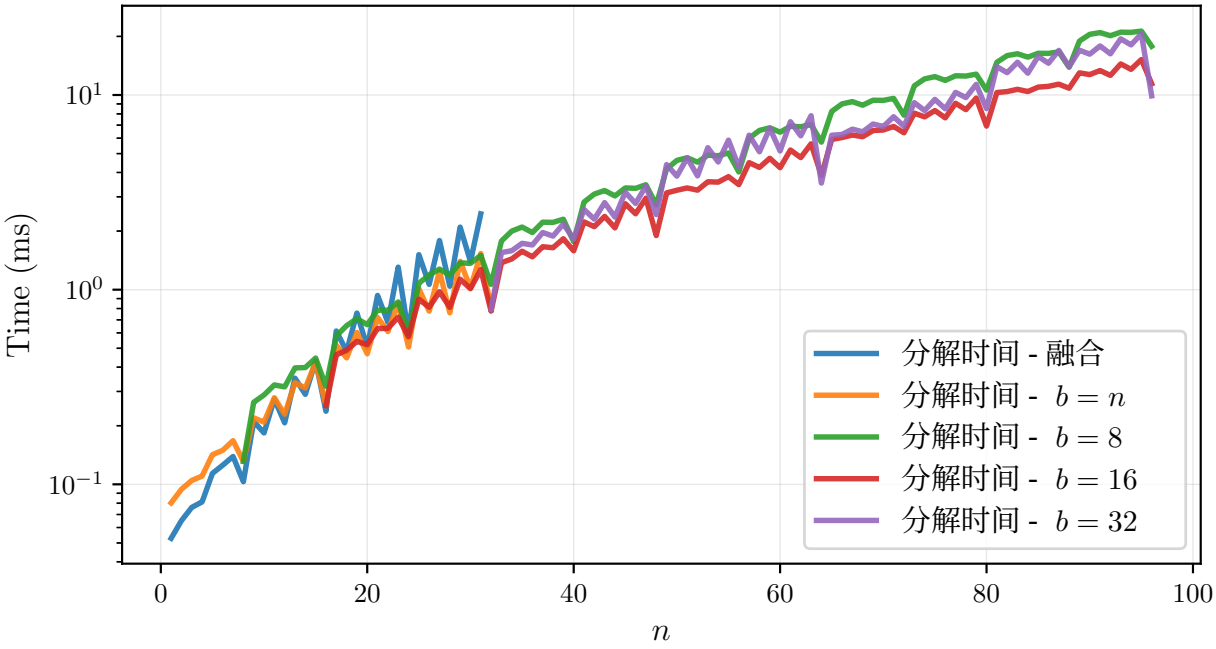


图 11：使用双精度（`f64`）在 RTX3080 上，采用融合内核和分块内核时，不同块大小 n 和时间范围 $N = 512$ 下的计算时间，其中 b 为瓦片大小。对于 $n \geq 32$，融合内核会耗尽共享内存。

为了解决这一限制，分块变体会对操作进行分块。例如，在 Cholesky 分解中，我们不加载大小为 $n \times n$ 的整个矩阵，而只加载大小为 $b \times b$ 的子矩阵，并逐块处理。除了矩阵级分块外，我们还按照第 4.4 节和算法 6 所述，在操作级进一步并行化。具体而言，我们通过三个 CUDA 流并行执行操作。为了进一步提升性能并减少 Python 开销，我们使用 CUDA 图来启动内核序列。CUDA 图支持记录并重放 C UDA 内核启动，从而降低各次内核调用的启动开销。

图 11 展示了时间范围为 $N = 512$ 时，不同块大小 n下的计算时间，并比较了融合内核变体与瓦片大小为 b的分块内核变体。结果采用双精度（`f64`），每个线程块包含 64 个线程。对于不超过 $n = 16$ 的较小块大小，融合内核因开销较低而实现最快的因式分解时间。超过 $n = 16$ 后，分块变体的效率更高，因为操作级并行性可带来加速比。

可以观察到显著的量化效应，即奇数块大小通常会慢很多。图 17 展示了采用单精度（`f32`）进行的相同实验，其中块大小为 4 的倍数时会出现这种离散化效应。这一现象凸显了内存对齐的重要性。矩阵必须按 128 位对齐，才能利用 NVIDIA GPU 加载 128 位对齐内存的专用指令。

图 18 和图 19 展示了 RTX5090 上的类似结果，表现出相似的规律。对于双精度，当 n 介于 16 和 32 之间时，设置 $b = n$ （即不进行分块）效率最高。然而，如前所述，对于更大的块大小，必须进行分块，以避免超出共享内存限制。瓦片大小为 $b = 8$ 似乎太小，无法充分利用 64 个可用线程。虽然 $b = 16$ 通常表现最佳，而当 n 是 32 的倍数时，$b = 32$ 更胜一筹。事实上，当双精度下 n可被 8 整除、单精度下可被 16 整除时，始终能获得最佳性能。因此，相应地填充问题规模可确保达到最高性能

图 12 展示了使用块大小为 $n = 32$ 的分块内核变体（ $b = 32$）时，不同时间范围 N 对应的计算时间。由于通常会先进行因式分解，然后直接针对给定的右端项求解，我们还实现了一种融合变体，将因式分解和前代交错执行，从而提升并行性和性能。交错内核启动如图 9 所示。如图 12 所示，与依次运行因式分解和求解例程相比，这种方法显著缩短了求解时间。

因式分解时间遵循第 4.3 节推导出的理论$\mathcal{O}(\lfloor \log_2(N) \rfloor)$复杂度，直到约 $N = 250$，之后则变得更加接近线性。求解时间也呈现出类似的趋势，不过这种现象出现得更晚，大约在 $N = 700$。这种偏离理论复杂度的情况是因为

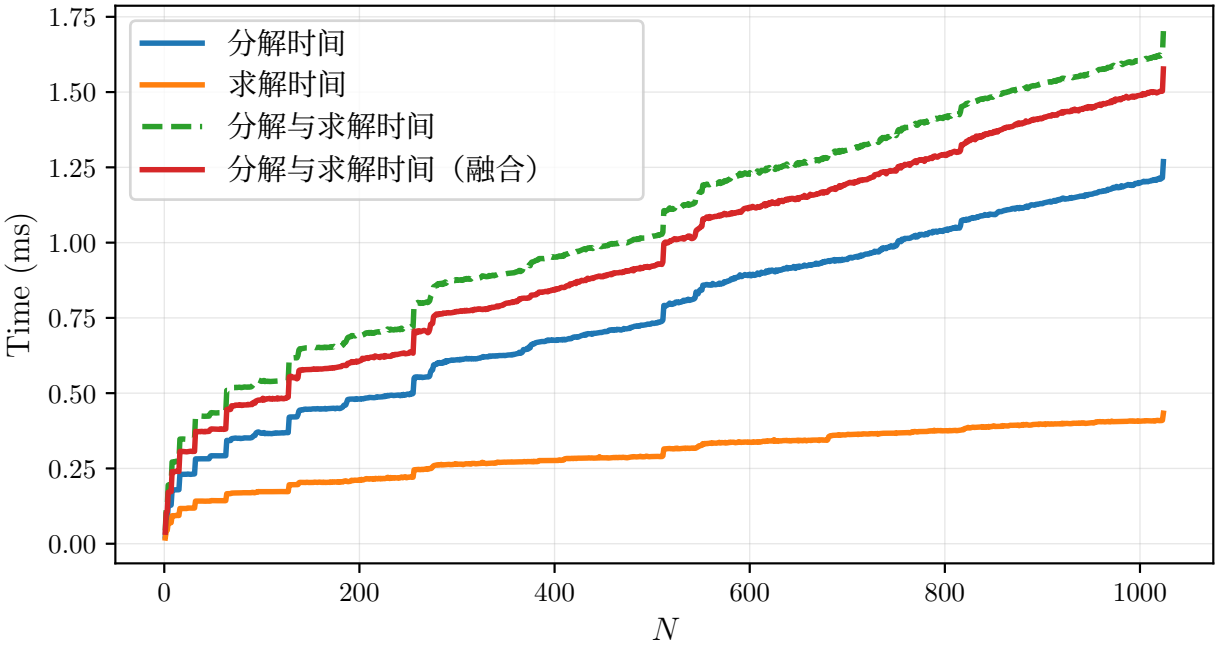


图 12：使用双精度（`f64`）在 RTX3080 上计算不同时间范围 N、块大小为 $n = 32$ 时的计算时间。

图形处理器中的流式多处理器已达到饱和，导致最初几次迭代无法完全并行处理。求解操作所需的共享内存较少，因此这种饱和出现得更晚；单个流式多处理器可以并行处理更多线程块。

图13比较了在 RTX3080 和 RTX5090 上使用单精度（`f32`）和双精度（`f64`）时的因式分解时间。对于较短的时域，RTX3080 和 RTX5090 的因式分解时间非常相近。然而，对于较长的时域 N，RTX5090 的性能显著优于 RTX3080，加速比约为 $2.5\times$，这与表 2 中所述其流式多处理器数量多出 $2.5\times$ 相符。我们还观察到，单精度相较于双精度可实现 $4\times$ 的加速比。这可以归因于较低的共享内存需求，以及单精度浮点运算比双精度浮点运算快 1.5–2× 倍 [23]。图20中的求解时间也呈现出类似的情况。

6 与其他中央处理器和图形处理器的比较

6.1 QDLDL

首先，我们将其与稀疏 LDL 求解器 QDLDL 进行比较。QDLDL 是广受欢迎的 QP求解器 OSQP [26] 的组成部分。报告的 QDLDL 计时仅包含数值分解时间，不包括符号分解阶段。

图14展示了随着时间范围 N 增加，不同图形处理器和精度下分解例程的加速比。随着时间范围增加，加速比显著提升，即使时间范围较小时也能观察到明显的加速效果。随着图形处理器达到饱和，计算时间从$\mathcal{O}(\lfloor \log_2(N) \rfloor)$状态转为线性增长，使用双精度时，RTX3080 和 RTX5090 的最大加速比分别稳定在约 $40\times$ 和 $100\times$。使用单精度时，RTX3080 的最大加速比为 $170\times$，RTX5090 则超过 $500\times$。

图21显示，求解时间也呈现出类似的行为，不过加速效果不那么明显，且要到更长的时间范围才会达到饱和。

6.2 BLASFEO

为确保比较公平，我们还使用 BLASFEO 库 [25] 作为后端，在中央处理器上实现了算法 1，而不是仅与通用性更强的稀疏 LDL 求解器 QDLDL 进行比较。我们为实验创建了 Python 绑定，并通过引用传递数据，以避免引入开销。

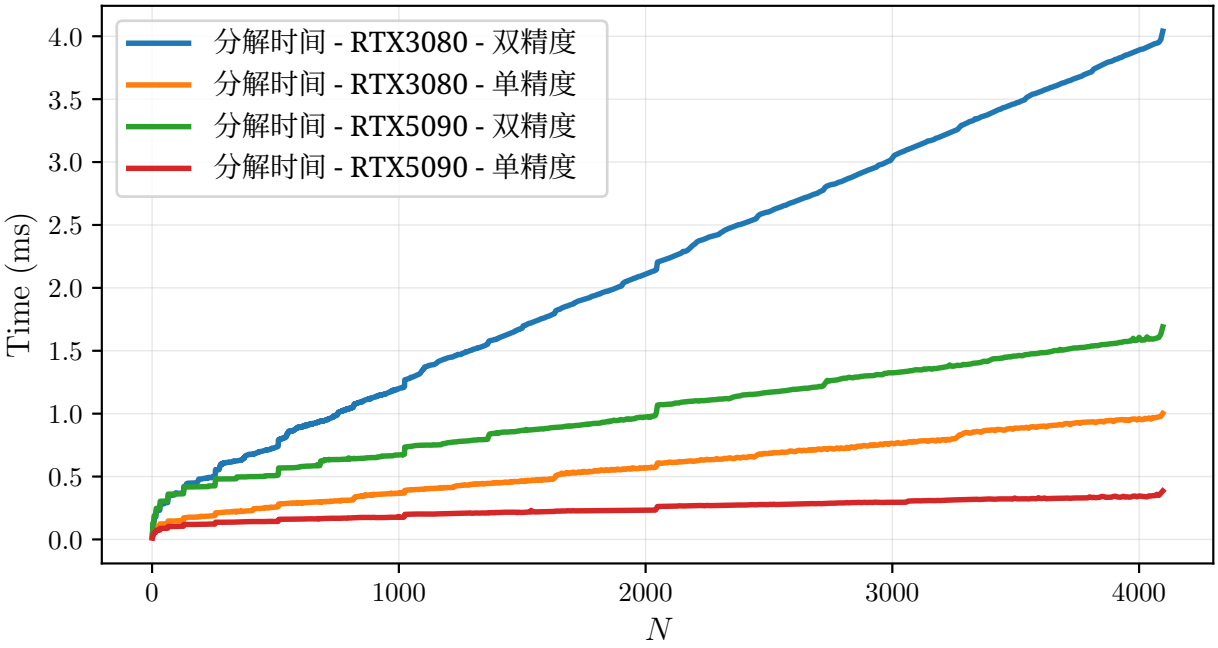


图13：因　　式分解在不同时间范围 N、块大小为 $n = 32$ 时的计算时间，使用不同图形处理器和精度。

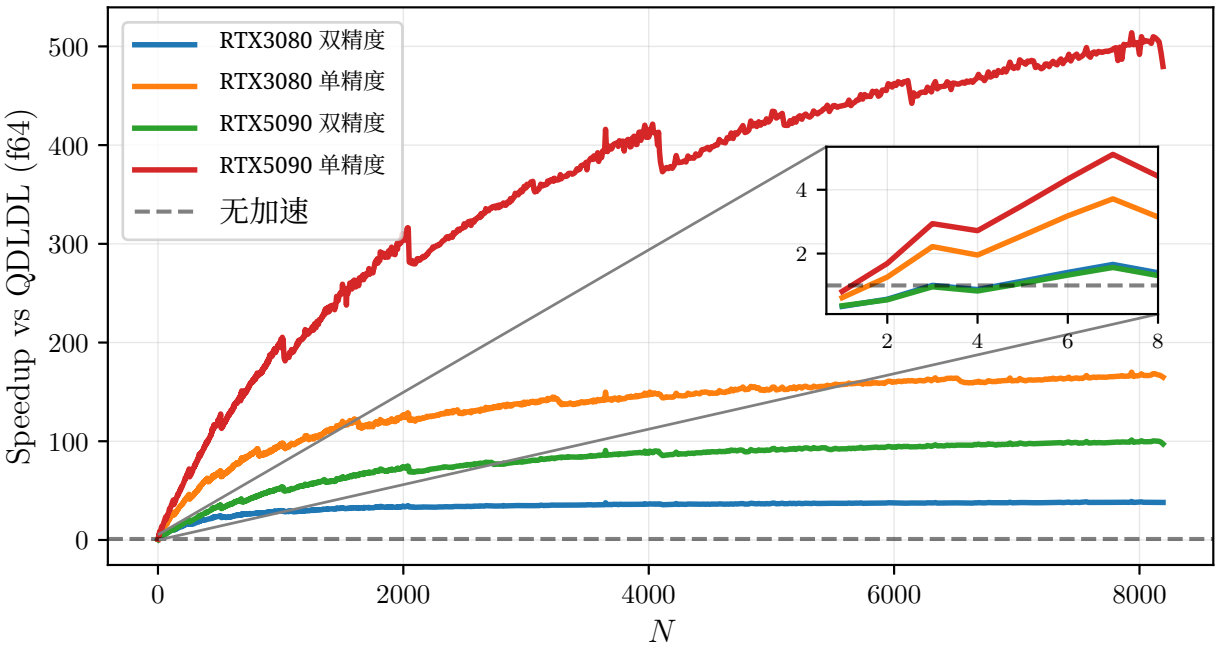


图14：不同时间范围下，使用不同图形处理器和精度时的因式分解时间加速比 N，块大小为 $n = 32$。相较于 QDLDL；QDLDL 仅提供双精度浮点接口。

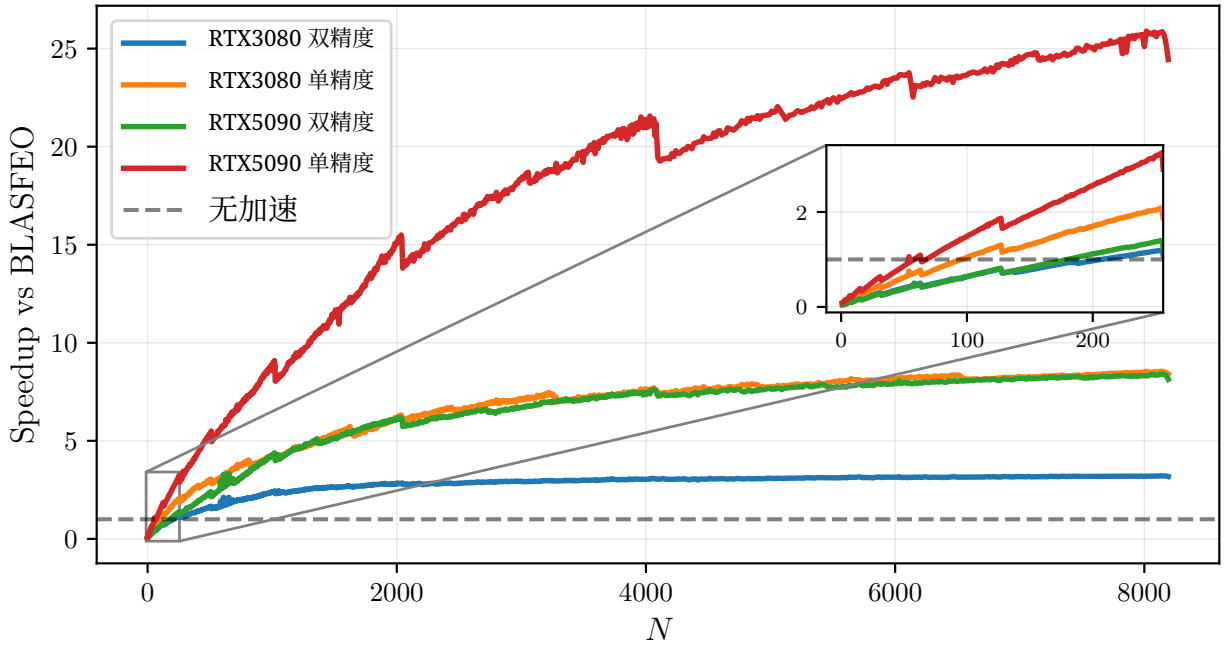


图 15：相对于使用不同图形处理器和精度的 BLASFEO，在块大小为 $n = 32$ 时，不同时间范围 N 下因式分解时间的加速比。

图 15 展示了我们的 GPU 求解器实现所获得的加速比。请注意，这里比较的是浮点精度相同的实现，例如 `f64` 图形处理器实现与 `f64` BLASFEO 实现。BLASFEO 后端的性能显著优于 QDLDL，这得益于它采用了 AVX2（单指令多数据）SIMD 指令，使中央处理器能够同时处理最多四个双精度数或八个单精度数，从而提供有限的指令级并行性。此外，BLASFEO 的数据局部性更好，因此缓存利用率也更高。由于内存传输是现代中央处理器上的一大瓶颈，这种更好的局部性还能进一步提升加速比。

尽管如此，我们的图形处理器实现使用 RTX5090 和单精度时仍实现了超过 $25\times$的加速比。即使在 RTX3080 上使用双精度，对于较长的时域 N，我们仍观察到 $2\times$ 的加速比。交叉点出现在时域约为 $N = 100$ 至 $N = 200$时。

6.3 CUDSS

我们还将所提出的方法与英伟达的闭源 CUDSS 库进行了比较。图 16 和图22分别展示了在 RTX3080 和 RTX5090 上使用双精度时的因式分解时间和求解时间。对于这两项指标，我们的方法始终实现约 $2\times$ 的加速比。

虽然 CUDSS 中采用的具体算法尚未公开，但我们认为，其分析阶段的置换很可能使用了嵌套剖分法，因此渐近扩展特性与我们的方法相似。事实上，计算时间在 2 的幂次边界处出现跃升，这表明 CUDSS 也利用了嵌套剖分置换所形成的树结构。性能差异主要源于 CUDSS 无法利用我们的问题本身所具有的稠密结构。

此外，图中未包含符号分析时间，而该时间通常比因式分解时间慢两个数量级。此阶段包括置换计算和符号分解，为数值分解做准备。我们未将这部分开销纳入比较，因为矩阵结构是先验已知的，因此可以离线执行符号分析。

7 结论

我们提出了一套全面的框架，通过系统的矩阵重排序策略，对块三对角矩阵进行 GPU 加速的 Cholesky 分解。我们的多阶段置换方法类似于嵌套剖分；当有足够的并行资源时，它可将计算复杂度从 $\mathcal{O}(Nn^3)$ 降至 $\mathcal{O}(\log_2(N)n^3)$。通过利用列级并行性和操作级并行性，

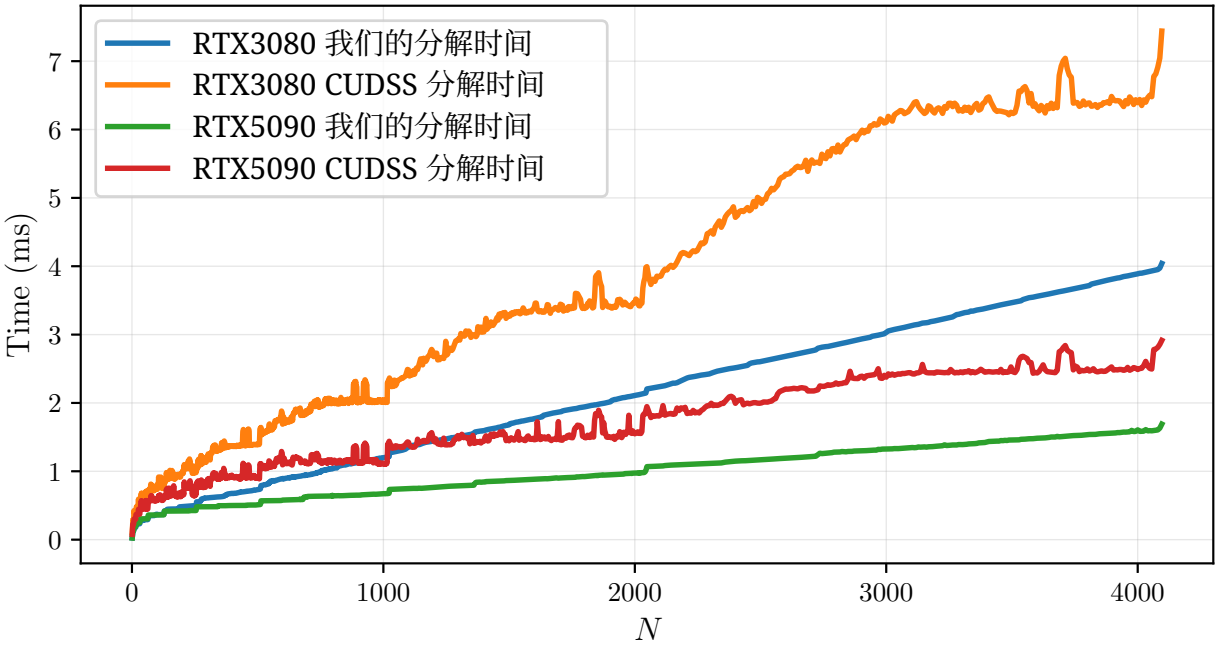


图 16：使用双精度，在不同图形处理器上比较我们分解方法与 CUDSS 在不同时间范围 N 下、块大小为 $n = 32$ 时的因式分解时间。

结合 CUDA 流和原子操作，我们进一步缩短了关键路径长度，并提升了实际性能。

数值实现表明，该方法在现代英伟达 GPU 上实现了显著的加速。与稀疏 LDL 求解器 QDLDL 相比，对于较长时域，采用单精度时，我们的方法在 RTX 3080 上的加速比超过 $100\times$，在 RTX 5090 上超过 $500\times$。即使与针对块三对角系统优化的 CPU 实现（使用 BLASFEO 和 SIMD 指令）相比，我们的 GPU 求解器在单精度下仍实现了 $25\times$ 至 $40\times$ 的加速比，性能交叉点约出现在 $N = 100$ 至 $N = 200$ 附近。值得注意的是，我们的方法在因式分解和求解操作上的表现也优于英伟达的闭源 CUDSS 库，优势约为 $2\times$。这一性能优势源于我们的算法能够利用块三对角矩阵固有的稠密块结构，而通用稀疏求解器无法同样有效地利用这种结构。

我们的方法尤其适用于模型预测控制、卡尔曼滤波，以及其他经常出现具有中等长度时间范围的块三对角系统的领域。这还包括将其集成到优化求解器中，用于机器人学和电力系统等应用。对数级扩展特性确保性能会随着时域长度的增加而持续提升，使其成为需要反复求解大型结构化线性系统的实时应用的理想方案。未来的工作可以探索更激进的内核融合、扩展到带宽更大的块带状矩阵，以及进一步提升性能的混合精度策略。

参考文献

[1] L. Amatucci、J. Sousa-Pinto、G. Turrisi、D. Orban、V. Barasuol 和 C.Semini，“足式机器人中用于图形处理器加速学习与控制的原始-对偶 iLQR”，2025年。[2] G. Perantoni 和 D. J. Limebeer，“参数可变的一级方程式赛车最优控制”，《车辆系统动力学》，第52卷，第5期，第653–678页，2014年。[3] D. Kouzoupis、I. Pendharkar、J. Frey、M. Diehl 和 F. Corman，“面向高效计算列车轨迹优化的直接多重打靶法”，《交通研究*C*辑：新兴技术》，第152卷，文章编号104170，2023年。[4] Z. Wang 和 M. J. Grant，“使用凸规划优化最短时间低推力转移”，《航天器与火箭杂志》，第55卷，第3期，第586–598页，2018年。

[5] R. Schwan、D. Kuhn 和 C. N. Jones，“利用近端求解器中的多阶段优化结构”，载于*IEEE* 决策与控制会议（*CDC*），2025年。[6] M. T. Heath、E. Ng 和 B. W. Peyton，“稀疏线性系统的并行算法”，*SIAM* 评论，第33卷，第3期，第420–460页，1991年。[7] T. Cao、J. Hall 和 R. van de Geijn，“块三对角矩阵的并行 Cholesky 分解”，载于《国际并行处理会议研讨会论文集》，2002年，第327–335页。[8] I. Nielsen 和 D. Axehill，“一种用于模型预测控制中牛顿步计算的 O(log N) 并行算法”，*IFAC* 会议论文集，第47卷，第3期，第10 505–10 511页，2014年，第19届 IFAC 世界大会。[9] ——，“一种利用并行结构的分解算法及其在模型预测控制中的应用”，载于*IEEE* 决策与控制会议（*CDC*），2015年，第3932–3938页。[10] S. Särkkä 和 A. F. García-Fernández，“动态规划与线性二次控制的时间并行化”，*IEEE* 自动控制汇刊，第68卷，第2期，第851–866页，2023年。[11] W. Jallet、E. Dantec、E. Arlaud、N. Mansard 和 J. Carpentier，“用于实时非线性模型预测控制的并行近端约束线性二次方法”，《机器人学：科学与系统》，2024年。[12] E. Adabag、M. Atal、W. Gerard 和 B. Plancher，“MPCGPU：通过图形处理器上的预条件共轭梯度法实现实时非线性模型预测控制”，载于*IEEE* 国际机器人与自动化会议（*ICRA*），2024年，第9787–9794页。[13] S. Yang、T. Ohtsuka、B. Plancher 和 C. N. Jones，“面向块三对角正定矩阵的多项式并行预条件方法”，2025年。[14] E. Adabag、M. Greiff、J. Subosits 和 T. Lew，“图形处理器上的可微模型预测控制”，2025年。[15] F. Jenelten、J. He、F. Farshidian 和 M. Hutter，“DTC：深度跟踪控制”，《科学·机器人学》，第9卷，第86期，文章编号5401，2024年。[16] S. H. Jeon、H. J. Lee、S. Hong 和 S. Kim，“残差模型预测控制：融合强化学习与图形处理器并行模型预测控制”，2025年。[17] R. Reiter、J. Hoffmann、D. Reinhardt、F. Messerer、K. Baumgärtner、S. Sawant、J. Boedecker、M. Diehl 和 S. Gros，“模型预测控制与强化学习的融合：综述与分类”，2025年。[18] F. Song、R. Schwan、Y. Chen 和 C. N. Jones，“用于多阶段优化的 PIQP 并行 KKT 求解器”，2025年。[19] R. Schwan、Y. Jiang、D. Kuhn 和 C. N. Jones，“PIQP：一种近端内点二次规划求解器”，载于*IEEE* 决策与控制会议（*CDC*），2023年，第1088–1093页。[20] A. George 和 J. W. H. Liu，“一种用于非规则有限元问题的自动嵌套剖分算法”，*SIAM* 数值分析杂志，第15卷，第5期，第1053–1069页，1978年。[21] D. Heller，“块三对角线性系统循环约简算法的若干方面”，*SIAM* 数值分析杂志，第13卷，第4期，第484–496页，1976年。[22] W. Gander 和 G. H. Golub，“循环约简的历史与应用”，载于《科学计算：研讨会论文集，*1997*年*3*月*10*日至*12*日，香港》。施普林格科学与商业媒体，1998年，第73页。[23] Z. Jia、M. Maggioni、J. Smith 和 D. P. Scarpazza，“通过微基准测试剖析 NVIDIA Turing T4 GPU”，2019年。[24] M. Macklin，“Warp：用于 GPU 仿真与图形学的高性能 Python 框架”，https: //github.com/nvidia/warp，2022年3月，NVIDIA GPU 技术大会（GTC）。[25] G. Frison、D. Kouzoupis、T. Sartor、A. Zanelli 和 M. Diehl，“BLASFEO：用于嵌入式优化的基础线性代数子程序”，*ACM* 数学软件汇刊，第44卷，第4期，2018年7月。

[26] B. Stellato、G. Banjac、P. Goulart、A. Bemporad 和 S. Boyd，《OSQP：一种用于二次规划的算子分裂求解器》，*Mathematical Programming Computation*，第 12 卷第 4 期，第 637–672 页，2020 年。

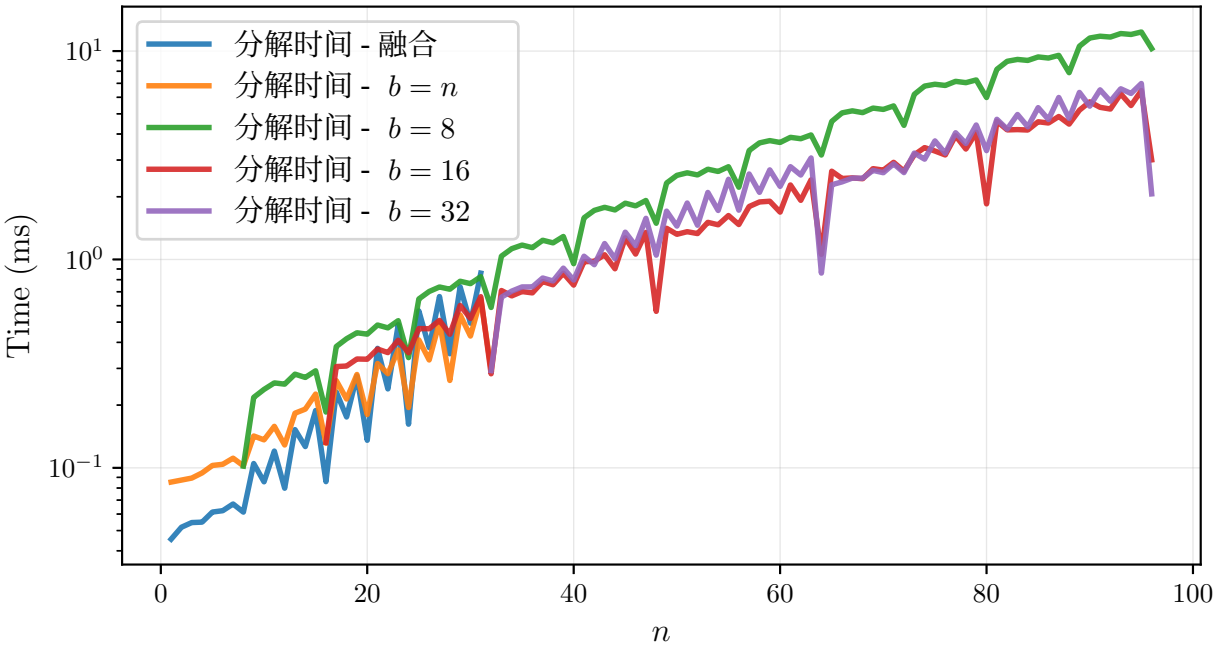


图 17：在 RTX3080 上使用双精度（`f32`）时，不同块大小 n 和时间范围 $N = 512$ 下融合内核与分块内核的计算时间，其中 b 为瓦片大小。

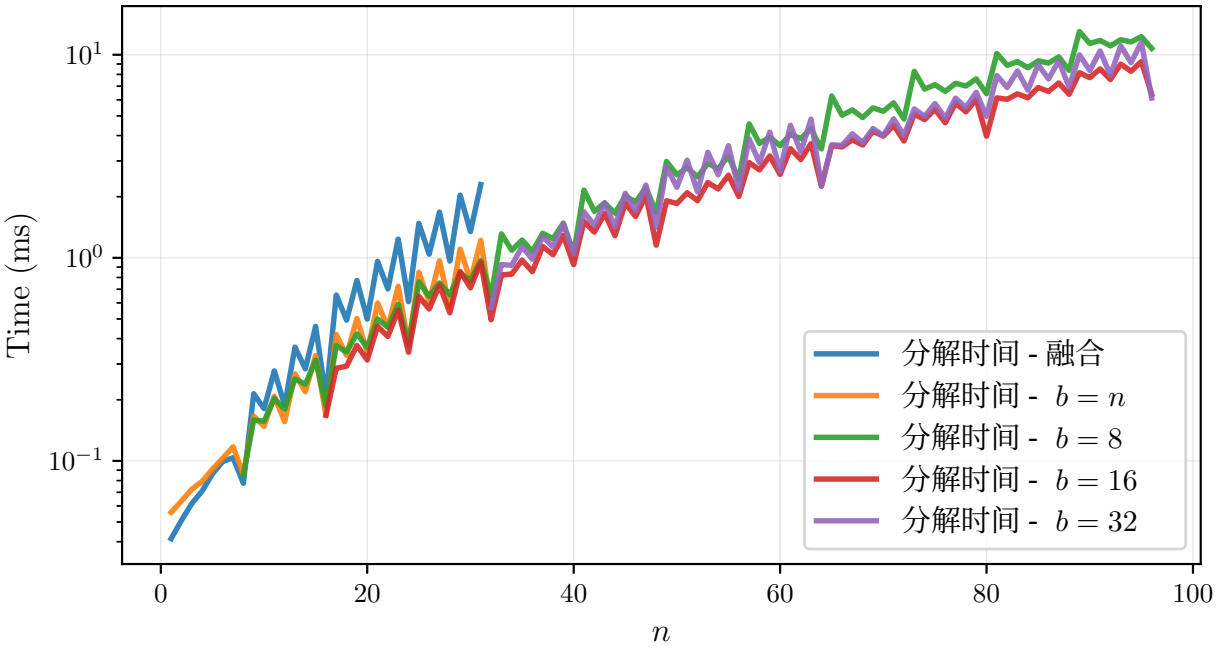


图18：在 RTX5090 上使用双精度（`f64`）时，不同块大小 n 和时间范围 $N = 512$ 下融合内核与分块内核的计算时间，其中 b 为瓦片大小。

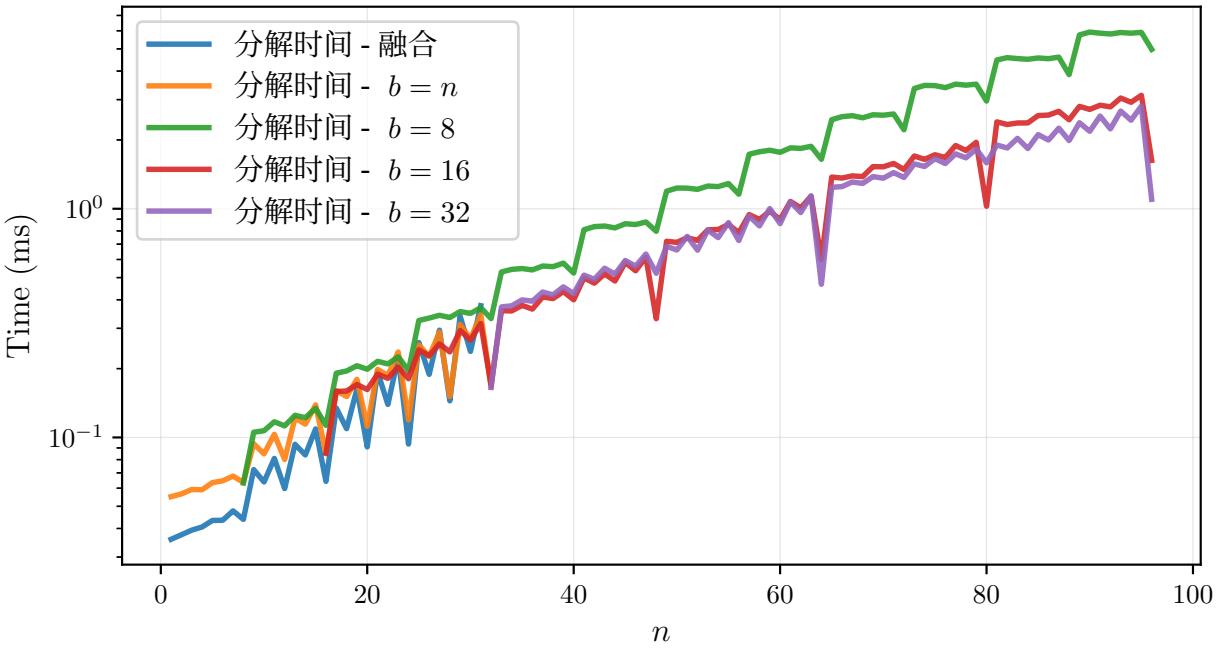


图19：使用双精度（`f32`）时，RTX5090 上不同块大小 n 和时间范围 $N = 512$ 下，融合内核和分块内核的计算时间，其中 b 是瓦片大小。

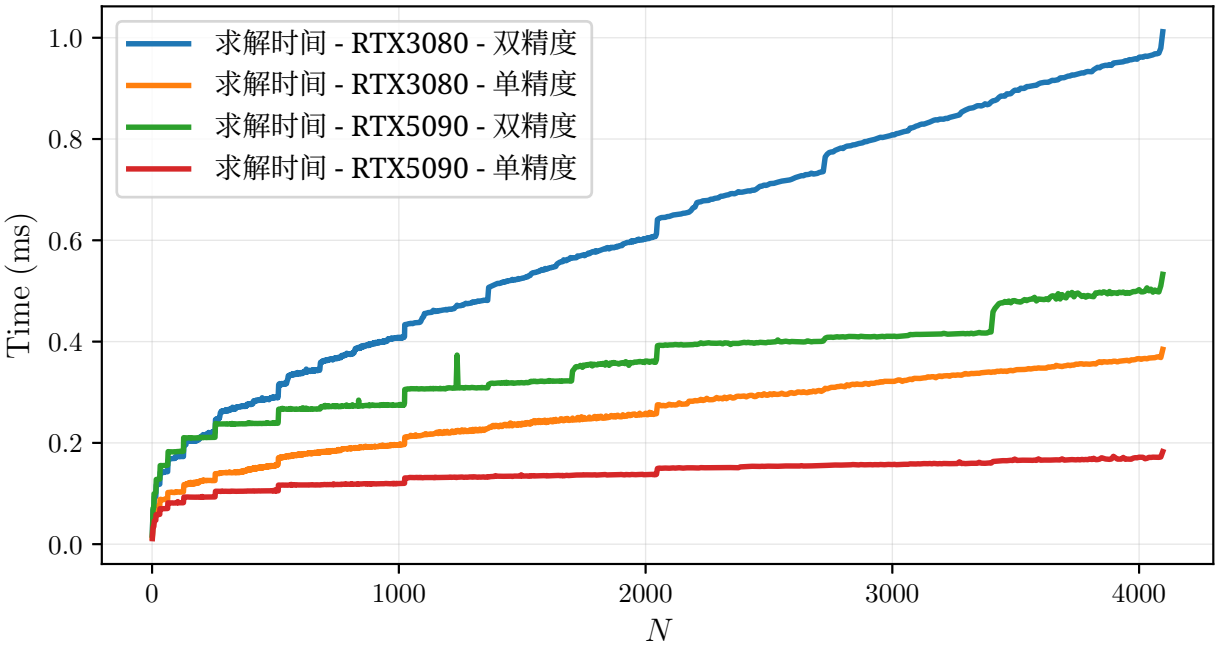


图20：不同图形处理器和精度下，块大小为 $n = 32$ 时，不同时域 N 下的求解计算时间。

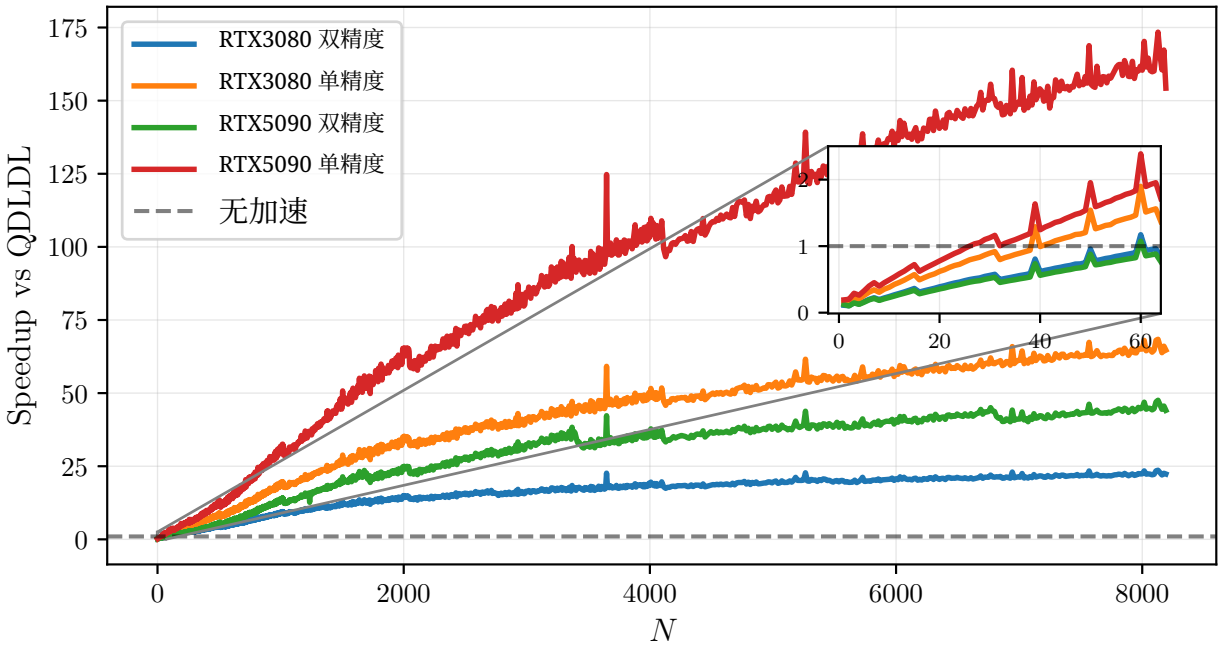


图21：不同时间范围下的求解时间加速比 N （块大小为 $n = 32$ ），相对于使用不同图形处理器和不同精度的 QDLDL。

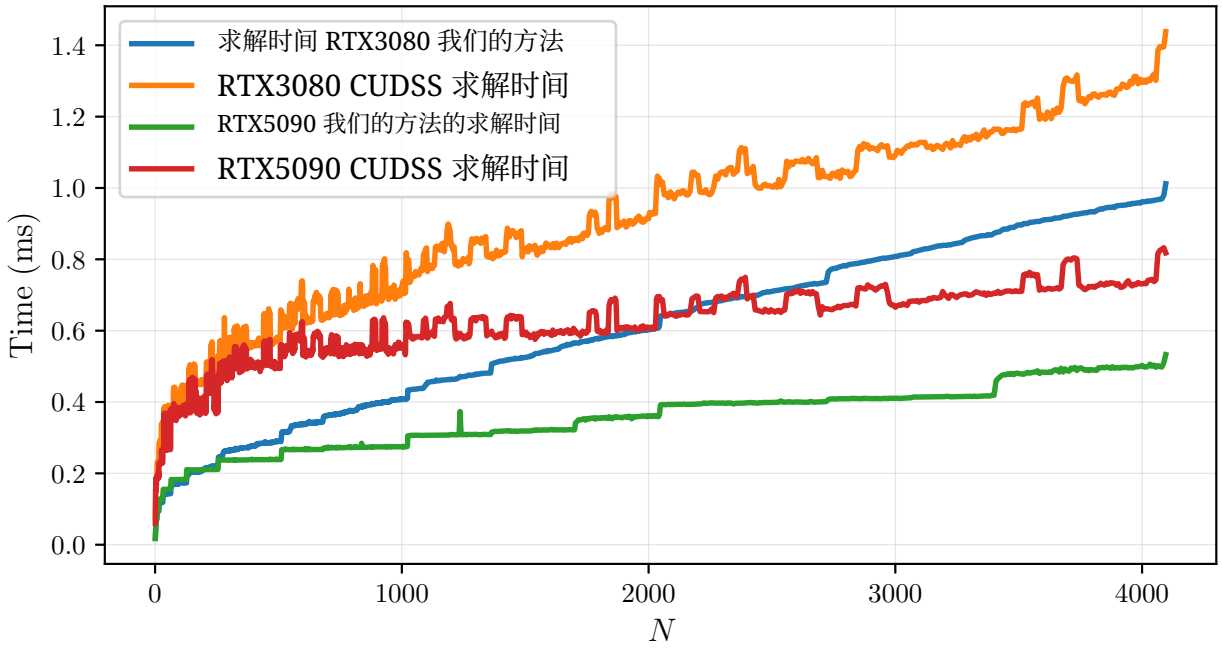


图22：使用双精度时，我们的分解方法与 CUDSS 在不同图形处理器上的求解时间对比。