软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(11):4952−4972 [doi: 10.13328/j.cnki.jos.006727]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于批量 LU 分解的矩阵求逆在 GPU 上的有效实现*

刘世芳[1,2], 赵永华[1], 黄荣锋[1,2], 于天禹[1,2], 张馨尹[1,2]

[1](中国科学院 计算机网络信息中心, 北京 100190)

[2](中国科学院大学, 北京 100049)

通信作者: 赵永华, E-mail: yhzhao@sccas.cn

摘　要: 给出批量矩阵的 LU 分解和批量求逆算法在 GPU 上实现及优化方法. 针对批量 LU 分解问题, 分析 Left-looking 和 Right-looking 等常用 LU 分解块算法在 GPU 上实现时对全局内存的数据读写次数, 针对 GPU 架构特点, 选择具有较少访存数据量的 Left-looking 块算法. 在 LU 分解的选主元过程, 采用适合 GPU 架构的并行二叉树搜索算法. 此外, 为了降低选主元引起的行交换过程对算法性能的影响, 提出 Warp 分组行交换和行交换延迟 2 个优化技术. 针对 LU 分解后的批量求逆问题, 分析矩阵求逆过程中修正方法, 为了减少修正过程对全局内存的访问, 在批量求逆的 GPU 实现中采用延迟修正的矩阵求逆块算法. 同时, 为了加快数据读写速度, 采用更多利用寄存器和共享内存的优化方法和减少访存数据量的列交换优化方法. 另外, 为了避免线程的闲置和共享内存等 GPU 资源浪费, 提出运行时动态 GPU 资源分配方法, 相较于一次性分配的静资源分配方法性能得到明显提升. 最终, 在 TITAN V GPU 上, 对 10 000 个规模在 33–190 之间的随机矩阵进行测试, 测试的数据类型为单精度复数、双精度复数、单精度实数和双精度实数. 所实现的批量 LU 分解算法的浮点计算性能分别可达到约 2 TFLOPS、1.2 TFLOPS、1 TFLOPS、0.67 TFLOPS, 与 CUBLAS 中的实现相比加速比最高分别达到了约 9×、8×、12×、13×, 与 MAGMA 中的实现相比加速比分别达到了约 1.2×–2.5×、1.2×–3.2×、1.1×–3×、1.1×–2.7×. 批量求逆算法的浮点计算性能分别可达到约 4 TFLOPS、2 TFLOPS、2.2 TFLOPS、1.2 TFLOPS, 与 CUBLAS 中的实现相比加速比最高分别达到了约 5×、4×、7×、7×, 与 MAGMA 中的实现相比加速比分别达到了约 2×–3×、2×–3×、2.8×–3.4×、1.6×–2×.

关键词: 批量 LU 分解; 选主元; 行交换; 批量矩阵求逆; 延迟修正; 动态方法

中图法分类号: TP301

中文引用格式: 刘世芳, 赵永华, 黄荣锋, 于天禹, 张馨尹. 基于批量LU分解的矩阵求逆在GPU上的有效实现. 软件学报, 2023, 34(11): 4952–4972. http://www.jos.org.cn/1000-9825/6727.htm

英文引用格式: Liu SF, Zhao YH, Huang RF, Yu TY, Zhang XY. Effective Implementation of Matrix Inversion Based on Batched LU Decomposition on GPU. Ruan Jian Xue Bao/Journal of Software, 2023, 34(11): 4952–4972 (in Chinese). http://www.jos.org.cn/1000-9825/6727.htm

Effective Implementation of Matrix Inversion Based on Batched LU Decomposition on GPU

LIU Shi-Fang[1,2], ZHAO Yong-Hua[1], HUANG Rong-Feng[1,2], YU Tian-Yu[1,2], ZHANG Xin-Yin[1,2]

[1](Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: This study presents the existing and optimized implementation methods for batched lower-upper (LU) matrix decomposition and batched inversion algorithms on the graphics processing unit (GPU). For batched LU decomposition, the study analyzes the number of

* 基金项目: 国家重点研发计划 (2020YFB0204802, 2017YFB0202202); 中国科学院战略性先导科技专项 (C 类)(XDC05000000); 光合基金 A 类 (20210701)
收稿时间: 2020-12-28; 修改时间: 2022-01-07; 采用时间: 2022-06-13; jos 在线出版时间: 2023-05-18
CNKI 网络首发时间: 2023-05-19

reads and writes to the global memory when the Left-looking, Right-looking, and other commonly used blocked LU decomposition algorithms are implemented on the GPU. The blocked Left-looking algorithm with less memory access data is selected due to the characteristics of the GPU architecture. In the process of pivoting during LU decomposition, a parallel binary tree search algorithm suitable for the GPU architecture is adopted. In addition, to reduce the impact of the row interchange process caused by the pivoting on the performance of the algorithm, this study proposes two optimization techniques, namely, the Warp-based packet row interchange and row interchange delay. For batched inversion after LU decomposition, this study investigates the correction method employed in the matrix inversion process. When batched inversion is implemented on the GPU, a blocked matrix inversion algorithm with delayed correction is adopted to reduce access to the global memory during the correction. Furthermore, to speed up data reading and writing, the study adopts the optimization method of using more registers and shared memory and that of performing column interchange to reduce memory access data. In addition, a method of dynamic GPU resource allocation during operation is proposed to avoid the idleness of threads and the waste of shared memory and other GPU resources. Compared with the static one-time resource allocation method, the dynamic allocation method improves the performance of the algorithm significantly. Finally, 10 000 random matrices with sizes between 33 and 190 data are tested on the TITAN V GPU, and the types of the tested data are single-precision complex, double-precision complex, single-precision real, and double-precision real. The floating-point arithmetic performance of the batched LU decomposition algorithm implemented in this study reaches about 2 TFLOPS, 1.2 TFLOPS, 1 TFLOPS, and 0.67 TFLOPS, respectively. This algorithm achieves the highest speedup of about 9×, 8×, 12×, and 13×, respectively, compared with the implementation in CUBLAS. The highest speedup achieved is about 1.2×–2.5×, 1.2×–3.2×, 1.1×–3× and 1.1×–2.7×, respectively, compared with the implementation in MAGMA. The floating-point arithmetic performance of the proposed batched inversion algorithm can reach about 4 TFLOPS, 2 TFLOPS, 2.2 TFLOPS, and 1.2 TFLOPS, respectively. This algorithm achieves the highest speedup of about 5×, 4×, 7×, and 7×, respectively, compared with the implementation in CUBLAS. The speedup is about 2×–3×, 2×–3×, 2.8×–3.4× and 1.6×–2×, respectively, compared with the implementation in MAGMA.

Key words: batched LU decomposition; pivoting; row interchange; batched inversion; delayed-correction; dynamic method

在 HPC (high performance computing) 领域, 大规模计算问题一直是科学和工程计算研究的热点. 然而, 在机器学习[1], 数据挖掘[2], 图像及信号处理[3−5], 计算流体力学[6], 天体物理学[7] 和量子化学[8]等许多科学计算应用中存在着大量小规模矩阵计算问题, 目前对同时处理大量小规模矩阵计算问题的研究得到越来越多的关注. 此外, 在一些超大规模计算问题的算法中, 也存在着对通过区域分解产生的大量小区域同时进行局部处理的问题. 这些领域需要处理的矩阵的行、列数通常在几十到几百之间, 但矩阵数量可能多至上万. 此类问题需要采用适合同时处理大量小规模矩阵计算的高效算法. 我们称这类同时处理大量小规模矩阵的问题为批量矩阵计算问题.

在解决 Cholesky 分解、LU 分解和 QR 分解等常见的矩阵计算问题时, 为了更好地实现数据的复用, 通常会采用将矩阵分成一系列列条块, 然后逐个处理这些列条块的块算法. 这些基于块算法的矩阵分解通常分为两步: 条块内分解和尾部矩阵更新. 对于在 GPU (graphics processing unit) 架构上实现大规模矩阵分解问题, 文献 [9−11] 已经给出了一个异构并行算法及其实现方法, 由于条块内分解计算为 BLAS-1 (向量和向量运算) 和 BLAS-2 (矩阵和向量运算), 这并不能充分利用 GPU 的计算能力, 而尾部矩阵的更新是一个可以充分发挥 GPU 计算能力的 BLAS-3 (矩阵和矩阵运算) 计算, 因此可将条块内分解放在 CPU 上计算, 而尾部矩阵更新放在 GPU 上进行计算, 该异构算法可将与条块内分解相关的 CPU-GPU 间数据传输和 GPU 上计算进行重叠. 对于大规模问题, 这种 CPU-GPU 异构算法能够得到较好的性能, 但对于小规模矩阵问题, 由于 GPU 计算时间相比于 CPU-GPU 之间的数据传输耗时较少, 因此对批量小规模矩阵, 这种异构算法带来的性能提升是有限的[12]. 因此对于批量小规模矩阵的 LU 分解和求逆在 GPU 上的实现, 通常采用将条块内分解和尾部矩阵更新都放在 GPU 上处理的方法[12−17].

GPU 是一种由大量运算单元组成的大规模并行计算架构, 可以支撑大量数据的并行计算, 适合批量处理具有相同计算任务的工作. 近年来已有许多工作关注于在 GPU 内批量矩阵处理运算, 文献 [18−21] 研究了在 GPU 内关于批处理小规模矩阵-矩阵乘法 GEMM 问题的实现. 文献 [12] 描述了批量处理矩阵常见的 Cholesky 分解、LU 分解和 QR 分解在 GPU 上的实现. 批量 LU 分解和批量矩阵求逆是许多科学计算和应用领域的关键计算问题[11,22−25]. 针对 GPU 架构, 基于批量 LU 分解的矩阵求逆算法实现方法得到了深入研究, 著名 GPU 生产商 NVIDIA 公司研发的 CUBLAS (CUDA basic linear algebra subroutine) 库[26]提供了批量 LU 分解与批量求逆的实现. GPU 数值代数

数学库 MAGMA (matrix algebra for GPU and multicore architectures) 中也提供了批量 LU 分解与批量求逆在 GPU 上的实现[27]. Villa 等人[14,15]给出了部分选主元与完全选主元的方法批量 LU 分解算法. Dong 和 Haidar 等人在文献 [12,13] 中给出了在 GPU 上批量多级 Right-looking LU 分解算法. Abdelfattah 等人在文献 [17] 中给出了极小规模的矩阵 (<32) 的批量 LU 分解与批量求逆算法.

本文第 1 节给出在 GPU 上实现批量 LU 分解与求逆的相关工作, 以及我们实现方法的优势与取得的效果. 第 2 节描述不同的求解 LU 分解和求逆的算法. 第 3 节给出在 GPU 上批量 LU 分解算法与批量求逆算法的设计实现和优化. 第 4 节给出我们实现的算法的实验结果与分析. 最后给出本文工作的总结.

1 相关工作

对于批量 LU 分解算法在 GPU 上的实现方法, Villa 等人在文献 [14] 中讨论了针对 GPU 架构的 3 种不同级别的并行方法. 第 1 种基于 CUBLAS 的 Warp 级并行 (一个 Warp 对应一个矩阵), 批处理函数为每个矩阵分配一个 Warp (32 个线程), 因此矩阵的规模不能超过 32×32. 此实现严重依赖共享内存, 但是共享内存的内容不会在不同的内核调用之间保留, 因此每个内核都必须进行额外的工作才能将数据重新缓存在共享内存中. 第 2 种 Thread-Block 级并行 (一个 Block 对应一个矩阵). 出于性能方面的考虑, 将矩阵完整地加载到共享内存中, 这意味着可以处理的矩阵的最大尺寸受到可用共享内存的限制. 对于 Fermi 和 Kepler 之前的 GPU 架构, 此实现可以处理尺寸为 76×76 (双精度) 的系统. 第 3 种针对矩阵规模小于 128 时, Thread 级并行 (一个线程对应一个矩阵), 此种方法每个线程在其对应的矩阵上进行操作, 在选主元的过程中, 由于每个矩阵的主元是不同的, 则每个线程寻找主元的过程是不同的, 则同一个 Warp 中的不同线程被分配了不同的任务, 这可能会由于 Warp 中的线程发散导致性能不佳. 第 1 种和第 2 种方法仅支持部分选主元, 第 3 种方法支持部分选主元和完全选主元中方法. 特别地, 第 3 种 Thread 级并行方法用于求解一个线性方程组时性能不如 CUBLAS 中的实现方法[13]. 该方法已用于地下运输模拟, 对流动路径中的许多化学和微生物反应进行了并行模拟[15].

Dong 和 Haidar 等人在文献 [12,13] 中实现了在 GPU 上批量 Right-looking LU 分解块算法. 提出了多级 Right-looking LU 分解块算法. 将 LU 分解分为 4 个过程, 在 GPU 上启动 4 个 Kernel 完成. 第 1 步是当前列条块的分解, 在分解过程中并没有将整个列条块从全局内存写到共享内存, 仅将一列写到了共享内存. 第 2 步是行交换, 给出了 NB 次行交换并行交换方法, 这里的 NB 表示块算法中块的大小. 第 3 步求解 $\boldsymbol{U}$ 的子矩阵块 $\boldsymbol{U}_{12}$. 第 4 步修正尾部矩阵, 这也是多级 Right-looking LU 分解算法与普通 Right-looking 块算法的不同之处. 通常的 Right-looking 块算法对当前列条块分解完成后, 对整个右边未求解的尾部矩阵进行修正, 而在多级块算法对尾部矩阵进行修正过程中, 仅对下一个列条块部分进行修正, 推迟对剩余部分的修正, 这是软件包 MAGMA 采用的方法.

Abdelfattah 等人在文献 [17] 中实现了部分选主元的批量矩阵 LU 分解, 也实现了求批量矩阵的逆矩阵, 但他们考虑的矩阵大小不超过 32. 在实现时因为矩阵非常小, 为了加速数据的访问, 因此将整个矩阵保存在寄存器中. Block 中的线程采用了一维结构, 因此实现时不需要同步点. 另外, 由于矩阵规模小, 若一个 Block 对应一个矩阵, 则一个 Block 中的线程的数目太少, 为了提高占用率, 采用一个 Block 同时分解多个矩阵的方法. 对于 LU 分解中的行交换步骤, 采取了 lazy swap 技术, 每个线程记录它对应行是否是最大主元所在行, 只有不是主元所在行对应的线程对尾部矩阵进行修正, 并且在下一次分解时选取最大主元. 分解完成后, 每个线程都知道其行的最终位置, 并将其直接写入全局内存, 此文采用的是 Right-looking LU 分解块算法.

国内对于在 GPU 上实现批量 LU 分解与求逆的研究比较少. 对于单个矩阵的 LU 分解在 GPU 上的实现, 文献 [28] 实现了部分选主元的 Right-looking LU 分解算法. 其中, 选取主元的部分在 CPU 上实现, CPU 每次找到主元并进行行交换后, 再把数据传到 GPU 上, GPU 负责处理主元列的并行计算和子矩阵元素的并行计算. 在 GPU 上计算时, 由于共享内存的大小限制, 当矩阵规模过大时, 不能将整个矩阵放到共享内存中, 因此采取了矩阵分段计算: 每次将两列元素放入共享内存计算. 将 LU 分解的过程分成在 CPU 和 GPU 上进行, 则每次迭代都需要 CPU 和 GPU 之间数据传输, 频繁的数据传输将会影响程序的性能. 此外, 该论文中实现的是非块的 Right-looking LU 分解算法, 大多数计算都是 BLAS-1 操作, 并不能充分利用 GPU 的计算性能. 此算法并不适用于批量的 LU 分解.

文献 [29] 中讨论了电力系统潮流计算当中对稀疏矩阵的基于 GPU 并行加速的批量 Left-looking LU 分解算法.

在 GPU 上实现批量 LU 分解算法时, 我们采用了一个 Block 对应一个矩阵的 Thread-Block 级并行方法, 不同于 Villa 等人在文献 [14] 中 Thread-Block 级并行方法, 我们通过使用块算法, 只需将要处理的当前列条块放在共享内存中, 而不是将整个矩阵放在共享内存中, 这样处理的矩阵规模就不受共享内存大小的限制. 在 GPU 内实现 LU 分解之后的批量求逆算法时, 我们同样采取了 Thread-Block 级并行块方法. 由于一个线程对应矩阵的一行, 并且列条块内每行的求解相互独立, 因此可将当前列条块存储在线程的私有寄存器中, 加快数据的读写速度. 我们的算法通过更好地利用 GPU 的各级内存, 提升了算法的性能. Dong 等人[13]在 GPU 上实现批量多级 Right-looking LU 分解算法时, 将 LU 分解分为 4 个过程, 在 GPU 上启动 4 个 Kernel 完成. 多次启动 Kernel, 共享内存的内容不会在不同的 Kernel 调用之间保留, 因此每个 Kernel 都必须将数据重新缓存在共享内存中, 这存在数据的重复读取, 对性能造成一定的影响, 当矩阵规模较小时造成的影响更大. 因为在 GPU 上影响批量小矩阵处理性能的主要因素是数据的读写时间[30]. 在 GPU 上实现批量 LU 分解时, 由于 Left-looking 算法比 Right-looking 算法具有更少的对全局内存数据访问量, 因此我们采用了基于 Left-looking 方法的批量 LU 分解算法. 并且将整个 LU 分解过程放在一个 Kernel 中完成, 这样只需要将要处理的当前列条块从全局内存中读取一次, 求解完成后再写回即可, 避免了数据的重复读取.

在 GPU 上实现批量 LU 分解算法时, 由于不选主元的 LU 分解算法数值不稳定, 因此我们采用了列选主元的 LU 分解算法. 以列优先存储的矩阵在 GPU 上进行 LU 分解时, 由于选主元存在矩阵的行交换过程, 而同一行的矩阵元素地址是不连续的, 造成线程的访存地址不连续 (non-coalesced memory accesses), 大大降低了在 GPU 中实现的 LU 分解算法的性能. 为此, 我们提出了 Warp 分组行交换和行交换延迟 2 个优化方案降低了行交换过程中访存地址不连续带来的性能影响. 在 TITAN V GPU 上对 10 000 个规模在 33–190 之间的随机矩阵进行了测试, 测试的数据类型为单精度复数、双精度复数、单精度实数和双精度实数. 我们实现的批量 LU 分解算法的浮点计算性能分别可达到约 2 TFLOPS、1.2 TFLOPS、1 TFLOPS、0.67 TFLOPS, 并同 CUBLAS 和 MAGMA 中的批量 LU 分解算法的实现进行了比较, 与 CUBLAS 中的实现相比加速比分别最高达到了约 9×、8×、12×、13×, 与 MAGMA 中的实现相比加速比分别达到了约 1.2×–2.5×、1.2×–3.2×、1.1×–3×、1.1×–2.7×.

LU 分解之后的求逆块算法包括列条块求逆和尾部矩阵修正两个阶段. 为了减少修正过程对全局内存的读写数据量, 我们在批量求逆算法的 GPU 实现中提出了延迟修正的矩阵求逆块算法. 此外, 为了减少对全局内存的访问和加快数据的读写速度, 求逆算法采用了更多利用寄存器和共享内存的优化方法, 并设计了减少访存数据量的列交换方法. 另外, 为了避免采用一次性分配 GPU 资源造成的线程闲置和共享内存浪费问题, 本文设计了运行时按需分配线程、共享内存等 GPU 资源的动态资源分配方法, 相比一次性分配的静态资源分配方法, 性能得到了一定的提升. 在 TITAN V GPU 上对 10 000 个规模在 33–190 之间的随机矩阵进行了测试, 测试的数据类型为单精度复数、双精度复数、单精度实数和双精度实数. 我们实现的批量求逆块算法浮点计算性能分别可达到约 4 TFLOPS、2 TFLOPS、2.2 TFLOPS、1.2 TFLOPS, 与 CUBLAS 中的实现相比加速比分别最高达到了约 5×、4×、7×、7×, 与 MAGMA 中的实现相比加速比分别达到了约 2×–3×、2×–3×、2.8×–3.4×、1.6×–2×.

2 基于 LU 分解的矩阵求逆算法

通过矩阵 $\boldsymbol{A}$ 的 LU 分解求解 $\boldsymbol{A}$ 的逆比直接对矩阵 $\boldsymbol{A}$ 求逆更加高效. 基于 LU 分解的矩阵求逆算法主要分为两步, 首先对矩阵 $\boldsymbol{A}$ 进行选主元的 LU 分解得到 $\boldsymbol{A}=\boldsymbol{PLU}$, 其中 $\boldsymbol{L}$ 为单位下三角矩阵, $\boldsymbol{U}$ 为上三角矩阵, $\boldsymbol{P}$ 为置换矩阵. 然后对三角矩阵 $\boldsymbol{L}$ 和 $\boldsymbol{U}$ 分别求逆, 最后由置换矩阵 $\boldsymbol{P}$ 得到原始矩阵 $\boldsymbol{A}$ 的逆.

2.1 选主元的 LU 分解算法

选主元的 LU 分解就是将一个 $n\times n$ 的可逆方阵 $\boldsymbol{A}$ 分解成 $n\times n$ 置换矩阵 $\boldsymbol{P}$、$n\times n$ 的单位下三角矩阵 $\boldsymbol{L}$ 和 $n\times n$ 的上三角矩阵 $\boldsymbol{U}$ 的乘积, 即 $\boldsymbol{A}=\boldsymbol{PLU}$. 在实现时, 置换矩阵 $\boldsymbol{P}$ 可以压缩存储在一个向量中. 算法 1 给出了对矩阵 $\boldsymbol{A}$ 进行 LU 分解的非块算法, 首先寻找当前第 i 列中最大主元所在的行号 $maxID$, 并存储在 $\boldsymbol{P}[i]$ 中 (第 2 行), 然后

交换第 i 和第 $maxID$ 行 (第 3 行), 交换完成后对第 i 列元素进行修正 (第 4 行), 最后对尾部矩阵进行秩–1 修正 (第 5 行) , 这里的计算为 BLAS-1 运算.

算法 1. 部分选主元的 LU 分解非块算法.

```
1  for i = 1 to n
2     P[i] = maxID = max(abs(A[i:n,i])); /*寻找最大主元所在的行*/
3     A[i,1:n] ↔ A[maxID,1:n]; /*行交换*/
4     A[i+1:n,i] = A[i+1:n,i]/A[i,i];
5     A[i+1:n,i+1:n] -= A[i+1:n,i]×A[i,i+1:n];
6  endfor
```

为了更好地实现数据的复用, 通常 LU 分解采用具有更多 BLAS-3 运算的块算法, LU 分解的块算法将矩阵分成一系列宽度为 NB 列条块, 然后逐个处理这 bn ($bn=\lceil n/NB\rceil$) 个列条块. 常用的 LU 分解的块算法分为 Right-looking LU 和 Left-looking LU 两种[20]. 算法 2 和算法 3 分别给出了这两种 LU 分解块算法, 在算法中求得的单位下三角矩阵 $\boldsymbol{L}$ 和上三角矩阵 $\boldsymbol{U}$ 分别覆盖矩阵 $\boldsymbol{A}$ 的上下三角部分, 矩阵 $\boldsymbol{A}$ 对角线存储的是矩阵 $\boldsymbol{U}$ 的对角线元素.

算法 2 给出了 Right-looking LU 分解的块算法, 首先对如图 1 所示的当前第 i 个列条块 $\begin{bmatrix}\boldsymbol{A}_{11}\\ \boldsymbol{A}_{21}\end{bmatrix}$ 进行部分选主元的 LU 分解得到 $\boldsymbol{L}_{11}$、$\boldsymbol{L}_{21}$ 和 $\boldsymbol{U}_{11}$ 并覆盖 $\boldsymbol{A}$ 的对应部分 (第 3 行), 分解完成后对当前列条块外的两侧进行行交换 (第 5 行), 行交换完成后立即求解矩阵 $\boldsymbol{U}$ 的子块 $\boldsymbol{U}_{12}$ (第 7 行), 最后用 $\boldsymbol{A}_{21}$ ($\boldsymbol{L}_{21}$) 和 $\boldsymbol{A}_{12}$ ($\boldsymbol{U}_{12}$) 来修正如图 1 所示的当前列条块右边未求解的尾部矩阵 $\boldsymbol{A}_{22}$ (第 8 行), 修正完成后向右继续处理下一个列条块, 直到最后一个列条块. 如图 1 所示, 由于算法 2 只关注了当前列条块及其右边部分, 因此称为 Right-looking LU 分解块算法.

算法 3 给出了 Left-looking LU 分解算法, 在求解如图 2 所示当前第 i 个列条块之前, 先用其左边前 $i-1$ 个列条块已经求解出的 $\boldsymbol{L}_{00}$ 和 $\boldsymbol{A}_{01}$ 来求解 $\boldsymbol{U}_{01}$ (第 3 行), 然后用左边前 $i-1$ 个列条块已经求解出的 $\begin{bmatrix}\boldsymbol{A}_{10}\\ \boldsymbol{A}_{20}\end{bmatrix}\left(\begin{bmatrix}\boldsymbol{L}_{10}\\ \boldsymbol{L}_{20}\end{bmatrix}\right)$ 和 $\boldsymbol{A}_{01}$ ($\boldsymbol{U}_{01}$) 对当前第 i 个列条块进行修正 (第 4 行), 修正完成后对当前列条块进行部分选主元的 LU 分解算法 (第 6 行), 最后对列条块外的两侧进行行交换 (第 8 行), 然后向右处理下一个列条块, 直到最后一个列条块. 如图 2 所示, 由于算法 3 只关注了当前列条块及其左边的部分, 因此可以称为 Left-looking LU 分解块算法.

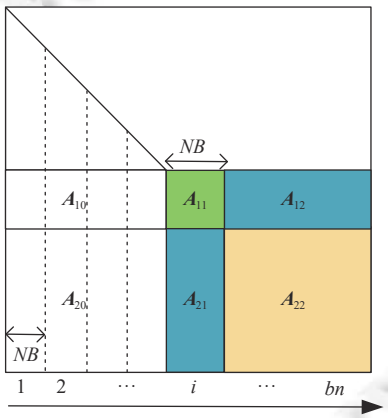

图 1 Right-looking LU 分解块算法示意图

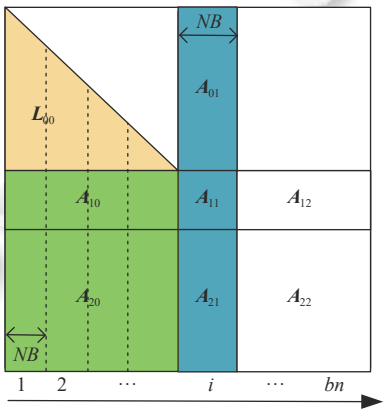

图 2 Left-looking LU 分解块算法示意图

算法 2. Right-looking LU 分解块算法.

```
1  for i = 1 to bn
```

2 /*当前列条块的选主元 LU 分解*/

3 $\begin{bmatrix} \boldsymbol{A}_{11} \\ \boldsymbol{A}_{21} \end{bmatrix} = \boldsymbol{P}\begin{bmatrix} \boldsymbol{L}_{11} \\ \boldsymbol{L}_{21} \end{bmatrix} \times \boldsymbol{U}_{11}$;

4 /*列条块外部分的行交换*/

5 $\begin{bmatrix} \boldsymbol{A}_{10} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22} \end{bmatrix} = \boldsymbol{P}\begin{bmatrix} \boldsymbol{A}_{10} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22} \end{bmatrix}$;

6 /*修正尾部矩阵*/

7 $\boldsymbol{A}_{12} := \boldsymbol{U}_{12} = \boldsymbol{L}_{11}^{-1} \times \boldsymbol{A}_{12}$;

8 $\boldsymbol{A}_{22} -= \boldsymbol{A}_{21} \times \boldsymbol{A}_{12}$;

9 endfor

算法 3. Left-looking LU 分解块算法.

1 for $i = 1$ to bn

2 /*修正当前列条块*/

3 $\boldsymbol{A}_{01} := \boldsymbol{U}_{01} = \boldsymbol{L}_{00}^{-1} \times \boldsymbol{A}_{01}$;

4 $\begin{bmatrix} \boldsymbol{A}_{11} \\ \boldsymbol{A}_{21} \end{bmatrix} -= \begin{bmatrix} \boldsymbol{A}_{10} \\ \boldsymbol{A}_{20} \end{bmatrix} \times \boldsymbol{A}_{01}$;

5 /*当前列条块的选主元 LU 分解*/

6 $\begin{bmatrix} \boldsymbol{A}_{11} \\ \boldsymbol{A}_{21} \end{bmatrix} = \boldsymbol{P}\begin{bmatrix} \boldsymbol{L}_{11} \\ \boldsymbol{L}_{21} \end{bmatrix} \times \boldsymbol{U}_{11}$;

7 /*列条块外部分的行交换*/

8 $\begin{bmatrix} \boldsymbol{A}_{10} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22} \end{bmatrix} = \boldsymbol{P}\begin{bmatrix} \boldsymbol{A}_{10} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22} \end{bmatrix}$;

9 endfor

2.2 LU 分解后矩阵求逆算法

在矩阵 $\boldsymbol{A}$ 的 LU 分解之后得到 $\boldsymbol{A} = \boldsymbol{PLU}$, 则 LU 分解之后的求逆过程可以分为如下 3 步.

(1) 求上三角矩阵 $\boldsymbol{U}$ 的逆 $\boldsymbol{U}^{-1}$;

(2) 求解下三角方程组 $\boldsymbol{XL} = \boldsymbol{U}^{-1}$, 得到 $\boldsymbol{X}$;

(3) 对 $\boldsymbol{X}$ 进行列变换 $\boldsymbol{P}$, 得到 $\boldsymbol{A}^{-1}$.

上述第 (1) 步求解矩阵 $\boldsymbol{U}$ 的逆可以通过求解方程组 $\boldsymbol{XU} = \boldsymbol{I}$ 或者 $\boldsymbol{UX} = \boldsymbol{I}$ 实现, 这里 $\boldsymbol{I}$ 为单位矩阵. 其中方程组 $\boldsymbol{XU} = \boldsymbol{I}$ 适合按列优先存储的矩阵, 方程组 $\boldsymbol{UX} = \boldsymbol{I}$ 适合按行优先存储的矩阵. 由于本文矩阵采用了按列优先的存储方式, 因此 $\boldsymbol{U}$ 的逆通过求解上三角方程组 $\boldsymbol{XU} = \boldsymbol{I}$ 获得.

与 LU 分解算法一样求逆过程也通过使用块算法使其具有更多 BLAS-3 运算. 求逆块算法将矩阵分成一系列大小 NB 的列条块, 然后逐次对这 bn ($bn = \lceil n/NB \rceil$) 个列条块进行求解. 求逆块算法包括列条块求逆和尾部矩阵修正 2 个模块. 根据对尾部矩阵是否及时修正, 可以分为及时修正与延迟修正 2 种块算法. 算法 4 和算法 5 分别给出了这 2 种求逆块算法. 这里假设 LU 分解得到的上三角矩阵为 $\boldsymbol{U}$, $\boldsymbol{U}^{-1}$ 通过求解上三角方程组 $\boldsymbol{XU} = \boldsymbol{W}$ 获得, 且求得的逆矩阵的结果 $\boldsymbol{U}^{-1}$ 覆盖矩阵 $\boldsymbol{W}$, 初始时 $\boldsymbol{W}$ 为单位矩.

算法 4 给出了及时修正的求解方程组 $\boldsymbol{XU} = \boldsymbol{W}$ 的块算法, 求解过程如图 3, 初始时 $\boldsymbol{W} = \boldsymbol{I}$, 且结果覆盖 $\boldsymbol{W}$. 首先如图 3(a) 中当前第 i 个列条块 (第 2 行), 当前列条块的结果可通过求解上三角方程组 $\begin{bmatrix} \boldsymbol{X}_{01} \\ \boldsymbol{X}_{11} \end{bmatrix} \boldsymbol{U}_{11} = \begin{bmatrix} \boldsymbol{W}_{01} \\ \boldsymbol{W}_{11} \end{bmatrix}$ 得到, 且求解的结果覆盖矩阵 $\boldsymbol{W}$ 的列条块 $\begin{bmatrix} \boldsymbol{W}_{01} \\ \boldsymbol{W}_{11} \end{bmatrix}$, 求解完成后与 $\boldsymbol{U}$ 的子矩阵 $\boldsymbol{U}_{12}$ 修正列条块右边剩余未求解的尾部矩阵 (第 3 行), 然后向右进行下一个列条块的求解, 直到最后一个列条块.

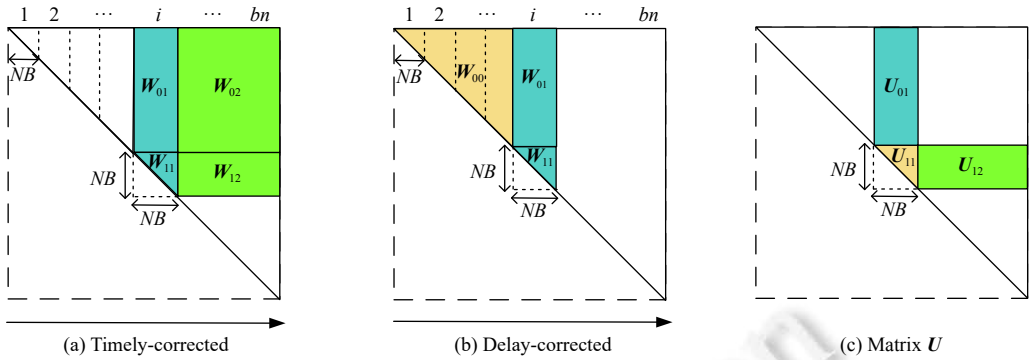

图 3 及时/延时修正的求 $\boldsymbol{XU}=\boldsymbol{W}$ 块算法示意图

算法 5 给出了延迟修正的求 $\boldsymbol{XU}=\boldsymbol{W}$ 的块算法, 在求解如图 3(b) 中当前第 i 个列条块之前, 首先用当前列条块左边已经求解出前 $i-1$ 个列条块的结果 $\boldsymbol{W}_{00}$ 与 $\boldsymbol{U}$ 的子矩阵 $\boldsymbol{U}_{01}$ 来修正当前第 i 个列条块 $\boldsymbol{W}_{01}$ (第 2 行), 修正完成后再求解当前列条块 (第 3 行), 然后向右进行下一个列条块的求解, 直到最后一个列条块.

算法 4. 及时修正的求 $\boldsymbol{XU}=\boldsymbol{W}$ 的块算法.

```
1  for i=1 to bn
2     [W01; W11] = [W01; W11] × U11^-1 ; /*求解当前列条块*/
3     [W02; W12] -= [W01; W11] × U12 ; /*修正剩余尾部矩阵*/
4  endfor
```

算法 5. 延迟修正的求 $\boldsymbol{XU}=\boldsymbol{W}$ 的块算法.

```
1  for i = 1 to bn
2     W01 -= W00 × U01; /*修正当前列条块*/
3     [W01; W11] = [W01; W11] × U11^-1 ; /*求解当前列条块*/
4  endfor
```

前文所述求逆过程的第 (2) 步中, 通过求解一个右端项为 $\boldsymbol{U}^{-1}$ 的下三角方程组 $\boldsymbol{XL}=\boldsymbol{U}^{-1}$ 得到 $\boldsymbol{X}$, 因为第 (1) 步得到 $\boldsymbol{U}^{-1}$ 的结果存储在矩阵 $\boldsymbol{W}$ 中, 因此可以通过求解方程组 $\boldsymbol{XL}=\boldsymbol{W}$ 实现, 并且求得的结果覆盖矩阵 $\boldsymbol{W}$. 第 (2) 步求解下三角方程组的过程与第 (1) 步求解上三角方程组的过程类似, 块算法包括列条块求解和尾部矩阵修正两个模块. 根据对尾部矩阵是否及时修正, 可以分为及时修正与延迟修正 2 种块算法. 算法 6 和算法 7 具体描述了这 2 种方法, 对应的块算法示意图如图 4 所示, 初始时 $\boldsymbol{W}=\boldsymbol{U}^{-1}$, 且结果覆盖 $\boldsymbol{W}$. 求解过程从右到左. 这里假设 LU 分解得到的单位下三角矩阵为 $\boldsymbol{L}$.

算法 6. 及时修正的求 $\boldsymbol{XL}=\boldsymbol{W}$ 的块算法.

```
1  for i = 1 to bn
2     W1 = W1 × L11^-1 ; /*求解当前列条块*/
3     W0 -= W1 × L10 ; /*修正剩余尾部矩阵*/
4  endfor
```

算法 7. 延迟修正的求 $\boldsymbol{XL}=\boldsymbol{W}$ 的块算法.

1 for $i=1$ to bn
2 $\quad \boldsymbol{W}_1 -= \boldsymbol{W}_2 \times \boldsymbol{L}_{21}$; /*修正当前列条块*/
3 $\quad \boldsymbol{W}_1 = \boldsymbol{W}_1 \times \boldsymbol{L}_{11}^{-1}$; /*求解当前列条块*/
4 endfor

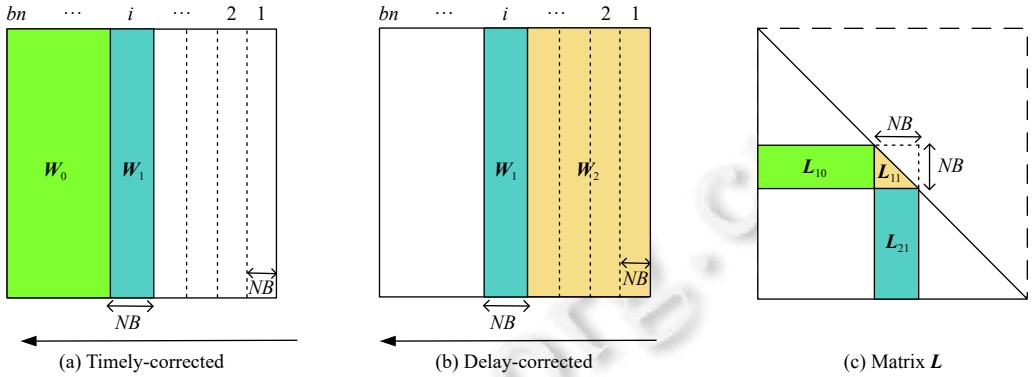

图 4 及时/延时修正求方程组 $\boldsymbol{XL}=\boldsymbol{W}$ 块算法示意图

3 基于批量 LU 分解的矩阵求逆算法在 GPU 上的设计实现和优化

在 GPU 上实现批量算法时, 如图 5 所示一个 Block 对应一个矩阵. 而对于 Block 内线程的配置, 可以是 1D 的也可以是 2D 的. 由于 2D 线程结构需要使用 $n\times n$ 个线程对应矩阵的 $n\times n$ 个元素, 限制了 CUDA 运行时每个 SM (streaming multiprocessor) 同时调度多个 Block 的能力. 另外对于大多数矩阵规模而言, 采用 2D 结构的线程个数通常会超出每个 Warp 的线程数目 (32 个), 这会导致在线程需要共享数据时 2D 会比 1D 结构更多必要的同步点. 并且, 当采用 2D 结构对当前列条块进行处理时, 由于一个线程对应矩阵的一个元素, 这会造成条块外的线程处于空闲状态, 导致线程利用率较低. 另外, 1D 结构中一个线程对应矩阵的一行, 这使得每个线程被分配更多的任务, 指令级并行性更好, 因此我们采用了如图 5 所示的 1D 结构. 假设批量矩阵已经存储于 GPU 的全局内存上, 并且矩阵元素是按照列主序存储, 共有 bs 个矩阵.

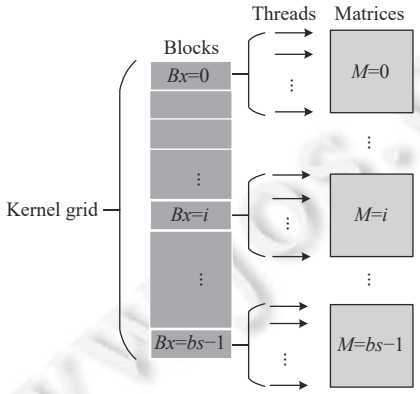

图 5 GPU 上 Grid 和 Block 与矩阵的对应关系示意图

3.1 批量 LU 分解块算法在 GPU 上的有效实现

文献 [15] 指出, 影响批量小矩阵在 GPU 上性能的问题的主要因素是对全局内存中数据的读写时间, 而不是计算时间, 因此, 在选择算法时, 应选择具有更少数据读写次数的算法. 由于 Left-looking LU 分解块算法比 Right-

looking LU 分解块算法具有更少的对全局内存的数据读写次数, 因此, 在 GPU 上的实现批量 LU 分解时, 我们选取了 Left-looking LU 分解块算法. 算法 8 和算法 9 给出了这两种算法在 GPU 上的实现, 并通过算法 8 和算法 9 分析说明了 Left-looking 对全局内存的数据读取量少于 Right-looking 对全局内存的数据读取次数, 算法中下标 “S” 的变量表示存储在共享内存中的变量.

如算法 8 描述, 在 GPU 上实现 Right-looking LU 分解块算法时, 由于对当前列条块中的数据需要进行多次读写, 则首先将列条块从全局内存读到共享内存 (第 3 行), 然后对当前列条块进行部分选主元的 LU 分解 (第 7 行), 分解完成后再将列条块写回到全局内存 (第 10 行), 此时对全局内存的数据读写数据量为 $2\times(n-(i-1)\times NB)\times NB$. 然后, 对除当前列条块外的两侧进行行交换 (第 12 行), 这个过程数据读写次数为 $4\times(n\times NB-NB\times NB)$. 之后将列条块右边的矩阵 $\boldsymbol{U}$ 的子块 $\boldsymbol{U}_{12}$ 读到共享内存 (第 4 行) 并进行计算 (第 15 行), 计算完成后再写回到全局内存 (第 17 行), 此时数据读写次数为 $2\times NB\times(n-i\times NB)$, 最后用列条块的求解结果与 $\boldsymbol{U}_{\mathrm{S},12}$ ($\boldsymbol{A}_{\mathrm{S},12}$) 修正尾部矩阵 $\boldsymbol{A}_{22}$ (第 18 行), 数据读写次数为 $2\times(n-i\times NB)\times(n-i\times NB)$. 则完成整个 Right-looking LU 分解块算法对全局内存的读写总次数约为: $V_{\mathrm{R}}=\frac{2}{3NB}n^3+5n^2-\frac{11}{3}\times NB\times n$.

算法 8. GPU 上的 Right-looking LU 分解块算法.

1 for $i=1$ to bn
2 /*将当前列条块读到共享内存*/
3 $\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}\leftarrow\begin{bmatrix}\boldsymbol{A}_{11}\\ \boldsymbol{A}_{21}\end{bmatrix}$;
4 $\boldsymbol{A}_{\mathrm{S},12}\leftarrow\boldsymbol{A}_{12}$;
5 __*syncthreads*();
6 /*当前列条块的选主元 LU 分解*/
7 $\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}=\boldsymbol{P}\begin{bmatrix}\boldsymbol{L}_{\mathrm{S},11}\\ \boldsymbol{L}_{\mathrm{S},21}\end{bmatrix}\times\boldsymbol{U}_{\mathrm{S},11}$;
8 __*syncthreads*();
9 /*将当前列条块写回到全局内存*/
10 $\begin{bmatrix}\boldsymbol{A}_{11}\\ \boldsymbol{A}_{21}\end{bmatrix}\leftarrow\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}$;
11 /*列条块外部分行交换*/
12 $\begin{bmatrix}\boldsymbol{A}_{10} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22}\end{bmatrix}=\boldsymbol{P}\begin{bmatrix}\boldsymbol{A}_{10} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22}\end{bmatrix}$;
13 __*syncthreads*();
14 /*修正尾部矩阵*/
15 $\boldsymbol{A}_{\mathrm{S},12}=\boldsymbol{L}_{\mathrm{S},11}^{-1}\times\boldsymbol{A}_{\mathrm{S},12}$;
16 __*syncthreads*();
17 $\boldsymbol{A}_{12}\leftarrow\boldsymbol{A}_{\mathrm{S},12}$; /*将 $\boldsymbol{A}_{12}$ 写回到全局内存*/
18 $\boldsymbol{A}_{22}-=\boldsymbol{A}_{\mathrm{S},21}\times\boldsymbol{A}_{\mathrm{S},12}$;
19 __*syncthreads*();
20 endfor

在 GPU 上实现如算法 9 所描述的 Left-looking LU 分解块算法时, 首先将当前列条块从全局内存读到共享内存 (第 3 行), 然后用当前列条块左边已经求解出的 $\boldsymbol{L}_{00}$ 来求解矩阵 $\boldsymbol{U}$ 的子块 $\boldsymbol{U}_{\mathrm{S},01}$ ($\boldsymbol{A}_{\mathrm{S},01}$) (第 6 行), 然后修正列条块 $\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}$ (第 8 行), 此时对全局内存的数据读取量为 $n\times NB+n\times(i-1)\times NB$. 修正完成后, 对列条进行部分选主元的 LU 分解 (第 11 行), 分解完成后将列条块写回到全局内存 (第 14 行), 此时读写数据量为 $n\times NB$. 最后对列条块

外的部分进行行交换 (第 16 行), 这个过程数据读写次数为 $4\times(n\times NB-NB\times NB)$. 则完成整个 Left-looking LU 分解块算法对全局内存的读写总量约为: $V_{\mathrm{L}}=\frac{1}{2NB}n^3+\frac{11}{2}n^2-4\times NB\times n$.

算法 9. GPU 上的 Left-looking LU 分解块算法.

1 for $i=1$ to bn
2 /*将当前列条块读到共享内存*/
3 $\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},01}\\ \boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}\leftarrow\begin{bmatrix}\boldsymbol{A}_{01}\\ \boldsymbol{A}_{11}\\ \boldsymbol{A}_{21}\end{bmatrix}$;
4 __*syncthreads*();
5 /*修正当前列条块*/
6 $\boldsymbol{A}_{\mathrm{S},01}=\boldsymbol{L}_{00}^{-1}\times\boldsymbol{A}_{\mathrm{S},01}$;
7 __*syncthreads*();
8 $\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}-=\begin{bmatrix}\boldsymbol{A}_{10}\\ \boldsymbol{A}_{20}\end{bmatrix}\times\boldsymbol{A}_{\mathrm{S},01}$;
9 __*syncthreads*();
10 /*当前列条块的选主元 LU 分解*/
11 $\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}=\boldsymbol{P}\begin{bmatrix}\boldsymbol{L}_{\mathrm{S},11}\\ \boldsymbol{L}_{\mathrm{S},21}\end{bmatrix}\times\boldsymbol{U}_{\mathrm{S},11}$;
12 __*syncthreads*();
13 /*将当前列条块写回到全局内存*/
14 $\begin{bmatrix}\boldsymbol{A}_{01}\\ \boldsymbol{A}_{11}\\ \boldsymbol{A}_{21}\end{bmatrix}\leftarrow\begin{bmatrix}\boldsymbol{A}_{\mathrm{S},01}\\ \boldsymbol{A}_{\mathrm{S},11}\\ \boldsymbol{A}_{\mathrm{S},21}\end{bmatrix}$;
15 /*列条块外部分行交换*/
16 $\begin{bmatrix}\boldsymbol{A}_{10} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22}\end{bmatrix}=\boldsymbol{P}\begin{bmatrix}\boldsymbol{A}_{10} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{20} & \boldsymbol{A}_{22}\end{bmatrix}$;
17 __*syncthreads*();
18 endfor

在 GPU 上实现 Left-looking LU 分解块算法时, 除了将当前列条块读到共享内存, 计算完成后再将其写回到全局内存外, 另外有 3 个关键步骤: 修正列条块、对列条块进行部分选主元的 LU 分解和列条块外部分的行交换. 接下来我们将对这 3 个步骤在 GPU 上的实现设计与优化进行描述.

(1) 修正当前列条块

修正当前第 i 列条块 (已存储于共享内存中) 的每列数据需要列条块左边已经求解出的前 $i-1$ 个列条块 (存储于全局内存) 中的每列数据, 为了减少从读取效率低的全局内存读取数据量, 所有线程同时从当前列条块的左边读取一列数据到寄存器, 然后修正当前列条块的每列数据. 修正当前列条块所有列数据之后, 所有线程再同时读取下一列数据到寄存器, 继续修正当前列条块的所有列. 此设计有 2 个优点, 首先从全局内存读取数据没有 Warp 发散. 其次, 只对存储在全局内存中的当前列条块左边部分进行了一次读取就更新了当前列条块中的所有数据. 如图 6 所示.

(2) 对列条块进行部分选主元的 LU 分解

对列条块进行部分选主元的 LU 分解时, 需要找出绝对值最大的元素所在行, 为了让更多的线程参与搜索过程, 减少搜索时间, 我们采用线程访存连续的 (coalesced) 并行二叉树搜索算法进行, 每一个线程对应一列中两个元素的比较. 如图 7 所示, 假设当前列需要比较的元素个数为 8, 则第 1 层共需要 4 个线程并行地对 a_1 和 a_5 (黑色)、

a_2 和 a_6 (蓝色)、a_3 和 a_7 (红色)、a_4 和 a_8 (橙色) 找到最大值, 显然这 4 个线程对应的元素 a_1-a_4 、a_5-a_8 是连续的 (coalesced) 的, 因此二叉树搜索算法的在这里效率很高. 然后第 2 层再对找到的 4 个最大值执行相同的过程, 直到找到本列的最大主元.

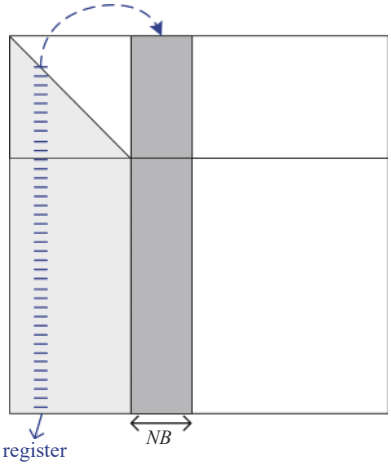

图 6 修正当前列条块示意图

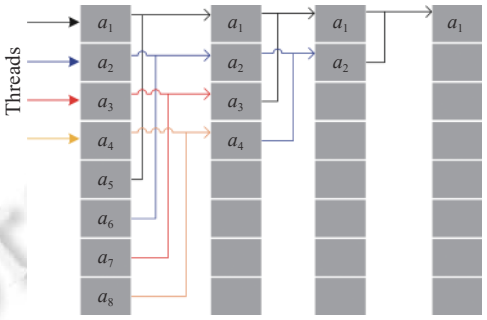

图 7 访存连续的并行二叉树搜索算法示意图

(3) 列条块外两侧的行交换

对列条外的部分进行行交换时, 通常的做法是 NB 次行交行逐次进行, 每个线程对应一列的 2 个元素交换, 这种方法虽然线程利用率高, 但是由于矩阵是以列主序存储在全局内存中, 则在行交换过程线程访问全局内存的地址是不连续的 (non-coalescent), 同一个 Warp 中的线程对应的数据地址跨步大, 这会使得行交换过程耗时较长, 对性能造成影响. 为了降低线程访问全局内存地址不连续带来的影响, 我们提出了 Warp 分组行交换技术, 将线程按照 Warp 进行分组, 组内重新设计线程与矩阵元素的对应关系, 再将线程细分成多组, 每组包含 NB 个线程. 由于行交换过程至多进行 NB 次行交换, 因此 NB 个线程刚好对应 NB 次行交换. 注意为了避免线程同步影响性能, 对矩阵同一列进行行交换的 NB 个线程必须属于同一个 Warp. 如图 8 所示, 假设块的大小 NB=8, 线程按照 Warp 分成了红、蓝两组, 组内再将线程分成 4 组, 每组 8 个连续线程, 同时考虑 8 次行交换, 也就是将线程重新设计成一个 [8, 4] 的二维结构. 红蓝两组同时进行, 完成这两部分的行交换后, 往右推进, 直到所有的行交换完成.

此外, 由于列条块右边的部分, 随着算法的进行会被取到共享内存中, 因此可将这部分的行交换推迟到后边取到共享内存中时再进行, 加快这部分的行交换的时间, 如图 9 所示. 此时 Left-looking LU 分解块算法对全局内存的数据读取量减少 $2n^2-2\times NB\times n$.

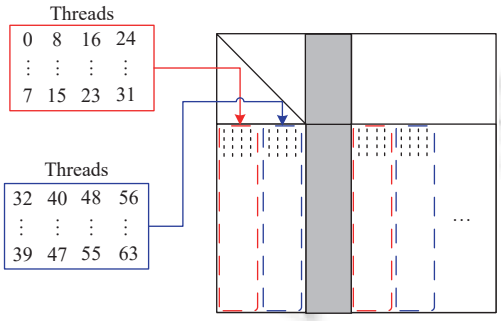

图 8 Warp 分组行交换示意图

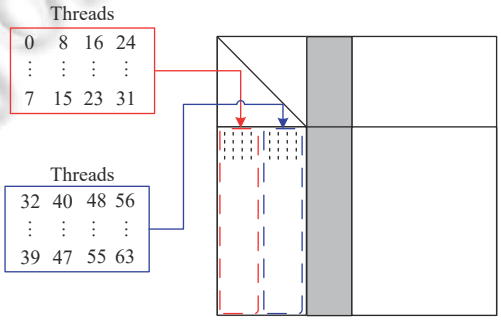

图 9 行交换推迟优化示意图

3.2 批量求逆算法在 GPU 上的有效实现

在 GPU 上实现 LU 分解之后的求逆算法与在 CPU 上实现一样, 也分为及时修正与延迟修正两种块算法, 但在 GPU 上实现时, 这两种方法对全局内存的数据读写次数不同. 因为影响批量小规模矩阵问题在 GPU 上实现性能的主要原因是对全局内存的数据读写时间, 由于延迟修正求逆块算法比及时修正求逆块算法具有更少的对全局内存的数据读写次数, 因此, 在 GPU 上的实现批量求逆算法时, 我们选取了延迟修正块算法.

算法 10–算法 13 给出了第 2.2 节求逆过程中第 (1) 步和第 (2) 步及时修正与延迟修正的块算法在 GPU 上的实现, 并通过算法 10–算法 13 分析说明了延迟修正求逆块算法对全局内存的数据读写次数少于及时修正求逆块算法对全局内存的数据读取量. 在条块内求解时, 由于矩阵 $\boldsymbol{W}$ 中的当前列条块中的数据需要多次读写, 并且每个线程对应的每行数据的求解、修正相互独立, 因此可以将其从全局内存读到线程私有的寄存器中, 加快数据的读写速度, 求解完成后再写回到全局内存即可. 对于使用到的矩阵 $\boldsymbol{U}$ 和矩阵 $\boldsymbol{L}$ 中子矩阵行/列条块, 由于对其进行了多次读取操作, 因此可以将其从全局内存取到共享内存中, 加快数据的读取速度. 在算法 10–算法 13 中, 下标“R”表示存储在寄存器中的变量, 下标“S”表示存储在共享内存中的变量.

算法 10 给出了求解方程组 $\boldsymbol{XU}=\boldsymbol{W}$ 及时修正块算法在 GPU 上的实现, 首先从全局内存中将当前列条块 $\begin{bmatrix}\boldsymbol{W}_{01}\\ \boldsymbol{W}_{11}\end{bmatrix}$ 读到寄存器 (第 3 行), 然后进行当前列条块内的计算 (第 9 行), 计算完成后再将列条块写回到全局内存 (第 12 行), 这时对全局内存的数据读写次数为 $2\times(i\times NB)\times NB$. 然后与 $\boldsymbol{U}_{12}$ 来修正列条块右边的部分 $\begin{bmatrix}\boldsymbol{W}_{02}\\ \boldsymbol{W}_{12}\end{bmatrix}$ (第 14 行), 此时, 对全局内存中矩阵 $\boldsymbol{W}$ 的数据读写次数为 $2\times(i\times NB)\times(n-i\times NB)$. 整个修正尾部矩阵的过程共对矩阵 $\boldsymbol{U}$ 的上三角部分进行了一次访问, 对全局内存中的矩阵 $\boldsymbol{U}$ 的数据读取次数约 $\frac{1}{2}n^2$, 则整个及时修正的求 $\boldsymbol{U}^{-1}$ 的块算法对全局内存的读写次数约为: $V_{\mathrm{T1}}=\frac{1}{3NB}n^3+\frac{3}{2}n^2+\frac{2NB}{3}n$.

算法 10. GPU 内及时修正的求 $\boldsymbol{XU}=\boldsymbol{W}$ 的块算法.

```
1  for i = 1 to bn
2    /*将当前列条块从全局内存读到寄存器*/
3    [ W_R,01 ] ← [ W_01 ];
     [ W_R,11 ]   [ W_11 ]
4    __syncthreads();
5    /*将U_11和U_12从全局内存读到共享内存*/
6    [U_S,11  U_S,12] ← [U_11  U_12];
7    __syncthreads();
8    /*求解当前列条块*/
9    [ W_R,01 ] = [ W_R,01 ] × U_S,11^-1;
     [ W_R,11 ]   [ W_R,11 ]
10   __syncthreads();
11   /*将当前列条块写回全局内存*/
12   [ W_01 ] ← [ W_R,01 ];
     [ W_11 ]   [ W_R,11 ]
13   /*修正剩余尾部矩阵*/
14   [ W_02 ] -= [ W_R,01 ] × U_S,12;
     [ W_12 ]   [ W_R,11 ]
15   __syncthreads();
16 endfor
```

算法 11 给出了求解方程组 $\boldsymbol{XU}=\boldsymbol{W}$ 延迟修正块算法在 GPU 上的实现, 首先对寄存器中的当前列条块进行初始化 (第 3 行), 这是由于 $\boldsymbol{W}$ 初始时为单位矩阵, 所以这里不需要从全局内存中读入数据到寄存器. 然后用当前列条块左边已经求解出的部分 $\boldsymbol{W}_{00}$ 与 $\boldsymbol{U}_{\mathrm{S},01}$ 来修正当前列条块 $\boldsymbol{W}_{\mathrm{R},01}$ (第 9 行), 此时, 对全局内存中矩阵 $\boldsymbol{W}$ 的数据读取量约为 $\frac{1}{2}\times(i-1)\times NB\times(i-1)\times NB$, 修正当前列条块的方法与 Left-looking LU 分解算法中修正列条块的方法类似, 所有线程同时从已经求解出的部分 $\boldsymbol{W}_{00}$ 中读取一列数据到寄存器, 然后与共享内存中的 $\boldsymbol{U}_{\mathrm{S},01}$ 修正存储在寄存器中的当前列条块的每列数据, 修正当前列条块的所有列数据之后, 所有线程再同时读取下一列数据到寄存器, 继续修正当前列条块的所有列. 修正完成后, 然后求解当前列条块 (第 12 行), 完成后写回到全局内存 (第 15 行), 此时数据写量为 $(i\times NB)\times NB$. 另外, 整个修正过程共对全局内存中的矩阵 $\boldsymbol{U}$ 进行了一次访问, 对全局内存中矩阵 $\boldsymbol{U}$ 的数据读取量约 $\frac{1}{2}n^2$. 则整个延迟修正的求 $\boldsymbol{U}^{-1}$ 的块算法对全局内存的读写量约为: $V_{\mathrm{D1}}=\frac{1}{6NB}n^3+\frac{3}{4}n^2+\frac{7NB}{12}n$.

算法 11. GPU 内延迟修正的求 $\boldsymbol{XU}=\boldsymbol{W}$ 的块算法.

```
1  for i = 1 to bn
2    /*初始化寄存器中的当前列条块*/
3    W_{R,01} = 0, W_{R,11} = I;
4    __syncthreads();
5    /*将 U_{01} 和 U_{11} 读到共享内存*/
6    [ U_{S,01} ] ← [ U_{01} ];
     [ U_{S,11} ]   [ U_{11} ]
7    __syncthreads();
8    /*修当前列条块*/
9    W_{R,01} -= W_{00} × U_{S,01};
10   __syncthreads();
11   /*求解当前列条块*/
12   [ W_{R,01} ] = [ W_{R,01} ] × U_{S,11}^{-1};
     [ W_{R,11} ]   [ W_{R,11} ]
13   __syncthreads();
14   /*将当前列条块写回全局内存*/
15   [ W_{01} ] ← [ W_{R,01} ];
     [ W_{11} ]   [ W_{R,11} ]
16   __syncthreads();
17 endfor
```

求逆过程的第 (2) 步求解方程组 $\boldsymbol{XL}=\boldsymbol{W}$ 的过程在 GPU 上的实现与第 (1) 步的在 GPU 上的实现类似, 也分为及时修正与延迟修正 2 种块算法, 算法 12 和算法 13 分别给出了这 2 种求解方程组 $\boldsymbol{XL}=\boldsymbol{W}$ 的块算法在 GPU 上的实现. 其中, 及时修正算法对全局内存的读写量约为: $\boldsymbol{V}_{\mathrm{T2}}=\frac{1}{NB}n^3+\frac{3}{2}n^2$. 延迟修正算法对全局内存的读写量为: $V_{\mathrm{D2}}=\frac{1}{2NB}n^3+2n^2$.

算法 12. GPU 内及时修正的求 $\boldsymbol{XL}=\boldsymbol{W}$ 的块算法.

```
1  for i = 1 to bn
2    /*将当前列条块从全局内存取到寄存器*/
3    W_{R,1} ← W_1;
4    __syncthreads();
```

5 /*将 $\boldsymbol{L}_{10}$ 和 $\boldsymbol{L}_{11}$ 从全局内存读到共享内存*/
6 $\left[\boldsymbol{L}_{\mathrm{S},10}\ \ \boldsymbol{L}_{\mathrm{S},11}\right] \leftarrow \left[\boldsymbol{L}_{10}\ \ \boldsymbol{L}_{11}\right]$;
7 __syncthreads();
8 /*求解列条块*/
9 $\boldsymbol{W}_{\mathrm{R},1} = \boldsymbol{W}_{\mathrm{R},1} \times \boldsymbol{L}_{\mathrm{S},11}^{-1}$;
10 __syncthreads();
11 /*将列条块的求解结果写回到全局内存*/
12 $\boldsymbol{W}_1 \leftarrow \boldsymbol{W}_{\mathrm{R},1}$;
13 /*修正剩余尾部矩阵*/
14 $\boldsymbol{W}_0 -= \boldsymbol{W}_{\mathrm{R},1} \times \boldsymbol{L}_{\mathrm{S},10}$;
15 endfor

算法 13. GPU 内延迟修正的求 $\boldsymbol{XL} = \boldsymbol{W}$ 的块算法.

1 for $i = 1$ to bn
2 /*将当前列条块从全局内存取到寄存器*/
3 $\boldsymbol{W}_{\mathrm{R},1} \leftarrow \boldsymbol{W}_1$;
4 __syncthreads();
5 /*将 $\boldsymbol{L}_{11}$ 和 $\boldsymbol{L}_{21}$ 读到共享内存*/
6 $\begin{bmatrix} \boldsymbol{L}_{\mathrm{S},11} \\ \boldsymbol{L}_{\mathrm{S},21} \end{bmatrix} \leftarrow \begin{bmatrix} \boldsymbol{L}_{11} \\ \boldsymbol{L}_{21} \end{bmatrix}$;
7 __syncthreads();
8 /*修正列条块*/
9 $\boldsymbol{W}_{\mathrm{R},1} -= \boldsymbol{W}_2 \times \boldsymbol{L}_{\mathrm{S},21}$;
10 __syncthreads();
11 /*求解列条块*/
12 $\boldsymbol{W}_{\mathrm{R},1} = \boldsymbol{W}_{\mathrm{R},1} \times \boldsymbol{L}_{\mathrm{S},11}^{-1}$;
13 __syncthreads();
14 /*将列条块的求解结果写回到全局内存*/
15 $\boldsymbol{W}_1 \leftarrow \boldsymbol{W}_{\mathrm{R},1}$;
16 __syncthreads();
17 endfor

显然, 求逆的延迟修正块算法对全局内存的数据读写次数小于及时修正块算法: $V_{\mathrm{D1}} + V_{\mathrm{D2}} < V_{\mathrm{T1}} + V_{\mathrm{T2}}$. 我们对采用这 2 种修正方法的批量求逆算法在 GPU 上进行了实现, 实验结果详见第 4.4 节.

在 GPU 上实现的批量算法中, 一个 Block 对应一个矩阵, Block 中一个线程对应矩阵的一行. 若上述求逆步骤在一个 Kernel 中完成, 为了保证算法的正确性, 则申请的 Block 中线程的个数只能是 n, 共享内存的大小只能为最大的 $n \times NB$. 我们称这种一次性分配的方法为静态资源分配法. 如算法 11 所描述的 GPU 内延迟修正的求 $\boldsymbol{U}^{-1}$ 块算法中, 所需的线程数目和共享内存的大小与当前列条块的大小有关, 而列条块的大小是动态变化的, 每一次列条块的求解需要的线程数目为 $i \times NB$, 需要的共享内存的大小为 $(i \times NB) \times NB$. 若采用静态资源分配法, 显然会造成线程的闲置和共享内存资源的浪费. 因此我们设计了运行时按需申请线程和共享内存资源的动态资源分配方法, 每一个列条块求解启动一次 Kernel, 根据列条块的大小申请线程和共享内存, 这种方法避免了 GPU 资源的浪费, 进一步提升批量求逆算法的性能. 实验结果详见第 4.4 节.

3.3 减少访存数据量的列交换优化方法

以上过程结束后, 根据 $\boldsymbol{P}$ 交换相应的列即可得到 $\boldsymbol{A}^{-1}$. 在 GPU 的实现中, 若要进行两列间交换, 首先需要从全局内存中将这两列读出, 然后将交换后两列写回到全局内存中对应的列, 则列交换需要访问全局内存的数据量为 $4n^2$. 为了进一步降低全局内存的访问量, 我们设计了减少访存数据量的列交换优化方法, 该方法将上述求解的结果直接存储到原始矩阵 $\boldsymbol{A}$, 而不是存储在 $\boldsymbol{A}^{-1}$ 中. 则可根据 $\boldsymbol{P}$ 直接将 $\boldsymbol{A}$ 中对应的列从全局内存中读出, 然后写入全局内存中的逆矩阵 $\boldsymbol{A}^{-1}$ 中去, 这样避免了显式的矩阵列交换, 对全局内存的访问数据量减少至 $2n^2$.

4 实验分析

为了测试算法的性能和准确性, 我们在 TITAN V GPU 上实现并测试了本文描述的批量 LU 分解与批量求逆算法, TITAN V GPU 的显存容量为 12 GB, 峰值双精度浮点性能 7.5 TFLOPS. 我们对 10 000 个规模在 33–190 之间的随机矩阵进行了测试. 并在同一平台上用 CUBLAS 库和 MAGAMA 库中的算法进行了性能测试和对比分析. 测试中使用的 CUBLAS 版本是 9.1.85, MAGMA 版本是 2.5.1.

4.1 块大小对算法实现性能的影响

(1) 块大小对批量 LU 分解块算法实现性能的影响

批量 LU 分解块算法中块的大小 *NB* 将会影响其在 GPU 上的实现性能. 这是因为, 块的大小 *NB* 决定着对全局内存的访问数据次数、每个列条块使用共享内存的大小以及条块外 Warp 分组行交换中线程的组织形式等. 因此, 在使用时需要衡量块的大小, 使程序达到最好的性能. 图 10 描述的是当矩阵类型为单精度复数、 块大小 *NB*=8, 10, 14, 18 时批量 LU 分解块算法的浮点计算性能曲线示意图. 从图 10 中可以看出, 开始时随着 *NB* 的增大, 批量 LU 分解块算法的性能越来越好, 当 *NB*=14 时达到的性能相对较好, 当 *NB* 再增加时性能反而下降, 这是因为随着 *NB* 的增大, 程序所需的共享内存等 GPU 资源越多, 会造成资源的竞争, 程序的性能反而降低了.

(2) 块大小对批量求逆块算法实现性能的影响

不同的块的大小 *NB* 也会影响批量求逆算法的实现性能. 这是因为, 在批量求逆块算法中块的大小 *NB* 决定着对全局内存的访问数据量、每个线程使用寄存器的数量, 以及每个列条块使用共享内存的大小. 通常来说使用越多的寄存器和共享内存越能加快数据的读写速度, 但是 GPU 内的资源是有限的, 若使用过多的寄存器和共享内存会造成资源的竞争, 反而会降低算法实现的性能. 因此, 在使用时需要衡量块的大小, 使算法实现达到最好的性能. 图 11 描述的是当矩阵类型为单精度复数、块大小 *NB*=8, 14, 20, 24, 28 时批量求逆块算法的浮点计算性能曲线示意图. 从图 11 中可以看出, 同批量 LU 分解块算法类似, 批量求逆块算法的性能开始时随着 *NB* 的增大性能不断提升, 当 *NB*=16–20 时取得的性能相对较好, 当 *NB* 再增加时性能反而下降了.

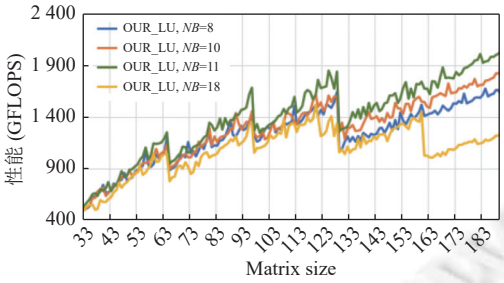

图 10 *NB* 取不同值时批量 LU 分解算法性能曲线

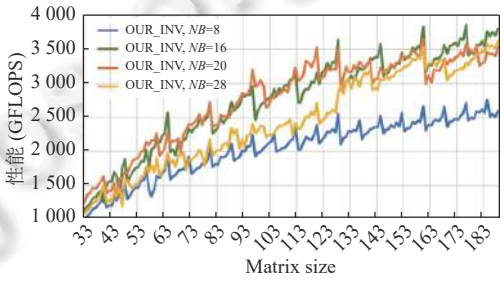

图 11 *NB* 取不同值时批量求逆算法性能曲线

4.2 本文批量算法与 CUBLAS 和 MAGMA 中算法实现的浮点计算性能比较

(1) 批量 LU 分解块算法与 CUBLAS 和 MAGMA 中实现的浮点计算性能比较

图 12 给出了矩阵类型为单精度复数 (float2)、双精度复数 (double2)、单精度实数 (float) 和双精度实数 (double) 时, 本文实现的批量 LU 算法 (OUR_LU) 与 CUBLAS (CUBLAS_LU) 和 MAGMA 中实现 (MAGMA_LU)

的批量 LU 算法的浮点计算性能曲线示意图. 从图 12 中可以看出, 不同矩阵类型的测试结果 MAGMA_LU 的性能高于 CUBLAS_LU 的性能, 而 OUR_LU 的性能又高于 MAGMA_LU 的性能. 此外, CUBLAS_LU 的浮点计算性能随着矩阵规模的增加比较平稳. 而 OUR_LU 与 MAGMA_LU 的浮点计算性能随着矩阵规模的增大不断提升, 不同矩阵类型的 OUR_LU 的浮点计算性能分别可达约 2 TFLOPS (float2)、1.2 TFLOPS (double2)、1 TFLOPS (float)、0.67 TFLOPS (double). 图 13 描述了本文 OUR_LU 算法与 CUBLAS_LU 和 MAGMA_LU 的不同矩阵类型的加速比曲线示意图. 从图 13 中可以看出, 开始时随着矩阵规模的增大, OUR_LU 与 CUBLAS_LU 的加速比越大, 不同矩阵类型分别最高达到约 9× (float2)、8× (double2)、12× (float)、13× (double), 当矩阵规模增大到一定程度后, 加速比趋于平稳, 分别约在 7×–8× (float2)、5×–6× (double2)、10×–12× (float)、9×–12× (double) 之间. OUR_LU 与 MAGMA_LU 的加速比相对平稳, 不同矩阵类型分别约在 1.2×–2.5× (float2)、1.2×–3.2× (double2)、1.1×–3× (float)、1.1×–2.7× (double) 之间.

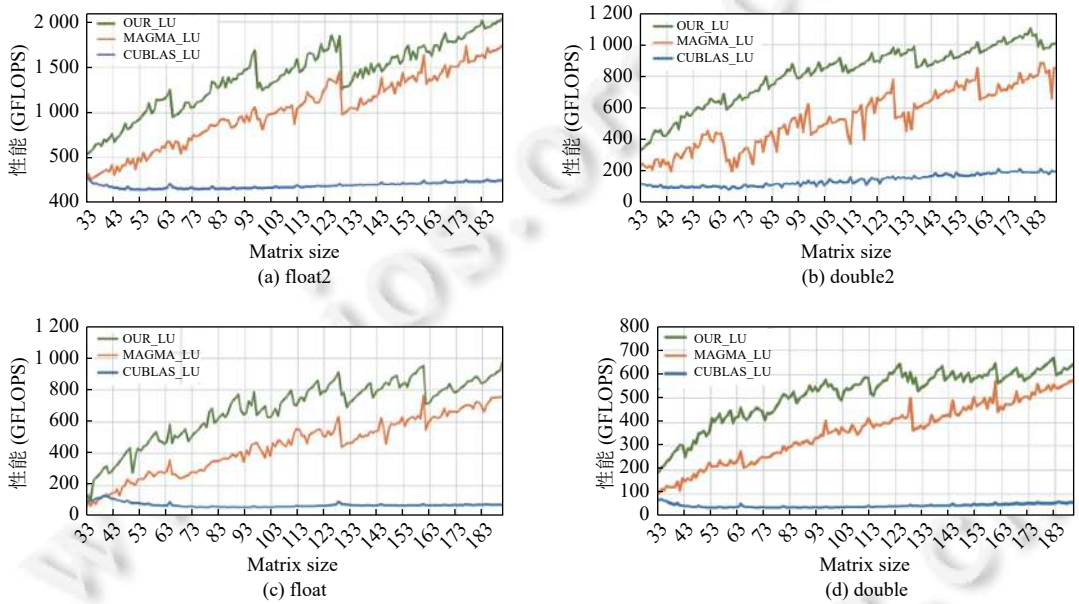

图 12 不同数据类型的批量 LU 分解算法的性能曲线

(2) 批量求逆块算法与 CUBLAS 和 MAGMA 中实现的浮点计算性能比较

图 14 给出了矩阵类型为单精度复数 (float2)、双精度复数 (double2)、单精度实数 (float) 和双精度实数 (double) 时本文实现的批量求逆算法 (OUR_INV) 与 CUBLAS (CUBLAS_INV) 和 MAGMA 中实现 (MAGMA_INV) 的批量求逆算法的浮点计算性能曲线示意图. 从图 14 中可以看出, 当矩阵规模较小时, 不同矩阵类型的测试结果中, CUBLAS_INV 的性能要高于 MAGMA_INV 的性能, 随着矩阵规模的增大, MAGMA_INV 的性能反超 CUBLAS_INV 的性能. 本文的 OUR_INV 的浮点计算性能一直高于 CUBLAS_INV 和 MAGMA_INV 的性能. 此外, CUBLAS_INV 的性能比较平稳, OUR_INV 与 MAGMA_INV 的浮点计算性能随着矩阵规模的增大不断提升. 不同矩阵类型的 OUR_INV 的浮点计算性能大约分别可达到 4 TFLOPS (float2)、2 TFLOPS (double2)、2.2 TFLOPS (float)、1.2 TFLOPS (double). 后文图 15 显示了 OUR_INV 与 CUBLAS_INV 和 MAGMA_INV 的加速比曲线示意图. 从图 15 中可以看出, 开始时随着矩阵规模越大, OUR_INV 与 CUBLAS_INV 的加速比越大, 最高分别达到约 5× (float2)、4× (double2)、7× (float)、7× (double). 当矩阵规模达到一定程度后, 加速比趋于平稳, 分别约在 4×–5× (float2)、3×–4× (double2)、5×–7× (float)、5×–6× (double) 之间. 而 OUR_INV 与 MAGMA_INV 的加速比, 当矩阵规模较小时比较大, 随着矩阵规模的增加趋于稳定, 不同矩阵类型分别约在 2×–3× (float2)、2×–3× (double2)、2.8×–3.4× (float)、1.6×–2× (double) 之间.

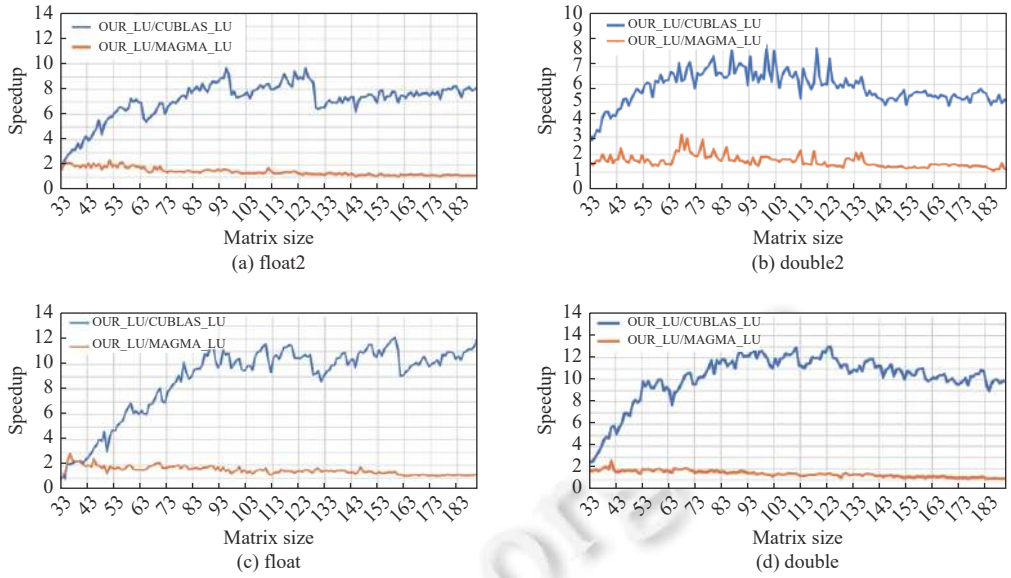

图 13　不同数据类型的批量 LU 分解算法的加速比曲线

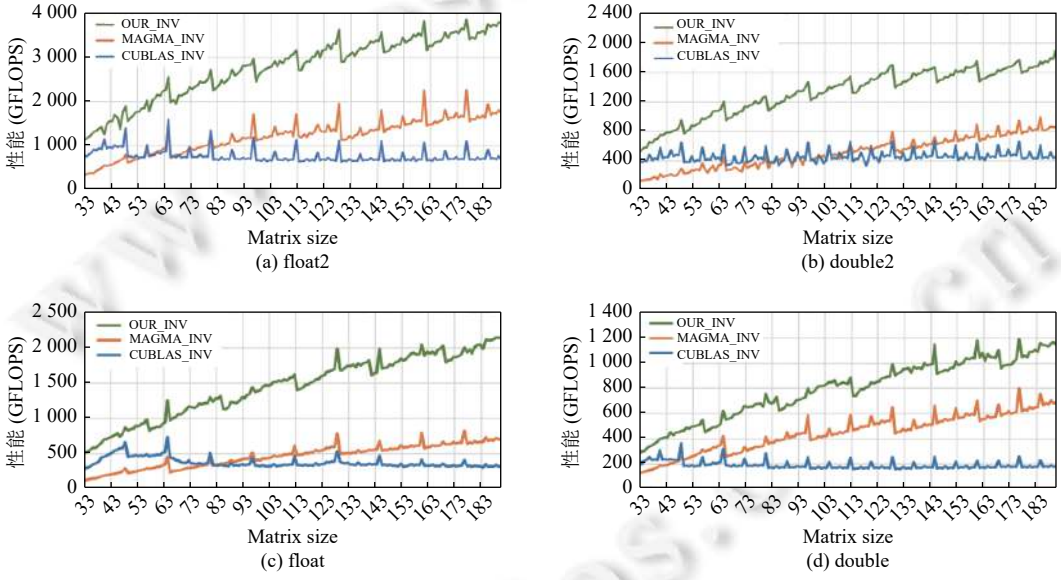

图 14　不同数据类型的批量求逆算法的性能曲线

4.3　批量 LU 分解算法优化分析

图 16 显示了当矩阵类型为单精度复数、块大小 *NB*=14 时, 批量 LU 分解块算法中列条块外的两侧进行行交换采用 Warp 分组行交换 (OUR_LU_v2) 以及行推迟优化技术 (OUR_LU_v3) 与优化前 (OUR_LU_v1) 的浮点计算性能曲线示意图. 从图 16 中可以看出, 对列条块外的两侧进行行交换进行优化后, OUR_LU_v3 的浮点计算性能相对于 OUR_LU_v1 的性能得到 2–5 倍的提升, 这是因为列条块外两侧的行交换在批量 LU 分解中占据了 60% 以上的时间[13], 因此, 对列条块外两侧的行交换进行优化, 将会明显提升批量 LU 算法的性能. 从图 16 还可以看出随着矩阵规模的增大提升幅度同样在增大, 这是因为随着矩阵规模的增大, 列条块外部分行交换对全局内存访问的数据量越多, 对列条块外在全局内存部分的行交换进行优化的效果越明显.

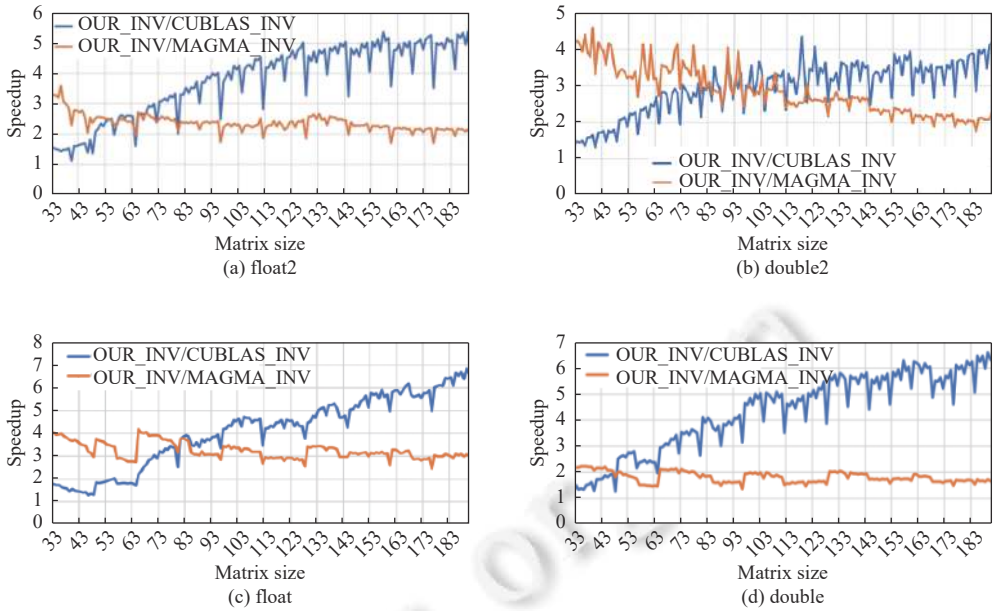

图 15　不同数据类型的批量求逆算法的加速比曲线

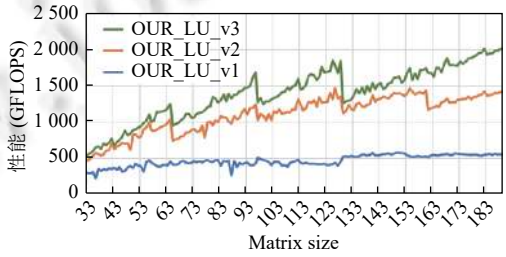

图 16　批量 LU 分解算法不同版本性能曲线

4.4　批量求逆算法优化分析

(1) 及时修正与延迟修正两种算法实现的浮点计算性能比较

图 17 显示了当矩阵类型为单精度复数、块的大小 *NB*=8, 16 时, 采用及时修正 (timely) 与延迟修正 (delay) 两种批量矩阵求逆算法方法在 GPU 内实现的浮点计算性能曲线示意图. 从图 17 中可以看出, 延迟修正算法的浮点计算性能优于及时修正算法的性能. 这是因为在 GPU 内实现及时修正的算法对于全局内存数据的读写数量多于延迟修正的算法. 并且 *NB* 越小, 提升越明显, 这是因为 *NB* 越小对全局内存的数据读写次数越多, 因此延迟修正算法的性能相较于及时修正算法的性能提升越明显. 这里及时修正与延迟修正算法使用的都是采用一次性分配资源的静态法.

(2) 静态资源分配法与动态资源分配方法实现的浮点计算性能比较

图 18 显示的是当矩阵类型为单精度复数、块的大小 *NB*=8, 16 时, 延迟修正算法采用一次性分配资源的静态资源分配方法 (static) 与运行时按需分配资源的动态资源分配方法 (dynamic) 的浮点计算性能曲线示意图. 从图 18 中可以看出, *NB* 取不同的值时动态资源分配方法的浮点计算性能相较于静态资源分配方法都有所提升. 并且随着矩阵规模的增大, 提升幅度也在不断增大, 最大约 30%. 这是因为随着矩阵规模的增大, 采用静态资源分配方法时线程闲置的情况和共享内存资源的浪费更加严重, 这时动态资源分配方法的优势就更加地显示出来, 因此提升的幅度越大.

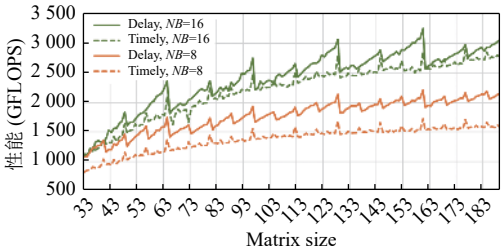

图 17 及时修正与延迟修正算法的性能曲线

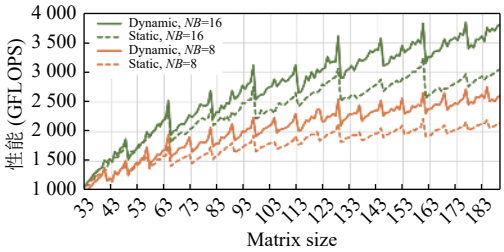

图 18 延迟修正算法的静态法和动态法的性能曲线

5 总 结

本文给出了批量矩阵的 LU 分解和求逆算法及其在 GPU 上实现及优化方法. 针对批量 LU 分解问题, 分析了 Left-looking 和 Right-looking 等常用 LU 分解块算法在 GPU 上实现时对全局内存的数据读写次数, 由于 Left-looking 算法具有更少访存次数, 能更好地适应 GPU 架构下批量小矩阵的 LU 分解, 因此我们选择了具有较少访存的 Left-looking 块算法. 在 LU 分解的选主元过程, 采用了适合 GPU 架构的并行二叉树搜索算法. 此外, 为了降低选主元引起的行交换过程对算法性能的影响, 本文提出了 Warp 分组行交换和行交换延迟两个优化技术, 大大提升了算法的性能. 针对 LU 分解后的批量求逆问题, 分析了矩阵求逆过程中修正方法, 为了减少修正过程对全局内存的数据读写次数, 在批量求逆的 GPU 实现中采用了延迟修正的矩阵求逆块算法. 同时, 为了加快对全局内存的数据读写速度, 采用了更多利用寄存器和共享内存的优化方法和减少访存数据量的列交换优化方法. 另外, 为了避免线程的闲置和共享内存等 GPU 资源浪费, 提出了运行时按需申请 GPU 资源的动态资源分配方法. 最终, 在 TITAN V GPU 上, 我们对 10000 个规模在 33–190 之间的随机矩阵进行了测试. 当矩阵类型为单精度复数、双精度复数、单精度实数和双精度实数时, 我们实现的批量 LU 分解算法的浮点计算性能分别可达到约 2 TFLOPS、1.2 TFLOPS、1 TFLOPS、0.67 TFLOPS, 与 CUBLAS 中的实现相比加速比分别最高达到了约 9×、8×、12×、13×, 与 MAGMA 中的实现相比加速比分别达到了约 1.2×–2.5×、1.2×–3.2×、1.1×–3×、1.1×–2.7×. 不同矩阵类型的批量求逆算法的浮点计算性能分别可达到约 4 TFLOPS、2 TFLOPS、2.2 TFLOPS、1.2 TFLOPS, 与 CUBLAS 中的实现相比加速比分别最高达到了约 5×、4×、7×、7×, 与 MAGMA 中的实现相比加速比分别达到了约 2×–3×、2×–3×、2.8×–3.4×、1.6×–2×. 此外, 不同的条块的大小将会影响程序的性能, 在使用时需要权衡. 本文批量 LU 分解算法与批量求逆算法在 GPU 上的实现与优化方法同样适用于 Cholesky 分解和 QR 分解等常见的矩阵分解算法在 GPU 上的批量实现.

References:

[1] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cuDNN: Efficient primitives for deep learning. arXiv:1410.0759, 2014.

[2] Papalexakis EE, Faloutsos C, Sidiropoulos ND. Tensors for data mining and data fusion: Models, applications, and scalable algorithms. ACM Trans. on Intelligent Systems and Technology, 2017, 8(2): 16. [doi: 10.1145/2915921]

[3] Molero JM, Garzón EM, García I, Quintana-Ortí ES, Plaza A. Efficient implementation of hyperspectral anomaly detection techniques on GPUs and multicore processors. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 2014, 7(6): 2256–2266. [doi: 10.1109/JSTARS.2014.2328614]

[4] Anderson MJ, Sheffield D, Keutzer K. A predictive model for solving small linear algebra problems in GPU registers. In: Proc. of the 26th IEEE Int'l Parallel and Distributed Processing Symp. Shanghai: IEEE, 2012. 2–13. [doi: 10.1109/IPDPS.2012.11]

[5] Molero JM, Garzón EM, García I, Quintana-Ortí ES, Plaza A. A batched Cholesky solver for local RX anomaly detection on GPUs. In: Proc. of the 13th Int'l Conf. on Computational and Mathematical Methods in Science and Engineering. Almería: CMMSE, 2013. 1037–1797.

[6] Yeralan SN, Davis TA, Sid-Lakhdar WM, Ranka S. Algorithm 980: Sparse QR factorization on the GPU. ACM Trans. on Mathematical Software, 2018, 44(2): 17. [doi: 10.1145/3065870]

[7] Messer OEB, Harris JA, Parete-Koon S, Chertkow MA. Multicore and accelerator development for a leadership-class stellar astrophysics code. In: Proc. of the 11th Int'l Workshop on Applied Parallel Computing. Helsinki: Springer, 2013. 92–106. [doi: 10.1007/978-3-642-36803-5_6]

[8] Auer AA, Baumgartner G, Bernholdt DE, Bibireata A, Choppella V, Cociorva D, Gao XY, Harrison R, Krishnamoorthy S, Krishnan S, Lam CC, Lu QD, Nooijen M, Pitzer R, Ramanujam J, Sadayappan P, Sibiryakov A. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. Molecular Physics, 2006, 104(2): 211–228. [doi: 10.1080/00268970500275780]

[9] Tomov S, Nath R, Ltaief H, Dongarra J. Dense linear algebra solvers for multicore with GPU accelerators. In: Proc. of the 2010 IEEE Int'l Symp. on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). Atlanta: IEEE, 2010. 1–8. [doi: 10.1109/IPDPSW.2010.5470941]

[10] Agullo E, Augonnet C, Dongarra J, Ltaief H, Namyst R, Thibault S, Tomov S. A hybridization methodology for high-performance linear algebra software for GPUs. In: Hwu WMW, ed. GPU Computing Gems Jade Edition. Morgan Kaufmann, 2012. 473–484. [doi: 10.1016/B978-0-12-385963-1.00034-4]

[11] Dongarra J, Haidar A, Kurzak J, Luszczek P, Tomov S, YarKhan A. Model-driven one-sided factorizations on multicore accelerated systems. Supercomputing Frontiers and Innovations, 2014, 1(1): 85–115. [doi: 10.14529/jsfi140105]

[12] Haidar A, Dong TX, Luszczek P, Tomov S, Dongarra J. Batched matrix computations on hardware accelerators based on GPUs. The Int'l Journal of High Performance Computing Applications, 2015, 29(2): 193–208. [doi: 10.1177/1094342014567546]

[13] Dong TX, Haidar A, Luszczek P, Harris JA, Tomov S, Dongarra J. LU factorization of small matrices: Accelerating batched DGETRF on the GPU. In: Proc. of the 2014 IEEE Int'l Conf. on High Performance Computing and Communications, the 6th IEEE Int'l Symp. on Cyberspace Safety and Security, the 11th IEEE Int'l Conf. on Embedded Software and Syst (HPCC, CSS, ICESS). Paris: IEEE, 2015. 157–160. [doi: 10.1109/HPCC.2014.30]

[14] Villa O, Fatica M, Gawande N, Tumeo A. Power/performance trade-offs of small batched LU based solvers on GPUs. In: Proc. of the 19th European Conf. on Parallel Processing. Aachen: Springer, 2013. 813–825. [doi: 10.1007/978-3-642-40047-6_81]

[15] Villa O, Gawande N, Tumeo A. Accelerating subsurface transport simulation on heterogeneous clusters. In: Proc. of the 2013 IEEE Int'l Conf. on Cluster Computing (CLUSTER). Indianapolis: IEEE, 2013. 1–8. [doi: 10.1109/CLUSTER.2013.6702656]

[16] Dong TX, Haidar A, Tomov S, Dongarra J. A fast batched Cholesky factorization on a GPU. In: Proc. of the 43rd Int'l Conf. on Parallel Processing. Minneapolis: IEEE, 2014. 432–440. [doi: 10.1109/ICPP.2014.52]

[17] Abdelfattah A, Haidar A, Tomov S, Dongarra J. Factorization and inversion of a million matrices using GPUs: Challenges and countermeasures. Procedia Computer Science, 2017, 108: 606–615. [doi: 10.1016/j.procs.2017.05.250]

[18] Abdelfattah A, Tomov S, Dongarra J. Fast batched matrix multiplication for small sizes using half-precision arithmetic on GPUs. In: Proc. of the 2019 IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). Rio de Janeiro: IEEE, 2019. 111–122. [doi: 10.1109/IPDPS.2019.00022]

[19] Masliah I, Abdelfattah A, Haidar A, Tomov S, Baboulin M, Falcou J, Dongarra J. Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices. Parallel Computing, 2019, 81: 1–21. [doi: 10.1016/j.parco.2018.10.003]

[20] Abdelfattah A, Tomov S, Dongarra J. Matrix multiplication on batches of small matrices in half and half-complex precisions. Journal of Parallel and Distributed Computing, 2020, 145: 188–201. [doi: 10.1016/j.jpdc.2020.07.001]

[21] Abdelfattah A, Costa T, Dongarra J, Gates M, Haidar A, Hammarling S, Higham NJ, Kurzak J, Luszczek P, Tomov S, Zounon M. A set of batched basic linear algebra subprograms and LAPACK routines. ACM Trans. on Mathematical Software, 2021, 47(3): 21. [doi: 10.1145/3431921]

[22] Anzt H, Dongarra J, Flegar G, Quintana-Ortí ES. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In: Proc. of the 8th Int'l Workshop on Programming Models and Applications for Multicores and Manycores. Austin: ACM, 2017. 1–10. [doi: 10.1145/3026937.3026940]

[23] Dongarra JJ, Hammarling S, Walker DW. Key concepts for parallel out-of-core LU factorization. Computers & Mathematics with Applications, 1998, 35(7): 13–31. [doi: 10.1016/S0898-1221(98)00029-7]

[24] Zhou G, Bo R, Chien L, Zhang X, Shi F, Xu CL, Feng YJ. GPU-based batch LU-factorization solver for concurrent analysis of massive power flows. IEEE Trans. on Power Systems, 2017, 32(6): 4975–4977. [doi: 10.1109/TPWRS.2017.2662322]

[25] Wang XF, Ziavras SG. Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines. Concurrency and Computation: Practice and Experience, 2004, 16(4): 319–343. [doi: 10.1002/cpe.748]

[26] NVIDIA Corporation. NVIDIA CUBLAS Library. 2022. https://docs.nvidia.com/cuda/cublas/index.html

[27] Tomov S, Nath R, Du P, Dongarra J. MAGMA users' guide. 2009. https://cseweb.ucsd.edu/~rknath/magma-v02.pdf

[28] Chen Y, Lin JX, Lv T. Implementation of LU decomposition and Laplace algorithms on GPU. Journal of Computer Applications, 2011,

31(3): 851–855 (in Chinese with English abstract). [doi: 10.3724/SP.J.1087.2011.00851]

[29] Li MY, Wang Y, Ma G, Zhou G. A GPU-accelerated algorithm of batch-LU decomposition. Electric Power Engineering Technology, 2019, 38(2): 57–63 (in Chinese with English abstract). [doi: 10.3969/j.issn.1009-0665.2019.02.009]

[30] Haidar A, Abdelfattah A, Zounon M, Tomov S, Dongarra J. A guide for achieving high performance with very small matrices on GPU: A case study of batched LU and Cholesky factorizations. IEEE Trans. on Parallel and Distributed Systems, 2018, 29(5): 973–984. [doi: 10.1109/TPDS.2017.2783929]

附中文参考文献:

[28] 陈颖, 林锦贤, 吕暾. LU分解和Laplace算法在GPU上的实现. 计算机应用, 2011, 31(3): 851–855. [doi: 10.3724/SP.J.1087.2011.00851]

[29] 李梦月, 王颖, 马刚, 周赣. 一种基于图形处理器加速的批量LU分解算法. 电力工程技术, 2019, 38(2): 57–63. [doi: 10.3969/j.issn.1009-0665.2019.02.009]

刘世芳(1994－), 女, 博士, 主要研究领域为数值并行算法库的实现, 异构并行机下并行算法库的优化.

于天禹(1985－) 男, 博士, 主要研究领域为数值并行算法, 求解器.

赵永华(1966－), 男, 博士, 研究员, 博士生导师, 主要研究领域为并行算法与软件, 并行编程模型, 谱聚类数据分析.

张馨尹(1994－), 女, 博士, 主要研究领域为高性能计算, 张量计算, 高光谱图像处理.

黄荣锋(1990－), 男, 博士, 主要研究领域为基础数学库在异构平台上的高效实现.