软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(11):4941−4951 [doi: 10.13328/j.cnki.jos.006720]

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

面向 GPU 平台的并行结构化稀疏三角方程组求解器*

陈道琨[1,2], 杨 超[3], 刘芳芳[1,2], 马文静[1,2]

[1](中国科学院 软件研究所 并行软件与计算科学实验室, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](北京大学 数学科学学院, 北京 100871)

通信作者: 杨超, E-mail: chao_yang@pku.edu.cn

摘 要: 稀疏三角线性方程组求解 (SpTRSV) 是预条件子部分的重要操作, 其中结构化 SpTRSV 问题, 在以迭代方法求解偏微分方程组的科学计算程序中, 是一种较为常见的问题类型, 而且通常是科学计算程序的需要解决的一个性能瓶颈. 针对 GPU 平台, 目前以 CUSPARSE 为代表的商用 GPU 数学库, 采用分层调度 (level-scheduling) 方法并行化 SpTRSV 操作. 该方法不仅预处理耗时较长, 而且在处理结构化 SpTRSV 问题时会出现较为严重 GPU 线程闲置问题. 针对结构化 SpTRSV 问题, 提出一种面向结构化 SpTRSV 问题的并行算法. 该算法利用结构化 SpTRSV 问题的特殊非零元分布规律进行任务划分, 避免对输入问题的非零元结构进行预处理分析. 并对现有分层调度方法的逐元素处理策略进行改进, 在有效缓解 GPU 线程闲置问题的基础上, 还隐藏了部分矩阵非零元素的访存延迟. 还根据算法的任务划分特点, 采用状态变量压缩技术, 显著提高算法状态变量操作的缓存命中率. 在此基础上, 还结合谓词执行等 GPU 硬件特性, 对算法实现进行全面的优化. 所提算法在 NVIDIA V100 GPU 上的实测性能, 相比 CUSPARSE 平均有 2.71 倍的加速效果, 有效访存带宽最高可达 225.2 GB/s. 改进后的逐元素处理策略, 配合针对 GPU 硬件的一系列调优手段, 优化效果显著, 将算法的有效访存带宽提高了约 1.15 倍.

关键词: 稀疏三角线性方程组求解 (SpTRSV); 模板计算; 结构化网格; GPU; 异构并行算法

中图法分类号: TP30

中文引用格式: 陈道琨, 杨超, 刘芳芳, 马文静. 面向GPU平台的并行结构化稀疏三角方程组求解器. 软件学报, 2023, 34(11): 4941–4951. http://www.jos.org.cn/1000-9825/6720.htm

英文引用格式: Chen DK, Yang C, Liu FF, MA WJ. Parallel Structured Sparse Triangular Solver for GPU Platform. Ruan Jian Xue Bao/Journal of Software, 2023, 34(11): 4941–4951 (in Chinese). http://www.jos.org.cn/1000-9825/6720.htm

Parallel Structured Sparse Triangular Solver for GPU Platform

CHEN Dao-Kun[1,2], YANG Chao[3], LIU Fang-Fang[1,2], MA Wen-Jing[1,2]

[1](Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

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

[3](School of Mathematical Sciences, Peking University, Beijing 100871, China)

Abstract: Sparse triangular solver (SpTRSV) is a vital operation in preconditioners. In particular, in scientific computing program that solves partial differential equation systems iteratively, structured SpTRSV is a common type of issue and often a performance bottleneck that needs to be addressed by the scientific computing program. The commercial mathematical libraries tailored to the graphics processing unit (GPU) platform, represented by CUSPARSE, parallelize SpTRSV operations by level-scheduling methods. However, this method is weakened by time-consuming preprocessing and serious GPU thread idle when it is employed to deal with structured SpTRSV issues. This study proposes a parallel algorithm tailored to structured SpTRSV issues. The proposed algorithm leverages the special non-zero element

* 基金项目: 国家重点研发计划高性能计算重点专项 (2020YFB0204601)
收稿时间: 2021-08-13; 修改时间: 2022-03-08; 采用时间: 2022-04-20; jos 在线出版时间: 2023-04-27
CNKI 网络首发时间: 2023-04-28

distribution pattern of structured SpTRSV issues during task allocation to skip the preprocessing and analysis of the non-zero element structure of the input issue. Furthermore, the element-wise operation strategy used in the existing level-scheduling methods is modified. As a result, the problem of GPU thread idle is effectively alleviated, and the memory access latency of some non-zero elements in the matrix is concealed. This study also adopts a state variable compression technique according to the task allocation characteristics of the proposed algorithm, significantly improving the cache hit rate of the algorithm in state variable operations. Additionally, several hardware features of the GPU, including predicated execution, are investigated to comprehensively optimize algorithm implementation. The proposed algorithm is tested on NVIDIA V100 GPU, achieving an average 2.71× acceleration over CUSPARSE and a peak effective memory-access bandwidth of 225.2 GB/s. The modified element-wise operation strategy, combined with a series of other optimization measures for GPU hardware, attains a prominent optimization effect by yielding a nearly 115% increase in the effective memory-access bandwidth of the proposed algorithm.

Key words: sparse triangular solver (SpTRSV); stencil computation; structured grid; GPU; heterogeneous parallel computing

稀疏三角线性方程组求解 (SpTRSV) 是迭代方法预条件子步骤的重要组成部分[1], 其中结构化 SpTRSV 问题[2]是一类极具代表性的问题, 在计算电磁学、计算气象学、计算流体动力学等领域[2−7]的应用十分广泛. 结构化 SpTRSV 问题耗时较多, 往往是迭代方法预条件子步骤的主要性能瓶颈[8]. 因此针对结构化 SpTRSV 问题, 设计高性能并行算法十分必要.

以 Summit[9]及 Sierra [10]等为代表的超级计算机平台, 采用 GPU 作为异构加速卡部件加速迭代方法的绝大部分运算操作. 针对 GPU 平台, 目前 AMD 的 ROCSPARSE[11]以及 NVIDIA 的 CUSPARSE[12]等商用高性能数学库, 以分层调度 (level scheduling) 方法作为并行求解 SpTRSV 问题的主要手段. 该方法求解 SpTRSV 问题之前需对输入问题的非零元分布特点进行细致的预处理分析. 然而分层调度方法预处理步骤耗时较多, 导致分层调度方法的整体加速效果出现明显下降[13−15]. 对此, 部分研究工作[13−15]采用简易直接的任务划分方式来避免预处理步骤引起的开销, 但由于缺少数据依赖信息的支撑, 此类方法对计算任务的调度规划方面存在一定不足, 在性能上需要做出妥协.

另外, 现有分层调度方法多采用基于逐元素处理的非零元分配策略[16,17]. 但结构化 SpTRSV 问题由于非零元数目稀少, 使用逐元素处理策略将导致大量 GPU 线程资源处于闲置状态, 进一步影响分层调度方法的加速效果. 由于结构化 SpTRSV 问题的非零元分布与偏微分方程组问题的离散化方法之间关系紧密, 此类问题通常呈现出较为规律的非零元分布结构. 目前已有一些针对特定异构平台的研究[2,3,18], 通过利用这种非零元分布结构, 有针对性地设计了结构化 SpTRSV 问题的并行算法.

本文在现有工作的基础上, 构造了一种面向 GPU 平台的高效并行结构化 SpTRSV 算法, 其实测性能显著优于分层调度方法, 而且该算法无需对输入问题进行任何预处理分析. 本文针对结构化 SpTRSV 问题非零元数目较少的特性, 对逐元素处理的分配策略进行了改进, 有效缓解了 GPU 线程资源的闲置问题. 此外, 本文还充分结合 GPU 体系架构的特点, 对算法的实现进行了深度优化.

1 背景介绍

1.1 结构化稀疏三角线性方程组问题

稀疏三角线性方程组问题要求解的是下列形式:

$$Ax = b$$

问题的解向量 x, 其中 A 是一个规模为 $n \times n$ 的稀疏三角矩阵, b 为 $n \times 1$ 的右端项向量. 假定 A 是下三角矩阵, 若其对角元素不为 0, 可通过回代求解步骤:

$$x_i \leftarrow A_{ii}^{-1} \cdot \left(b_i - \sum\nolimits_{0 \leqslant j < i} A_{ij} \cdot x_j\right),$$

计算解向量分量 x_i 的取值. 为节约存储空间, 稀疏矩阵多以行压缩格式 (compressed row, CSR)[19]存储. CSR 格式包括 3 个数组 $Ai[\cdot], Aj[\cdot], Aa[\cdot]$, 矩阵第 r 行包含 $(Ai[r+1]-Ai[r])$ 个非零元素, 第 r 行第 j 个非零元素的列下标和数值分别为 $Aj[Ai[r]+j]$ 及 $Aa[Ai[r]+j]$. 本文中等式的索引下标以 0 起始.

结构化稀疏三角线性方程组主要来源于采用规则网格离散方法 (如有限差分方法) 求解的偏微分方程组 (PDE).

此类问题若采用 ILU(k)[1]等方法构造迭代方法的预条件子矩阵, 在迭代方法预条件阶段计算该矩阵的逆与向量乘积的过程就是典型的结构化 SpTRSV 问题. 结构化三角线性方程组问题[2]指问题区域为规则网格的 PDE 问题, 当采用 ILU(k) 等矩阵不完全分解方法[1]生成的近似矩阵, 作为迭代方法的预条件子矩阵时, 所产生的稀疏三角线性方程组问题. 由于每个网格点对应 SpTRSV 问题中的一个等式, 假设规则网格的维度为 $X\times Y\times Z$, 记网格点坐标为 (x,y,z), 方程组等式的行索引为 r, 网格点与等式之间存在如下对应关系成立:

$$\begin{cases} f(r)=\left(f_x(r),f_y(r),f_z(r)\right) \\ g(x,y,z)=x+y\cdot X+z\cdot XY \end{cases},$$

其中, $f_x(r)=r \mod X$, $f_y(r)=\left\lfloor \frac{r}{X} \right\rfloor \mod Y$, $f_z(r)=\left\lfloor \frac{r}{XY} \right\rfloor$.

上式中函数 $f(\cdot)$ 是从等式索引 (r) 至网格点空间坐标 (x,y,z) 之间的映射关系, 函数 $g(\cdot)$ 是函数 $f(\cdot)$ 的逆映射.

结构化 SpTRSV 问题的矩阵非零元分布与 PDE 的离散化格式密切相关, 通常呈现较为规律的结构. 图 1 展示了与若干结构化 SpTRSV 问题的非零元分布规律, 图中带有不同形状标记的网格点 (等式) 均依赖于网格点 $g(x,y,z)$ 所对应等式的解向量分量 $x_{g(x,y,z)}$. 对结构化 SpTRSV 问题而言, PDE 的离散化格式以及网格尺寸可视为已知条件.

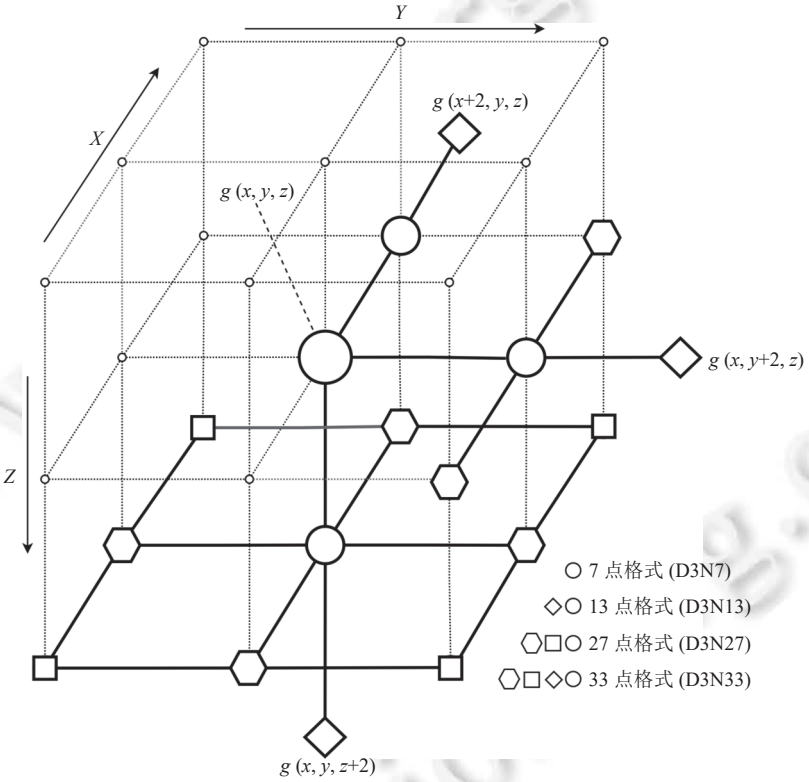


图 1 部分常见结构化问题的非零元分布示意图

1.2 GPU 体系架构

GPU 在采用异构架构的超级计算机平台上是一种中常见的加速卡硬件, 可提供数量庞大的硬件线程资源. GPU 线程资源以流处理器 (stream multiprocessor, SM) 为单元分组, 流处理器内部按 32 个线程为一个线程束 (Warp)[20], 由硬件负责其执行调度. 流处理器通过总线互联. GPU 线程共享全局内存资源, 由于 GPU 全局内存带宽有限[21,22], GPU 采用 L1+L2 的缓存结构, 用于缓冲 GPU 线程的读写操作, 减少实际产生的全局访存请求[23,24]. L1 级缓存不提供数据一致性保证. GPU 配备单独的纹理缓存, 用于缓冲只读数据, 其数据吞吐效率略高于 L1 级缓存[20].

目前比较新颖的 GPU 微架构[21,22], 已对访存弱一致性模型[25]的各项特性提供了充分支持. 该模型允许 GPU

线程对全局内存读取操作附加“获取 (acquiring)”语义[26]. 经修饰的内存读取操作能够观察到其余 GPU 线程对访存目标地址的最新改动[27]. 附带获取语义的读取操作可将目标操作数的最新副本, 推送至发起该操作的 GPU 线程所属 L1 级缓存[26].

在同组线程束内, GPU 线程在单位时钟周期内仅能执行一条公共指令. 条件跳转等分支语句可能导致组内线程跳转至不同目标地址, 导致线程束发散 (warp divergence), 降低指令流水线的执行效率[20]. GPU 架构的谓词执行 (predication) 功能可等价实现条件分支指令的功能, 适用于优化包含少量语句的条件体或循环体代码段[28].

1.3 采用逐元素处理策略的分层调度方法

分层调度算法[29]是一类基于有向无环图 (directed acyclic graph, DAG) 拓扑排序操作的并行 SpTRSV 算法. 该方法首先通过 DAG 描述方程组等式之间的数据依赖关系, 通过拓扑排序操作筛选出相互独立的方程等式, 将它们分配至 GPU 线程并行处理. 面向 GPU 架构的特性, 现有研究[16,17,30,31]已对分层调度算法进行了优化, 在其基础上加入了逐元素策略, 以充分发掘 GPU 硬件线程层面的并行性 (算法 1).

算法 1. 采用逐元素处理策略的分层调度算法.

Level-scheduling $(Ai[\cdot],Aj[\cdot],A[\cdot],b[\cdot],x[\cdot])$

1 Generate levels $S_i \in \{S\}$ by topological sorting.
2 $P[\cdot] \leftarrow 0$ ▷ Clear the state variable.
3 **foreach** level $S_i \in S$ **do**
4 **parallel-foreach** equation $s_{ij} \in S_i$ **do** ▷ at warp level.
5 $\sigma_t \leftarrow 0$ ▷ partial sum for each warp thread t.
6 **parallel-foreach** off-diagonal $Aj[e_k]$ of s_{ij} **do** ▷ at warp thread level.
7 wait for dependencies $x[Aj[e_k]]$. ▷ spinning while-loop on $P[Aj[e_k]]$.
8 $\sigma_t \leftarrow \sigma_t + Aa[e_k] \cdot x[Aj[e_k]]$
9 **end**
10 $x\left[s_{ij}\right] \leftarrow \mathrm{DIAG}\left[s_{ij}\right]^{-1} \cdot \left(b\left[s_{ij}\right] - \mathrm{WARP\text{-}REDUCTION}[\sigma_t]\right)$. ▷ solve equation s_{ij} after reducing σ_t.
11 $P\left[s_{ij}\right] \leftarrow 1$. ▷ update the state variable.
12 **end**
13 **end**

2 基于多级逐元素处理策略的并行结构化 SpTRSV 算法

2.1 基于结构信息的任务划分方案

本文在结构化 SpTRSV 问题的网格上进行粗粒度任务划分. 首先, 本文观察到结构化问题中, 行索引落入区间范围 $Q_{yz} := [g(0,y,z), g(X,y,z))$ 内的等式存在严格的串行数据依赖关系. 因此本文将行索引位于该区间范围内的等式视作一个整体, 构成子任务 Q_{yz}. 本文按 z 轴坐标, 将子任务 Q_{yz} 归为 Z 个任务队列 $Q(z) := \{Q_{0z}, Q_{1z}, \ldots, Q_{(Y-1)z}\}$.

本文按任务队列 $Q(\cdot)$ 的 z 轴坐标增序, 依次向各队列分配线程束资源. 在任务队列 $Q(\cdot)$ 内部, 线程束以动态调度的形式, 从队列中获取要处理的子任务. y 轴坐标较小的子任务具有更高的调度优先级. 本文的并行结构化 SpTRSV 算法如算法 2 所示, 其中 D 代表算法向每个任务队列分配的 GPU 线程束数量.

算法 2. 本文提出的并行结构化 SpTRSV 算法.

Para-Structured-SpTRSV $(Ai[\cdot],Aj[\cdot],A[\cdot],b[\cdot],x[\cdot],X,Y,Z)$

1 $P'[\cdot] \leftarrow -1$ ▷ clear the state variable.

2 $z \leftarrow 0$
3 **while** $\lfloor z/D \rfloor < Z$ **do**
4 **parallel-foreach** task $Q_{yz} \in Q(z)$ **do** ▷ at warp level.
5 **foreach** equation $s \in Q_{yz}$ **do**
6 wait for dependencies $x[Aj[e_k]]$▷ spinning while-loop testing $P\left[f_y(Aj[e_k]), f_z(Aj[e_k])\right] \geqslant f_x(Aj[e_k])$.
7 solve equation s.▷ with multistage element-based strategy (Section 2.2).
8 $P'\left[f_y(s), f_z(s)\right] \leftarrow s$.▷ for cache footprint reduction (Section 2.3).
9 **end**
10 **end**
11 $z \leftarrow z+1$
12 **end**

2.2 多级逐元素处理策略

算法 1 中 GPU 线程束每处理完一个等式, 后续非零元素与当前线程束内各线程之间的映射, 都将从标号为 0 的线程重新开始. 由于结构化 SpTRSV 问题每个等式所包含的非零元数目较少, 标号较大的线程, 在计算过程中无法获得任何非零元素, 始终处于闲置状态, 造成 GPU 资源的浪费 (图 2). 为解决现有逐元素策略的线程闲置问题, 本文修改了上述线程和非零元素的映射关系. 算法 2 将当前等式的非零元素映射至组内线程之后, 将继续把后续等式的非零元素, 映射至余下处于闲置状态的线程, 这样每个线程都能够获得一定数量非零元素, 从而避免 GPU 线程闲置 (图 3).

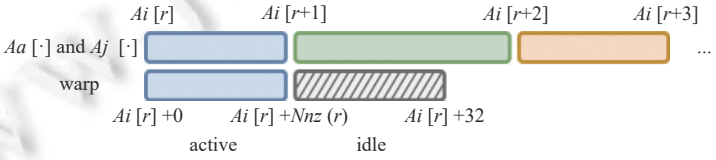


图 2 现有逐元素处理策略导致 GPU 线程闲置 (斜线阴影部分)

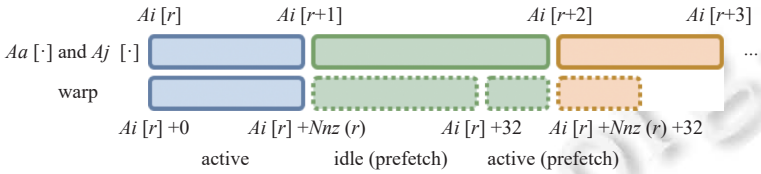


图 3 经改进的逐元素处理策略可避免 GPU 线程闲置

GPU 线程的部分和 σ_t 的计算:

$$\sigma_t \leftarrow \sigma_t + Aa[e_k] \cdot x[Aj[e_k]].$$

可拆分为 2 个阶段. 在第 1 阶段, 线程将会读取非零元信息 $Aj[\cdot]$ 和 $Aa[\cdot]$, 在第 2 阶段才会进行右端项 $x[\cdot]$ 相关的计算和内存操作. 前一阶段的全局内存访问的耗时较长. 本文采用多级处理的方式来隐藏访存开销. 多级处理在现有逐元素处理策略的基础上, 加入了非零元信息的预取步骤. 本文利用少量 GPU 寄存器资源, 构造了一个微型数据缓冲队列, 用于容纳预取步骤的操作数并记录操作数的下标. 由线程束内部线程根据下标决定, 是否从缓冲队列中提取非零元信息, 并进行第 2 阶段的计算. GPU 线程从队列中取出非零元素后, 将发起下一批次非零元信息的预取操作. 因为非零元信息的预取操作与问题等式间数据依赖无关, 所以多级处理的非零元素访存和计算可重叠进行, 能够隐藏非零元信息的访存延迟.

2.3 状态变量压缩技术

SpTRSV 算法使用若干变量 $P[\cdot]$ 来标记等式的状态. SpTRSV 计算开始前, $P[\cdot]$ 的初始值为 0, 算法得到分量

$x[s]$之后, 会将$P[s]$的值设置成 1. GPU 线程在访问分量$x[\cdot]$之前, 会通过检测$P[\cdot]$是否为 1 (算法 1 第 2 行), 来推断$x[\cdot]$当前是否处于可用状态, 以维持 SpTRSV 问题的数据依赖关系. 当大量 GPU 线程以循环形式 (算法 1 第 7 行), 反复访问状态变量时, 将引起频繁的缓存缺失, 进而产生大量全局内存操作请求. 而这些访存操作将与矩阵非零元等关键访存操作, 争抢有限的 GPU 访存带宽资源.

为解决访存带宽争用的问题, 本文采用一个维度为$Y\times Z$的二维数组$P[\cdot,\cdot]$来记录$x[\cdot]$的状态 (算法 2 第 6 行), $P[\cdot,\cdot]$初始值置为−1 (算法 2 第 8 行). 算法解出分量$x[s]$之后, 会将状态变量$P\left[f_y(s),f_z(s)\right]$的值设置为$s$. GPU 线程在读取$x[s]$之前, 将通过测试如下条件:

$$P\left[f_y(s),f_z(s)\right]\geqslant s$$

判断$x[s]$当前的可用状态. 本文提出的状态标记方法将状态变量的总数据量, 压缩至原有标记方法的$1/X$, 减少了状态变量在算法工作集数据中的占比, 有效提高了状态变量操作的缓存命中率 (第 3.4 节). 因此算法得以大幅减少和状态变量$P'[\cdot]$有关的全局内存请求, 节省 GPU 的访存带宽资源.

2.4 针对 GPU 体系架构的进一步优化

处理子任务Q_{yz}的 for-each 循环体 (算法 2 第 5–9 行) 是热点代码段. 在现有方法[13−17]中线程束每次获取的计算任务较少, 对于循环内部非零元素计算进行优化的预期收益较少, 而且非零元素计算效率上的不足可以通过 GPU 硬件高并发的特性进行弥补, 因此循环体的优化对整体性能的影响十分有限. 本文提出的算法计算任务Q_{yz}的粒度较大. 受限于 GPU 的硬件设计, 线程束的指令/访存吞吐效率不高, 所以处理非零元素的循环体就会成为算法性能的主要瓶颈. 本文结合 NVIDIA V100 GPU 的硬件特性, 对该循环体进行了优化.

首先, GPU 线程通过前文构造的寄存器缓冲队列, 获取矩阵非零元的信息 ($Aj[\cdot]$,$Aa[\cdot]$). 而且由于矩阵非零元信息在计算过程中保持不变, 本文将缓冲队列预取操作中出现的全局内存取数指令 (ld.global.∗), 替换成了相应无数据一致性保证的版本 (ld.global.nc.∗), 把与$Aj[\cdot]$以及$Aa[\cdot]$有关的访存操作, 交由 GPU 纹理缓存进行缓冲, 进一步提高非零元信息的访存吞吐效率.

其次, GPU 线程需要根据状态变量的数值推断其余线程的计算进度, 因此 GPU 线程需共享压缩后的状态变量$P[\cdot,\cdot]$, 并且要能够观察到任意线程对$P[\cdot,\cdot]$的最新改动. 由于 L1 级缓存不提供数据一致性保证, 在 GPU 的全局内存访存指令当中, 针对 L2 级缓存的访存指令 (ld.global.cg.*) 是满足上述需求的一个选项[27], 但 ld.global.cg.*访存指令的操作延迟较高, GPU 线程从修改到读取$P[\cdot,\cdot]$的时间间隔至少为 250 个 GPU 时钟周期 (图 4 实线路径). 附带“获取”语义的访存操作 (ld.require.gpu.*) 也可用于实现与 ld.global.cg.*指令类似的功能[27], 且使用 d.require.gpu.*指令 (图 4 虚线路径) 对$P[\cdot,\cdot]$进行操作的延迟 (140 个 GPU 时钟周期) 显著少于 ld.global.cg.*指令, 因此本文采用该指令对$P[\cdot,\cdot]$进行访问. 状态变量操作的高缓存命中率也让 ld.require.gpu.*指令的执行效率能够维持在较为稳定的状态. 需要说明的是结合 ld. require. gpu.*指令的操作延迟数据以及 NVIDIA V100 GPU 的体系架构特点分析, 本文推断流处理器的总线在数据推送过程中极有可能承担了数据转发的角色.

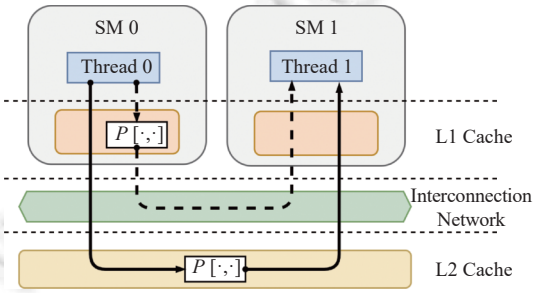


图 4 使用 ld.global.cg.*指令 (实线) 以及 ld.require.gpu.*指令 (虚线) 访问 ?[·,·] 的数据传输路径示意图

最后, 在多级逐元素处理以及寄存器缓冲队列的预取操作中, GPU 线程需要根据当前非零元的索引以及列下标 $Aj[\cdot]$, 决定是否发起新一轮数据预取, 或者判断是否需要进行一些与对角元素相关的额外操作. 实现上述功能的条件判断语句, 将在循环体内部引入众多分支指令, 导致线程在执行过程中经常出现取值地址发散的情况. 本文采用谓词执行功能消除了这些分支指令, 以缓解取值地址发散导致的性能损耗问题. 本文将条件判断语句的真值结果, 存入 GPU 的谓词寄存器[27]当中, 然后给分支语句的每一条指令添加相应的标记信息, 其中包含用于控制指令执行的谓词寄存器的索引. 被标记的指令总会进入 GPU 指令流水线, 因此不会造成取值地址发散的问题. 如果相应谓词寄存器存储的状态不为真, 那么 GPU 指令流水线不会提交被标记指令的执行结果, 该指令将不会对算法 2 的状态产生任何影响.

3 实验验证

3.1 实验设置

本文在 NVIDIA V100 GPU 平台上, 对算法性能进行测试评估, 实验平台的软硬件环境配置如表 1 所示. 由于子任务数据依赖关系会随结构化 SpTRSV 问题的类型发生变化, 本文发现算法 2 的实测性能与任务划分阶段, 算法给任务队列所分配的线程束数量之间存在关联. 为保证算法 2 能够达到最佳的实验效果, 本文依据结构化 SpTRSV 问题的非零元分布结构, 调整了算法向任务队列分配的线程束数量, 表 2 列举了算法参数 D 在各类测试问题中的取值.

表 1 实验平台的软硬件环境配置

项目	配置
CPU	Intel Xeon Gold 6240
GPU	NVIDIA Tesla V100 SXM2
GMEM	32 GB HBM2
OS	RHEL 7.6
GCC	7.5
NVCC	10.1

表 2 实验环节对算法参数 D 的设置

问题类型	64^3	128^3	192^3	256^3
D3N7	64	80	80	80
D3N13	64	80	80	80
D3N27	64	64	80	80
D3N33	64	64	44	44

3.2 性能对比实验结果

本文按照图 1 所示的非零元分布结构, 生成 D3N7 等 4 种类型的结构化 SpTRSV 问题, 以实际验证算法 2 的加速效果. 参照文献 [2], 本文采用有效访存带宽作为算法性能的评判标准, 有效访存带宽定义如下:

$$\text{有效访存带宽} := \frac{\left(\text{稀疏矩阵数据总量} + \text{右端项数据总量} \times 2\right)}{\text{SpTRSV计算时间}}.$$

测试结构化 SpTRSV 问题的规模从 64^3 逐渐增长至 256^3. 本文同时还测试了 CUSPARSE 数学库 (csrsv2 函数)[12]以及现有同步避免 (synchronization free) 方法的性能, 作为算法 2 加速效果的对照. 测试中所采用的同步避免方法, 包括 Liu 等人提出的原始同步避免算法[13,14], 以及最近由 Su 等人改进的 Capellini 框架[15]. Capellini 框架融合了一系列 SpTRSV 并行算法, 本文呈现的有效带宽数据为该框架的最优性能结果. 实验测量数据取 10 次运行的平均值. 性能实验的结果如后文图 5 所示, 算法 2 的有效访存带宽为 40.1–225.2 GB/s, 与问题规模以及等式所包含的非零元数目呈正相关. 算法 2 相比 CUSPASRE 的平均加速比为 2.71 倍, 对比同步避免方法的加速可达 15.1 倍, 性能较已有方法有大幅度提升.

3.3 性能优化效果分析

本文通过对比实验评估前文所述各项优化手段的技术效果. 实验的基线 (baseline) 版本仅使用第 2.1 节所描述的任务划分策略, 不包含任何优化措施. 多级处理版本 (multistage) 在基线版本的基础上, 加入了多级逐元素处理策略, 完整版本 (compression & others) 包含所有优化手段. 实验测试了以上各版本, 处理 D3N33 问题时的性能数据. 图 6 是实验中收集的有效访存带宽数据, 图中标注了各项优化手段对有效访存带宽的提升效果. 多级逐元素

处理策略以及谓词执行等后续优化手段, 在小规模问题中的性能增益有限. 对于大规模结构化 SpTRSV 问题, 多级逐元素处理策略的有效访存带宽, 相较基线版本有 68% 左右的提升, 若使用状态变量压缩技术以及第 2.4 节所述的优化手段, 算法性能还可进一步提升约 25%. 两种优化手段叠加之后, 基线版本有效访存带宽的提升幅度约为 115%.

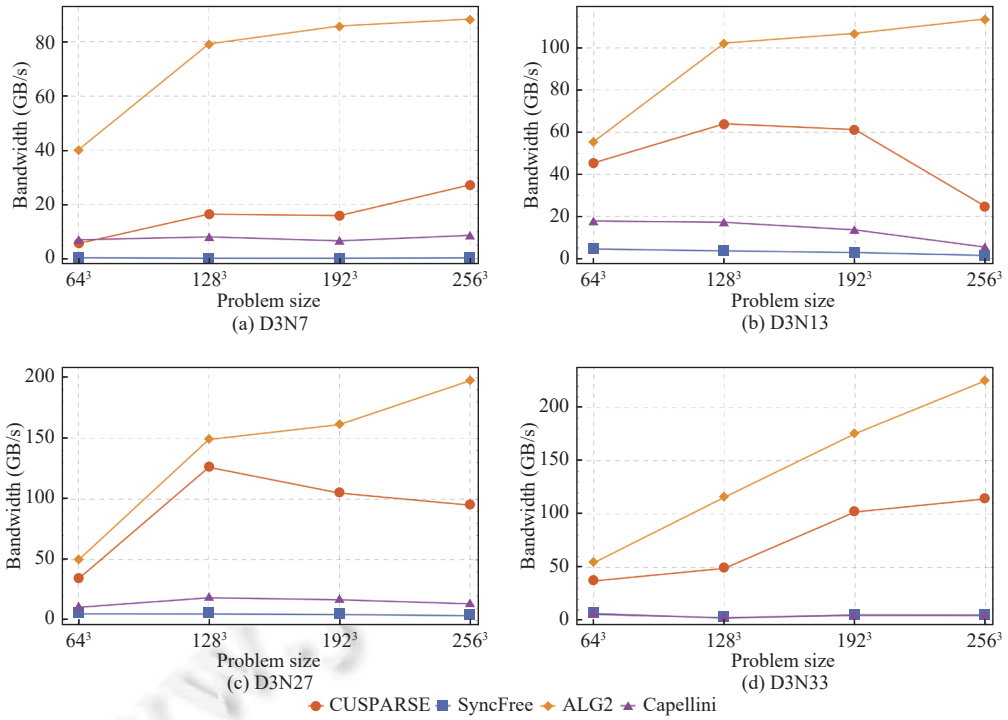


图 5 Synchronization Free, CUSPARSE 以及本文提出的算法的有效访存带宽

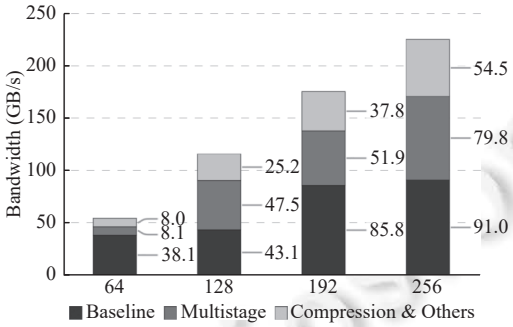


图 6 本文所述各项的优化手段对 D3N33 类型问题有效访存带宽的影响

3.4 状态变量压缩效果评估

由于算法 2 采用动态任务调度, 因此本文无法直接通过性能监测工具测量算法的缓存命中率数据. 本文通过测量算法 2, 在使用或未使用状态变量压缩技术的情况下, GPU 线程状态变量操作所消耗的时间来间接推测其缓存命中情况. 图 7 为在 D3N33 类型的结构化 SpTRSV 问题中, GPU 线程在状态变量操作上消耗的时间. 采用状态变量压缩技术并启用 GPU 缓存之后 (compressed, cached), 状态变量操作的用时相较禁用 GPU 缓存的情况 (compressed, no caching) 有显著下降, 显示绝大部分与 $P[\cdot,\cdot]$ 相关的访存操作, 均有效命中 GPU 的缓存. 如不采用状态变量压缩技术 (uncompressed, no caching/cached), 那么 GPU 缓存对状态变量操作的时间开销无明显影响, 说

明相关操作的 GPU 缓存命中率较低. 本实验说明采用状态变量压缩技术, 可有效提升 GPU 缓存命中率, 显著减少 GPU 线程状态变量操作的耗时, 减少状态变量操作占用的 GPU 访存带宽.

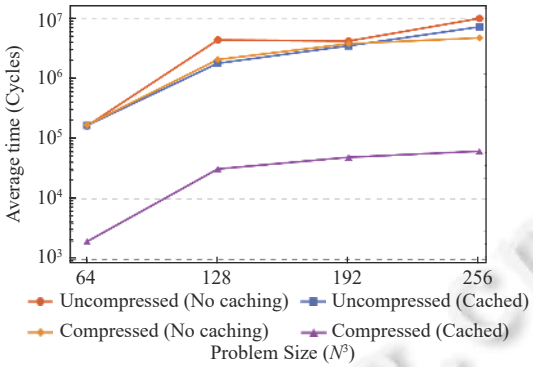


图 7 D3N33 类型问题中状态变量压缩前/后 GPU 线程状态变量操作的耗时

4 相关工作总结

分层调度方法是目前运用范围最为广泛的并行 SpTRSV 算法, 但该方法预处理开销是算法的主要性能瓶颈[16,17]. 分层调度方法的早期版本需通过全局线程同步来维持等式间的数据依赖关系, 同步操作严重影响算法性能[32], 现有研究也提出了一系列方法[13−15], 来消除线程同步带来的额外开销.

除分层调度算法等基于 CSR 格式的并行 SpTRSV 算法以外, 现有研究也提供了许多针对 SpTRSV 问题的专用稀疏数据结构, 采用多种存储方式混合形式, 记录稀疏矩阵的非零元分布结构[33,34], 为并行算法的设计提供便利条件. 在特定场景中, 预条件子步骤可采用 SpTRSV 问题的近似解替换精确解, 着色算法 (coloring)[35−37]是一种比较常用的近似方法. 该方法通过对 SpTRSV 问题的 DAG 进行着色, 将等式分为若干集合. 着色算法将忽略集合间的等式数据依赖关系, 以计算得到 SpTRSV 问题的近似解. 从理论的角度, SpTRSV 问题也存在其他求解方法[38−40].

5 结 论

本文面向 GPU 平台, 针对结构化 SpTRSV 问题, 提出了一种无需预处理步骤的结构化 SpTRSV 问题并行算法. 该算法采用多级逐元素处理策略, 在隐藏矩阵元素访存延迟的同时, 有效缓解了结构化 SpTRSV 问题导致的 GPU 线程闲置问题. 此外, 本文还引入了状态压缩技术, 并结合 GPU 的硬件特性进一步改善算法的性能.

为摆脱现有单片封装工艺的技术限制, 目前最新一代[22,41]的 GPU 已开始采用多片封装技术 (multi-chip module, MCM)[42,43], 将多个 GPU 模块整合到一起来提高 GPU 集成的功能单元数量. 虽然 MCM 模式提供了更多访存及计算资源, 但 GPU 模块间存在与非均衡内存访问 (non-uniform memory access, NUMA) 类似的性能损耗现象[44], 这给并行 SpTRSV 算法的任务划分以及访存优化提出了新的挑战. 如何克服 MCM 封装模式的不均衡效应, 找到行之有效的并行 SpTRSV 算法, 是本文未来需要关注的一个研究方向.

References:

[1] Björck Å. Numerical Methods in Matrix Computations. Cham: Springer, 2015. [doi: 10.1007/978-3-319-05089-8]

[2] Wang XL, Xu P, Xue W, Ao YL, Yang C, Fu HH, Gan L, Yang GW, Zheng WM. A fast sparse triangular solver for structured-grid problems on Sunway many-core processor sw26010. In: Proc. of the 47th Int'l Conf. on Parallel Processing. Eugene: Association for Computing Machinery, 2018. 53. [doi: 10.1145/3225058.3225071]

[3] Ao YL, Yang C, Wang XL, Xue W, Fu HH, Liu FF, Gan L, Xu P, Ma WJ. 26 PFLOPS stencil computations for atmospheric modeling on Sunway TaihuLight. In: Proc. of the 2017 IEEE Int'l Parallel and Distributed Processing Symp. Orlando: IEEE, 2017. 535−544. [doi: 10.1109/IPDPS.2017.9]

[4] Ao YL, Yang C, Liu FF, Yin WW, Jiang LJ, Sun Q. Performance optimization of the HPCG benchmark on the Sunway TaihuLight supercomputer. ACM Trans. on Architecture and Code Optimization, 2018, 15(1): 11. [doi: 10.1145/3182177]

[5] Liu YD, Wang B, Ji ZZ. Research on atmospheric motion in horizontal discrete grids. Advances in Atmospheric Sciences, 2003, 20(1): 139–148. [doi: 10.1007/bf03342058]

[6] Swillam MA, Gohary RH, Bakr MH, Li X. Efficient approach for sensitivity analysis of lossy and leaky structures using FDTD. Progress in Electromagnetics Research, 2009, 94: 197–212. [doi: 10.2528/pier09061708]

[7] Curduman L, Nastac S, Debeleac C, Modiga M. Computational dynamics of the rotational heavy loads mastered by hydrostatical driving systems. Procedia Engineering, 2017, 181: 509–517. [doi: 10.1016/j.proeng.2017.02.427]

[8] Saad Y. Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics, 2013. [doi: 10.1137/1.9780898718003]

[9] Oak ridge national laboratory. Summit user guide. 2002. https://docs.olcf.ornl.gov/systems/summit-user-guide.html

[10] IBM. Sierra fact sheet. 2018. https://asc.llnl.gov/sites/asc/files/sierra-fact-sheet.pdf

[11] AMD. ROCSPARSE documentation. 2021. https://rocsparse.readthedocs.io/en/master/

[12] NVIDIA. CUSPARSE library. 2020. https://docs.nvidia.com/pdf/CUSPARSE_Library.pdf

[13] Liu WF, Li A, Hogg JD, Duff IS, Vinter, B. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. Concurrency and Computation: Practice and Experience, 2017, 29(21): e4244. [doi: 10.1002/cpe.4244]

[14] Liu WF, Li A, Hogg J, Du IS, Vinter B. A synchronization-free algorithm for parallel sparse triangular solves. In: Proc. of the 22nd European Conf. on Parallel Processing. Grenoble: Springer, 2016. 617–630. [doi: 10.1007/978-3-319-43659-3_45]

[15] Su JY, Zhang F, Liu WF, He BS, Wu RF, Du XY, Wang RJ. CapelliniSpTRSV: A thread-level synchronization-free sparse triangular solve on GPUs. In: Proc. of the 49th Int'l Conf. on Parallel Processing. Edmonton: Association for Computing Machinery, 2020. 2. [doi: 10.1145/3404397.3404400]

[16] Li RP, Zhang CY. Efficient parallel implementations of sparse triangular solves for GPU architectures. In: Proc. of the 2020 SIAM Conf. on Parallel Processing for Scientific Computing. Seattle: Society for Industrial and Applied Mathematics, 2020. 106–117. [doi: 10.1137/1.9781611976137.10]

[17] Dufrechou E, Ezzatti P. Solving sparse triangular linear systems in modern GPUs: A synchronization-free algorithm. In: Proc. of the 26th Euromicro Int'l Conf. on Parallel, Distributed and Network-based Processing. Cambridge: IEEE, 2018. 196–203. [doi: 10.1109/pdp2018.2018.00034]

[18] Saltz JH. Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM Journal on Scientific and Statistical Computing, 1990, 11(1): 123–144. [doi: 10.1137/0911008]

[19] Choi JW, Singh A, Vuduc RW. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Bangalore: Association for Computing Machinery, 2010. 115–126. [doi: 10.1145/1693453.1693471]

[20] NVIDIA. CUDA C++ programming guide. 2021. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[21] NVIDIA. NVIDIA tesla V100 GPU architecture. 2020. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[22] NVIDIA. NVIDIA a100 tensor core GPU architecture. 2020. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[23] Choo K, Panlener W, Jang B. Understanding and optimizing GPU cache memory performance for compute workloads. In: Proc. of the 13th IEEE Int'l Symp. on Parallel and Distributed Computing. Marseille: IEEE, 2014. 189–196. [doi: 10.1109/ispdc.2014.29]

[24] Choi H, Ahn J, Sung W. Reducing off-chip memory traffic by selective cache management scheme in GPGPUs. In: Proc. of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units. London: ACM Press, 2012. 110–119. [doi: 10.1145/2159430.2159443]

[25] Gaster BR, Hower D, Howes L. HRF-relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. ACM Trans. on Architecture and Code Optimization, 2015, 12(1): 7. [doi: 10.1145/2701618]

[26] Orr MS, Che S, Yilmazer A, Beckmann BM, Hill MD, Wood DA. Synchronization using remote-scope promotion. In: Proc. of the 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Istanbul: Association for Computing Machinery, 2015. 73–86. [doi: 10.1145/2694344.2694350]

[27] NVIDIA. Parallel thread execution ISA application guide. 2021. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[28] Tian ZW, Sun G. An if-conversion algorithm based on predication execution. Information Technology Journal, 2010, 9(5): 984–988. [doi: 10.3923/itj.2010.984.988]

[29] Anderson E, Saad Y. Solving sparse triangular linear systems on parallel computers. Int'l Journal of High Speed Computing, 1989, 1(1): 73–95. [doi: 10.1142/S0129053389000056]

[30] Chien LC. How to avoid global synchronization by domino scheme. 2014. https://on-demand.gputechconf.com/gtc/2014/presentations/S4188-avoid-global-synchronization-domino-scheme.pdf

[31] Naumov M. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Technical Report, NVR-2011-001, NVIDIA, 2011.

[32] Park J, Smelyanskiy M, Sundaram N, Dubey P. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In: Proc. of the 29th Int'l Supercomputing Conf. Leipzig: Springer, 2014. 124–140. [doi: 10.1007/978-3-319-07518-1_8]

[33] Picciau A, Inggs GE, Wickerson J, Kerrigan EC, Constantinides GA. Balancing locality and concurrency: Solving sparse triangular systems on GPUs. In: Proc. of the 23rd IEEE Int'l Conf. on High Performance Computing. Hyderabad: IEEE, 2016. 183–192. [doi: 10.1109/HiPC.2016.030]

[34] Kabir H, Booth JD, Aupy G, Benoit A, Robert Y, Raghavan P. STS-K: A multilevel sparse triangular solution scheme for NUMA multicores. In: Proc. of the 2015 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Austin: IEEE, 2015. 1–11. [doi: 10.1145/2807591.2807667]

[35] Iwashita T, Nakanishi Y, Shimasaki M. Comparison criteria for parallel orderings in ILU preconditioning. SIAM Journal on Scientific Computing, 2005, 26(4): 1234–1260. [doi: 10.1137/03060076x]

[36] Iwashita T, Nakashima H, Takahashi Y. Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in ICCG method. In: Proc. of the 26th IEEE Int'l Parallel and Distributed Processing Symp. Shanghai: IEEE, 2012. 474–483. [doi: 10.1109/IPDPS.2012.51]

[37] Iwashita T, Shimasaki M. Block red-black ordering: A new ordering strategy for parallelization of ICCG method. Int'l Journal of Parallel Programming, 2003, 31(1): 55–75. [doi: 10.1023/a:1021738303840]

[38] Anzt H, Chow E, Dongarra J. Iterative sparse triangular solves for preconditioning. In: Proc. of the 21st European Conf. on Parallel Processing. Vienna: Springer, 2015. 650–661. [doi: 10.1007/978-3-662-48096-0_50]

[39] Raghavan P, Teranishi K. Parallel hybrid preconditioning: Incomplete factorization with selective sparse approximate inversion. SIAM Journal on Scientific Computing, 2010, 32(3): 1323–1345. [doi: 10.1137/080739987]

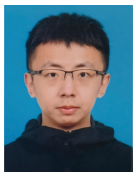
[40] Alvarado FL, Schreiber R. Optimal parallel solution of sparse triangular systems. SIAM Journal on Scientific Computing, 1993, 14(2): 446–460. [doi: 10.1137/0914027]

[41] AMD. Introducing RDNA Architecture. 2019. https://www.cxybb.com/article/chengyq116/122413939

[42] IEEE. Interconnects for 2D and 3D architectures. 2019. https://eps.ieee.org/images/files/HIR_2019/HIR1_ch22_2D-3D.pdf

[43] MEPTEC. The evolution of multi-chip packaging: From MCMs to 2.5/3D to Photonics. 2016. http://www.meptec.org/Resources/9%20-%20McCann.pdf

[44] Isaac SB, Black-Schaffer D, Casas M, Moretó M, Stupnikova A, Popov M, Modeling and optimizing NUMA effects and prefetching with machine learning. In: Proc. of the 34th ACM Int'l Conf. on Supercomputing. Barcelona: Association for Computing Machinery, 2020. 34. [doi: 10.1145/3392717.3392765]



陈道琨(1994－), 男, 博士, 助理研究员, 主要研究领域为高性能并行算法及相关自动编译优化技术.

刘芳芳(1982－), 女, 正高级工程师, CCF 专业会员, 主要研究领域为高性能数学库, 稀疏迭代解法器, 异构众核并行.

杨超(1979－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为高性能计算, 科学与工程计算.

马文静(1981－), 女, 副研究员, CCF 专业会员, 主要研究领域为高性能计算, 代码生成与优化.