软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(8):3721−3738 [doi: 10.13328/j.cnki.jos.006945]

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

一种六边形循环分块的 Jacobi 计算优化方法*

屈 彬, 刘 松, 张增源, 马 洁, 伍卫国

(西安交通大学 计算机科学与技术学院, 陕西 西安 710049)

通信作者: 伍卫国, E-mail: wgwu@xjtu.edu.cn

摘 要: Jacobi 计算是一种模板计算, 在科学计算领域具有广泛的应用. 围绕 Jacobi 计算的性能优化是一个经典的课题, 其中循环分块是一种较有效的优化方法. 现有的循环分块主要关注分块对并行通信和程序局部性的影响, 缺少对负载均衡和向量化等其他因素的考虑. 面向多核计算架构, 分析比较不同分块方法, 并选择一种先进的六边形分块作为加速 Jacobi 计算的主要方法. 在分块大小选择上, 综合考虑分块对程序向量化效率、局部性和计算核负载均衡等多方面的影响, 提出一种六边形分块大小选择算法 Hexagon_TSS. 实验结果表明所提算法相对于原始串行程序计算方法, 最好情况可将 L1 数据缓存失效率降低至其 5.46%, 最大加速比可达 24.48, 并且具有良好的可扩展性.

关键词: Jacobi 计算; 六边形分块方法; 分块大小选择; 性能优化; 多核架构

中图法分类号: TP301

中文引用格式: 屈彬, 刘松, 张增源, 马洁, 伍卫国. 一种六边形循环分块的Jacobi计算优化方法. 软件学报, 2024, 35(8): 3721–3738. http://www.jos.org.cn/1000-9825/6945.htm

英文引用格式: Qu B, Liu S, Zhang ZY, Ma J, Wu WG. Hexagonal Loop Tiling for Jacobi Computation Optimization Method. Ruan Jian Xue Bao/Journal of Software, 2024, 35(8): 3721–3738 (in Chinese). http://www.jos.org.cn/1000-9825/6945.htm

Hexagonal Loop Tiling for Jacobi Computation Optimization Method

QU Bin, LIU Song, ZHANG Zeng-Yuan, MA Jie, WU Wei-Guo

(School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China)

Abstract: Jacobi computation is a kind of stencil computation, which has been widely applied in the field of scientific computing. The performance optimization of Jacobi computation is a classic topic, where loop tiling is an effective optimization method. The existing loop tiling methods mainly focus on the impact of tiling on parallel communication and program locality and fail to consider other factors such as load balancing and vectorization. This study analyzes and compares several tiling methods based on multi-core computing architecture and chooses an advanced hexagonal tiling as the main method to accelerate Jacobi computation. For tile size selection, this study proposes a hexagonal tile size selection algorithm called Hexagon_TSS by comprehensively considering the impact of tiling on load balancing, vectorization efficiency, and locality. The experimental results show that the L1 data cache miss rate can be reduced to 5.46% of original serial program computation in the best case by Hexagon_TSS, and the maximum speedup reaches 24.48. The proposed method also has excellent scalability.

Key words: Jacobi computation; hexagonal tiling method; tile size selection; performance optimization; multi-core architecture

模板计算 (stencil computation) 是一种常见的循环计算模式, 其基本特征是遍历计算区域, 每个位置均执行相同的计算操作, 所有参与计算的数组元素共享同一套指令模板. 模板计算在医学成像、数值方法或机器学习等领域都有广泛的应用[1]. 其中, Jacobi 计算是一种具有高度优化潜力和研究价值的模板计算, 对 Jacobi 计算的优化往往能取得良好的加速效果, 但至今仍没有一套公认的针对 Jacobi 计算的最佳优化策略. Jacobi 计算在众多科学计

* 基金项目: 国家自然科学基金 (62002279); 陕西省自然科学基础研究计划一般项目 (青年) (2020JQ-077)
收稿时间: 2021-07-18; 修改时间: 2022-02-23; 采用时间: 2023-04-10; jos 在线出版时间: 2023-09-13
CNKI 网络首发时间: 2023-09-15

算领域也有着广泛的应用, 例如大规模线性和偏微分方程组求解、计算流体力学等[2].

大多数科学计算及相关应用的性能热点主要位于嵌套循环, 因此采用循环优化技术可以有效提高计算性能[3]. 循环优化包括循环展开 (loop unrolling)、循环融合 (loop fusion)、循环偏斜 (loop skewing) 和循环分块 (loop tiling) 技术, 它可以通过改变原始循环的执行顺序, 来优化循环程序的局部性和并行性[4,5]. 由于目前的计算机体系结构普遍采用分层存储架构, 在运行大规模科学计算时往往存在“存储墙 (memory wall)”问题[6]. 在循环优化技术中, 循环分块作为一种十分有效的访存优化技术, 在缓解存储墙问题上能够发挥关键的作用[7].

现有的分块方法在进行分块时往往只考虑了分块局部性的优化, 然而在实际运行过程中, 分块形状和大小还会对计算核负载均衡和向量化效率产生影响, 从而影响到实际的计算速率. 因此本文综合考虑了负载均衡、分块向量化效率和分块局部性的影响, 提出了一种新的六边形分块大小选择方法. 本文的主要工作包括以下内容: (1) 采用一种先进的六边形分块作为 Jacobi 计算循环分块优化方法, 为使分块内的访存地址具有强连续性, 选择时间维与最外空间维构成的平面作为分块平面. (2) 分析了分块大小对程序运行时向量化效率、局部性和计算核负载均衡的影响, 为六边形分块设计实现了分块大小选择算法 Hexagon_TSS. (3) 对 Jacobi, Heat 等经典模版计算程序进行测试, 通过实验验证六边形分块策略在降低缓存失效率和提升程序计算性能方面的效果明显, 并具有良好的可扩展性.

本文第 1 节介绍 Jacobi 计算和循环优化的相关工作. 第 2 节介绍本文提出的 Jacobi 计算优化方法, 包括六边形循环分块和分块大小选择算法. 第 3 节给出实验结果, 分析验证所提方法的有效性. 第 4 节总结全文.

1 相关工作

本节对本文中使用的关键技术进行介绍. 首先, 介绍 Jacobi 计算的特征; 其次, 介绍循环分块技术, 分别从分块形状和大小、局部性量化和多面体模型 3 个角度详细说明; 最后, 介绍本文所采用的波阵面并行方式.

1.1 Jacobi 计算的研究现状

Jacobi 计算来源于求解线性方程组常用的雅克比迭代法 (Jacobi iteration) 的离散化形式[8,9]. 除雅克比迭代法外, Jacobi 计算在数值模拟领域具有广泛的应用, 包括偏微分方程组求解、Lanczos 图像插值和快速傅里叶变换在内的大量科学计算方法的离散化形式都可归类为 Jacobi 计算.

Jacobi 计算的核心循环代码通常包含一层时间维循环, 并在时间维循环内嵌若干层空间维循环, 最内层循环执行 Jacobi 内核. 根据空间维的层数不同可分为 Jacobi-1d, Jacobi-2d, Jacobi-3d 等. 以 Jacobi-1d 计算程序为例, 其核心循环代码如图 1 所示. 程序在空间维循环中遍历数据, 在遍历过程更新数组元素, 完成一次遍历后进入下一个时间步继续从头开始遍历. 在 Jacobi 内核中, 任意一数组元素的更新依赖于其周围若干个数组元素的数据.

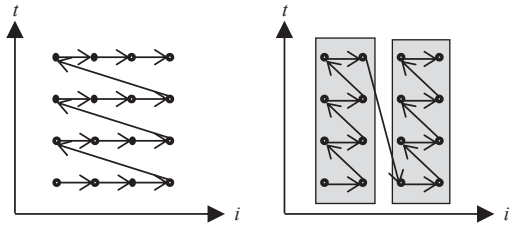
```
for (t = 0; t < tsteps; ++t)
  for (i = 1; i < size − 1; ++i)
    A[(t + 1) % 2][i] = 0.5 * (A[t % 2][i − 1] + A[t % 2][i + 1]);
```

图 1 Jacobi-1d 程序核心循环代码示例

Jacobi 计算主要通过奇偶复制方式实现, 即分配两个大小相同的数组, 每个时间步遍历一个数组, 并将计算结果写入另一个数组中; 进入下一个时间步后, 程序遍历另一个数组, 并将计算结果写入到原数组中, 依此类推. 奇偶复制方式使得 Jacobi 计算中同一个时间步内对空间维的遍历是完全可并行的, 因此 Jacobi 计算适合并行化处理.

1.2 循环分块技术

循环分块是指通过增加循环嵌套的维度来提升数据局部性的循环变换技术[10], 在程序并行任务划分、通信优化、局部性优化等领域发挥着重要作用. 其中, 局部性优化特别是时间维局部性优化, 是循环分块能够提升程序性能的主要原因. 在局部性优化方面, 如图 2 所示, 循环分块能够改变程序执行顺序, 进而改变数据重用距离, 使程序的重用发生在更小的访存地址跨度内, 以降低基于 LRU 替换算法的高速缓存失效率, 并减少访存时间.



(a) 未分块循环迭代执行顺序 (b) 分块后循环迭代执行顺序

图 2 循环分块对循环迭代执行顺序的影响

针对不同类型的计算模板, 循环分块可以有不同的形状, 包括矩形、平行四边形、梯形、菱形、六边形等. 在循环分块中, 矩形、平行四边形、梯形被看作是基本四边形, 而六边形可看作是一个正梯形和一个倒梯形的组合, 菱形则是特殊的六边形. 矩形分块具有实现简单的特点, 并且在所有分块形状中能够达到最高的分块内数据重用率. 在条件允许的情况下, 实现循环分块应尽可能地采用矩形. 矩形分块中不允许存在跨空间维的数据依赖, 所以矩形分块适用于 Point-wise 模板计算和矩阵乘类型计算, 但不适用于 Jacobi 计算.

针对 Jacobi 计算的数据流特点, Bondhugula 等人[11]提出和实现了平行四边形分块 (parallelogram tiling) 方法, 通过倾斜分块表面使分块内的数据依赖得到保持. 平行四边形分块之间存在空间维超平面内的数据依赖, 因此分块间不能完全并行执行. 采用平行四边形分块的循环程序可通过波阵面方式实现并行, 其执行分为起始、满载、排空 3 个阶段, 程序只有在满载阶段才达到并行度的最大化, 在起始和排空阶段不能完全并行.

为提高 Jacobi 循环分块程序的并行度, Grosser 等人[12]在平行四边形分块的基础上提出了梯形分块 (trapezoidal tiling) 方法. 梯形分块是一种分裂分块 (split tiling) 方法, 将平行四边形分裂为一个正梯形和一个倒梯形, 并改变分块间数据依赖的方向. 分裂分块中同一层的正梯形间和倒梯形间可完全并行, 正梯形与倒梯形交替并行执行. 梯形分块也可通过波阵面实现并行, 在并行执行时全程满载. 梯形分裂分块在并行度上优于平行四边形分块, 并且比平行四边形能够达到更高的分块内数据重用率.

Grosser 等人在梯形分块基础上进行改进, 将正梯形与倒梯形上下拼接在一起, 实现了六边形分块 (hexagonal tiling)[13,14]. 与梯形分块一样, 六边形分块也可完全并行, 但六边形分块的块内数据重用率高于梯形分块. 在六边形分块的基础上, Bondhugula 等人[15]通过最大化分块内的时间维长度得到菱形分块 (diamond tiling). 菱形分块是六边形分块的特殊形式, 目前适合 Jacobi 计算的分块形状中, 菱形分块可达到最高的分块内数据重用率. 但菱形分块上下顶端短边边长较短, 在数据规模相同的情况下控制开销大于一般的六边形分块, 且不利于分块内的向量计算, 而梯形分块和六边形分块则能够达到更好的向量化效率.

分块大小对循环分块的效果有重要影响, 因此大部分针对循环分块的研究围绕分块大小选择 (tile size selection, TSS) 展开. 目前分块大小选择方法主要有静态方法和经验搜索方法. 静态方法上, 斯坦福大学的 Wolf 等人[16]最早提出了描述程序数据重用的局部性静态数学模型, 在此基础上首先实现了面向矩形分块的定量化分块大小选择算法. 刘松等人[17]提出了一种基于高速缓存均匀映射的分块因子选择算法 UMC-TSS, 充分利用局部性分析的收益模型, 选择高收益的循环代码实施分块优化. 在经验搜索方面, ATLAS 系统[18]可通过遍历不同问题规模的线性代数求解程序, 从中选择性能较优的分块大小. 经验搜索方法通常与机器学习结合[19,20], 可获得局部最优性能的分块大小, 但遍历测试用例的时间开销较大, 因此通常先结合静态分析模型裁剪搜索空间, 以减少遍历次数.

1.3 局部性量化

局部性原理作为重要的计算机科学理论, 一直受到广泛的研究, 许多研究工作专注于局部性分析与量化的理论研究[21,22]. 狄鹏等人[23]针对 Jacobi 计算提出了一个能够较客观地反映局部性的指标, 时间维数据重用率 (temporal data reuse rate, *TDRR*). 块内的时间维数据重用率可定义为分块中所有数据的时间维重用次数除以分块的访存地址跨度, 如公式 (1) 所示.

$$TDRR = \frac{reuse}{range} \tag{1}$$

其中, *reuse* 表示分块的时间维数据重用次数, 仅考虑时间开销较大的写重用; *range* 表示分块的访存地址跨度, 分块局部性与 *TDRR* 值正相关. 对于 Jacobi 计算, 分块的数据重用次数等于分块的迭代实例个数减去分块的访存地址跨度. 分块的访存地址跨度定义为分块内最大访存地址与最小访存地址之差, 因为访存地址跨度考虑的是静态存储区分配内存空间, 数据行在内存中是连续的, 其值也等于分块投影在空间维上超平面的面积. 分块迭代实例个数与访存地址跨度均与具体的分块形状有关. 该公式的定义仅面向分块方法, 而分块方法是为并行服务的, 为了保证各线程计算结果正确性, 每个分块的块间依赖必须在执行前得到满足, 因此该公式中未考虑块间依赖.

从定性分析的角度上看, 分块访存地址跨度衡量了分块内的数据在内存中的分布范围. 若分块数据的分布范围越小, 则访存地址跨度越小, 那么访问该分块内的数据时更容易命中缓存. 而数据重用次数则表示程序访问相同数据的次数. 结合起来看, *TDRR* 表示在一定的缓存命中率下访问内存的次数, 当缓存命中率较高时, 访问相同数据的次数越多, 表明分块局部性越好.

1.4 波阵面并行

波阵面并行 (wave-front parallelism) 是一种面向 DOACROSS 循环的并行控制方式[24,25], 适用于以 Jacobi 计算为代表的具有同步操作的模板计算程序. 波阵面将循环迭代空间划分为多个波前线 (wave-front line), 每个波前线内放置若干个分块. 同一个波前线内的分块可以同时并行执行, 程序按顺序执行每个波前线, 在推进波前线时进行同步. 若循环迭代空间存在空间维超平面内的数据依赖, 则依赖目标迭代实例必须滞后于依赖源迭代实例执行, 对应的波阵面也必须倾斜. 倾斜波阵面的执行分为起始、满载和排空阶段, 在起始和排空阶段不能完全并行, 相对于普通波阵面并行度较低. 倾斜波阵面通常应用于 Sweep 模板计算和平行四边形分块的并行.

在任务调度方面, 波阵面既可采用静态调度方法, 也可以采用动态调度方法. 静态调度方法在程序执行前预先为每个计算核分配负载范围, 可实现分块与计算核的绑定, 有利于提高分块间的局部性. 动态调度方法[26−28]在运行时根据程序产生的实际数值调节负载分配, 有利于维持运行时计算核负载均衡. 静态调度适用于包括模板计算在内的访存地址序列可预测、随机性弱的任务类型, 而动态调度适用于访存地址序列不可预测、随机性强的任务类型, 例如实时任务调度. Jacobi 计算作为一类访存地址序列可预测的计算任务, 适合采用静态调度方法.

2 六边形循环分块优化方法

本节首先分析分块平面对分块访存地址连续性的影响, 探讨适合 Jacobi 六边形分块的分块平面. 接着从负载均衡、向量化效率和分块局部性收益 3 个角度讨论分块大小对分块性能的影响. 最后, 推导面向六边形分块的分块大小选择算法 Hexagon_TSS. 六边形可视作一个正梯形与一个倒梯形的组合, 而梯形分块较为简单, 因此下面每节的分析都先从梯形分块入手, 再将梯形分块的结论扩展到六边形分块中.

2.1 分块平面选择

分块平面指循环分块在空间中投影出分块形状的平面, 例如 2 维 Jacobi 计算的六边形分块在分块平面上投影出六边形, 在其他平面上投影出矩形; 2 维 Jacobi 计算的平行四边形分块在分块平面上投影出平行四边形, 在其他平面上投影出矩形.

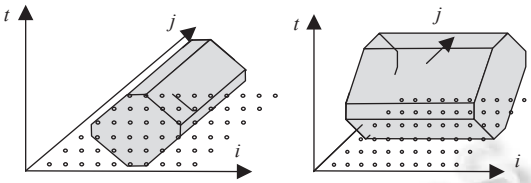
在进行循环分块前首先为 2 维及 2 维以上的 Jacobi 计算选择合适的分块平面, 多边形分块可以选择时间维和空间上任意一维构成分块平面. 需要注意的是, 越往内的空间维循环, 迭代实例的访存连续性越强. 特别地, 最内维循环的迭代实例具有连续的访存地址. 选择空间上的内维与时间维构成的平面作为分块平面将破坏该空间维的访存连续性, 扩大分块的访存地址跨度, 不利于局部性.

以后文图 3 中的 Jacobi-2d 六边形分块为例, *t* 轴是时间维, *i* 轴、*j* 轴分别是空间的第 1 维和第 2 维. 其中 *j* 轴是最内维, *j* 轴上任意相邻两点的访存地址连续, 图中圆点表示迭代实例. 图 3(a) 中的分块以 *t* 轴和 *i* 轴构成的平面作为分块平面, 那么分块内同一个 *i*-*j* 平面上的访存地址都是连续的. 而图 3(b) 中的分块以 *t* 轴和 *j* 轴构成的平

面作为分块平面, 此时分块内只有 j 方向上迭代实例的访存地址才是连续的, 而 i 方向上迭代实例的访存地址不连续. 由此可见, 将空间上的最外维与时间维构成的平面作为分块平面, 在访存地址连续性上优于时间维与空间内维构成的分块平面, 且前者的访存地址跨度更小. 为了最小化分块的访存地址跨度, 本文为六边形分块选择空间维的最外维与时间维构成的平面作为分块平面.

2.2 六边形分块大小定义

记 TS_i 表示分块在第 i 维的长度, N_i 表示 Jacobi 计算第 i 维的迭代范围. 由于选择时间维和空间第 1 维构成的平面作为分块平面, 分块第 1 维的长度 TS_1 和第 2 维的长度 TS_2 是可调节的, TS_1 和 TS_2 决定了分块大小; 而分块第 x 维 (x≥3) 的长度固定等于全局循环迭代空间在第 x 维上的迭代范围. 其中 TS_1 和 TS_2 分别是分块跨过的时间步范围和分块在空间第 1 维上投影的长度, 它们与分块几何形状之间的对应关系如图 4 所示.



(a) t-i 分块平面 (b) t-j 分块平面

图 3 Jacobi-2d 中两种六边形分块平面

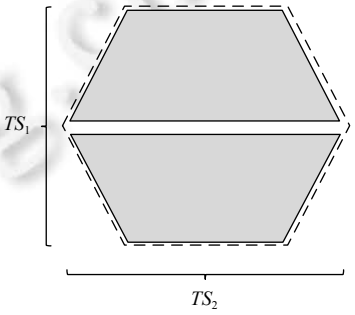


图 4 六边形分块大小与几何形状对应关系

由于分块第 x 维 (x≥3) 的长度已固定, 在分块大小选择上主要考虑 TS_1 和 TS_2 的选择. 下面将分别讨论分块大小 TS_1 和 TS_2 与分块形状、负载均衡、向量化效率以及局部性收益之间的关系.

2.3 分块大小选择

2.3.1 分块大小与分块形状

在梯形分块中, 底边可分为长底边和短底边. 梯形分块的高为 tra_TS_1, 长底边的长度为 tra_TS_2, 短底边的长度 tra_TS_2' 与 tra_TS_1、tra_TS_2 和 Jacobi 的模板因子有关. 在梯形分块中, tra_TS_2 经过 tra_TS_1 时间步后缩短为 tra_TS_2', 而在大多数 Jacobi 计算中, 模板因子为 1, 因此每个时间步缩短的长度为 2, 故短底边长度 tra_TS_2' 可通过公式 (2) 计算.

$$tra\_TS_2' = tra\_TS_2 - 2\times(tra\_TS_1 - 1) \tag{2}$$

为了维持梯形的形状, 梯形分块的短底边长度必须大于 0, 同时梯形的高, 即时间维长度大于 1, 因此梯形分块第 1 维的长度 tra_TS_1 和第 2 维的长度 tra_TS_2 须满足公式 (3) 所示的约束条件.

$$\begin{cases} tra\_TS_1 \geqslant 2 \\ tra\_TS_2 \geqslant 2\times(tra\_TS_1 - 1) \end{cases} \tag{3}$$

同样的原理运用到六边形分块上, 当 TS_2=tra_TS_2 时, 六边形分块的是梯形分块的两倍, 所以六边形分块的 TS_1 是偶数. 为了维持六边形的形状, 分块第 1 维的长度 TS_1 和第 2 维的长度 TS_2 须满足公式 (4) 所示的约束条件. 特别地, 对于六边形分块, 当 TS_2=TS_1−1 时, 分块的形状为菱形.

$$\begin{cases} TS_1 \geqslant 4 \\ \mathrm{mod}(TS_1, 2) = 0 \\ TS_2 \geqslant TS_1 - 1 \end{cases} \tag{4}$$

2.3.2 分块大小与负载均衡

在采用静态任务调度的波阵面并行方式下, 设波阵面中每个分块都采用相同规格的分块大小, 为平衡线程的

计算负载, 分块的分配应遵循平均原则. 在就绪波前线中分块数量等于线程数的情况下, 每个线程处理的分块数相同. 对于问题规模较小的 Jacobi 计算, 若分块大小过大, 那么就绪波前线中分块的数量就可能小于线程的个数, 将导致部分线程无负载, 出现负载不均衡的情况. 对于一般问题规模的 Jacobi 计算, 若就绪波前线中分块数量不等于线程数的整数倍, 那么在运行时也将出现负载不均衡的现象, 部分线程在每次推进波前线时总是比其他线程多执行一个分块, 造成其他线程的等待. 下面将分析分块大小 TS_1 和 TS_2 与负载均衡之间的定量关系.

在采用梯形分块的 Jacobi 计算中, 任意时刻都是由一个正梯形和一个倒梯形交错的形式排布, 且其执行过程也是交错执行的, 如图 5 所示. 因此就绪波前线中分块数量 *num* 为单个时间分块内总分块数的一半, 可通过公式 (5) 计算.

$$num=\left\lceil\frac{N_2}{2\times tra\_TS_1\times(tra\_TS_2+tra\_TS_2')/2}\right\rceil=\left\lceil\frac{N_2}{2\times(tra\_TS_2-tra\_TS_1+1)}\right\rceil \tag{5}$$

公式 (5) 中, N_2 表示 Jacobi 计算第 2 维的迭代范围. 通过调节 tra_TS_1 和 tra_TS_2 可使就绪波前线中分块数量等于线程数 *threads* 的整数倍, 此时 *num* 除以 *threads* 的余数 *remain* 应等于 0, *remain* 通过公式 (6) 计算. 若调节 tra_TS_1 和 tra_TS_2 不能使 *remain* 的值等于 0, 此时应使等待线程的数量最小, 即调节 tra_TS_1 和 tra_TS_2 使 *remain* 的值最大.

$$remain=\mathrm{mod}(num,threads) \tag{6}$$

对于采用六边形分块的 Jacobi 计算, 在 TS_2=tra_TS_2 的情况下, 六边形分块的 TS_1 是梯形分块的 tra_TS_1 的两倍, 那么任意时刻就绪波前线中六边形分块数量 *num* 通过公式 (7) 计算. 与梯形分块同理, 就绪波前线中六边形分块数量除以线程数得到的余数 *remain* 也通过公式 (6) 计算.

$$num=\left\lceil\frac{N_2}{2\times(TS_2-TS_1/2+1)}\right\rceil=\left\lceil\frac{N_2}{2\times(TS_2+1)-TS_1}\right\rceil \tag{7}$$

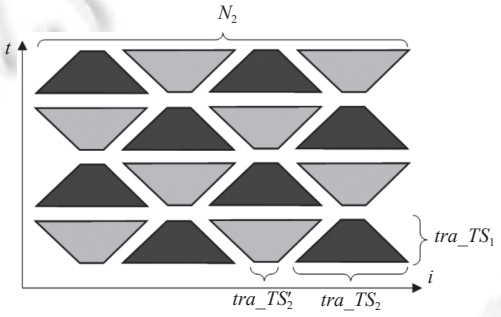


图 5 梯形分块调度过程

2.3.3 分块大小与向量化效率

对于具有 SIMD 指令集扩展的 CPU, 可通过单条 SIMD 指令可以完成多组数据的运算, 前提是参与运算的数据宽度与向量寄存器宽度 *W* 要对齐, 即一个向量寄存器中可存放的浮点运算操作数个数对齐. 若不对齐, 则处理器只能通过标量运算指令完成不对齐部分的运算. 对于循环分块, 若分块边长与向量寄存器宽度对齐良好, 那么大部分的运算可通过 SIMD 指令执行, 完成块内数据的运算所需的指令个数更少, 进而减少处理器运算该分块所需的时间. 目前主流的编译器都支持自动向量化, 在编译过程中将满足条件的标量计算指令替换成 SIMD 指令. 由于编译器具有良好的自动向量化功能, 在编写高性能程序时, 大多数情况下程序员不需要手动输入和编排 SIMD 指令, 只需把主要的精力放在数据对齐上.

记分块中平均每个迭代实例所需的指令条数 (instructions per iteration) 为 *IPI*, *IPI* 与分块最内维的长度有关. *IPI* 能够客观地反映分块的向量化效率, 它的值越小, 说明分块的向量化效率越好, 程序所含的指令条数越少. 对于 *n* 维 ($n\geqslant 3$) 梯形分块, 它的 *IPI* 等于各时间步对应的空间维超平面运算所需指令条数的叠加, 再除以分块的迭代实例个数, 其简化后的计算公式如公式 (8) 所示.

$$IPI=\frac{\sum_{t=0}^{TS_1-1}(\lfloor L_n/W\rfloor+\mathrm{mod}(L_n,W))}{S} \tag{8}$$

其中, W 是向量寄存器宽度; L_n 是分块最内维的实际长度, 随时间步变化, 在 $n\geqslant 3$ 的情况下固定等于最内层循环的迭代范围 N_n, 不可调节; S 为分块的迭代实例个数, 可通过计算分块对应的几何形状面积得到. 特别地, 当 $n=2$ 时, 即面向 Jacobi-1d 程序时, tra_TS_2 作为分块最内维的边长, 梯形分块 IPI 的计算方式比较特殊, 通过公式 (9) 计算, 其中 $K=\mathrm{mod}(tra\_TS_2-2t, W)$, tra_TS_2'可由公式 (2) 计算.

$$IPI=\frac{\sum_{t=0}^{TS_1-1}(\lfloor(tra\_TS_2-2t)/W\rfloor+K)}{tra\_TS_1\times(tra\_TS_2+tra\_TS_2')/2}=\frac{\sum_{t=0}^{TS_1-1}(\lfloor(tra\_TS_2-2t)/W\rfloor+K)}{tra\_TS_1\times(tra\_TS_2-1)+1} \tag{9}$$

六边形分块可看作两个梯形的组合, 在循环维度 $n\geqslant 3$ 时, 六边形分块的 IPI 与具有相同 TS_2 的梯形分块的 IPI 相等. 但在循环维度 $n=2$ 时, 六边形分块 IPI 的计算方式与梯形分块略有差别, 此时六边形分块的 IPI 由公式 (10) 计算, 其中, $K=\mathrm{mod}(TS_2-2t, W)$.

$$IPI=\frac{\sum_{t=0}^{TS_1-1}(\lfloor(TS_2-2t)/W\rfloor+K)}{(TS_1/2\times(TS_2-1)+1)\times 2} \tag{10}$$

向量寄存器宽度 W 与处理器的 SIMD 指令集类型有关, 表 1 给出了部分常用 SIMD 指令集对应的向量寄存器宽度. 由于只有在循环维度 $n=2$ 时, 调节分块大小 TS_1 和 TS_2 才能影响分块的 IPI, 因此分块大小对分块向量化效率的影响仅存在于 1 维 Jacobi 计算中. 在针对 Jacobi-1d 程序选择分块大小时, 可将最小化 IPI 作为一个优化目标, 通过调节 TS_1 和 TS_2, 改善数据对齐状况, 减少分块所需的指令条数.

表 1 常用 SIMD 指令集的数据位宽

SIMD指令集类型	向量寄存器宽度W		
	位宽 (bit)	单精度操作数宽度 (个)	双精度操作数宽度 (个)
Intel SSE	128	4	2
Intel AVX	256	8	4
Intel AVX512	512	16	8
ARM NEON	128	4	2

2.3.4 分块大小与局部性收益

在分块大小与局部性收益的研究中, 冲突失效 (conflict miss)[26]、数据重用 (data reuse)[26]、缓存容量 (cache capacity)[26]和缓存一致性 (cache coherence) 等都是重要的影响因素. 然而冲突失效与缓存一致性比较难以量化, 此外为了防止考量过多因素造成分块大小选择算法中局部性权重过大, 因此本节对于局部性的分析仅考量了时间维的数据重用和缓存容量两个因素.

由于处理器访问主存所需的时间远大于访问缓存的时间, 因此完全命中缓存相比于部分访存不命中的情况, 访存速度有显著的提升. 在 Jacobi 循环分块程序中, 通过调节空间维分块大小可改变分块的访存地址跨度. 根据访存地址跨度的定义, Jacobi 计算中梯形分块和六边形分块的访存地址跨度 $range$ 与空间维分块大小 TS_2 之间的定量关系可由公式 (11) 描述.

$$range=2\times TS_2\times\prod_{i=3}^{n}TS_i \tag{11}$$

在选择分块大小时, 应尽量使分块内的访存操作完全命中某一级缓存, 分块的访存地址跨度应小于目标缓存的容量. 据此, 可为分块大小 TS_2 构建一个与目标缓存容量相关的约束条件, 进一步裁剪分块大小选择的搜索空间, 该约束条件如公式 (12) 所示.

$$\left\lfloor\frac{range}{cline}\right\rfloor\times cline\times typesize\leqslant ccapt \tag{12}$$

其中, $cline$ 表示目标缓存行大小, $ccapt$ 表示目标缓存容量, $typesize$ 表示数据类型大小. 由于缓存是以缓存行为单

位进行数据置换的, 所以在计算分块在缓存中的实际访存地址跨度时, 须将分块的访存地址跨度换算成分块占用的缓存行个数, 并乘上缓存行大小. 在实际应用中, 一个分块通常不能独占目标缓存, 因此 TS_2 的选取应偏小一些.

前文提到, 分块的时间维数据重用率 *TDRR* 能够客观地反映分块的局部性, 它等于分块中所有数据的时间维重用次数 *reuse* 除以分块的访存地址跨度. 按照时间维重用次数的定义, 梯形分块的时间维重用次数等于分块的迭代实例个数减去分块在空间维超平面上投影的面积, 其化简后的计算方法如公式 (13) 所示.

$$reuse = (tra\_TS_1 - 1) \times (tra\_TS_2 - 1) \times \prod_{i=3}^{n} tra\_TS_i \tag{13}$$

对于六边形分块, 在 TS_2 相等的情况下 TS_1 是梯形分块的两倍, 同时六边形分块的迭代实例个数等于两个梯形分块的迭代实例个数之和. 在考虑分块的时间维重用次数时, 还要再加上两个梯形之间的一层时间维长度为 1 的矩形分块的迭代实例个数. 六边形分块的时间维重用次数计算与梯形分块相似, 如公式 (14) 所示.

$$reuse = (TS_1 \times (TS_2 - 1) + 2) \times \prod_{i=3}^{n} TS_i \tag{14}$$

在得到六边形分块的访存地址跨度和时间维数据重用次数后, 联立公式 (11) 和公式 (14), 六边形分块的 *TDRR* 通过公式 (15) 计算. 在规划分块大小时, 应通过调节 TS_1 和 TS_2 以最大化分块的时间维数据重用率 *TDRR*.

$$TDRR = \frac{(TS_1 \times (TS_2 - 1) + 2) \times \prod_{i=3}^{n} TS_i}{2 \times TS_2 \times \prod_{i=3}^{n} TS_i} \tag{15}$$

2.3.5 Hexagon_TSS 算法

本节将推导面向六边形分块的分块大小选择算法 Hexagon_TSS (hexagon tile size selection). Hexagon_TSS 算法的核心是多目标规划, 利用约束条件构造一个搜索空间, 在搜索空间里根据规划目标搜索最优解. 其中约束条件包括分块大小满足维持六边形的形状、分块的访存地址跨度小于目标缓存容量. 优化目标则包括最佳多线程运行时负载均衡、最小化分块内平均执行每个迭代实例的指令个数 *IPI*、最大化分块的时间维数据重用率 *TDRR*. 而 Hexagon_TSS 算法只有两个可调节的自变量, 分别为时间维分块长度 TS_1 和空间第 1 维分块长度 TS_2. 综合公式 (4)、公式 (6)、公式 (10)–公式 (12) 和公式 (15), 以及最小化 *IPI* 和最大化 *TDRR* 的约束, 推导出关于六边形分块可调分块大小 TS_1 和 TS_2 的数学规划表达式的问题约束和优化目标. 其中, 问题约束如公式 (16) 所示.

$$\begin{cases} TS_1 \geqslant 4 \\ TS_1 \leqslant N_1 \\ \mathrm{mod}(TS_1, 2) = 0 \\ TS_2 \geqslant TS_1 - 1 \\ remain = \mathrm{mod}(num, threads) \\ TS_2 \leqslant \min\left(N_2, \dfrac{ccapt}{2 \times typesize \times \prod_{i=3}^{n} TS_i}\right) \\ \min\left(\dfrac{\sum_{t=0}^{TS_1-1} (\lfloor (TS_2 - 2t)/W \rfloor + K)}{(TS_1/2 \times (TS_2 - 1) + 1) \times 2}\right) \\ \max\left(\dfrac{(TS_1 \times (TS_2 - 1) + 2) \times \prod_{i=3}^{n} TS_i}{2 \times TS_2 \times \prod_{i=3}^{n} TS_i}\right) \end{cases} \tag{16}$$

Hexagon_TSS 算法规划优化目标如表 2 所示. 表 2 中, 优化目标 1 和优化目标 2 存在互斥性, 在进行数学规划时, 若能够满足优化目标 1, 则不需要再考虑优化目标 2; 反之, 若不能满足优化目标 1, 再考虑优化目标 2. 下面将讨论分别以 4 个优化目标为主要优化目标进行数学规划时, 自变量的取值.

表 2 Hexagon_TSS 算法优化目标

优化目标编号	目标函数	最优解
1	公式(6)	0
2	公式(6)且$remain\neq 0$	最大值
3	公式(10)	最小值
4	公式(15)	最大值

首先, 是以最佳多线程运行时负载均衡为优化目标的数学规划. 最理想的情况是就绪分块的数量刚好等于并行线程数的整数倍, 即就绪分块的数量除以线程数的余数 *remain* 等于 0. 然而余数的计算涉及与 TS_1 和 TS_2 相关的取模运算, 而取模运算的结果具有较强的随机性, 难以构造针对取模运算的精确优化模型. 因此, 在以最佳多线程运行时负载均衡为优化目前时, 应通过遍历的方法记录使得 *remain* 等于 0 的 TS_1 和 TS_2; 若搜索空间内不存在 TS_1 和 TS_2 使得 *remain* 等于 0, 则取使 *remain* 达到最大值的 TS_1 和 TS_2 作为该优化目标下的分块大小.

其次, 是以分块内平均执行每个迭代实例的指令个数 *IPI* 最小化为优化目标的数学规划. 观察公式 (10), 由于该公式中存在与 TS_1 和 TS_2 相关的取模运算, *IPI* 与 TS_1 和 TS_2 之间不存在明显的单调关系, 也不能求 *IPI* 对 TS_1 和 TS_2 的偏导数. 取模运算的结果具有较强的随机性, 难以推导 *IPI* 与 TS_1 和 TS_2 之间精确的换算关系. 因此对搜索空间中的 TS_1 和 TS_2 进行遍历, 记录使 *IPI* 最小的 TS_1 和 TS_2 作为该优化目标下的最佳分块大小.

最后, 是以分块的时间维数据重用率 *TDRR* 最大化为优化目标的数学规划. 给定六边形分块, 观察公式 (15), TS_1 和 TS_2 必然满足 $TS_1\geqslant 1$ 和 $TS_2\geqslant 1$, 因此通过分别求 *TDRR* 对 TS_1 和 TS_2 的偏导数, 可发现 *TDRR* 随 TS_1 或 TS_2 单调递增, 提高分块的时间维重用次数的途径是增大 TS_1 或 TS_2. 但分块的访存地址跨度同样随着 TS_2 的增大而扩大, 为了最小化分块的访存地址跨度, 在最大化 *TDRR* 时应优先增大 TS_1, 再增大 TS_2.

Hexagon_TSS 的实现算法如下所示. Hexagon_TSS 算法首先根据循环的迭代范围设置第 3 维以及更内维的分块大小, 根据公式 (16) 所示的约束条件构造一个搜索空间, 并构造一个候选解的集合 *candidates*, 如算法 1 中第 1, 2 行. 然后在搜索空间中搜索满足优化目标 1 的解; 若满足, 则将解加入 *candidates* 中, 跳过优化目标 2, 如算法 1 中第 3–9 行. 若优化目标 1 没有得到满足, 则基于优化目标 2 遍历搜索空间, 记录 *remain* 的最大值并将对应的解加入 *candidates* 中, 如算法 1 中第 10–24 行. 若循环维度 n=2, 则基于优化目标 3, 遍历候选解集合 *candidates*, 记录 *IPI* 的最小值, 并删除 *candidates* 中不满足 *IPI* 达到最小值的解; 若循环维度 $n\geqslant 3$, 则跳过优化目标 3, 如算法 1 中第 25–37 行. 最后基于优化目标 4, 在候选解集合 *candidates* 中找出能够使 *TDRR* 达到最大值的解, 并在其中找到 TS_2 最小的解作为最终解, 如算法 1 中第 38–50 行.

算法 1. 六边形分块大小选择算法 Hexagon_TSS.

输入: 循环维度 n, 循环迭代范围 $N=\{N_1, N_2, \ldots, N_n\}$, 并行线程数 *threads*, 向量寄存器宽度 W, 目标缓存容量 *ccapt*, 数据类型大小 *typesize*;

输出: 六边形分块大小 $TS=\{TS_1, TS_2, \ldots, TS_n\}$.

1 初始化$\{TS_1, TS_2, \ldots, TS_n\}$, 构造候选解集合 *candidates* 和优先级队列 *priority_queue*;

2 根据公式 (16) 初始化 $TSmax_1$ 和 $TSmax_2$;

3 **FOR** $ts1 = 4 : TSmax_1$ **BY** 2 **DO**

4 **FOR** $ts2 = ts1 - 1 : TSmax_2$ **BY** 1 **DO**

5 **IF** ($remain(ts1, ts2) == 0$) **THEN**

```
6          candidates.add({ts1, ts2});
7        END IF
8    END FOR
9  END FOR
10 IF (candidates.empty()) THEN
11    maxremain = 0;
12    FOR ts1 = 4 : TSmax_1 BY 2 DO
13       FOR ts2 = ts1 − 1 : TSmax_2 BY 1 DO
14          IF (remain(ts1, ts2) > maxremain) THEN // remain(a, b) 表示计算以 a, b 为分块大小时的就绪波前线中分块数目, 并对 threads 取余
15             candidates.clear();
16             candidates.add({ts1, ts2});
17             maxremain = remain(ts1, ts2);
18          ELSE IF(remain(ts1, ts2) == maxremain) THEN
19             candidates.add({ts1, ts2});
20          END IF
21       END IF
22    END FOR
23 END FOR
24 END IF
25 IF (n == 2) THEN
26    minIPI = INT_MAX; // INT_MAX 表示编程语言支持的最大整数
27    FOR {ts1, ts2} IN candidates DO
28       IF (IPI(ts1, ts2) ⩽ minIPI) THEN
29          minIPI = IPI(ts1, ts2);
30       END IF
31    END FOR
32    FOR {ts1, ts2} IN candidates DO
33       IF (IPI(ts1, ts2) > minIPI ) THEN
34          candidates.remove({ts1, ts2});
35       END IF
36    END FOR
37 END IF
38 maxtdrr = 0;
39 FOR {ts1, ts2} IN candidates DO
40    IF (TDRR({ts1, ts2}) > maxreuse) THEN
41       maxtdrr = TDRR({ts1, ts2});
42       priority.queue, clear();
43       priority_queue.push({ts1, ts2}); // priority_queue.push({a, b}) 表示将 a 和 b 的配对装入优先级队列, 并对队列中的元素根据 b 按升序排序, 若 b 相等则根据 a 按降序排序
44    ELSE IF (TDRR({ts1, ts2}) == maxreuse) THEN
```

```
45          priority_queue.push({ts1, ts2});
46        END IF
47      END IF
48 END FOR
49 {TS_1, TS_2} = priority_queue.top();
50 RETURN ({TS_1, TS_2, … , TS_n});
```

Hexagon_TSS 算法中包含两层嵌套循环, 第 1 层循环的迭代范围不超过 N_1, 第 2 层循环的迭代范围不超过 N_2, 循环内部运算的时间复杂度为常数级, 算法的时间复杂度为 $O(N_1N_2)$; 算法中最多有 $0.5N_1N_2$ 个候选解, 则候选解集合与优先级队列的最大空间开销为 $0.5N_1N_2$, 算法的空间复杂度为 $O(N_1N_2)$. 此外, 在选择目标缓存时, 应优先选择距离计算核最近、访问速度最快的 L1 数据缓存. 若 L1 数据缓存的容量不足以容纳最小的分块, 则选择 L2 缓存. 若 L2 缓存的容量仍不能容纳最小分块, 那么不考虑访存地址跨度小于缓存容量的约束条件, 在 Hexagon_TSS 算法得到的候选解中选择最小的分块大小作为最终解.

3 实验分析

本节的实验将围绕两个目标展开: (1) 验证 Hexagon_TSS 分块大小选择算法的有效性; (2) 对比结合了 Hexagon_TSS 分块大小选择算法的六边形分块方法与其他优化方法的性能表现.

3.1 实验设置与环境

首先说明实验设置和环境. 所有实验都在同一台服务器上进行, 实验涉及的具体软硬件信息如表 3 所示.

表 3 实验环境信息

指标	实验环境
CPU	2×Intel Xeon Gold 6248 2.5 GHz
微架构	Cascade Lake
核心数	2×20
SIMD指令集	Intel AVX512
缓存容量	32 KB | 1 024 KB | 28 160 KB
缓存行大小	64字节
主存	6×32 GB DDR4
操作系统	CentOS 7
编译器	GCC/G++ 10.2.0
编译选项	-O3 -mavx512f -fopenmp -ftree-vectorize -Wall -std=c++11
PLuTo版本	0.11.4
PLuTo flags	./polycc [source code] –tile --parallel
perf版本	3.10.0

本文选择 Jacobi-1d、heat-2d、heat-3d 和 seidel-2d 作为测试程序[29], 它们属于常见的 1 维、2 维和 3 维的 Jacobi 计算. 测试程序的计算访存比对循环分块的优化效果具有较大的影响, 表 4 给出了 4 种测试程序的计算访存比, 计算访存比为单次迭代的计算次数与读访存次数的比值, 可以反映不同模板计算的差异.

实验对每种测试程序都实现了 4 种优化版本, 分别为 baseline、pgram、diamond 和 hexagon, 其中 hexagon 版本用于验证本文的实验目标, 另外 3 个版本作为对照组与 hexagon 版本进行对比. baseline 版本不进行任何循环分块优化, 但基于 OpenMP 实现了多线程并行. pgram 是基于 PLuTo 算法[11]实现的平行四边形循环分块, 而 diamond 是基于 PLuTo+算法[15]实现的菱形分块, 两种版本的代码都使用 PLuTo 编译器生成. pgram 和 diamond 分块代码与本文 hexagon 分块代码一样使用空间循环最外维和时间维构成分块平面.

4 种测试程序具有不同的 Jacobi 模板和循环维度, 因此具有不同的问题规模表达格式. 为了便于问题规模的表达, 将问题规模按从小到大的顺序划分为 4 个等级. 表 5 列出了 4 种问题规模在不同 Jacobi 模板中对应的数值大小.

表 4　测试程序信息

程序名称	Jacobi模板类型	计算访存比
Jacobi-1d	1d3pt	0.33
heat-2d	2d5pt	1.20
heat-3d	3d7pt	1.29
seidel-2d	2d9pt	1.00

表 5　测试程序问题规模

问题规模	问题规模数值		
	1d	2d	3d
sizeA	40k	200×200	40×40×40
sizeB	400k	600×600	80×80×80
sizeC	4 000k	2k×2k	160×160×160
sizeD	40 000k	6k×6k	400×400×400

不同的实验可能在部分实验参数上的数值相同, 例如相同的问题规模、相同的并行线程数等. 为避免赘述实验参数, 若无特殊说明, 则每个实验参数的取值都为默认值: 数据采用 double 型, 并行线程数为 20, 问题规模为 sizeC, 时间维迭代范围为 300. 此外, 在默认情况下, hexagon 版本测试程序通过 Hexagon_TSS 算法计算得到, pgram 和 diamond 版本测试程序通过 PLuTo 得到.

实验使用 perf 采集程序访问缓存的次数以及缓存失效次数等缓存相关信息. perf 是一款基于 Linux 平台的开源 profiling 工具, 它能够通过 CPU 事件记录各级缓存的读次数和读失效次数, 将读失效次数除以读次数可得到读失效率. 虽然 perf 无法采集程序的缓存写失效次数, 但 Jacobi 计算的访存操作以读为主, 因此测试程序的读失效率接近于实际读写失效率.

3.2　Hexagon_TSS 算法有效性验证

本节将进行两方面的实验: (1) 记录和对比 baseline、pgram、diamond、hexagon 版本下 4 种测试程序的各级缓存失效率, 验证六边形分块结合 Hexagon_TSS 算法对 Jacobi 计算缓存失效率的改善作用; (2) 针对 hexagon 版本的测试程序, 对比 Hexagon_TSS 算法得到的分块大小与实际最佳分块大小对应的程序性能.

3.2.1　缓存失效率验证

表 6 列出了 4 种测试程序的缓存失效率. 由于 L3 共享缓存平均到每个计算核上的容量小于计算核私有的 L2 缓存, 并且存在争用现象, 所以 L3 缓存失效率普遍大于 L2 缓存失效率.

表 6　不同优化版本测试程序缓存失效率对比 (%)

测试程序	优化版本	缓存读失效率		
		L1 cache	L2 cache	L3 cache
Jacobi-1d	baseline	62.51	13.74	27.01
	pgram	13.39	31.75	49.34
	diamond	1.75	25.13	73.93
	hexagon	3.41	14.02	69.84
heat-2d	baseline	48.55	1.46	44.09
	pgram	47.56	1.52	28.71
	diamond	42.67	0.71	44.71
	hexagon	41.40	0.53	37.74
heat-3d	baseline	39.30	9.00	34.73
	pgram	33.42	13.25	34.84
	diamond	30.82	11.63	47.51
	hexagon	31.18	9.71	32.53
seidel-2d	baseline	38.84	1.21	43.73
	pgram	38.18	1.21	29.14
	diamond	34.35	0.67	39.24
	hexagon	33.59	0.46	41.60

表 6 中, pgram、diamond 和 hexagon 版本的 L1 数据缓存失效率都低于 baseline 版本, 在 Jacobi-1d 测试程序中尤为明显, 而 hexagon 版本在 Jacobi-1d 中 L1 数据缓存的失效率仅为 baseline 的 5.46%. 这是因为 1 维 Jacobi 可以实现访存地址跨度较小的分块, 使得针对 1 维 Jacobi 计算的循环分块优化能够显著地降低缓存失效率. 整体上 diamond 版本的程序达到了最低的 L1 缓存失效率, 这是因为菱形分块是最具局部性收益的分块形状, 能够最大化分块内的时间维重用次数. hexagon 版本在 L1 缓存失效率上相比于 diamond 版本略有不足, 但在 L2 缓存失效率上表现相对较好. 实验表明, 六边形分块结合 Hexagon_TSS 分块大小选择算法可显著降低缓存失效率, 其效率可与目前较先进的 diamond 的分块方法相当, 甚至略优于 diamond 的分块方法.

3.2.2 Hexagon_TSS 效率验证

Hexagon_TSS 算法的效率指以使用 Hexagon_TSS 算法得到的分块大小对应的计算速率除以遍历搜索空间得到最佳分块大小 Best_Hexagon 对应的计算速率. 其中, Best_Hexagon 是使程序计算速率最优的分块大小, 其值通过线性规划确定搜索空间, 然后遍历空间内的所有解进行测试获得. 由于遍历并获得最佳分块大小的过程需要巨大的时间开销, 本实验仅以计算访存比适中的 seidel-2d 作为测试程序. 表 7 列出了使用 Hexagon_TSS 算法为 hexagon 版本的 seidel-2d 测试程序计算分块大小时, 在 4 种问题规模下的效率.

表 7 Hexagon_TSS 算法效率

问题规模	Hexagon_TSS分块大小 (TS_1×TS_2)	Hexagon_TSS计算速率 (GFLOPS)	Best_Hexagon计算速率 (GFLOPS)	Hexagon_TSS efficiency (%)
sizeA	10×9	13.42	14.83	90.47
sizeB	30×29	85.27	129.45	65.87
sizeC	16×32	216.70	222.75	97.29
sizeD	10×10	116.62	117.55	99.21
平均	—	—	—	88.21

表 7 中, Hexagon_TSS 算法效率随问题规模的增大而上升, 在大多数问题规模下都能达到 90% 以上. 但 Hexagon_TSS 算法在问题规模为 sizeB 时的效率明显低于其他问题规模. 这是因为对于 2 维 Jacobi 计算, 在 sizeB 的问题规模下 Hexagon_TSS 算法计算的分块大小使六边形分块的访存地址跨度超过了 L1 缓存的容量, 程序跨过了 L1 缓存性能悬崖 (cache performance cliffs); 但分块的访存地址跨度又远没有达到 L2 缓存的容量, 分块大小相对于 L2 缓存容量偏小, 未能最大化分块的时间维重用次数. 综合来看, Hexagon_TSS 算法在绝大多数问题规模下都具有良好的效率, 但在个别问题规模上仍然存在改进空间.

3.3 程序性能测试

本节将进行两次控制变量实验, 在默认实验参数下改变某一项实验参数, 记录各个优化版本对应的测试程序的浮点计算速率: (1) 改变并行线程数; (2) 改变问题规模.

3.3.1 不同并行线程数下性能对比

设置 4 种并行线程数: 10、20、30、40, 记录 baseline、pgram、diamond、hexagon 版本的测试程序在不同并行线程数下的浮点计算速率, 其结果在表 8 中展示.

表 8 中, 最高计算速率为 287. 79 GFLOPS, 对应测试程序为 seidel-2d, 优化方法为 diamond, 并行线程数为 30. 用 pgram、diamond、hexagon 版本的计算速率分别除以 baseline 版本的计算速率, 得到 pgram、diamond、hexagon 版本相对于 baseline 版本的加速比, 结果如图 6 所示. 其中横轴表示并行线程数, 纵轴表示相对于 baseline 版本的加速比. 由图 6 可知, 最大加速比为 8.78, 对应的测试程序为 Jacobi-1d, 优化方法为 hexagon, 并行线程数为 20. 在 4 种测试程序中, hexagon 版本在 40 线程下的加速比与单核下的加速比的比值都达到了 50%, 表明 hexagon 方法具有良好的并行可扩展性.

循环分块的加速效果总体上随着线程数的增加趋于下降, 且 hexagon 方法优势变小, 主要原因是随着线程

数增多, 程序执行的并行度饱和, 主要性能瓶颈转变为访存, 增加线程数获得的性能收益减小, 同时线程同步开销逐渐增大, 因此整体呈下降趋势. 而从表 6 结果显示, hexagon 方法在访存方面的效率仅略优于其他分块方法, 当主要性能瓶颈逐渐转变为访存后, hexagon 相比与其他方法的优势就减少了. 特别地, Jacobi-1d 测试程序中 diamond 和 hexagon 版本的加速效果则表现为先上升后下降, 这是因为循环分块对 1 维 Jacobi 计算具有较好的访存优化效果, Jacobi-1d 程序在线程数较少时主要性能瓶颈是并行度, 此时增加线程数有利于提升加速效果; 当 Jacobi-1d 程序的并行度饱和后, 主要性能瓶颈转变为访存, 增加线程数获得的性能收益减小, 同时线程同步开销逐渐增大.

表 8　不同并行线程数下各优化版本测试程序的计算速率

测试程序	优化版本	计算速率 (GFLOPS)			
		10线程	20线程	30线程	40线程
Jacobi-1d	baseline	6.20	9.40	21.34	22.55
	pgram	18.38	24.58	21.23	18.94
	diamond	29.64	54.08	74.91	94.41
	hexagon	46.06	82.51	127.03	124.80
heat-2d	baseline	39.71	74.24	99.45	108.06
	pgram	43.64	71.79	87.02	98.11
	diamond	100.52	161.68	218.69	212.33
	hexagon	103.56	168.23	217.02	200.21
heat-3d	baseline	41.63	44.12	44.23	35.93
	pgram	12.82	20.26	29.87	31.01
	diamond	78.92	70.14	51.01	39.92
	hexagon	104.53	78.24	55.92	34.67
seidel-2d	baseline	51.75	87.87	130.05	150.78
	pgram	53.40	89.90	107.11	126.28
	diamond	122.79	209.58	287.79	271.15
	hexagon	133.22	217.18	276.85	277.78

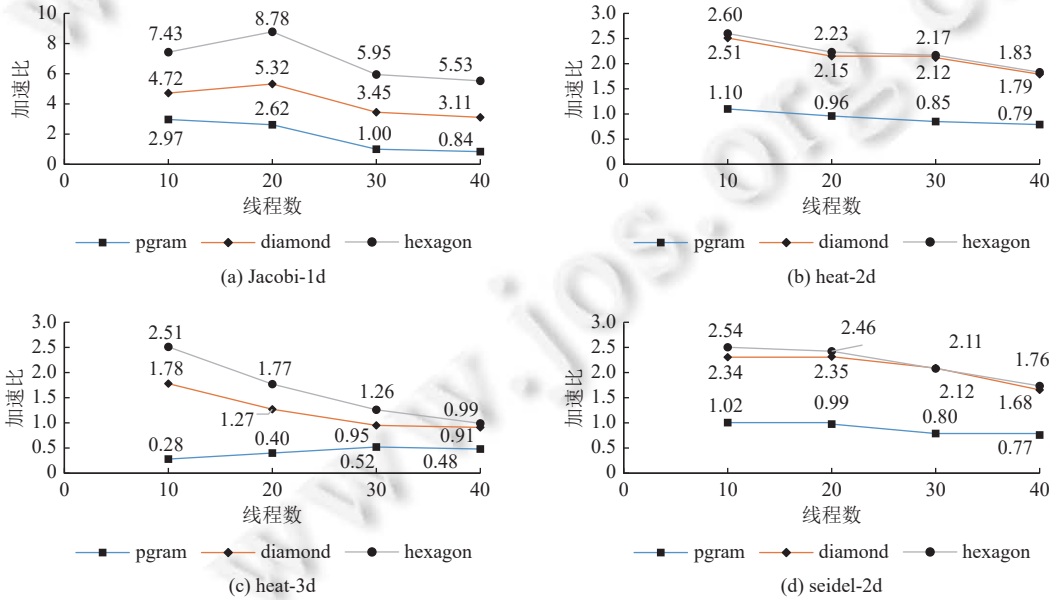


图 6　不同并行线程数下测试程序加速效果

此外, 从图 6 中可发现, diamond 和 hexagon 版本的加速效果明显优于 pgram 版本, 表明菱形分块和六边形分块相比于平行四边形分块在 Jacobi 计算优化上具有更好的性能表现. 而 hexagon 版本在 Jacobi-1d 测试程序中的加速效果明显优于 diamond 版本, 在其他测试程序中的加速效果也略微优于 diamond 版本或与其基本一致, 这主要是因为 Hexagon_TSS 算法针对 1 维 Jacobi 进行了数据对齐优化, 使得六边形分块在 1 维 Jacobi 计算上相对于菱形分块具有更高的向量化效率和更低的 *IPI*.

注意到循环分块优化在 4 种测试程序中具有不同级别的加速效果, 在 Jacobi-1d 上最佳, 其次为 seidel-2d 和 heat-2d, 然后是 heat-3d. 循环分块优化的加速效果与程序本身的计算访存比有关, 计算访存比越小, 那么程序的访存作为性能瓶颈越突出, 循环分块的优化效果也越好.

3.3.2 不同问题规模下性能对比

设置 4 种问题规模: sizeA、sizeB、sizeC、sizeD, 记录 baseline、pgram、diamond、hexagon 版本的测试程序在不同问题规模下的浮点计算速率, 其结果在表 9 中展示. 表 9 中, 最高计算速率为 211.40 GFLOPS, 对应测试程序为 seidel-2d, 优化方法为 hexagon, 问题规模为 sizeC.

表 9 不同问题规模下各优化版本测试程序的计算速率

测试程序	优化版本	计算速率 (GFLOPS)			
		sizeA	sizeB	sizeC	sizeD
Jacobi-1d	baseline	2. 22	12. 32	11. 82	3. 38
	pgram	14. 30	17. 60	26. 46	32. 58
	diamond	17. 17	32. 34	52. 78	51. 81
	hexagon	22. 96	49. 23	79. 62	82. 76
heat-2d	baseline	12. 89	48. 23	73. 41	20. 01
	pgram	25. 75	35. 61	68. 40	84. 05
	diamond	8. 78	42. 78	162. 61	77. 85
	hexagon	9. 21	64. 75	164. 96	77. 93
heat-3d	baseline	12. 61	38. 36	44. 74	21. 55
	pgram	6. 90	11. 61	19. 05	14. 89
	diamond	8. 04	28. 65	76. 06	40. 09
	hexagon	8. 27	31. 59	90. 04	39. 95
seidel-2d	baseline	17. 78	55. 32	96. 31	29. 47
	pgram	31. 14	39. 26	86. 14	102. 92
	diamond	13. 14	62. 48	204. 77	120. 71
	hexagon	14. 30	82. 24	211. 40	116. 47

根据表 9 给出的数据计算 pgram、diamond、hexagon 版本相对于 baseline 版本的加速比, 加速比随问题规模变化的折线趋势如图 7 所示. 其中横轴表示问题规模, 纵轴表示相对于 baseline 版本的加速比. 由图 7 可知, hexagon 版本优化的最大加速比为 24.48, 对应的测试程序为 Jacobi-1d, 问题规模为 sizeD. hexagon 版本优化在 Jacobi-1d 程序中具有最佳的加速效果, 在其他测试程序中加速效果与 diamond 版本基本一致. pgram 版本优化在 2 维 Jacobi 计算的小问题规模上加速效果优于 diamond 和 hexagon 版本.

整体上, 循环分块在 Jacobi-1d 测试程序中具有最佳的加速效果, 在 heat-3d 测试程序中的加速效果最低, pgram 版本在 heat-3d 程序中所有问题规模都表现出负优化的效果. 循环分块的加速效果随问题规模的扩大而上升, 但加速效果的上升趋势在问题规模达到 sizeD 时放缓, 主要是由于问题规模的扩大导致程序的主要性能瓶颈从计算转换到访存. 特别地, 在 2 维 Jacobi 计算中循环分块的加速效果在问题规模达到 sizeD 后出现明显下降, 这是因为在 2 维及以上的 Jacobi 计算中, 分块大小随着问题规模而增大, 当问题规模超过 sizeD 后, 分块基本失去了局部性收益. 理论上 3 维 Jacobi 计算在问题规模达到 sizeD 后分块局部性收益应出现明显下降, 但由于循环分块在 3 维 Jacobi 程序上的加速效果本就不佳, 所以实验中没有表现出加速效果的明显下降.

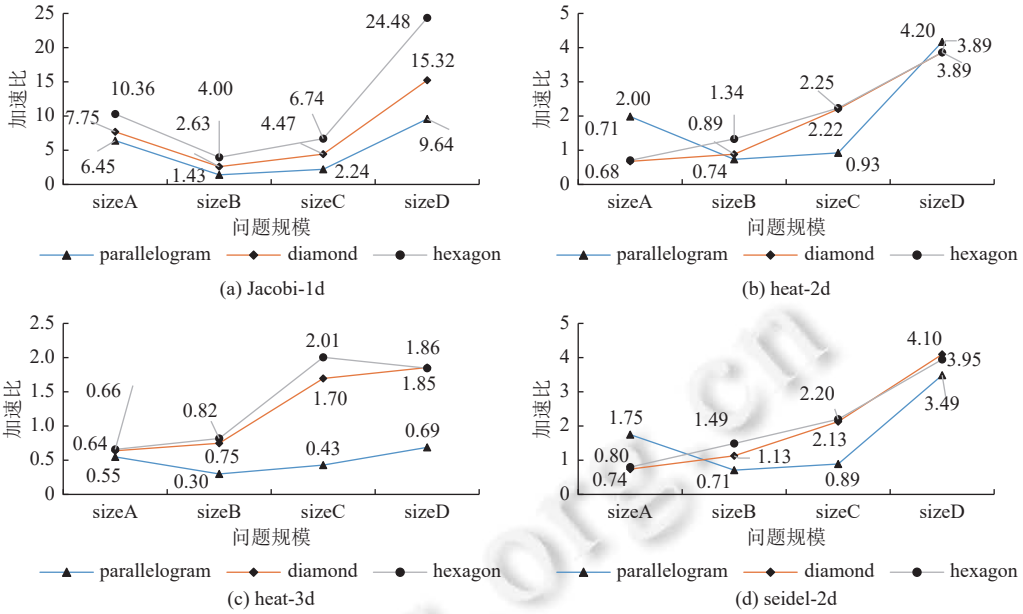


图 7 不同问题规模下测试程序加速效果

4 总 结

Jacobi 计算的性能热点是嵌套循环, 且具有可观的数据重用, 采用循环分块的优化方法能够显著地提升 Jacobi 计算的性能. 分块大小对循环分块的效果具有重要的影响, 现有关于分块大小的研究大多以分块局部性和并行性为优化目标. 但随着处理器技术的发展, 分块大小对向量化效率、计算负载均衡等方面的影响逐渐变得不可忽视. 针对上述问题, 本文选择六边形分块作为 Jacobi 计算的优化方法, 通过设计和实现一种六边形分块大小选择算法 Hexagon_TSS, 提升 Jacobi 计算的局部性, 充分利用多核处理器的计算能力, 从而提高 Jacobi 计算的浮点运算速率.

本文工作还存在可优化的空间. 例如, 本文设计和实现的六边形循环分块优化方法并非在各种测试环境下都能达到最优的性能表现, 现有的其他优化方法在部分环境下的效果优于六边形循环分块. 因此, 设计一种优化策略分支选择模型, 根据运行环境选择最佳的优化方案将是本文未来研究工作. 此外, 在实际应用中, 影响分块性能效果的因素非常复杂, 难以建立精确的数学模型来求解最佳分块大小. 本文提出的 Hexagon_TSS 算法是一种静态分块大小选择算法, 可以获得接近最佳分块大小的性能. 采用机器学习方法, 通过训练深度神经网络来求解最佳分块参数是当前的一个热门研究方向, 但这需要特征提取方面的工作, 并且需要大量的训练数据集. 因此将机器学习方法与静态建模方法相结合, 将会是分块大小选择算法的下一步研究方向.

References:

[1] Stoltzfus L, Hagedorn B, Steuwer M, Gorlatch S, Dubach C. Tiling optimizations for stencil computations using rewrite rules in lift. ACM Trans. on Architecture and Code Optimization, 2019, 16(4): 52. [doi: 10.1145/3368858]

[2] Levchenko V, Zakirov A, Perepelkina A. GPU implementation of conetorre algorithm for fluid dynamics simulation. In: Proc. of the 15th Int'l Conf. on Parallel Computing Technologies. Almaty: Springer, 2019. 199–213. [doi: 10.1007/978-3-030-25636-4_16]

[3] Pouchet LN, Bondhugula U, Bastoul C, Cohen A, Ramanujam J, Sadayappan P, Vasilache N. Loop transformations: Convexity, pruning and optimization. In: Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Austin: ACM, 2011. 549–562. [doi: 10.1145/1926385.1926449]

[4] Liu S, Wu WG, Zhao B, Jiang Q. Loop tiling for optimization of locality and parallelism. Journal of Computer Research and Development, 2015, 52(5): 1160–1176 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2015.20131387]

[5] Pop S, Cohen A, Bastoul C, Girbal S, Silber GA, Vasilache N. GRAPHITE: Polyhedral analyses and optimizations for GCC. In: Proc. of

the 4th GCC Developper's Summit. Ottawa, 2006. 1–18.

[6] Sun XH, Liu YH. Utilizing concurrency: A new theory for memory wall. In: Proc. of the 29th Int'l Workshop on Languages and Compilers for Parallel Computing. Rochester: Springer, 2017. 18–23. [doi: 10.1007/978-3-319-52709-3_2]

[7] Xue JL. Loop Tiling for Parallelism. 2nd ed., New York: Springer, 2012.

[8] Neumaier A. Introduction to Numerical Analysis. Cambridge: Cambridge University Press, 2001.

[9] Chen GL. Parallel Computing: Architecture, Algorithm and Programming. 3rd ed., Beijing: Higher Education Press, 2011 (in Chinese).

[10] Zhao J, Li YY, Zhao RC. "Black magic" of polyhedral compilation. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2371−2396 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5563.htm [doi: 10.13328/j.cnki.jos.005563]

[11] Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. In: Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Tucson: ACM, 2008. 101–113. [doi: 10.1145/1375581.1375595]

[12] Grosser T, Cohen A, Kelly PHJ, Ramanujam J, Sadayappan P, Verdoolaege S. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In: Proc. of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. Houston: ACM, 2013. 24–31. [doi: 10.1145/2458523.2458526]

[13] Grosser T, Cohen A, Holewinski J, Sadayappan P, Verdoolaege S. Hybrid hexagonal/classical tiling for GPUs. In: Proc. of the 2014 Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. Orlando: ACM, 2014. 66–75. [doi: 10.1145/2544137.2544160]

[14] Grosser T, Verdoolaege S, Cohen A, Sadayappan P. The relation between diamond tiling and hexagonal tiling. Parallel Processing Letters, 2014, 24(3): 1441002. [doi: 10.1142/S0129626414410023]

[15] Bondhugula U, Bandishti V, Pananilath I. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. IEEE Trans. on Parallel and Distributed Systems, 2017, 28(5): 1285–1298. [doi: 10.1109/TPDS.2016.2615094]

[16] Wolf ME, Lam MS. A data locality optimizing algorithm. In: Proc. of the 1991 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Toronto: ACM, 1991. 30–44. [doi: 10.1145/113445.113449]

[17] Liu S, Zhao B, Jiang Q, Wu WG. A semi-automatic coarse-grained parallelization approach for loop optimization and irregular code sections. Chinese Journal of Computers, 2017, 40(9): 2127–2147 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2017.02127]

[18] Whaley RC, Petitet A, Dongarra JJ. Automated empirical optimizations of software and the ATLAS project. Parallel Computing, 2001, 27(1–2): 3–35. [doi: 10.1016/S0167-8191(00)00087-9]

[19] Wang Z, O'Boyle M. Machine learning in compiler optimization. Proc. of the IEEE, 2018, 106(11): 1879–1901. [doi: 10.1109/JPROC.2018.2817118]

[20] Liu H, Xu JL, Zhao RC, Yao JY. Compiler optimization sequence selection method based on learning model. Journal of Computer Research and Development, 2019, 56(9): 2012–2026 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2019.20180789]

[21] Xiang XY, Ding C, Luo H, Bao B. HOTL: A higher order theory of locality. In: Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Houston: ACM, 2013. 343–356. [doi: 10.1145/2451116.2451153]

[22] Sabarimuthu JM, Venkatesh TG. Analytical miss rate calculation of L2 cache from the RD profile of L1 cache. IEEE Trans. on Computers, 2018, 67(1): 9–15. [doi: 10.1109/TC.2017.2723878]

[23] Di P, Hu CJ, Li JJ. Research and implementation of effective Jacobi iteration algorithms on GPU. Journal of Chinese Computer Systems, 2012, 33(9): 1962–1967 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2012.09.018]

[24] Liu S, Cui YZ, Zou NJ, Zhu WH, Zhang D, Wu WG. Revisiting the parallel strategy for DOACROSS loops. Journal of Computer Science and Technology, 2019, 34(2): 456–475. [doi: 10.1007/s11390-019-1919-7]

[25] Li YZ, Schwiebert L. Memory-optimized wavefront parallelism on GPUs. Int'l Journal of Parallel Programming, 2020, 48(6): 1008–1031. [doi: 10.1007/s10766-020-00658-y]

[26] Assis ÍAS, Fernandes JB, Barros T, Xavier-De-Souza S. Auto-tuning of dynamic scheduling applied to 3D reverse time migration on multicore systems. IEEE Access, 2020, 8: 145115–145127. [doi: 10.1109/ACCESS.2020.3015045]

[27] Huang PH, Chen YS, Liao JH. QT-adaptation engine: Adaptive QoS-aware scheduling and governing in thermally constrained mobile devices. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2020, 39(3): 585–598. [doi: 10.1109/TCAD.2019.2897697]

[28] Guo Y, Zhao JS, Cave V, Sarkar V. SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Bangalore: ACM, 2010. 341–342. [doi: 10.1145/1693453.1693504]

[29] Karimov J, Rabl T, Markl V. PolyBench: The first benchmark for polystores. In: Proc. of the 10th TPC Technology Conf. on Performance

Evaluation and Benchmarking. Rio de Janeiro: Springer, 2019. 24–41. [doi: 10.1007/978-3-030-11404-6_3]

附中文参考文献:

[4] 刘松, 伍卫国, 赵博, 蒋庆. 面向局部性和并行优化的循环分块技术. 计算机研究与发展, 2015, 52(5): 1160−1176. [doi: 10.7544/issn1000-1239.2015.20131387]

[9] 陈国良. 并行计算: 结构·算法·编程. 第3版, 北京: 高等教育出版社, 2011.

[10] 赵捷, 李颖颖, 赵荣彩. 基于多面体模型的编译“黑魔法”. 软件学报, 2018, 29(8): 2371−2396. http://www.jos.org.cn/1000-9825/5563.htm [doi: 10.13328/j.cnki.jos.005563]

[17] 刘松, 赵博, 蒋庆, 伍卫国. 一种面向循环优化和非规则代码段的粗粒度半自动并行化方法. 计算机学报, 2017, 40(9): 2127–2147. [doi: 10.11897/SP.J.1016.2017.02127]

[20] 刘慧, 徐金龙, 赵荣彩, 姚金阳. 学习模型指导的编译器优化顺序选择方法. 计算机研究与发展, 2019, 56(9): 2012–2026. [doi: 10.7544/issn1000-1239.2019.20180789]

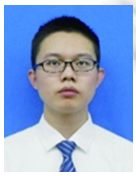
[23] 狄鹏, 胡长军, 李建江. GPU上高效Jacobi迭代算法的研究与实现. 小型微型计算机系统, 2012, 33(9): 1962–1967. [doi: 10.3969/j.issn.1000-1220.2012.09.018]

屈彬(1996−), 男, 硕士, 主要研究领域为程序性能优化, 并行计算, 编译优化.

马洁(1997−), 女, 硕士, 主要研究领域为并行计算, 任务调度.

刘松(1987−), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为程序性能优化, 计算机体系结构, 编译优化.

伍卫国(1963−), 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架构, 海量存储系统, 计算机网络, 嵌入式系统.



张增源(1998−), 男, 硕士, 主要研究领域为程序性能优化, 并行计算.