软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(8):2289−2306 [doi: 10.13328/j.cnki.jos.006002]

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

异构 HPL 算法中 CPU 端高性能 BLAS 库优化*

蔡 雨, 孙成国, 杜朝晖, 刘子行, 康梦博, 李双双

(信息技术有限公司,江苏 苏州 215000)

通讯作者: 孙成国, E-mail: sunchengguo1@163.com

摘 要: 异构 HPL(high-performance Linpack)效率的提高需要充分发挥加速部件和通用 CPU 计算能力,加速部件集成了更多的计算核心,负责主要的计算,通用 CPU 负责任务调度的同时也参与计算.在合理划分任务、平衡负载的前提下,优化 CPU 端计算性能对整体效率的提升尤为重要.针对具体平台体系结构特点对 BLAS(basic linear algebra subprograms)函数进行优化往往可以更加充分地利用通用 CPU 计算能力,提高系统整体效率.BLIS(BLAS-like library instantiation software)算法库是开源的 BLAS 函数框架,具有易开发、易移植和模块化等优点.基于异构系统平台体系结构以及 HPL 算法特点,充分利用三级缓存、向量化指令和多线程并行等技术手段优化 CPU 端调用的各级 BLAS 函数,应用 auto-tuning 技术优化矩阵分块参数,从而形成了异构环境下优化的 BLIS 算法库 HBLIS.与 MKL 相比,HPL 整体性能提高了 11.8%.

关键词: BLAS;遗传算法 auto-tuning;向量化指令;数据预取;多线程并行

中图法分类号: TP303

中文引用格式: 蔡雨,孙成国,杜朝晖,刘子行,康梦博,李双双.异构 HPL 算法中 CPU 端高性能 BLAS 库优化.软件学报,2021, 32(8):2289−2306. http.//www.jos.org.cn/1000-9825/6002.htm

英文引用格式: Cai Y, Sun CG, Du ZH, Liu ZX, Kang MB, Li SS. CPU-side high performance BLAS library optimization in heterogeneous HPL algorithm. Ruan Jian Xue Bao/Journal of Software, 2021,32(8):2289−2306 (in Chinese). http.//www.jos.org.cn/1000-9825/6002.htm

CPU-side High Performance BLAS Library Optimization in Heterogeneous HPL Algorithm

CAI Yu, SUN Cheng-Guo, DU Zhao-Hui, LIU Zi-Xing, KANG Meng-Bo, LI Shuang-Shuang

(Information Technology Co., Ltd., Suzhou 215000, China)

Abstract: Improving the efficiency of heterogeneous HPL needs to fully utilize the computing power of acceleration components and CPU, the acceleration components integrate more computing cores and are responsible for the main calculation. The general CPU is responsible for task scheduling and also participates in calculation. Under the premise of reasonable division of tasks and load balancing, optimizing CPU-side computing performance is particularly important to improve overall efficiency. Optimizing the basic linear algebra subprogram (BLAS) functions for specific platform architecture characteristics can often make full use of general-purpose CPU computing capabilities to improve the overall system efficiency. The BLIS (BLAS-like library instantiation software) algorithm library is an open source BLAS function framework, which has the advantages of easy development, portability, and modularity. Based on the heterogeneous system platform architecture and HPL algorithm characteristics, this study uses three-level cache, vectorized instructions, and multi-threaded parallel technology to optimize the BLAS functions called by the CPU, applies auto-tuning technology to optimize the matrix block parameters, and eventually forms the optimized BLIS algorithm library in heterogeneous environment. Compared with MKL, the overall performance of the HPL using the optimized HBLIS has been improved by 11.8%.

Key words: BLAS; genetic algorithm auto-tuning; vectorization instruction; data prefetching; multi-threading parallelization

BLAS(basic linear algebra subprograms)是基本线性代数子程序的缩写,是目前应用广泛的核心线性代数数

* 本文由“国产复杂异构高性能数值软件的研制与测试”专题特约编辑孙家昶研究员、李会元研究员推荐.
收稿时间: 2019-07-25; 修改时间: 2019-12-05, 2020-01-22, 2020-03-19; 定稿时间: 2020-03-27

学库.随着不断的发展完善,BLAS 在高性能计算、科学和工程领域都得到了广泛的应用.由于不同厂商、不同体系结构下的 CPU 差异较大,通用 BLAS 函数库不能达到很好的性能.因此,在大规模矩阵运算时,针对不同 CPU 的架构特点来优化 BLAS,可以充分发挥处理器的计算性能,从而使计算效率大大提升.目前主流的 BLAS 分为专用和通用两种.专用型 BLAS 是由特定的 CPU 公司开发,针对特定平台芯片进行代码优化,极大地提高了性能,Intel 的 MKL 和 AMD 的 ACML 就是此种类型.通用 BLAS 由非盈利机构开发,不同平台都可以对开源代码进行优化,最具代表性的是 ATLAS[1]、GotoBLAS[2,3]和 OpenBLAS[4,5].

BLIS(BLAS-like library instantiation software)数学库是美国 Texas 大学超算中心高性能组开发的开源架构,是在 GotoBLAS 的基础上进行了改写和优化,具有可移植性、易用性和模块化设计易于开发的特点,支持混合类型矩阵存储和多线程[6].文献[7]提出了 BLIS 架构,并做了全面的介绍;文献[8]研究了如何使用 BLIS 在通用、低功耗和多核架构中实现高效的 level-3 级 BLAS 函数;文献[9]系统地探讨了矩阵乘法算法在 BLIS 五层循环中并行化的机会.BLIS 具有的这些特性使其更容易针对具体平台进行 BLAS 优化开发.BLAS 包含三级函数,分别为第 1 级向量与向量计算,第 2 级向量与矩阵计算,第 3 级矩阵与矩阵的计算,其中第 3 级函数 DGEMM 计算量最大,可以达到理论计算量的 95%以上[10],因此三级函数的优化是性能提升的关键.

高性能计算系统可以分为同构或异构两类,同构系统是由大量 CPU 构建,计算和数据处理都在 CPU 上进行;异构系统是 CPU 和加速部件组成,加速部件一般由 Cell(cell broadband engine architecture)、FPGA(Field programmable gate array)和 GPU(graphic processing unit)等为代表,集成大量的浮点计算单元的同时舍弃了一些 CPU 上复杂的控制单元.异构系统中 CPU 负责任务调度和简单计算,主要计算在加速部件上进行,计算能力更强.国际上对高性能系统的评测标准程序是 HPL(high performance Linpack),由美国田纳西大学教授 Dongarra 设计提出[11],是在高性能计算系统中使用高斯消元法求解一个随机的 N 元一次线性方程组问题,用以评价高性能计算机的浮点性能.

异构 HPL 浮点计算在 CPU 端是通过 BLAS 函数完成的,BLAS 库的性能会影响整体系统性能.本文异构系统计算单元由 CPU+协处理器(co-processor)组成(如图 1 所示).CPU 计算核心之间和协处理器通过高速互联进行数据交换和通信.协处理器的计算能力一般都比通用 CPU 强大很多,往往造成计算负载严重不均衡,损失性能.因此在 CPU 端利用更高效的数学库,能够充分地发挥 CPU 性能,使负载更均衡,是提高计算能力的可行方式之一.文献[12−15]分别针对 ARM 架构处理器以及异构平台的 GEMM 优化做了深入研究.文献[16,17]研究了在申威众核处理器上,1 级、2 级和 3 级 BLAS 的优化方法,文献[18]对异构多核平台上的数学库寄存器分配优化方法进行了研究.

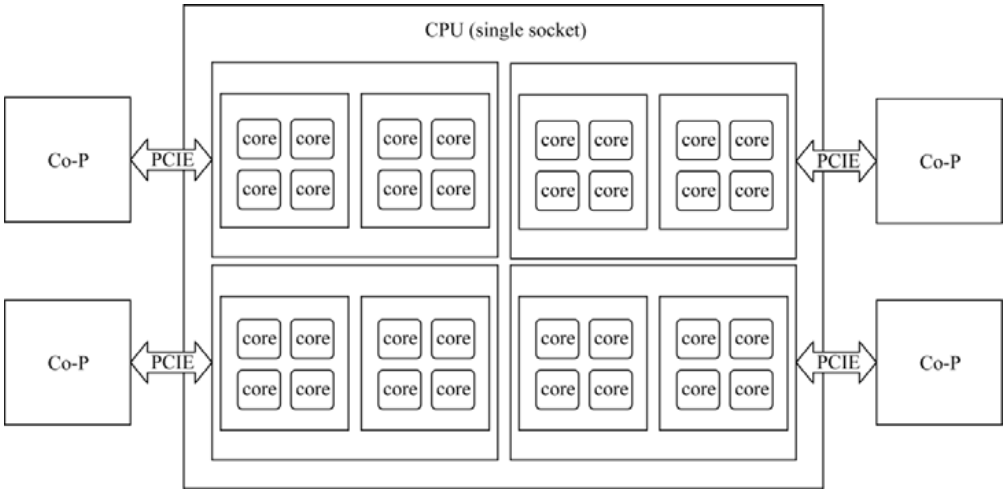

Fig.1 Heterogeneous computing node architecture

图 1 异构计算节点架构

本文在 BLIS 算法库基础上,分析异构 HPL 算法中调用的各级 BLAS 函数,针对体系结构特点,利用访存优化、指令集优化和多线程并行等技术手段形成了优化的 HBLIS 库,提高了异构系统中 CPU 端的计算效率,后文详细分析了优化策略和方法.本文第 1 节首先对异构 HPL 的 BLIS 库进行分析,特别是异构环境的 HPL 特点以及各个数学库函数调用关系.第 2 节~第 4 节对高性能 BLIS 库的优化策略做详细的分析和介绍,优化 HPL 中涉及的各级 BLAS 函数,实现高性能的 HBLIS 算法库.第 5 节在异构平台上做详细的性能测试与分析,通过性能对比,验证优化的 HBLIS 库性能有大幅度的提高.第 6 节对全文进行总结和展望.

1 算法分析和 CPU 微体系结构

1.1 HPL算法与BLAS函数

HPL 算法核心计算公式如式(1)所示,其求解过程实际是对系数矩阵[*A*,*b*]使用高斯消元法进行 LU 分解,变成[*A*,*b*]=[[*L*,*U*],*Y*],其中 *L* 是下三角矩阵,*U* 为上三角矩阵,随着因式分解的进行,方程等价于求解 *Ux*=*Y*.LU 分解过程中每一次迭代计算都要经过 panel 分解、panel 广播、行交换和尾矩阵更新.HPL 程序在进程通信和矩阵计算上需要 MPI[19]库和 BLAS 或 VSIPL[20]库的支持,除基本算法不可以改变外,可以通过配置文件调节问题规模 *N*(矩阵大小)、进程数等测试参数,以及使用各种优化方法来执行 HPL 程序,以获取最佳的性能.当求解问题规模为 *N* 时,浮点运算次数为 $2/3\times N^3-2\times N^2$.因此,给出问题规模 *N*,测得系统计算时间 *T*,计算系统的浮点计算能力=计算量$(2/3\times N^3-2\times N^2)$/计算时间(*T*),测试结果以每秒浮点运算次数(FLOPS)为单位.

$$Ax=b(A\in R^{n\times n},x\in R^n,b\in R^n) \tag{1}$$

HPL 算法过程分为 LU 分解和回代过程.异构 HPL 环境下,LU 分解过程中大量计算要在 CPU 和协处理器之间进行任务分配,CPU 负责调度和适当的辅助计算,而主要计算要在加速部件上进行.当问题规模 *N* 非常大时,HPL 的最大计算量是更新尾矩阵的计算,包括双精度的三角矩阵求解和稠密矩阵乘 DGEMM,因此将 DGEMM 放在协处理器上计算,CPU 负责计算量不大的 panel 分解、panel 广播和行交换.异构高性能计算系统优化的难点主要是在主从芯片对计算任务的划分和通信开销这两个方面,一些学者从任务静态划分和动态划分实现负载均衡[12,21−23],还有学者在数据重用、存储优化、软件流水和应用双缓冲等方面深入研究[24−26],达到实现更高效的通信隐藏的目的.异构 HPL 主程序算法中任务分配调用加速部件专用编程接口进行计算,CPU 端则调用 BLAS 库进行选主元、panel 分解计算.

本文针对的异构 HPL 算法由中国科学院软件研究所开发[27],其 LU 分解过程会将 panel 的一部分矩阵(*N*×*NBMIN*)分解为单位下三角 *L*1、上三角 *U* 和 *L*2,这 3 个部分的运算在 CPU 上进行,主要涉及一级 BLAS 函数中的 DCOPY、IDMAX、DSCAL;二级 BLAS 函数中的 DTRSM、DTRSV、DGEMV;三级 BLAS 函数中的 DGEMM.各个 BLAS 函数作用如下:

(1) DCOPY 执行数据拷贝操作,将一个向量拷贝到另一个向量.

(2) IDMAX 用于选主元操作,在一列元素中选择最大的元素.

(3) DSCAL 用于向量更新,用一个标量乘以一个向量.

(4) DTRSM 用于三角矩阵求解操作.

(5) DTRSV 用于求解三角方程操作.

(6) DGEMV 用于计算向量矩阵乘法.

(7) DGEMM 用于每个进程上剩余矩阵数据更新.

同时,panel 分解的递归部分也会调用三级 BLAS 函数 DGEMM 加速 panel 分解过程.因此 DGEMM 的效率极大地影响了 panel 分解的效率.在异构 HPL 算法中,参数 *NBMIN* 表示递归分解算法可以分解到的最小方阵大小,随着它的增大,一级 BLAS 在整体过程中的操作次数与规模不变,二级 BLAS 操作次数保持不变,而运算规模将增加,导致 panel 分解非递归部分的整体时间延长.当 *NBMIN* 减小时,HPL_pdpanllN[28]函数的耗时将减少,但是也会增加外层非递归部分对三级 BLAS 的调用次数.因此需要通过权衡三级与二级 BLAS 的调用次数与效率来调整 *NBMIN* 的大小,获得更好的性能.另一方面,通过 DGEMV 的优化可以增大 *NBMIN* 来减少三级 BLAS 的

调用次数.针对这些函数,本文应用多种优化策略对BLIS库中的各级BLAS函数进行了优化,获得了性能的大幅度提升,提高了异构 HPL 的效率,更好地发挥了异构系统的性能.

1.2 异构单元处理器概述

异构计算单元 CPU 是 64 位 X86 架构的通用服务器芯片,采用 14nm 工艺制造.从图 1 中可以简单地了解 CPU 架构,共 32 个计算核心.其基本的微体系结构如下:

(1) 支持所有标准 X86 指令集,如 AVX、AVX2、SMEP、SSE 和 SSE2 等.

(2) 每 4 个核心组成一组,每个核心独享 *L*1 和 *L*2 级缓存,共享 *L*3 级缓存,支持 SMT 技术,每个物理核心支持同步 2 线程.

(3) 三级缓存结构如图 2 所示,*L*1 cache 由 64K 指令缓存和 32K 数据缓存组成,其中 *L*2 cache 大小为 512K,共享 *L*3 cache 大小为 8M.所有 Cache line 大小为 64 字节.

(4) X86 处理器有三级指令 TLB 和两级数据 TLB.访存页面大小为 4KB,映射的物理地址空间最大可以达到 1GB.

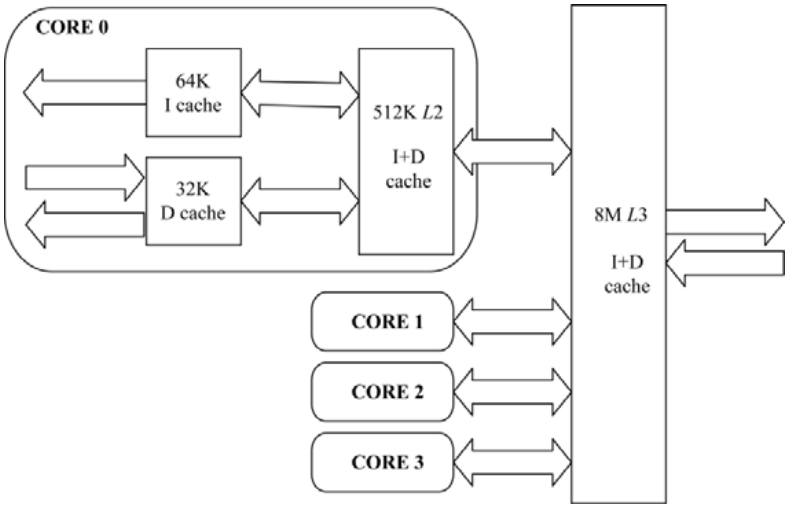

Fig.2 CPU cache hierarchy

图 2 CPU 缓存层次结构

2 访存优化

HPL 算法中 DGEMM 运算占据了 BLAS 运算中最大的计算量和访存耗时.优化的 HBLIS 通过实现互补的两类 GEMM kernels 优化了 DGEMM 性能,即大 kernel 和小 kernel.大 kernel 用于处理矩阵 *A* 和 *B* 的规模都相对较大的情况,而小 kernel 用于处理矩阵 *A* 更加“高窄”和矩阵 *B* 规模较小的情况.在 DGEMM 函数接口内根据矩阵规模切换两类 kernel.同时引入 Auto-tuning 自适应调优技术对矩阵分块参数进行优化,优化后的分块参数可以更充分地利用 CPU 缓存,提高 cache 命中率,从而提高 DGEMM 访存性能.

2.1 矩阵分块

GEMM 大 kernel 采用经典高效的 GEMM 分块算法[8],如图 3 左侧的 6 层循环伪代码所示;GEMM 小 kernel 的算法如图 3 右侧的伪代码所示.

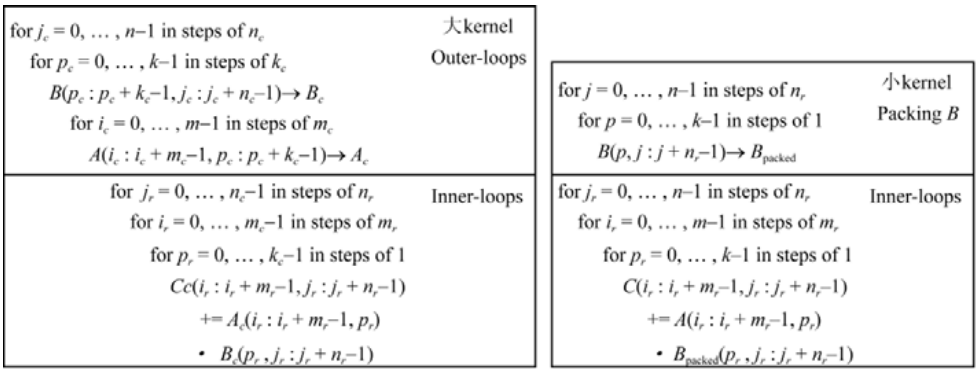

Fig.3 High performance GEMM algorithm of big and little kernels

图 3 高性能 GEMM 的大 kernel 算法和小 kernel 算法

大 kernel 算法描述如下:

- 最外层 Loop1 遍历 n 维度(以 j_c 索引),以宽度 n_c 为单位,对矩阵 C 和矩阵 B 进行分块,形成列向 panels.
- 第 2 层 Loop2 遍历 k 维度(以 p_c 索引),以高度 k_c 为单位,将矩阵 A 划分成列向 panels,同时将 Loop1 中划分出的矩阵 B 的列向 panel 进一步划分成 block;Loop2 中,还会将 B block($B(p_c:p_c+k_c-1,j_c+n_c-1)$表示)进行打包(packing).关于大 kernel 中的 packing 操作见第 2.2 节.
- 第 3 层 Loop3 遍历 m 维度(以 i_c 索引),以高度 m_c 为单位,将 Loop1 中划分出的矩阵 C 的列向 panel 进一步划分出 block,以 C_c 表示(未进行 packing 操作),将 Loop2 中划分出的矩阵 A 的列向 panel 也进一步划分成 block.同样地,Loop3 中,还会将 A block 进行打包,以 A_c 表示打包后的矩阵块.
- 第 4 层 Loop4 再次遍历 n 维度(以 j_r 索引),以宽度 n_r 为单位,将 packed block B_c 和 block C_c 再次进一步划分成 micro-panels.
- 第 5 层 Loop5 再次遍历 m 维度(以 i_r 索引),以高度 m_r 为单位,将 packed block A_c 划分成 micro-panels,将 Loop4 中 C_c 中的 micro-panel 进一步划分成 micro-tiles.
- 最内层 Loop6 再次遍历 k 维度(以 p_r 索引),以 1 为单位,计算 A_c 中的 micro-panel 与 B_c 中的 micro-panel 的点积(dot product),并将结果累加到 C_c 中的 micro-panel,即所谓的“rank-1 update”.

小 kernel 算法描述如下:

- 前两次 loops 的作用在于将小矩阵 B 进行打包,其目的是便于 inner-loops 中最内层 kernel 对 B 矩阵的连续、对齐访问,关于小 kernel 中的 packing B 操作见第 2.2 节.
- inner-loops 中的 3 层循环类似于大 kernel 中的实现,只是此时循环操作的上限不再是大 kernel 中分块后的 n_c、m_c 和 k_c,而直接是整个矩阵的大小.

与大 kernel 算法相比,小 kernel 算法有如下优点:

(1) 省去了对矩阵 A 的 packing 操作.

异构 HPL 中的矩阵 A 往往是“高窄阵”,其规模比矩阵 B 大非常多,对矩阵 A 进行打包操作的耗时占整个 GEMM 操作的比例非常高.

(2) 极大地避免了大 kernel 中 out-loops 的循环次数.

大 kernel 算法与小 kernel 算法间的动态切换,主要依据输入矩阵的规模,即计算量的大小.但并不存在完美的切换阈值(threshold),因为 3 个矩阵 A、B 和 C 的行、列维度千变万化,某一个特定的切换阈值无法做到全面适用于所有的使用场景,需要在具体的应用中实际测试来找到比较合适的阈值设置.根据本文系统平台实测,切换进入小 kernel 的阈值条件满足下面两者之一即可:

(1) C 矩阵的面积,即 $M \times N \leqslant 700 \times 700$ 时.

(2) 维数 $M \leqslant 160$ 同时维数 $K \leqslant 128$ 时.

2.2 Auto-tuning自适应分块参数调优

DGEMM计算量大,计算访存比高,除上述kernel算法层面的优化外,对BLIS框架内矩阵分块参数进行优化还可以提高cache命中率,提高性能.矩阵乘法循环分块的大小是通过BLIS库中DGEMM的分块参数来决定的,分块参数的选择可以影响 cache 命中率,不合适的参数可能导致流水线停顿,从而使 DGEMM 性能下降.因此,DGEMM分块参数的选择至关重要.BLIS中DGEMM分块是通过开发人员的经验和测试选择的一组数值,手动测试不能保证测试覆盖率,也不能保证得到的是最优解.因此本文采用了Auto-tuning遗传算法自适应调优技术,更高效地优化DGEMM分块参数.

遗传算法基本算法思想是从可能具有潜在解集的一个种群开始的,种群是基因编码后形成的一组个体集合,在这个种群基础上通过选择(selection)、交叉(crossover)和变异(mutation)操作,优胜劣汰,不断地进化出最优的个体.Auto-tuning自适应调优就是应用遗传算法对DGEMM分块参数进行智能优化选择,将分块参数作为个体,一定范围内的多个个体形成一个种群进行自适应迭代进化,从而针对具体平台体系结构产生最优的DGEMM矩阵分块参数.文献[29]在Intel平台上对遗传算法在DGEMM的分块参数智能优化选择上的实现做了详细介绍.如图4所示是遗传算法的基本流程.

- 首先初始化种群,以分块参数[m_c,k_c,n_c]为一个个体,随机生成一组个体的集合.
- 然后计算种群每个个体的适应度,适应度大小决定遗传概率.
- 接着判断迭代次数,如果没有达到预设的最大值,则进行selection、crossover和mutation的遗传算法进化步骤.
- 最后,经过selection、crossover和mutation这3个步骤后生成一个新的种群,与原始种群大小一致,但个体都经过优胜劣汰,进化到了更好的水平.重复上述步骤,直到满足终止条件退出.

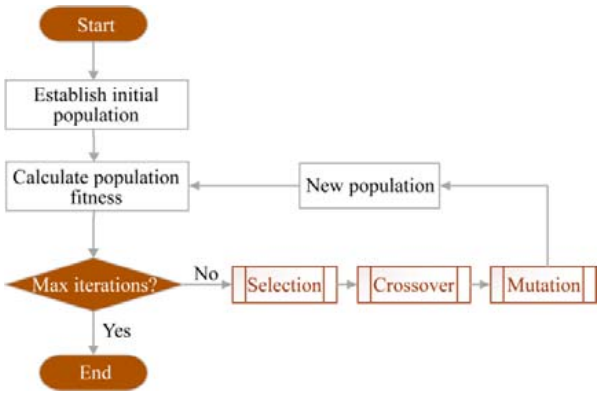

Fig.4 Generic algorithm flow chart

图4 遗传算法流程图

根据上述遗传算法的介绍,自适应程序的实现需要经过编码个体、建立种群、计算适应度、选择(selection)、交叉(crossover)、变异(mutation)和终止条件判断这几个步骤.

(1) 个体编码

编码方法有二进制编码、格雷编码、浮点数编码和符号编码等多种方式,根据DGEMM分块参数特点,采取浮点数编码更适合.也就是将分块参数[m_c,k_c,n_c]这3个浮点数直接作为参数传入.

(2) 建立种群(population)

个体集合的建立过程就是要在一个确定范围内去随机生成一定数量的个体.种群范围的确定可以根据cache的大小进行理论计算得出[7],HBLIS分块参数原始值是[1080,120,8600],在原始参数值的基础上扩大范围至[512<m_c<2400,60<k_c<2400,4800<n_c<12000],随机生成的分块参数要保证能够被内核参数整除,也就是被分配

的寄存器数[4,6,6]整除.

(3) 计算适应度(fitness)

适应度函数就是 DGEMM 测试程序,其性能作为判断适应度高低的标准.性能高的分块参数会遗传给下一代种群,性能低的分块参数就会被淘汰.

(4) 选择(selection)

基本遗传算法仅仅使用 selection、crossover 和 mutation 这 3 种算子是没有办法收敛于全局最优解的,因为简单地进行杂交,可能反而会把较好的组合破坏掉.本文采用精英保留策略,也就是把目前最优个体不进行任何操作直接复制到下一代种群中.选择方法有很多种,包括比例选择法、排序选择等.本文采用比例选择法和精英保留策略结合,种群中每一个个体的适应度占总适应度的比例越大,其被选择的概率就越大,同时这一代种群最优秀个体无条件复制到下一代.

(5) 交叉(crossover)

单点交叉、双点交叉、均匀交叉等都适合二进制编码和浮点数编码,根据 GEMM 分块参数只有 3 个,选择单点交叉,按某一概率将两个个体的某一位置参数互换.交叉率设置越大,新个体产生越快,其全局搜索能力就会越强,但是优秀个体被破坏的可能性也就越高,综合考虑,设置交叉概率为 0.8.

(6) 变异(mutation)

变异的主要目的是改善局部搜索能力以及维持种群个体的多样性,变异方法采用均匀变异,按某一概率在一个范围内随机生成一个数替换原来个体中某一位置的参数.变异概率不宜设置过大,导致丢失优秀个体而变成随机搜索.考虑设置参数范围已经在合理范围内,本文按 0.1 的概率对每一个个体的某一参数进行变异.

(7) 终止条件判断(iteration)

种群中所有的个体适应度计算完成后要判断设置的终止条件是否满足,以判断程序是否结束退出.如果不满足终止条件就进行 selection、crossover 和 mutation 操作,形成下一个同等规模的个体集合(种群).

Auto-tuning 自适应调优算法完成后,需要进行 DGEMM 测试,BLIS 库中 config 目录是存放不同平台 kernel 的配置信息,其中 DGEMM 分块参数通过 bli_kernel.h 头文件中的 BLIS_DEFAULT_MC_D、BLIS_DEFAULT_KC_D 和 BLIS_DEFAULT_NC_D 这 3 个宏定义配置.

Auto-tuning 的具体测试流程如下:

(1) 在包含初始分块的更大的范围内随机生成一组满足遗传算法个体要求的分块参数[m_c,k_c,n_c],并用其替换掉 blis_kernel.h 中的分块参数.

(2) 自动编译 BLIS 库,生成此分块参数下的动态链接库 libblis.so.

(3) 调用 libblis.so 执行测试程序 DGEMM,主程序将性能值记录并返回.

通过 Auto-tuning 的不断迭代,根据算法模型,我们会得到一个全局最优解,这组参数就是性能更高的 DGEMM 分块参数,实测数据分块参数由[1080,120,8400]优化为[792,822,8628],将这组参数替换进 BLIS 中,并使用高性能计算标准测试程序 HPL 和单独的 DGEMM 测试程序进行测试,具体测试对比数据参考第 5 节性能分析.

2.3 矩阵打包

连续的内存访问比非连续的内存访问要快.异构 HPL 中传给 DGEMM 的矩阵 A、B 和 C 的索引是初始化矩阵的某个位置指针,矩阵元素通过 leading-dimension 索引,leading-dimension 表示列优先存储方式的矩阵同一行(或行优先存储方式的矩阵同一列)中两相邻元素的距离,也就是实际存储矩阵的二维数组的第 1 维(或第 2 维)的大小.矩阵 A、B 和 C 的行或列间跨度通过 LDA、LDB 和 LDC 表示,这 3 个数值可能会非常大.比如异构 HPL 求解问题 $Ax=b$,其矩阵规模 N 的最大取值可以使矩阵 A 的存储空间占用内存 80%左右,则 LDA 不能小于 N.在算法实现中,大 kernel 算法会不断地对 A 和 B 的不同分块进行打包,形成 A_c 和 B_c;而小 kernel 算法只将原始矩阵 B 进行一次打包,形成行优先存储(row-major)的 B_{packed}.如图 5 所示,其中假设 n_r 等于 6,矩阵 B 是以列优先存储(column-major)方式存储数据,且行数 k 和列数 n 都是 n_r 的整数倍.

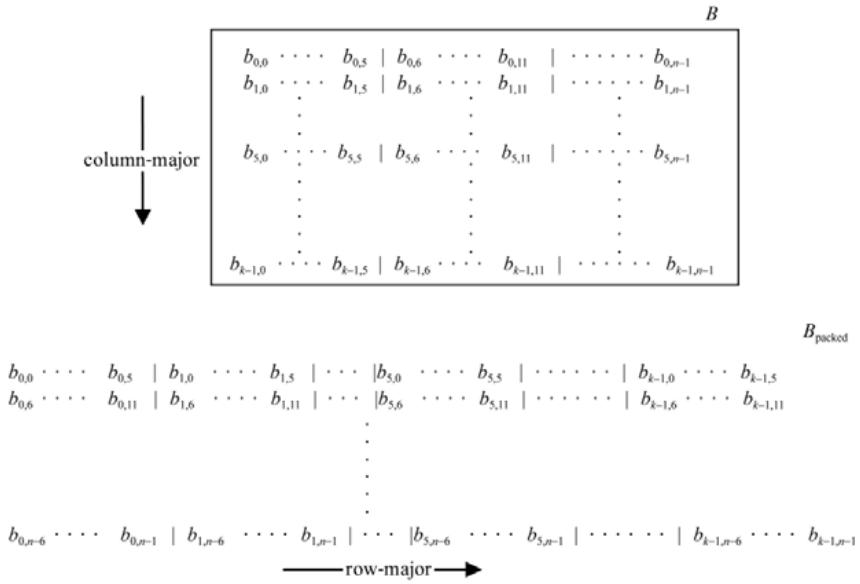

Fig.5 Packing matrix B in little GEMM kernel

图 5 GEMM 小 kernel 中矩阵 B 的打包操作

Packing 数据打包操作的优点有 3 个.

(1) 创造连续的内存访问方式.

BLAS 接口传入的矩阵存储方式是不确定的,可能是 row-major,也可能是 column-major,且矩阵的行或列间的跨度会非常大,最内层的 kernel 为了适应不同的数据存储方式以及不连续的数据分布,需要通过 packing 制造统一的、连续的内存布局视图.

(2) 数据对齐.

对于大多数 CPU 微架构实现而言,对齐到寄存器宽度,cache 边界和页边界的数据访问往往比非对齐的数据访问方式更快.

(3) 提前 load 数据到 cache.

通过 packing 操作,最内层 kernel 可以在尽可能接近 CPU 寄存器的 cache 上获取到需要进行运算的操作数,这比直接从内存中获取数据要快很多.

2.4 数据预取

X86 CPU 提供了硬件取功能和符合 X86 ISA 的软件预取指令.其中硬件预取对开发者是透明的,软件预取指令 PREFETCHn 指示处理器将后续需要访问的数据提前读入到指定的某级缓存中,当程序使用这些数据时,可以直接读取缓存.PREFETCHn 指令格式见表 1.

Table 1 X86 software prefetch instructions

表 1 X86 软件预取指令

指令格式	功能描述
PREFETCHT0 mem8	将内存字节地址 mem8 指定的缓存行,预取到所有的 caches 中
PREFETCHT1 mem8	将 mem8 指定的缓存行,预取到除 $L1$ 之外的所有 caches 中
PREFETCHT2 mem8	将 mem8 指定的缓存行,预取到除 $L1$、$L2$ 之外的所有 caches 中
PREFETCHNTA mem8	将 mem8 指定的缓存行,以缓存污染最小化的方式预取到 caches 中

软件预取指令不要求数据行对齐,如果预取数据行已经存在于比指定的 n 更低的某层 cache 中,那么指令将被当作 NOP 空指令处理.

在最内层的核心汇编代码中,PREFETCHn 指令的使用主要考虑两点:

(1) 何处插入 PREFETCHn 指令.

(2) 预取的地址跨度设置.

X86 CPU 的 cache line 大小为 64B,一个缓存行可以存放 8 个双精度浮点数据.如图 6 所示的汇编代码第 1 行的软件预取操作,表示把 A_c 的第 9 个双精度数据开始的一个缓存行预取到 $L1$ 缓存中.此处的预取操作是为第 22 行的 load 指令准备数据.第 5 行的 load 操作会在 $L1$ 缓存发生 cache miss,但是因为对 A 进行了 packing 操作,理论上能够在 $L2$ cache 命中.同理,第 9 行、第 14 行和第 18 行处的 load 操作会在 $L1$ 中命中,第 11 行预取操作将 A_c 的第 17 个数据预取到内存.

```
      ……
1     prefetcht0 64(%%rax)
      ……
2     vmovapd 0 * 8(%%rbx),%%xmm1
3     vmovapd 2 * 8(%%rbx),%%xmm2
4     vmovapd 4 * 8(%%rbx),%%xmm3
5     vmovddup 0 * 8(%%rax),%%xmm0
6     vfmadd231pd    %%xmm1,%%xmm0,%%xmm4
7     vfmadd231pd    %%xmm2,%%xmm0,%%xmm5
8     vfmadd231pd    %%xmm3,%%xmm0,%%xmm6
9     vmovddup 1 * 8(%%rax),%%xmm0
10    vfmadd231pd    %%xmm1,%%xmm0,%%xmm7
11    prefetcht0 128(%%rax)
12    vfmadd231pd    %%xmm2,%*%xmm0,%%xmm8
13    vfmadd231pd    %%xmm3,%%xmm0,%%xmm9
14    vmovddup 2 * 8(%%rax),%%xmm0
15    vfmadd231pd     %%xmm1,%%xmm0,%%xmm10
16    vfmadd231pd     %%xmm2,%%xmm0,%%xmm11
17    vfmadd231pd     %%xmm3,%%xmm0,%%xmm12
18    vmovddup 3 * 8(%%rax),%%xmm0
19    vfmadd231pd     %%xmm1,%%xmm0,%%xmm13
20    vfmadd231pd     %%xmm2,%%xmm0,%%xmm14
21    vfmadd231pd     %%xmm3,%%xmm0,%%xmm15
      ……
22    vmovddup 8 * 8(%%rax),%%xmm0
```

Fig.6 Code segment of inner-most DGEMM kernel

图 6 DGEMM 核心汇编代码片段

软件预取指令的插入位置需要综合考虑体系结构特点,合理安排汇编指令.X86 CPU 的 $L2$ load-to-use 的 latency 至少是 12 个 core cycles,而每时钟周期只可以执行一条 FMA 指令,在 CPU 流水线能够连续 retire FMA 指令的理想状态下,预取指令和后续 load 指令之间至少插入 12 条 FMA 指令.

3 指令集优化

高效 BLAS kernel 的实现方法具有通用性,不同于其他数学库大部分代码通过汇编实现优化,BLIS 只在最内层的 kernel 核心代码处使用了汇编代码,框架内的其他代码使用 C 实现.这给可移植性带来了极大的便利,在适配不同架构的 CPU 时,开发者可以将主要精力集中在最内层 kernel 的汇编代码的优化上.

3.1 X86向量指令

不同架构的 CPU 大多会提供高效的 SIMD(single instruction multiple data)指令,X86 CPU 支持 AVX2 指令集,可以使用 128 位的 XMM 寄存器和 256 位的 YMM 寄存器,但由于浮点运算部件的数据路径宽度是 128 位,每个时钟周期最多可以同时处理 2 个双精度浮点数,单核理论双精度浮点性能是(128/64)×2=4 DP-FLOPs/cycle,实际测试结果表明,使用 XMM 寄存器的效率要优于 YMM 寄存器.AVX2 指令集中主要有两类指令:数据传输类指令和数据操作类指令.其中,VMOVDDUP、VMOVAPD、VMOVUPD、VMULPD 和 VFMADD231PD 是在 X86 架构 CPU 平台上高效 BLAS kernels 的实现中使用频率非常高的几条指令.

HPL 算法中传递给 BLAS 函数的参数会有固定值的情况,可以通过适当的指令替换来精简指令,举例如下.

(1) 对于 Level-3 级函数 DGEMM

系数 Alpha 固定为–1.0,Beta 固定为 1.0.此时的 DGEMM 计算公式由 $C:=\alpha\cdot AB+\beta\cdot C$ 变成了 $C:=C-AB$ 可以在 packing B(见第 2.3 节)的过程中使用 xorpd 指令进行符号位翻转,精简了对系数 Alpha 的乘法操作.矩阵 C 不变,精简了对系数 Beta 的乘法操作.精简指令后,原来需要 4 个 core cycle 的乘法操作和 5 个 core cycle 的乘加指令只需 3 个 core cycle 的加法指令即可完成.在矩阵规模较大,最内层 kernel 被循环多次的情况下,指令时钟周期的减少可以优化整体性能.

(2) 对于 Level-2 级函数 DGEMV

转置参数 TransA 固定为 NoTrans,可以省略对矩阵 A 进行转置操作的代码,提高 CPU 流水线分支预测的准确率,避免分支预测失败的惩罚.Alpha 固定为–1.0,Beta 固定为 1.0,则可以在内层 kernel 中使用一条 VFNMADD231PD 指令取代 VMULPD 和 VFMADD231PD 两条指令.

(3) 对于 Level-1 级函数 DSCAL

IncX 固定为 1,可以省去对数据成员间隔较大时所需的软件预取操作的指令(见第 2.4 节).

异构 HPL 传递给 BLAS 接口的参数还有其他的固定值情况存在,这些固定值的存在可以起到精简指令的作用,为 BLAS 库提供了优化空间.

3.2 循环展开

循环展开是编译器常用的优化方式之一,可以减少分支预测失败带来的开销,提高指令级的并行度.但高级语言通过编译器循环展开后的汇编代码在 BLAS 这种访存密集型的程序特征下,往往性能表现较差.因此 BLAS 的核心循环代码通常采用手写汇编的方式进行展开.

手写 BLAS 核心汇编代码,不仅需要考虑底层 CPU 架构层面的架构寄存器的数量和复用方式,而且需要考虑底层 CPU 的微架构特点,比如浮点部件的宽度、存储层级的大小和层级间的延迟等.这些 CPU 特性共同决定了手写汇编循环展开的程度,也就是“循环展开因子”.本文第 2.1 节中提到的 5 个参数 n_c、k_c、m_c、n_r 和 m_r,其中内核参数 n_r 和 m_r 与汇编代码循环展开直接相关,而矩阵分块参数 n_c,k_c 和 m_c 则与 CPU 存储层级特性直接相关.针对具体平台体系结构可以理论计算出这 5 个参数的大致取值[30],但实际代码测试中发现,理论数据往往并不是最优设置,X86 CPU 有 16 个 128 位的向量寄存器 XMM_0~XMM_{15},内核参数 n_r,m_r 取值的首要策略是最大化利用架构寄存器,因此综合考虑,n_r 和 m_r 分别取值为 6 和 4.同时矩阵参数[n_c,k_c,m_c]依据第 2.2 节中的自动调优技术同样获得了比理论值更好的参数设置.

如图 6 所示,通用寄存器 RAX 用于索引 A_c 的 micro-panel 中的数据,不断获取一个双精度数据然后复制成两份填充到 XMM0 中;RBX 用于索引 B_c 的 micro-panel 中的数据,一次获取并存储 6 个双精度数据到 XMM1~XMM3 中.XMM4~XMM15 用于存储不断累加的中间结果.在此过程中,向量寄存器 XMM0 实际共存储过 4 个 A_c 数据,XMM1~XMM3 共存储过 6 个 B_c 数据.

4 多线程并行

常用的多线程并行手段有 OpenMP[31]和 POSIX Threads(pthread)[32].BLIS 同时支持两种并行方式,且当前只在 Level-3 实现了并行.与 OpenMP 相比,Pthreads 技术在实际编程中需要考虑临界区、线程同步原语等非常底层的操作,故我们在 GEMM 以及 TRSM 等小 kernel 和 Level-1 与 Level-2 的并行化实现中,选择使用更加便利的 OpenMP 进行并行化.

4.1 Control-tree优化

针对 Level-3 级 BLAS 的并行实现方式,BLIS 提出了新颖的“control-tree”结构[7].该结构以统一的方式实现了如图 3 所示的 GEMM 大 kernel 算法的并行以及 HERK、TRMM 和 TRSM 运算的并行.Control-tree 结构最突出的优点是可以在任意层 loop 内分别使用环境变量 *BLIS_JC_NT*、*BLIS_IC_NT*、*BLIS_JR_NT* 和 *BLIS_IR_NT*

进行多线程并行的控制,甚至是多个维度同时并行.这种统一结构的便利性的代价就是在所有能够发生并行的地方设置同步屏障(barrier),以便在当前 loop 的下层所有操作全部返回后才能继续回溯.虽然文献[7]中的性能测试结果表明 control-tree 结构的开销是非常小的,但在异构 HPL 环境中,control-tree 的开销变得尤为突出.BLIS 原生 control-tree 的结构大致如图 7 所示.

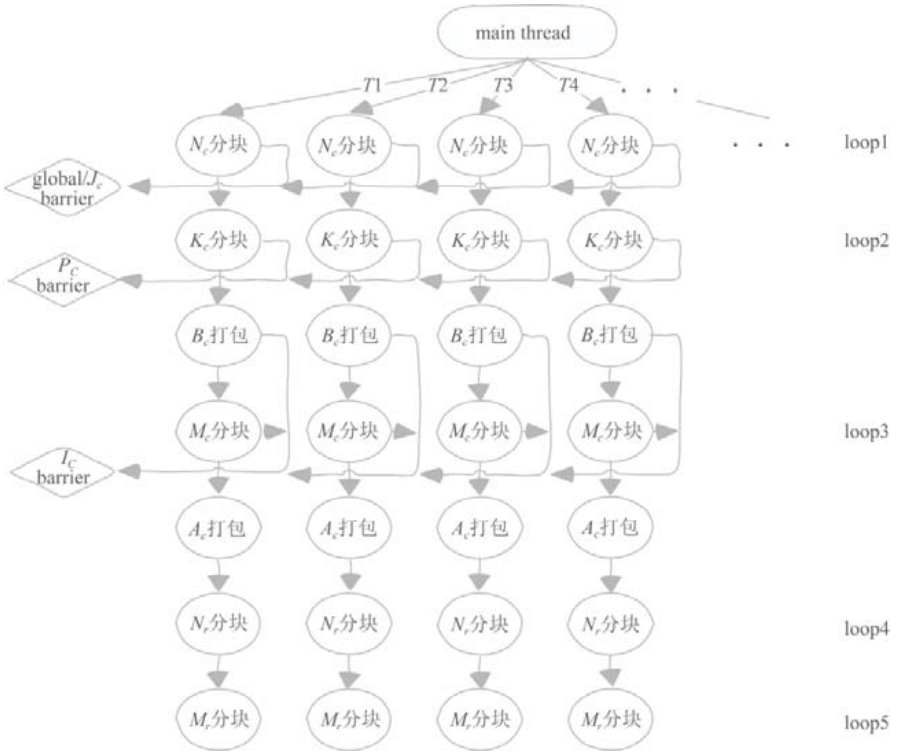

Fig.7 Original BLIS GEMM control-tree

图 7 BLIS 原生 GEMM control-tree

对于异构 HPL 中的 GEMM 运算,我们选择只在 loop3 处(M_c 维度)实现并行,当各个线程的 A_c packing 打包操作和后续操作都完成后,需要在 loop3 的 I_c barrier 处进行第 1 次同步,随后的 P_c barrier 和 global barrier 处再次进行同步,才可以保证最终 GEMM 运算的正确性.可以看到,P_c barrier 和 global barrier 的同步开销其实是可以避免的,只需将 loop3 放到算法的最前端,然后使用 OpenMP 自身的同步操作实现各个线程的同步即可.基于此分析,对 GEMM 运算的 control-tree 结构做出了如图 8 所示的调整.

通过异构 HPL 的实际测试,未优化 control-tree 时 DGEMM 和 DTRSM 运算的同步 barrier 等待的时间开销为 2.29s,约占 CPU 端计算总时间的 20%.优化 control-tree 结构后,GEMM 函数同步 barrier 的等待时间减少为 1.49s,TRSM 运算应用类似于优化思路,同步 barrier 的等待时间减少到基本可以忽略不计.

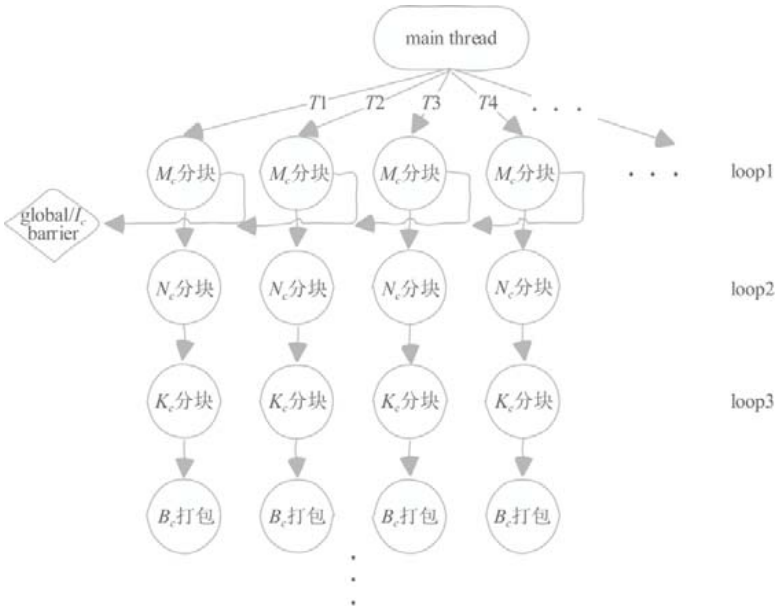

Fig.8 Optimized HBLIS GEMM control-tree

图 8 优化后的 HBLIS GEMM control-tree

4.2 Level-1和Level-2级BLAS并行优化

原 BLIS 库没有实现 Level-1 级和 Level-2 级 BLAS 函数运算的并行化.本文使用如下相似的策略实现了 DGEMV、DTRSV、DSCAL、DAXPY 以及 IDAMAX 运算的并行化:

(1) 通过运行脚本中设置的环境变量“*OMP_NUM_THREADS*”获取当前 MPI 进程中的线程数.

(2) 将矩阵或向量中的某个维度按设置的线程数进行切分.

(3) 最后使用 OpenMP 的 Pragma(编译指示)启动多线程,在不同的线程空间中复用同一段汇编 kernel 来处理不同的数据.在 Pragma 包裹的并行域(parallel region)中需要合理而准确地为每个线程划分任务空间.可以通过 omp_get_thread_num()函数获得当前线程的线程编号(TID)划分任务,避免计算冲突.

多线程并行对于 BLAS 运算的性能提升是非常显著的,矩阵或向量计算达到一定规模后,性能往往随着线程数的增加而提升,但提升并不会完全是线性的,通常是计算越密集的运算,提升的线性程度越高,因此计算量最大的 Level-3 级 BLAS 运算通过多线程并行会大大提升性能.如图 9 所示,在 N=43776 时的 Level-1 运算 IDMAX 随着线程数的增加表现出性能的提升.

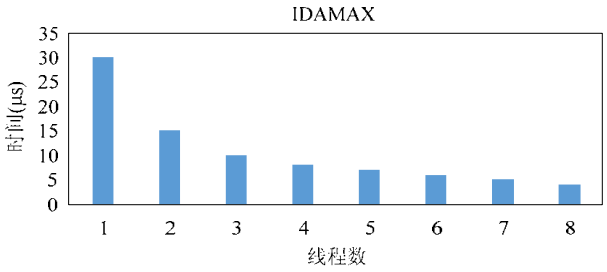

Fig.9 Performance improvement by multi-threading of IDAMAX

图 9 多线程对 IDAMAX 的性能提升

4.3 并行化问题

多线程并行化并不是总会对计算带来帮助,为了进一步提高性能,需要综合考虑以下问题.

(1) 动态多线程

创建和启动多线程的开销在计算量很小的情况下会变得突出,从而导致程序性能下降.所以需要设置一个或多个特定的阈值来进行线程数的切换.具体是设置一个阈值将多线程直接切换到单线程,还是设置多个阈值将线程数分段递减,需要具体的 BLAS 函数结合特定的输入数据进行大量的测试归纳(profiling).在 DGEMM 的多线程优化中,我们设计了精确的运算量模型.DGEMM 需要计算一个 $m\times k$ 阶矩阵 A 和 $k\times n$ 阶矩阵 B 的乘积,其主要计算量和乘积 mkn 成正比.考虑到矩阵 A 的规模 $m\cdot k$,矩阵 B 的规模 $k\cdot n$ 等都对最终运算性能造成影响,我们使用最小二乘法拟合矩阵计算的性能公式,假设不同线程数目下运算时间根据式(2)计算.

$$T = a_1mkn + a_2mk + a_3mn + a_4nk + a_5m + a_6n + a_7k + a_8 \tag{2}$$

于是对于输入参数为 m_i、k_i 和 n_i,运行时间为 t_i 的测试数据,对应拟合数据中的一行如式(3)所示.

$$\left(m_ik_in_i, m_ik_i, m_in_i, n_ik_i, m_i, n_i, k_i, 1\right) \tag{3}$$

所有这些拟合数据组成 $N\times 8$ 阶矩阵 P,其中 N 是采样数,而所有的采集时间构成对应的列向量 $\boldsymbol{b}$,于是构成式(4)的拟合方程.

$$P^TPx = P^T\boldsymbol{b} \tag{4}$$

解方程得出的 8 阶变量 x 的各分量就是 $a_1,a_2,\ldots,a_8$ 的最小二乘拟合结果.分别对单线程、4 线程、8 线程和 32 线程测试拟合公式,在运行中根据拟合结果动态切换线程数据,得到如图 10 所示的性能结果.

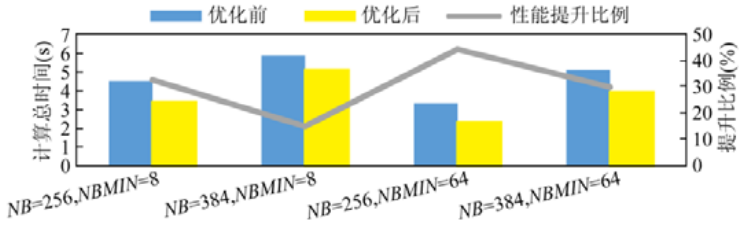

Fig.10 Performance speedup of DGEMM in 32 threads

图 10 DGEMM 在 32 个线程中的性能加速比

实践中还发现,不同的 kernel 实现方式以及硬件设置等因素也会影响到阈值的选取.比如,在异构 HPL 的 DGEMV 函数的优化过程中,根据不同的“OMP_NUM_THREADS”、不同的 CPU 主频以及矩阵 A 的列数 N(M 方向做并行)设置不同的多个阈值.

(2) 均衡余量(remainder)处理

当使用多线程将求解问题切分成多块后,往往会遇到不能均分的情况.所谓不能“均分”,是指整体的运算量不能均匀地分配到每个线程中去.此时有两个选择:或者将所有多出的运算量全部放到某一个线程中,比如最后一个线程;或者将余量再次均分到所有线程中.Remainder 处理的问题在线程数较少时(比如 6 个线程)不会那么关键,但在线程数较多时(比如 32 个线程)会变得突出.因为此时的木桶效应更加明显,整体运算的同步结束取决于拥有最大余量的那个线程.根据本文 HPL 算法测试分析后,采取了将 remainder 分配给每个线程处理,得到的性能结果最优.

(3) 循环展开与多线程的均衡

循环展开的程度和线程的数量需要均衡考虑.在使用动态多线程去加速运算的过程中,往往会有多种求解方法去完成同一个运算.考虑以下场景:异构 HPL 中的 DGEMM 运算在 M 较小时,比如 M=28,此时可以使用 4 个线程各处理 6 个 M 维度的行,然后使用 1 个线程处理剩下的 4 个 M 行;或者使用 5 个线程各处理 5 个 M 维度的行,然后使用 3 个线程处理各处理一个 M 行,如式(5)所示.

$$M=28=4T(\text{thread})\times 6+1T\times 4=5T\times 5+3T\times 1 \tag{5}$$

不同的求解方法涉及同一个函数的不同 kernel 函数编写以及动态多线程 threshold 的选取.通过实际测试发现:DGEMM 运算在 M 较小时,更多的循环展开比更多的线程性能要好,如图 11 所示.

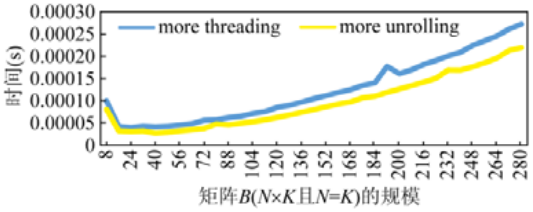

Fig.11 In the case that B matrix is square matrix (dimension N=K), more unrolling is better performance than more threading in small DGEMM

图 11 B 矩阵是方阵(维数 N=K)的情况下,在小规模的 DGEMM 中,更多的循环展开比更多的线程性能更优

5 性能测试与分析

异构 HPL 算法中综合使用了以上介绍的访存优化、指令集优化和多线程并行等优化技术,对算法中调用的 7 个 BLAS 函数进行了针对性优化,并与 OpenBLAS 0.3.6 和 MKL(Intel Math Kernel Library 2018.0.128)进行了对比测试.实验测试平台是高性能计算集群系统中的单个计算节点,其配置是如图 1 所示的 32 核 X86 CPU 与 4 个协处理组成的异构计算单元.在测试平台上分别进行了单核 DGEMM 和多核心并行 DGEMM 测试,最后测试了异构 HPL 程序.

5.1 Cache性能分析

通过系统性能分析工具读取 CPU 中各种计数器的数值,可以分析应用程序的各项指标.X86 处理器支持非常多的性能监控计数器,包括浮点、访存、指令等 CPU 各种资源的监控.如表 2 所示为 Auto-tuning 优化后的矩阵分块参数与初始分块参数进行 DGEMM 计算时的 cache miss 对比情况,分析得出分块参数[m_c,k_c,n_c]从[1080,120,8400]优化为[792,822,8628],k_c 和 n_c 的增大导致矩阵打包到 $L1$ 缓存时,因 $L1$ 容量小而发生更多的 cache miss.但是,$L2$ 和 $L3$ 缓存容量比 $L1$ 大得多,其命中率的提高可以掩盖 $L1$ 的性能损失并提高整体 DGEMM 的性能.

Table 2 Cache miss rate of HPL counted by performance monitor unit

表 2 通过性能监控单元统计 HPL 的缓存失效率

cache	初始分块参数	优化分块参数
$L1$ miss rate (%)	6.90	17.20
$L2$ miss rate (%)	8.76	4.41
$L3$ miss rate (%)	8.05	2.43

5.2 DGEMM效率

如图 12 和图 13 所示分别为单核和 32 核情况下不同数学库的 DGEMM 性能对比.因为 Intel MKL 只有在 Intel 平台上才能发挥最佳效果,在 X86 异构平台上效率不是最好,仅作为参考;而 Netlib 为未经过任何优化的 Fortran 语言实现的通用标准数学库.可以看到,HBLIS 的 DGEMM 实现无论是在单核心还是在 32 核心下,其性能均超过了其他 BLAS 库,单核 DGEMM 最高效率达到了 98%,32 核最高效率达到了 96.9%.与未优化前的 BLIS 和 MKL 相比,单核性能分别提升了 19.2%和 33.3%,32 核性能分别提升了 42.1%和 33.6%.

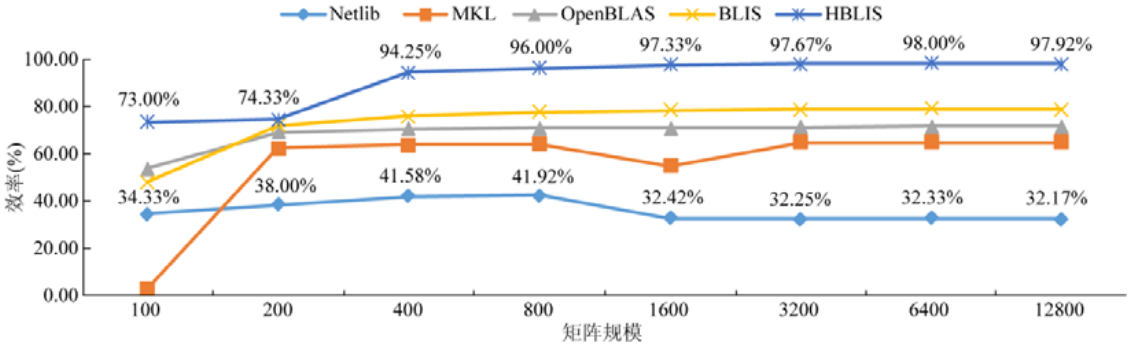

Fig.12 DGEMM efficiency comparison of single-core

图 12 单核 DGEMM 效率对比

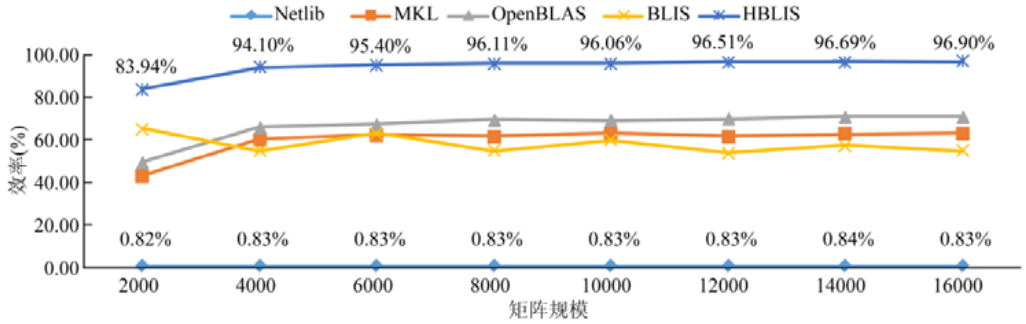

Fig.13 DGEMM efficiency comparison of 32 cores

图 13 32 核 DGEMM 效率对比

5.3 异构HPL测试

HPL 实测性能值是最终评价标准,分别统计每个 BLAS 函数的时间作为 BLAS 优化的评价标准.为避免网络通信的干扰,本文基于单节点进行异构 HPL 测试,对比包括 panel 分解中 pfact 时间以及 HPL 算法中使用的 7 个 BLAS 函数 DGEMM、DGEMV、DTRSM、DTRSV、IDMAX、DSCAL 和 DCOPY 的计算时间.

异构 HPL 算法 panel 分解部分主要在 CPU 端进行计算.如图 14 左侧所示为 HPL 程序 panel 分解中的 pfact 总时间统计,调用 HBLIS 的 pfact 与调用 MKL 的相比,计算时间节省了 53%.如图 14 右侧所示为 DGEMM 函数计算时间统计,可以看到,HBLIS 的 DGEMM 性能比 MKL 的 DGEMM 性能提升了约 25%.

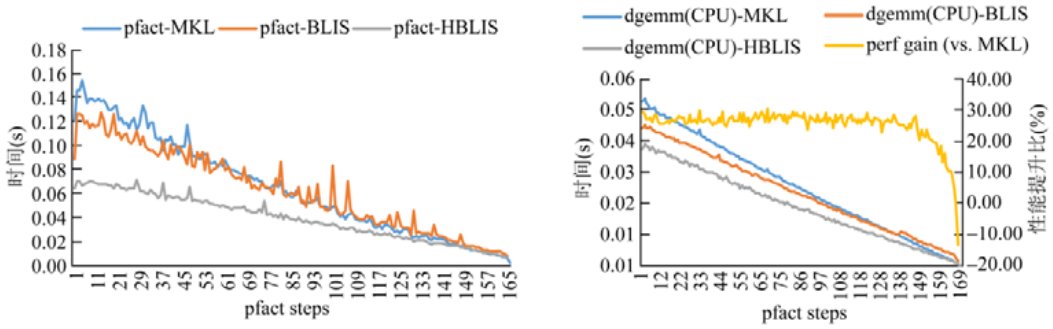

Fig.14 Performance comparison of pfact and DGEMM calling different math libraries in heterogeneous HPL

图 14 异构 HPL 中调用不同数学库的 pfact 和 DGEMM 的性能对比

如图 15 所示为 DGEMV 和 DTRSV 函数性能对比.DGEMV 在 panel 分解中时间占比仅次于 DGEMM,因此

DGEMV 性能对整体性能的影响也仅次于 DGEMM.图 15 左侧为 DGEMV 性能,优化的 HBLIS 比 MKL 提升了约 62%,比原 BLIS 提升了约 65%.图 15 右侧为 DTRSV 函数时间统计,原 BLIS 性能与 MKL 差距较大,经过优化后的 BLIS 性能超过了 MKL.因为 DTRSV 计算量较小,在总时间中占比也相对很小.

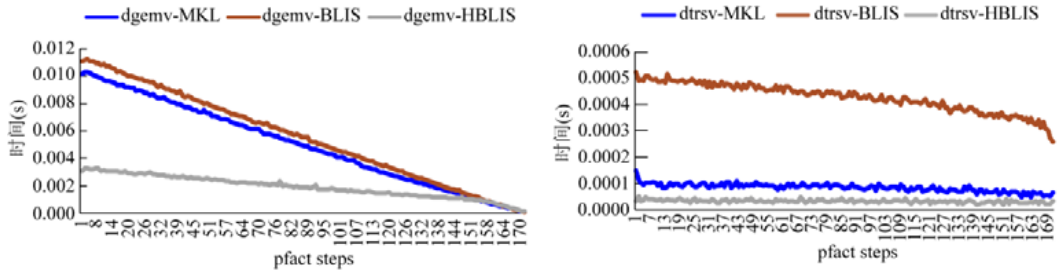

Fig.15 Performance comparison of DGEMV and DTRSV calling different math libraries in heterogeneous HPL

图 15 异构 HPL 中调用不同数学库的 DGEMV 和 DTRSV 的性能对比

如图 16 所示为 DTRSM 和 IDMAX 函数性能对比.原生 BLIS 算法库中的三角矩阵求解 DTRSM 性能较差,优化后的 DTRSM 性能与 MKL 基本一致.在 pfact 第 106 步~第 141 步的性能是超过 MKL 的,而在 panel 分解最后段,MKL 运算更快,原因是 DTRSM 在切换到小 kernel 的 threshold 和动态多线程的 threshold 时,需要做进一步的调整,以适应运算量的变化.IDMAX 函数主要是用于选主元操作,计算量可以忽略不计.

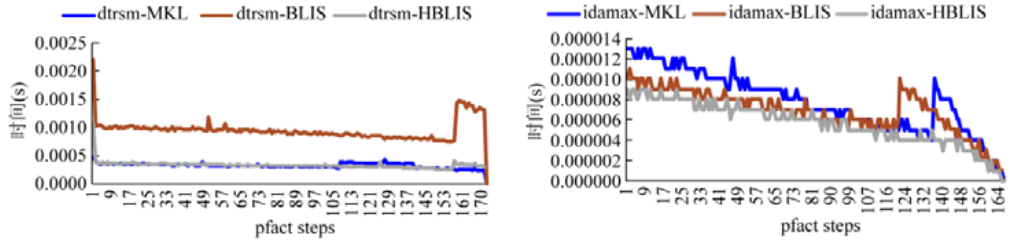

Fig.16 Performance comparison of DTRSM and IDAMAX calling different math libraries in heterogeneous HPL

图 16 异构 HPL 中调用不同数学库的 DTRSM 和 IDAMAX 的性能对比

如图 17 所示,Level-1 级 BLAS 函数 DSCAL 和 DCOPY 优化后的性能基本与 Intel MKL 一致.其中,DSCAL 优化后的性能比原生 BLIS 有大幅度提升,但在 panel 分解后段的性能还是略低于 MKL,原因是 MKL 使用的多线程库 IOMP 比开源的 GOMP 在多线程启动、调度和动态多线程的切换上更加高效.而 DCOPY 代码实现简单,优化空间非常有限,硬件平台性能直接决定了其时间开销,故可以看到三者 DCOPY 的性能基本一致.DAXPY 运算只在回代(back substitution)过程中调用一次,性能影响可以忽略,故未列出详细对比.

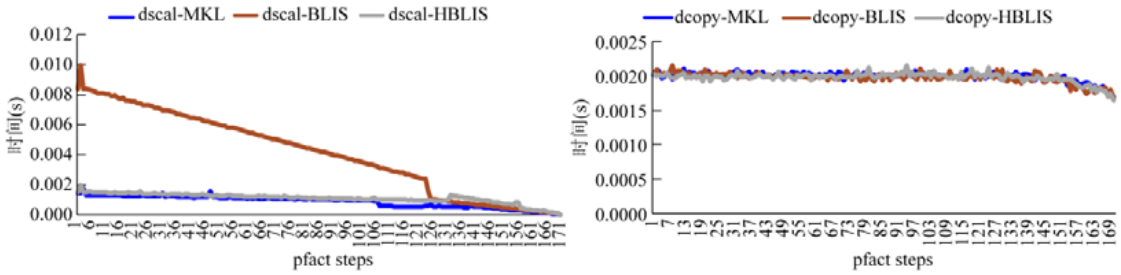

Fig.17 Performance comparison of DSCAL and DCOPY calling different math libraries in heterogeneous HPL

图 17 异构 HPL 中调用不同数学库的 DSCAL 和 DCOPY 的性能对比

6 总结与展望

本文针对 CPU 平台系统架构的特点,在开源 BLIS 算法库的基础上开发了优化的 HBLIS 数学库,详细介绍了 HPL 调用的各级 BLAS 函数优化方法.通过访存优化、指令集优化和多线程并行优化,HPL 调用的各级 BLAS 函数的计算性能与 MKL 的相应函数相比有着普遍和显著的提升,个别计算量小的函数性能基本与 MKL 相应函数性能一致,其中最重要的 DGEMM 函数提升了约 25%,DGEMV 函数性能提升了约 62%.在异构单节点 HPL 测试中,与调用 MKL 库的 HPL 相比,采用 HBLIS 的 HPL 整体效率值提升了 11.8%,更好地发挥了异构高性能平台的计算能力.同时本文也有不足之处,虽然 HBLIS 库性能测试发现 DGEMM、DGEMV、DTRSV 和 IDMAX 有着很好的优化效果,其性能与 MKL 和开源 BLIS 库相应函数相比都有大幅度提升,但是 DTRSM 和 DSCAL 优化后性能并没有优于 Intel MKL,这可能与计算量、多线程库支持和编译器优化支持都有密切关联.底层子例程库优化是一个系统工程,不仅需要硬件架构的优化设计,而且需要编译器和线程库等底层软件的优化.本文未涉及的其他 BLAS 函数的优化将是 HBLIS 库下一步优化工作的方向.

References:

[1] Whaley RC, Dongarra JJ. Automatically tuned linear algebra software. In: Proc. of the 1998 ACM/IEEE Conf. on Supercomputing (SC'98). San Jose, 1998. 1−27.

[2] Goto K, van de Geijn RA. Anatomy of high-performance matrix Multiplication. ACM Trans. on Mathematical Software, 2008,34(3): 1−25.

[3] Goto K, van de Geijn RA. High-performance implementation of the Level-3 BLAS. ACM Trans. on Mathematical Software, 2008, 35(1):1−14.

[4] Wang Q, Zhang X, Zhang Y, Qing Y. AUGEM: Automatically generate high performance dense linear algebra kernels on X86 CPUs. In: Proc. of the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC 2013). Denver, 2013. 1−12.

[5] https://github.com/xianyi/OpenBLAS

[6] https://github.com/flame/blis

[7] Van Zee FG, van de Geijn RA. BLIS: A framework for rapidly instantiating BLAS functionality. ACM Trans. on Mathematical Software, 2015:41(3):1−33.

[8] Van Zee FG, Smith TM, Marker B, Low TM, van de Geijn RA, Igual FD, Smelyanskiy M, Zhang XY, Kistler M, Austel V, Gunnels JA, Killough L. The BLIS Framework: Experiments in Portability. ACM Trans. on Mathematical Software, 2016,42(2):1−19.

[9] Smith TM, van de Geijn RA, Smelyanskiy M, Hammond JR, Van Zee FG. Anatomy of high-performance many-threaded matrix multiplication. In: Proc. of the IEEE 28th Int'l Parallel and Distributed Processing Symp. 2014. 1049−1059.

[10] Gu NJ, Li K, Chen GL, Wu C. Optimization of BLAS based on Loongson 2F architecture. Journal of University of Science and Technology of China, 2008,38(7):854−859 (in Chinese with English abstract).

[11] Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience, 2003,15(9):803−820.

[12] Tan G, Li L, Triechle S, Phillips E, Bao Y, Sun N. Fast implementation of DGEMM on Fermi GPU. In: Proc. of the 2011 Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC 2011). Seattle, 2011. 1−11.

[13] Jiang H, Wang F, Zuo K, Su X, Xue L, Yang C. Design and implementation of a highly efficient DGEMM for 64-bit ARMv8 multi-core processors. In: Proc. of the 44th Int'l Conf. on Parallel Processing. Beijing, 2015. 200−209.

[14] Jiang H, Wang F, Li K, Yang C, Zhao K, Huang C. Implementation of an accurate and efficient compensated DGEMM for 64-bit ARMv8 multi-core processors. In: Proc. of the IEEE 21st Int'l Conf. on Parallel and Distributed Systems (ICPADS). Melbourne, 2015. 491−498.

[15] Wang L, Wu W, Xu Z, Xiao J, Yang Y. BLASX: A high performance level-3 BLAS library for heterogeneous multi-GPU computing. In: Proc. of the 2016 Int'l Conf. on Supercomputing (ICS 2016). Istanbul, 2016. 1−11.

[16] Sun JD, Sun Q, Deng P, Yang C. Research on the optimization of BLAS level 1 and 2 functions on Shenwei many-core processor. Computer Systems & Applications, 2017,26(11):101−108 (in Chinese with English abstract).

[17] Liu H, Liu FF, Zhang P, Yang C, Jiang LJ. Optimization of BLAS level 3 functions on SW1600. Computer Systems & Applications, 2016,25(12):234−239 (in Chinese with English abstract).

[18] Guo ZH, Guo SZ, Xu JC, Zhang ZT. Register allocation in base mathematics library for platform of heterogenerous multi-core. Journal of Computer Applications, 2014,34(S1):86−89 (in Chinese with English abstract).

[19] https://www.mcs.anl.gov/research/projects/mpi/index.htm

[20] https://www.cs.colostate.edu/cameron/Vsipl.html

[21] Fatica M. Accelerating Linpack with CUDA on heterogenous clusters. In: Proc. of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2009). Washington, 2009. 46−51.

[22] Yang C, Wang F, Du Y, Chen J, Liu J, Yi H. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In: Proc. of the 2010 IEEE Int'l Conf. on Cluster Computing. Heraklion, 2010. 19−28.

[23] Yamazaki I, Tomov S, Dongarra J. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators. Procedia Computer Science, 2012,9(11):37−46.

[24] Yang C, Chen C, Tang T, Chen X, Fang J, Xue J. An energy-efficient implementation of LU factorization on heterogeneous systems. In: Proc. of the IEEE 22nd Int'l Conf. on Parallel and Distributed Systems (ICPADS). Wuhan, 2016. 971−979.

[25] Jo G, Nah J, Lee J, Kim J, Lee J. Accelerating LINPACK with MPI-OpenCL on clusters of multi-GPU nodes. IEEE Trans. on Parallel & Distributed Systems, 2015,26(7):1814−1825.

[26] Li J, Li X, Tan G, Chen M, Sun N. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. In: Proc. of the 26th ACM Int'l Conf. on Supercomputing (ICS 2012). New York, 2012. 377−386.

[27] Li LS, Yang WH, Ma WJ, Zhang Y, Zhao H, Zhao HT, Li HY, Sun JC. Optimization of HPL on complex heterogeneous computing system. Ruan Jian Xue Bao/Journal of Software, 2021,32(8):2307−2318 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6003.htm [doi: 10.13328/j.cnki.jos.006003]

[28] http://www.netlib.org/benchmark/hpl/HPL_pdpanllN.html

[29] Sun CG, Lan J, Jiang H. Genetic algorithm for deciding blocking size parameters of GEMM in BLAS. Computer Engineering & Science, 2018,40(5):798−804 (in Chinese with English abstract).

[30] Low T, Igual F, Smith T, Quintana-Orti E. Analytical modeling is enough for high-performance BLIS. ACM Trans. on Mathematical Software, 2016,43(2):1−18.

[31] Dagum L, Menon R. OpenMP: An industry standard API for shared-memory programming. IEEE Computational Science and Engineering, 1998,5(1):46−55.

[32] https://computing.llnl.gov/tutorials/pthreads/

附中文参考文献:

[10] 顾乃杰,李凯,陈国良,吴超.基于龙芯 2F 体系结构的 BLAS 库优化.中国科学技术大学学报,2008,38(7):854−859.

[16] 孙家栋,孙乔,邓攀,杨超.基于申威众核处理器的 1、2 级 BLAS 函数优化研究.计算机系统应用,2017,26(11):101−108.

[17] 刘昊,刘芳芳,张鹏,杨超,蒋丽娟.基于申威 1600 的 3 级 BLAS GEMM 函数优化.计算机系统应用,2016,25(12):234−239.

[18] 郭正红,郭绍忠,许瑾晨,张兆天.异构多核平台下基础数学库寄存器分配方法.计算机应用,2014,34(S1):86−89.

[27] 黎雷生,杨文浩,马文静,张娅,赵慧,赵海涛,李会元,孙家昶.复杂异构计算系统 HPL 的优化.软件学报,2021,32(8):2307−2318 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6003.htm [doi: 10.13328/j.cnki.jos.006003]

[29] 孙成国,兰静,姜浩.一种基于遗传算法的 BLAS 库优化方法.计算机工程与科学,2018,40(5):798−804.

蔡雨(1988－),男,高级主管工程师,主要研究领域为 CPU 架构,性能优化.

刘子行(1977－),男,高级主管工程师,主要研究领域为安全软件.

孙成国(1985－),男,高级主管工程师,主要研究领域为高性能计算,性能优化.

康梦博(1989－),女,高级工程师,主要研究领域为性能优化.

杜朝晖(1975－),男,主任工程师,主要研究领域为安全软件.

李双双(1984－),男,高级工程师,主要研究领域为数学库.