软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2023,34(7):3451−3463 [doi: 10.13328/j.cnki.jos.006519]

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

# 申威 1621 处理器上矩阵乘法优化研究*

闫 昊[1,2], 刘芳芳[1,3], 马文静[1,3], 陈道琨[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](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 马文静, E-mail: wenjing@iscas.ac.cn

**摘 要**: 稠密矩阵乘法 (GEMM) 是很多科学与工程计算应用中大量使用的函数, 也是很多代数函数库中的基础函数, 其性能高低对整个应用往往有决定性的影响. 另外, 因其计算密集的特点, 矩阵乘法效率往往也是体现硬件平台性能的重要指标. 针对国产申威 1621 处理器, 对稠密矩阵乘法进行了系统性地优化. 基于对各部分开销的分析, 以及对体系结构特点与指令集的充分利用, 对 DGEMM 函数从循环与分块方案, 打包方式, 核心计算函数实现, 数据预取等方面进行了深入优化. 此外, 开发了代码生成器, 为不同的输入参数生成不同版本的汇编代码和 C 语言代码, 配合自动调优脚本, 选取最佳参数. 经过优化和调优, 单线程 DGEMM 性能达到了单核浮点峰值性能的 85%, 16 线程 DGEMM 性能达到 16 核浮点峰值性能的 80%. 对 DGEMM 函数的优化不仅提高了申威 1621 平台 BLAS 函数库性能, 也为国产申威系列多核处理器上稠密数据计算优化提供了重要参考.

**关键词**: 矩阵乘法; 缓存; 分块算法; 优化; 数据预取

**中图法分类号**: TP303

中文引用格式: 闫昊, 刘芳芳, 马文静, 陈道琨. 申威1621处理器上矩阵乘法优化研究. 软件学报, 2023, 34(7): 3451–3463. http://www.jos.org.cn/1000-9825/6519.htm

英文引用格式: Yan H, Liu FF, Ma WJ, Chen DK. Optimization of GEMM on SW1621 Processors. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3451–3463 (in Chinese). http://www.jos.org.cn/1000-9825/6519.htm

## Optimization of GEMM on SW1621 Processors

YAN Hao[1,2], LIU Fang-Fang[1,3], MA Wen-Jing[1,3], CHEN Dao-Kun[1,2]

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

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

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

**Abstract**: General matrix multiply (GEMM) is one of the most used functions in scientific and engineering computation, and it is also the base function of many linear algebra libraries. Its performance usually has essential influence on the whole application. Besides, because of its intensity in computation, its efficiency is often considered as an important metric of the hardware platform. This study conducts systematic optimization to dense GEMM on the domestic SW1621 processor. Based on analysis of the baseline code and profiling of various overhead, as well as utilization of the architectural features and instruction set, optimization for DGEMM is carefully designed and performed, including blocking scheme, packing mechanism, kernel function implementation, data prefetch, etc. Besides, a code generator is developed, which can generate different assembly and C code according to the input parameters. Using the code generator, together with auto-tuning scripts, it is able to find the optimal values for the tunable parameters. After applying the optimizations and tuning, the proposed single thread DGEMM achieved 85% of the peak performance of a single core, and 80% of the performance of the entire chip of 16 cores. The optimization to DGEMM not only improves the performance of BLAS on SW1621, but also provides an important

* 基金项目: 国家重点研发计划 (2020YFB0204601)
收稿时间: 2021-06-07; 修改时间: 2021-08-07; 采用时间: 2021-10-18; jos 在线出版时间: 2022-11-30
CNKI 网络首发时间: 2022-12-01

reference for optimizing dense data computation on SW series multi-core machines.

**Key words**: general matrix multiply (GEMM); cache; tiling; optimization; prefetch

矩阵乘法 (GEMM) 是使用最广泛的基本矩阵操作函数之一. 它不仅在很多计算应用中被频繁调用, 而且是基础代数库 BLAS 中三级函数的基础. 很多其他 BLAS 三级函数的性能提高也依赖于 GEMM 的优化. 因此, 对很多科学和工程应用来说, 提高稠密矩阵乘法的性能, 对加快整体计算速度有重要作用. 而 GEMM 函数的优化, 又严重依赖于硬件平台, 函数实现必须充分利用处理器特性才能发挥出计算平台的计算能力. 因此, 各处理器制造厂商都在致力于提供针对自己生产的处理器进行过优化的 BLAS 函数库, 尤其是进行了深度优化的 GEMM 函数. 例如, Intel 公司提供优化的数学库 MKL[1]. AMD 公司开发了数学库软件 AOCL[2], 面向 AMD 处理器进行优化. 在 GPU 上, NVIDIA 公司提供深度优化的数学库软件 cuBLAS[3], AMD 开发了 AMD GPU 上的高性能库 rocblas[4], 并且开放到开源社区. 此外, GotoBLAS[5,6], OpenBLAS[7], KBLAS[8]等开源基础函数库, 也都对 GEMM 进行了针对不同硬件平台的优化.

申威 1621 处理器是我国自主研发的多核高性能处理器, 采用自主 sw 指令集. 由于申威处理器的架构, 存储层次及所使用的指令集与国外商用 CPU 有很大不同, 因此在申威处理器上无法使用专为国外商用处理器优化的 BLAS 库. 而对开源库进行简单适配的版本性能又远远低于处理器所提供的性能. 基于此种状况, 研制面向申威 CPU 的 BLAS 库成为一个亟待实现的任务. 刘昊等人对早期申威处理器 (申威 1600) 上的 GEMM 进行了优化[9], 解庆春等人进行了向量化等方面的探索[10]. 但是, 申威 1621 芯片与早期的 1600 芯片在指令发射方式, 缓存大小等方面, 有较大不同. 因此, 在申威 1621 平台上需要对 GEMM 函数的实现进行重新设计并进一步优化, 以充分发挥其硬件特性.

我们在申威 1621 平台上对 GEMM 从访问优化, 指令排布以及减少不必要计算和访存等方面, 进行了深度优化, 并开发了子函数代码生成器, 为不同规模的矩阵生成优化的代码并进行参数空间搜索. 相比于 GotoBLAS, 单线程 DGEMM 实现 7.39 倍加速, 达到了系统峰值性能的 85%, 16 线程性能达到了系统峰值的 80%.

## 1 研究现状及背景介绍

稠密矩阵乘法 GEMM 的计算公式是:

$$C = \alpha AB + \beta C \tag{1}$$

公式 (1) 中, 矩阵 $A$, $B$, $C$ 大小分别是 $M\times K$, $K\times N$, $M\times N$, $\alpha$ 和 $\beta$ 是标量, 矩阵 $A$ 和 $B$ 可取原矩阵或其转置, 矩阵 $A$ 和矩阵 $B$ 均非转置的稠密矩阵乘法朴素算法如算法 1 所示.

**算法 1**. GEMM 朴素算法.

```
1.  for i = 0 : M – 1 step 1
2.      for j = 0 : N – 1 step 1
3.          C_ij = βC_ij
4.          for l = 0 : K – 1 step 1
5.              C_ij = αA_il × B_lj + C_ij
6.          end for
7.      end for
8.  end for
```

GEMM 的浮点运算量是 $2MNK$, 计算复杂度是 $O(N^3)$; 访存量是 $MK+KN+2MN$, 访存复杂度是 $O(N^2)$, 是典型的计算密集型问题. 理论上讲, GEMM 浮点操作性能应逼近硬件浮点峰值. 而在 GEMM 函数实现中, 除了 $O(N^3)$ 的运算操作, 访存操作也要占用大量时间, 属于不可避免的额外开销. 因此, 设计高效的 GEMM 实现, 不仅要充分优化计算指令流, 还要考虑如何减少或掩盖访存用时.

目前 GEMM 的优化主要有 3 个方面: (1) 针对特定的平台, 在充分考虑硬件架构特征的基础上设计分块方案和单线程/多线程算法; (2) 对核心代码进行精细手工优化, 例如, GotoBLAS 就提供了几种不同 CPU 上的汇编核心代码; (3) 基于自动调优 (auto-tuning), 通过在不同配置下运行程序收集性能表现数据, 自动选择算法和参数. 该方法的代表是 ATLAS 数学库. 在库函数的开发当中, 往往需要将 3 种优化思路结合使用, 以期在提高开发效率的同时, 充分挖掘平台性能潜力.

各经典 BLAS 库中的 GEMM 函数及各方研究人员开发的 GEMM 函数中, 用到了多种体系结构相关的优化. 在广泛使用的开源软件 GotoBLAS 中, Goto 等人考虑到硬件上缓存容量和 TLB 表项容量的限制, 给出了一系列分块的实际原则以及分块参数选择方法, 具有普适的指导性[5,11]. Lim 等人在 Intel KNL 架构上通过使用矩阵分块, 数据预取, 循环展开, 采用 AVX512 向量指令等方法来优化 GEMM 函数, 最终性能达到了 MKL DGEMM 性能的 99%[12]. Smith 等人对多核处理器上 GEMM 的并行和优化方式进行了深入分析并提出了比较系统的优化方案[13]. 近年来, 国产处理器迅速发展, 基于国产芯片的 GEMM 优化方法也应运而生. 张明针对龙芯平台, 围绕常用基础数学库函数的访存和计算关键问题展开研究[14]. 对 GEMM 函数, 他提出一套适用于龙芯处理器的异步计算访存优化方法——通过流水化和分组处理计算任务, 充分掩盖了访存用时. 刘昊等人基于国产申威 1600 平台, 使用乘加指令, 向量化指令, 循环展开与软件流水, 数据预取与数据重排等优化技术, 给出了一种 GEMM 的高性能实现方法[9]. Jiang 等人在神威太湖之光超级计算机上实现了高性能的 GEMM, 针对申威 26010 众核平台, 在不同存储层次上设计了分块方案, 并使用申威众核平台提供的异步 DMA 等功能充分掩盖了访存用时[15].

ATLAS 数学库采用了自动调优的方法, 在不同配置下运行程序, 收集性能数据, 选择性能表现最优的参数组合作为调优结果[16]. 近些年出现的 TVM 深度学习编译器也借鉴了这一思路, 在一些 GPU 平台上, 对部分 GEMM 输入规模加速效果明显[17,18].

国产申威 1621 平台[19]有 16 个计算核心, 具有三级缓存结构. 每个核有独立的一级缓存和二级缓存, 所有核共享三级缓存. 申威 1621 处理器向量长度为 32 字节, 即可以同时操作 4 个双精度浮点数. 此外, 指令集中还包含预取指令, 可以将某个内存地址所属缓存行的数据从主存读入缓存. 针对申威 1621 多核平台的体系结构特点和访存特性, 我们设计并实现了 GEMM 单线程算法和多线程并行算法, 并进行了深入优化. 本文以矩阵 $A$ 和矩阵 $B$ 均非转置, 数据类型为双精度的 DGEMM 为例来说明实现以及优化技术, 其他输入情况和精度可以使用类似的优化方法.

## 2 面向申威 1621 处理器的 DGEMM 优化

本节介绍单线程 DGEMM 在申威 1621 平台上的实现方案及优化方法. 我们将文献 [9] 中基于申威 1600 处理器的 GEMM 实现与优化方法作为基础版本. 在此基础上, 我们针对 1621 处理器的特点, 从分块方式, 打包方式, 及核心计算优化等各方面, 进行了系统的分析与优化, 提出一套 GEMM 高效实现方案及基于此方案的一系列优化技术.

### 2.1 基础实现方案

本节介绍文献 [9] 所采用的 DGEMM 方案及实现, 即前文所述申威 1621 平台上进行 DGEMM 优化的基础版本.

#### 2.1.1 分块方案及实现

为有效利用现代处理器上的分层缓存结构, GEMM 采用分块算法, 每个维度上的循环迭代步长是此维度的分块大小, 最内层循环计算一个分块的矩阵乘法. 在循环顺序上, 我们采用了与 GotoBLAS2 相同的 N-K-M 的循环顺序. 鉴于传统 BLAS 库中矩阵使用列优先存储, 矩阵 $A$ 和 $C$ 都在 M 方向上连续, 这样的循环顺序可以保证计算时尽可能多的访问矩阵中连续元素, 提高缓存命中率, 从而减少访存开销. 程序中使用打包函数 packA 和 packB 将矩阵中不连续的数据放置在连续缓冲区中. 这样一来, 执行计算任务的 KERNEL 函数就可以从缓冲区中读取或写回数据, 避免了大量的不连续访存.

算法 2 展示了单线程 DGEMM 的分块算法. 矩阵 $A$, $B$, $C$ 分别被分为 $BM\times BK$, $BK\times BN$, $BM\times BN$ 大小的子矩阵

块 (第 2, 3, 5 行), 记为§$A_{il}$, §$B_{lj}$, 和§$C_{ij}$. 在循环开始之前, 将 $C$ 矩阵全部乘以 $\beta$. 在计算过程中, 子矩阵块§$A_{il}$ 和§$B_{lj}$ 中的元素经过打包被放入缓冲区 $\tilde{A}$和$\tilde{B}$ (第 4, 6 行) 后, 函数 KERNEL 完成子矩阵块乘法的计算 (第 7 行). 在文献 [9] 所使用的原始版本中, 对 $C$ 矩阵未使用缓冲区, 而是在 KERNEL 中直接写入. 我们认为, 不对 $C$ 矩阵使用缓冲区, 是因为对 $C$ 的解包要在最内层循环每次迭代进行, 与 packA 的调用频率相同, 开销过大, 性能反而会受损. 我们也通过实验验证了上述推断.

**算法 2**. GEMM 分块算法.

1. $C = \beta C$
2. for $j = 0 : N - 1$ step $BN$
3. for $l = 0 : K - 1$ step $BK$
4. pack §$B_{lj}$ into $\tilde{B}$
5. for $i = 0 : M - 1$ step $BM$
6. pack §$A_{il}$ into $\tilde{A}$
7. 调用计算函数 KERNEL(), 计算 $C_{ij}$ += $\alpha\tilde{A} \times \tilde{B}$
8. end for
9. end for
10. end for

### 2.1.2 汇编 KERNEL 实现

函数 KERNEL() 完成 $M$, $N$, $K$ 这 3 个维度大小分别为 $BM$, $BN$, $BK$ 的矩阵乘法操作. 函数使用汇编语言开发, 使用向量化指令和乘加指令, 以充分发挥申威多核平台的浮点运算能力, 具体实现见算法 3. 函数主体是 N-M-K 三层循环. 在最内层循环中, 矩阵 $A$ 中的元素被装载到寄存器 $A_r$ 中, $B$ 中元素被装载到寄存器 $B_r$ 中 (第 4 行), 对 $A_r$ 和 $B_r$ 进行乘法运算, 结果放入中间变量 $C_t$ 中 (第 5 行). 待 K 方向循环结束后, 进行写回操作, 即将 $C$ 矩阵中元素装载到 $C_r$, $C_r$ 和中间变量 $C_t$ 累加后, 结果被写回 $C$ 矩阵在内存中的地址 (第 7, 8, 9 行). KERNEL() 的计算过程如图 1.

**算法 3**. 计算函数 KERNEL($A$, $B$, $C$).

1. for $j = 0 : BN - 1$ step $RN$
2. for $i = 0 : BM - 1$ step $RM$
3. for $l = 0 : BK - 1$ step $RK$
4. load $A$, $B$ to $A_r$, $B_r$
5. $C_t = C_t + A_r \times B_r$
6. end for
7. load $C$ to $C_r$
8. $C_r = C_r + C_t$
9. store $C_r$ to $C$
10. end for
11. end for

图 1 所示, 在 K 方向上一次迭代中, 使用向量化指令来装载矩阵 $A$ 一列上的 $RM$ 个元素, 使用标量指令装载矩阵 $B$ 一行中的 $RN$ 个元素并将每个元素复制扩展为向量. $RM$ 个 $A$ 矩阵向量与 $RN$ 个 $B$ 矩阵向量相乘后得到 $RM$×$RN$ 个中间结果 $C_t$. K 方向 $RK$ 次迭代后, 相加并写回到矩阵 $C$. 考虑到申威 1621 处理器向量化长度是 4, 所以 K 方向上一次迭代中, 矩阵 $A$ 中元素需使用 $VM$=$RM$/4 个寄存器, 矩阵 $B$ 中元素需使用 $RN$ 个寄存器. 此外, 在

K 方向上循环展开 2 次, 为了充分利用指令级并行, 我们对展开的两个最内层循环使用不同的寄存器, 即矩阵 $A$ 和矩阵 $B$ 中元素分别使用 $VM\times2$ 和 $RN\times2$ 个寄存器. 存储 $C$ 矩阵中间结果需要使用 $VM\times RN$ 个寄存器. 因此总共需要的寄存器数为

$$reg\_used = VM\times2+RN\times2+VM\times RN \tag{2}$$

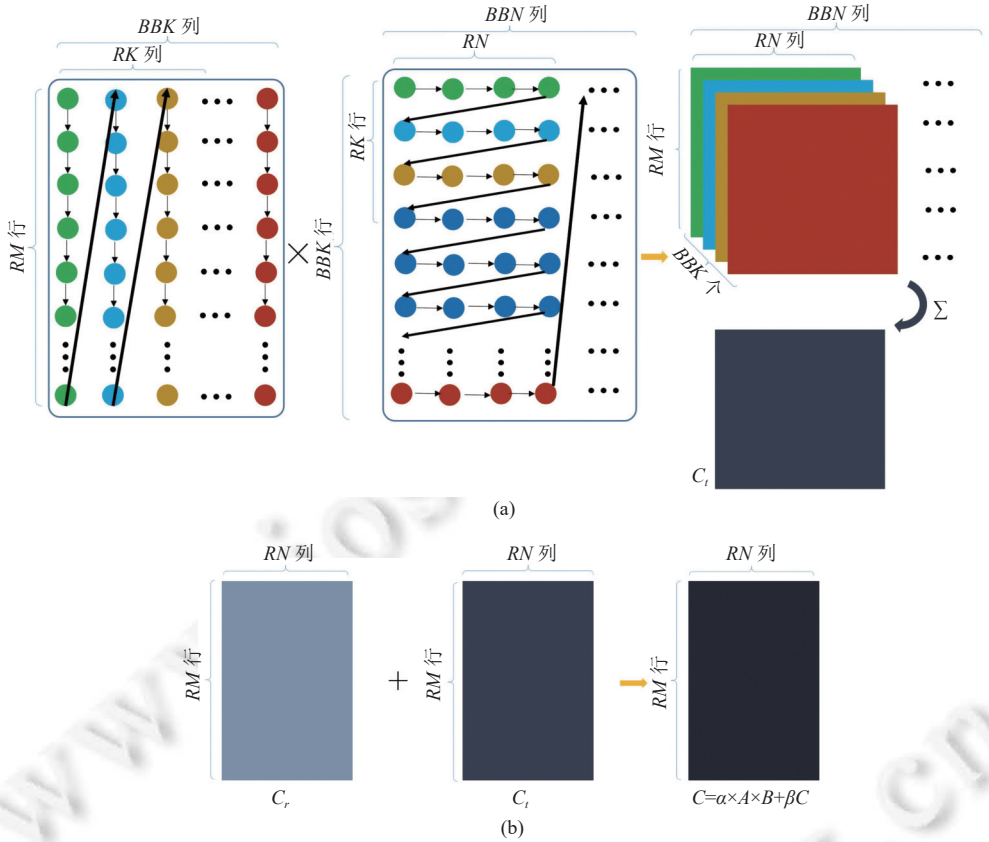


图 1 函数 KERNEL 图示

循环最内层共有 $VM\times RN$ 条计算指令, $VM+RN$ 条访存指令, 为使得计算指令与访存指令的比例 $\frac{VM\times RN}{VM+RN}$ 最大, 应使 $VM=RN$. 根据公式 (1), 当 $VM=RN=4$ 时, 共需要 32 个寄存器. 而申威 1621 平台共有 31 个可用浮点寄存器. 为了充分利用寄存器资源, 我们在不影响性能的前提下, K 方向上两次迭代中重用了一个寄存器, 此时计算指令比与访存指令比例达到 2:1. 如果有更多的寄存器被重用, 可以进一步增大计算访存比. 在 K 方向展开的两次迭代中, 按照最大可能的重用度, $A$ 和 $B$ 各只有一个寄存器不重用, 那么只需要 $VM+1+RN+1+VM\times RN$ 个寄存器. 基于这种方案, 可将现有的最内层分块增大, 假设在 M 方向增加 1, 那么将会用到 $VM+1+RN+1+VM\times RN=31$ 个寄存器, 也就是全部浮点寄存器. 这种做法, 虽然能够在一定程度上提高计算访存比, 但是由于每个 panel 对 $A$ 和 $B$ 只有一个非重用的寄存器, 这就意味着最后的 3 条读取指令与倒数第 4、3、2 条乘加指令存在寄存器重用. 而神威 6A 处理器采用 2 条浮点乘加指令和 2 条访存指令同时发射的方式, 即浮点操作与访存双发射. 上述情况下, 由于浮点计算指令与读取指令所用的寄存器有重叠, 因此在最后两条指令上无法实现双发射, 从而影响性能. 我们对这种分块方式的 KERNEL 进行了测试, 性能略低于 $VM=RN=4$ 的分块. 因此, 本文延续使用了 $VM=RN=4$ 的分块方式.

为了提高数据访问的效率, KERNEL 中插入了针对 3 个矩阵的预取操作. 由于 cache line 长度为 128 字节, 每次预取操作能够读取 16 个双精度浮点数. 因此, 对 $A$ 的预取发生在 K 方向上的每次迭代中. 而对 $B$ 的读取是每 4 次迭代处理 16 个双精度浮点数, 因此其预取频率是 $A$ 的 1/4. $C$ 矩阵的预取放置在 K 方向迭代之前, 即算

法 4 第 3 行之前. 由于从主存中读取数据至 cache 的时间较长, 预取操作不能直接预取下一次迭代所需数据, 而是要设置一定的预取步长 (fetch_stride). 也就是说, 每次预取操作要将 fetch_stride 个字节之后的 128 字节数据载入缓存中.

### 2.2 单线程 GEMM 优化方案

在现代计算机体系结构上实现高性能矩阵乘法, 分块方法的优化是必不可少的[11,20]. 尽管算法 2 对分块算法做了优化, 使用汇编代码实现计算函数 KERNEL, 并在 KERNEL 中使用了预取指令来优化访存, 但是其性能仍然差强人意. 通过对各部分进行计时和分析, 我们发现, 在申威 1621 平台上, GEMM 函数还有很大的性能提升空间. 首先, 打包函数时间过长, 这种额外开销降低了函数的整体效率. 此外, 我们将 KERNEL 中的访存指令删去, 其性能几乎可以达到峰值性能. 加入访存指令后, 性能大幅下降, 可见访存效率还有待提高. 1621 处理器 L1 缓存仅有 32 KB, 基础版本代码的分块大小 $A$ 子矩阵就已经超过了 L1 缓存大小. 为提高函数整体性能, 我们考虑以下因素, 并提出有针对性的优化方法.

首先, 算法 2 中的分块算法无法兼顾打包开销的减少和 KERNEL 性能的优化. 一方面, 较大分块意味着更多的缓存缺失率, 影响 KERNEL 访存效率; 另一方面, 如果分块较小, 又会产生大量的重复打包操作, 增加打包操作引起的访存开销. 虽然通过调节分块大小能在一定程度上改善程序性能, 但是要想在打包和 KERNEL 两个环节都有性能提升, 需要对循环结构进行进一步优化以减少这两种开销. 其次, pack 函数的运行时间是不可忽略的, 故对此类内存拷贝函数, 也需要进行精细的分析和优化. 此外, KERNEL 的实现中也有一些可改进之处.

基于上述分析, 我们提出了兼顾减少打包开销和提高 KERNEL 效率的二层分块方案, 并对打包函数及 KERNEL 函数的实现都进行了优化. 基于优化之后的程序, 我们进一步对 KERNEL 的实现在减少访存量和提高数据预取质量方面进行了优化. 下面详细介绍各项优化技术.

#### 2.2.1 二层分块及分块参数确定

算法 2 中, 函数 packA() 执行次数是 $N/BN\times K/BK\times M/BM$. 单次执行访问的数据量是 $BM\times BK\times sizeof(double)$. 设机器带宽是 bandwidth, 那么单次用时是 $BM\times BK\times sizeof(double)/bandwidth$ , 所以函数 packA() 总用时为 $N\times K\times sizeof(double)/bandwidth$ . 同理, 函数 packB() 总用时为 $N\times K\times sizeof(double)/bandwidth$ .

当输入规模固定时, 函数 packB() 用时只取决于机器带宽; 函数 packA() 用时反比于分块参数 $BN$, $BN$ 越大函数 packA() 用时越少. 但另一方面, 如果 $BN$ 过大, 函数 KERNEL() 访问的子块数据量就会超出一级缓存容量, 导致缓存换入换出频繁, 增加不必要的开销. 为充分利用 $BN$ 较大时函数 packA() 用时更少, $BN$ 较小时 KERNEL() 中缓存效率更高的特性, 在调用计算函数时加入了第 2 层分块, 将程序改为 6 层循环, 如算法 4 所示. 外层使用较大分块, 在外层分块中调用函数 packA() 和函数 packB(), 将 $A$, $B$ 矩阵中的子块打包放入大小分别为 $BM\times BK$ 和 $BK\times BN$ 的缓冲区 $\tilde{A}$和$\tilde{B}$ 中 (第 3 行和第 5 行). 内层使用较小分块, 3 个维度分别为 $BBM$, $BBN$ 和 $BBK$. 函数 KERNEL() 处理 $BBM\times BBN$ 大小的子块, 进行 $BBK$ 次累加. 因此, packA() 和 packB() 将 $\tilde{A},\tilde{B}$ 按 $BBM\times BBK$ 和 $BBK\times BBN$ 大小的子块排列, 也就是每个 $BBM\times BBK$ 大小的 $A$ 子矩阵连续存放. 这样一来, 只要 $BBM$, $BBK$ 和 $BBN$ 足够小, 那么每次调用 KERNEL 时, $A$ 矩阵和 $B$ 矩阵在缓存中的数据可以得到比较充分的重用. 在我们的实现中, $BBM$ 取 4 个向量的长度, 即 KERNEL 中一次循环处理的分块 M 方向大小. $A$ 子块驻留于 L1 缓存中, $B$ 子块驻留于 L2 缓存. 在 KERNEL 内部, 还有 2 层循环, 外层为 N 方向循环, 内层为 B 方向循环. 由于 KRENEL 内部 N 方向分块大小仅为 4 ($RN$=4), 在 KERNEL 内部可以保证 $A$ 子矩阵驻留于 L1. 当 $BBN$ 足够小时, 能保证 $B$ 子块驻留于 L2 缓存.

**算法 4**. 二层分块算法.

1. $C=\beta C$
2. for $j=0:N-1$ step $BN$
3. for $l=0:K-1$ step $BK$

```
4.        pack §B_lj into B̃
5.        for i = 0 : M – 1 step BM
6.          pack §A_il into Ã
7.          for v = 0 : BN – 1 step BBN
8.            for w = 0 : BK – 1 step BBK
9.              for u = 0 : BM – 1 step BBM
10.               调用计算函数 KERNEL(), 计算 C_ij += αÃ_uw ×B̃_wv,
11.             end for
12.           end for
13.         end for
14.       end for
15.     end for
16.   end for
```

2.2.2 设置 $C$ 矩阵缓冲区

如前所述, 在基础版本中, 对 $C$ 矩阵未使用缓冲区, 而是在 KERNEL 中直接写入. 这是因为频繁的解包操作开销过大. 但通过分析和实验, 我们发现, 如果使用更大的缓冲区, 那么就可以将 $C$ 矩阵的解包放到 M 方向循环之外, 这样就能保证只对 $C$ 矩阵解包一次. 这种情况下, 使用缓冲区的性能收益超过 $C$ 矩阵的解包操作开销. 因此, 我们在第 2.1.2 节优化的基础上, 对 $C$ 矩阵也使用了缓冲区. 要保证只对 $C$ 矩阵解包一次, 需要将解包操作放在 K 方向最外层循环 (算法 4 第 14 行和第 15 行之间), 而 M 方向的两层循环都在 K 方向外层循环的内部, 这就意味着 $C$ 缓冲区 $\tilde{C}$ 的大小应为 $M\times BN$. 也就是说, 缓冲区 $\tilde{C}$ 需要在内存中占据的空间达到整个矩阵规模的 $BN/N$. 在 $M$ 维度很大时, 采用减小 $BN$ 的方法来避免内存分配失败.

2.2.3 打包函数优化

通过简单的计时统计, 我们发现, 打包操作的时间是不可忽略的, 因此, 对 packA 和 packB 操作, 我们对其实现进行了深入分析和优化.

由于 KERNEL 函数处理的是 $BBM\times BBK$ 大小的 $A$ 矩阵子块, 因此 packA 要将矩阵中每个 $BBM\times BBK$ 大小的子块连续存放, 如图 1 所示, $\tilde{A}$ 中元素按数据块存储, 每一块大小是 $RM\times RK$, 块与块之间在行方向上连续, 块内在列方向上连续. 原始的打包操作, 对每个子块分别调用 packA, 也就是每次读取矩阵 $A$ 同一列上的 $BBM$ 个元素, 并放入缓冲区. 然后再读入下一列的 $BBM$ 个元素, 存入缓冲区. 这种做法每次读取的连续数据仅有 $BBM$ 个, 容易造成 cache 缺失, 在 $M$ 较大时还会有较高的 TLB 缺失. 而 packA 本身要被调用多次, 因此这种 cache 缺失和 TLB 缺失造成的开销会被进一步放大. 为了提高打包操作的性能, 我们对改进的循环进行分析, 发现二层分块给了我们改进 packA 操作的机会. 在这种循环模式下, 我们可以通过使用更大的缓冲区, 将打包函数改为集中式操作. 也就是说, 在分配缓冲区 $\tilde{A}$ 时, 同时分配 $BM/BBM$ 个缓冲区. 先读取 M 方向上第 1 个列向量, 存入缓冲区 $\tilde{A}$ 中相应的位置. 然后读取 M 方向上下一个列向量. 依次类推, 直到全部 $BM$ 个元素都已存入缓冲区, 然后开始处理下一列, 如图 2 所示. 这样, packA() 函数可以实现 M 方向上的连续读取. 此外, 由于读取与写入都是列主序, 也就是读出和写入都是连续数据, 因此可以使用向量化指令完成读写, 进一步提高效率.

类似的, 函数 packB() 也要将 $B$ 中数据打包成 $BBK\times BBN$ 大小的子块. 而且, 由于 KERNEL 中每次迭代处理的是 N 方向上的 $RN$ 个元素, 打包时需要将 N 方向上每 $RK\times RN$ 个元素打包. 在基础版本程序中, $RK$=2, $\tilde{B}$ 中元素按行优先存储, 打包操作以 4×4 的小块为单位, 如图 1 所示. 这种存储方式使得在 KERNEL 中 $\tilde{B}$ 的元素读取都是连续的, 但它使 packB 函数的输入和输出使用了不同的存储方式, 因此无法使用向量化操作. 为了提高 packB 函数的运行效率, 我们将 KERNEL 内的循环从原来的 2 次展开改为 4 次展开, 如图 3 所示, 不但减少了分支判断语句的开销, 而且在 KERNEL 中将会连续访问 K 方向上的 4 个 $B$ 元素. 这种访问方式允许我们对 $\tilde{B}$ 重新布局, 将其改为

列优先存储. 也就是说, 一列当中 (K 方向上) 的连续 4 个 $B$ 元素在打包之后依然连续存储. 采用这种布局, 就可以使用向量化指令来完成 packB() 的操作.

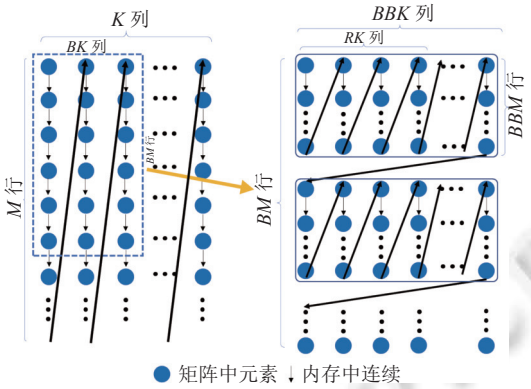


图 2 改进的函数 packA 示意图

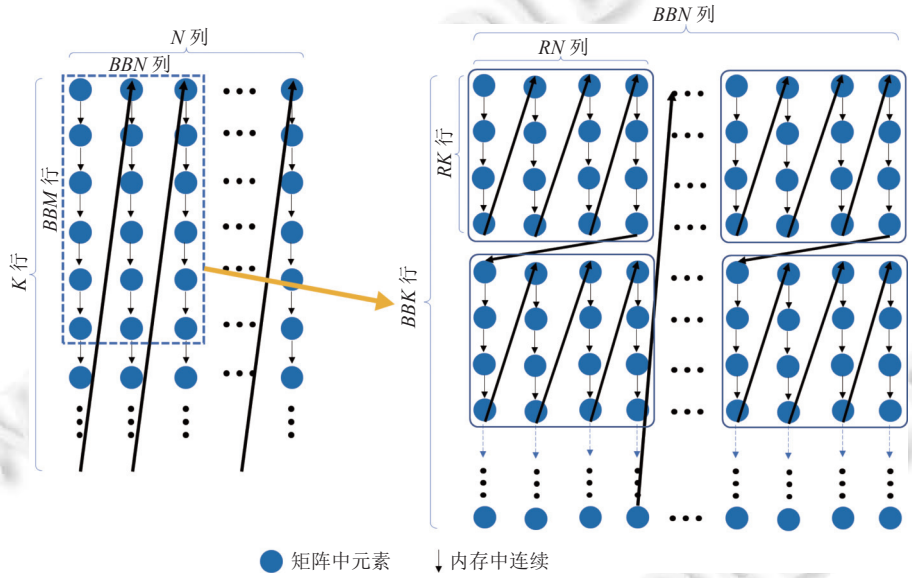


图 3 改进的函数 packB 示意图

2.2.4 减少对 $C$ 矩阵的不必要读写

在本节及第 2.2.5 节中, 我们介绍对计算 KERNEL 的优化, 包括减少对 $C$ 矩阵的不必要读写, 及针对预取操作进行的优化.

在 KERNEL 计算中, K 循环结束时, 需要将矩阵 $C$ 中的数据读出, 并与 $A$ 和 $B$ 的乘积结果相加, 再将加和的结果写回 $C$. 在我们的改进方案中, 由于 $C$ 矩阵使用了缓冲区, 我们可以在缓冲区 $\tilde{C}$ 中只保存中间结果, 而将与原始 $C$ 矩阵块加和的操作推迟. 基于这种设计, 我们开发了 3 个 KERNEL 函数, 分别用在 K 方向的不同迭代阶段, 从而尽可能减少对 $C$ 矩阵的读写.

在 K 方向第 1 次迭代中, 缓冲区 $\tilde{C}$ 中的初始值全部是 0, 因此 KERNEL 不需要读取 $C$ 的值, 而是直接将乘积写入缓冲区 $\tilde{C}$, 此为第 1 个版本 KERNEL_INIT. 之后的迭代中, $C$ 矩阵的读写都发生在缓冲区 $\tilde{C}$, 即将缓冲区 $\tilde{C}$ 中的数据读出, 加上本次迭代中 $A$ 与 $B$ 的乘积, 然后写回 $\tilde{C}$ 缓冲区. 这些迭代中的 KERNEL 操作即为原始的 KERNEL 实现, 此处我们命名为 KERNEL_GENERIC. 当到达 K 方向最后一次迭代时, 我们开发了 KERNEL_UPDATE, 在此函数中把 unpackC 融合进来, 读取原始矩阵中 $C$ 矩阵的值, 与本次迭代的乘积及 $\tilde{C}$ 中的值加和, 并写回原始 $C$

矩阵, 也就是完成 $C:=\alpha\tilde{A}\times\tilde{B}+\beta\tilde{C}$ 的操作. 使用 3 个 KERNEL 后, 算法 4 第 9 行计算函数调用逻辑如算法 5 所示. 此方案中, $C$ 与 $\beta$ 相乘的操作也被融合到 KERNEL_UPDATE, 进一步减少了对 $C$ 的读写.

**算法 5**. 在 K 方向不同阶段使用 3 个版本的 KERNEL 函数.

```
1.  if C̃ 上第 1 次 K 方向迭代
2.      KERNEL_INIT();
3.  else if C̃ 上最后一次 K 方向迭代
4.      KERNEL_UPDATE();
5.  else
6.      KERNEL_GENERIC ();
7.  end if
```

2.2.5 针对预取操作的优化

原始版本的 KERNEL 虽然对寄存器使用等进行了精细调优[9], 但我们发现, 访存操作仍有可优化空间. 此外, 二层分块的结构也使 KERNEL 的访存行为有了变化. 基于对 KERNEL 访存特征的分析, 我们对 KERNEL 进行了以下两方面的改进.

首先, 原有 KERNEL 中对 $C$ 矩阵中每一小块数据 ($RM\times RN$ 个浮点数) 的读取都发生在一次迭代的写回阶段, 即 K 方向全部迭代之后 (算法 3 第 7 行), 与前面循环体中的计算没有任何重叠. 实际上, 我们可以在不破坏寄存器使用依赖关系的前提下, 对 $C$ 中的数据提前读入, 以充分利用指令级并行, 减少写回操作时对 $C$ 数据的等待. 为实现此方案, 我们将 K 方向最后一次迭代从循环中剥离, 也就是将其放置于循环体之外, 然后在其中插入若干个对 $C$ 矩阵元素的读取指令. 为了保证寄存器重用的安全性, 我们选取了 9 个 $C$ 中的元素, 使它们的读取与 K 方向最后一次迭代的计算重合, 并在插入对矩阵元素 $C_i$ 的读取指令时, 确保 $C_i$ 使用的寄存器 $R_i$ 在此读取指令和写回部分之间不被使用.

其次, 在原始程序中, 由于对 $A$ 的预取发生在算法 3 第 3 到 6 行的循环中, 因此在每次迭代中对 $A$ 的预取都使用同样的指令 (即预取距离当前 $A$ 指针同样步长的数据). 实际上, 每次 K 方向循环开始时, $A$ 都要被从头重新读取. 所以在 K 方向的最后几次迭代读取的是无效的数据, 而 K 方向起始的几次迭代中所使用的矩阵 $A$ 数据却没有被预取. 为了解决这个问题, 我们通过插入新的预取指令对 $A$ 的访存做了进一步优化. 如上一段所述, 我们已将 K 方向最后一次迭代置于循环体之外. 在这段代码中, 我们删除了原有的对矩阵 $A$ 在 K 方向下一块数据的预取指令, 改为预取当前 $A$ 子块 (算法 4 第 9 行中的 $\tilde{A}_{uw}$) 起始部分的数据.

矩阵 $B$ 的预取也有类似的问题, 在每次 KERNEL 调用的末尾, 使用同样的预取指令, 读取的是下一块矩阵 B 的值. 而实际上, 只要不是 N 方向最后一次迭代, 就无需预取下一 $B$ 子矩阵中的数据. 基于这种考虑, 我们将每个 KERNEL 进一步分为两个. 第 1 个 KERNEL 处理 FBN 列, 第 2 个 KERNEL 处理 $LBN=BBN-FBN$ 列. 在第 2 个 KERNEL 中, 插入对当前 $B$ 子矩阵起始部分数据的预取指令. 预取指令放在 K 方向循环体中. 如前所述, 改进的 KERNEL 中 K 方向是 4 次展开, 也就是每 4 次迭代, 进行一次矩阵 $B$ 的预取. 两次 KERNEL 调用使用同样的矩阵 $B$ 预取频率. $FBN$ 和 $LBN$ 的大小决定了预取指令的间隔. 当 $LBN$ 减少时, 预取距离不够; 当 $LBN$ 增加时, 预取的数据可能在二级缓存中再次被换出, 都会使性能都有所降低. 我们通过自动调优选取最佳的 KERNEL 切割点, 即 $FBN$ 和 $LBN$ 的值.

### 2.3 多线程优化技术

对多线程矩阵乘法, 我们采用了与 GotoBLAS 相同的并行方案[11], 即在 M 方向上划分任务分配给各个线程, 如图 4 所示. 原问题在 N 方向上分块后, 对 $C$ 矩阵在 N 方向上每一块 (即每个列块), 进一步在 M 方向上进行划分, 分配给各个线程并行计算. 这样划分任务后, 每个线程计算子数据块时, 各自使用无重叠区域的 $A$ 矩阵子块, 线程间没有数据依赖; 多个线程使用相同的 $B$ 矩阵子块, $B$ 矩阵子块的打包任务被划分到多线程共同执行, 打包后的缓冲区中 $B$ 子矩阵由多个线程共同使用.

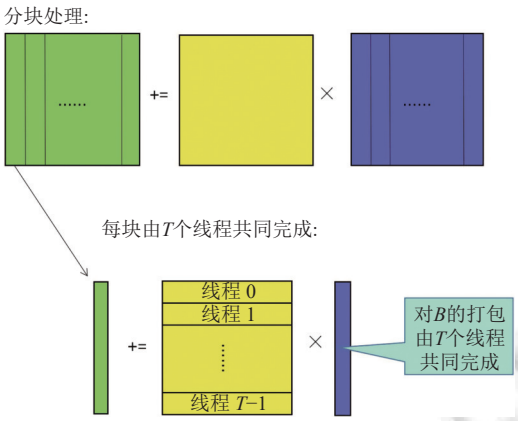


图 4 多线程并行矩阵乘法任务划分示意图

单线程 GEMM 的优化技术均适用于多线程 DGEMM. 区别在于, 多线程 GEMM 在部署分块方案时, N 方向需要在二层分块的基础上, 增加一层线程级分块.

## 3 子函数代码生成器及自动调优

在实现过程中, 我们开发了 KERNEL 和打包函数的代码生成器, 使用 Python 语言编写代码生成脚本. 通过调整参数, 该脚本可以生成不同版本的 KERNEL 函数和打包函数代码.

KERNEL 函数的代码生成器将预取步长, 循环展开次数等作为参数传递, 可以生成使用不同优化策略和参数的 KERNEL 汇编代码. 对 *A*, *B*, *C* 这 3 个矩阵, 我们分别设置了预取步长 FS_A, FS_B, 和 FS_C. 由于 *C* 矩阵在 K 方向循环末尾使用, 因此 FS_C 设为 0, 即 K 方向循环开始之前便预取本次 M, N 方向所需 *C* 子矩阵块. FS_A 和 FS_B 通过遍历搜索选取. 理论上讲, FS_A 的值应根据指令数与访存带宽计算. 设从主存装载数据到 L1 cache 需要 *W* 个时钟周期, K 方向一次循环执行 *M* 条指令, 每时钟周期发射指令数为 IPC, 那么预取应该在 $W\times P/M$ 迭代之后, 因此 FS_A 应等于 $W\times IPC/M\times 16\times sizeof(double)$. 我们根据以上推导, 确定出 FS_A 的大致初始值范围, 参数搜索将在初始值的周围进行遍历. FS_B 的初始值可用类似方式确定. 打包函数使用 C 语言实现, 并采用内嵌汇编实现向量化的数据读写.

为了选取最优参数, 我们编写了自动调优脚本, 对参数空间进行遍历搜索. 需要调优的参数包括第 1 层分块和第 2 层分块的分块大小 (*BM*, *BN*, *BK*, *BBM*, *BBN*, *BBK*), KERNEL 中的预取步长 (FS_A, FS_B), 及第 2.2.5 节所述的 N 方向 KERNEL 切分位置 (*FBN*) 等.

经过实验遍历搜索后得出性能最佳的分块和预取参数组合如表 1 和表 2 所示. 第 2 层分块参数 *BBM* 的值和寄存器层分块参数 *RM* 相同, 即 KERNEL 内部在 M 方向完全展开, 不做循环处理. 由于本文所做优化主要面向较大规模的矩阵, 因此搜索参数空间测试采用的是 $N$=16 384 的矩阵. 当矩阵规模较小, 或者呈现极不规则的形状时, 由于 GEMM 操作的计算/访存占比相较大规模矩阵的情形已经出现了较为明显的变化[21], 本文提出的策略不再适用于此类问题的优化. 此类小规模及不规则形状矩阵乘法的优化不在本文讨论的范围之内.

表 1 单线程 DGEMM 循环分块参数表

| 维度 | 第1层数据分块 | 第2层数据分块 |
|---|---|---|
| *M* | 1 024 | *RM* |
| *N* | 4 096 | 256 |
| *K* | 256 | 128 |

表 2 DGEMM KERNEL 内部预取步长参数表

| 矩阵 | L1预取步长 | L2预取步长 |
|---|---|---|
| *A* | 96 | 160 |
| *B* | 64 | 160 |

对多线程 GEMM, 我们采用类似的方法, 对分块参数进行遍历搜索, 得出的多线程 GEMM 最优分块参数如表 3 所示. 由于 L1 和 L2 缓存为各核心私有, 故 KERNEL 内部最优预取步长与单线程相同. 而 L3 缓存是各核心

共用的, 在外层循环中, 多线程环境下每次迭代中同时处理的数据量增加, 因此容易产生更多的 L3 缓存缺失及更多的缺页. 所以经遍历选得的最优分块大小小于单线程外层分块大小.

表 3 多线程 DGEMM 关键分块参数表

| 维度 | 第1层数据分块 | 第2层数据分块 |
|---|---|---|
| *M* | 512 | *RM* |
| *N* | 4096 | 256 |
| *K* | 128 | 128 |

## 4 实验结果及分析

本节介绍我们在申威 1621 处理器上对双精度浮点实数矩阵乘法进行的测试, 包括单线程 DGEMM 使用各项优化技术后的性能, 以及优化的多线程 DGEMM 程序性能.

### 4.1 实验设置

实验采用的申威 1621 处理器有 16 个计算核心, 主频 1.6 GHz, 内存 128 GB. 单核双精度浮点运算峰值 25.6 Gflops, 16 核双精度浮点运算峰值 409.6 Gflops. 三级缓存的大小分别是 32 KB, 512 KB, 512 MB.

对于单线程 DGEMM, 我们选择了 7 个版本进行了测试. 第 1 个版本 Gemm_Goto 使用稍加改动的 GotoBLAS 代码 (GotoBLAS 原始代码在申威处理器上不能直接运行). Gemm_baseline 采用文献 [9] 中所述方案及实现, 作为基础版本. Gemm_op1–Gemm_opt5 是 5 个优化版本, 每一版本都在上一个版本的基础上增加一种新的优化技术, 详细描述如后文表 4 所示. 其中, Gemm_opt1 在 Gemm_baseline 的基础上加入了二层分块, Gemm_opt2 引入了 C 矩阵缓冲区. Gemm_opt3 使用改进的打包函数, Gemm_opt4 使用了不同的 KERNEL, 而 Gemm_opt5 包含了前文提到的所有优化技术. 实验中, 我们选取了 4 种输入规模, 均为方阵, 即 *M*, *N*, *K* 这 3 个维度相等, 下文为叙述方便用一个维度数字代表所有维度的大小. 在实验中, 我们发现矩阵的主维大小 (即矩阵第 1 列长度) 会影响内存到缓存映射关系, 进而影响访存性能, 尤其是矩阵 *C* 的主维 LDC. 若矩阵 *C* 的 LDC 为 256 的倍数, 会造成较多的 cache miss, 性能会有所下降. 因此我们将 LDC 的取值设为 *M* 加上一定的偏移量 (16 的倍数).

### 4.2 实验结果

本节我们将展示在使用了上文所述的优化方法后, 单线程和多线程 GEMM 的性能提升.

#### 4.2.1 优化方法对单线程 GEMM 性能的影响

单线程 DGEMM 实验结果见图 5. 与基础版本相比, 使用二层分块带来 8.6% 的性能提升. 对 *C* 矩阵使用缓冲区使性能又提高了 10%. 对 *A* 和 *B* 打包函数的改进将性能提升了 13%. Gemm_opt4 和 Gemm_opt5 进一步增加了对 KERNEL 函数的改进, 分别得到了 3.1% 和 5.3% 的性能提升. 与 GotoBLAS 相比, 我们的最优版本实现了平均 7.39 倍的加速, 在规模较大 (*M*=16384) 时, 性能达到单核浮点计算峰值的 85%.

表 4 DGEMM 各版本说明

| 版本号 | 算法描述 |
|---|---|
| Gemm_Goto | Goto版本 |
| Gemm_baseline | 文献[9]中基于申威1600的优化版本 |
| Gemm_opt1 | 使用二层分块 |
| Gemm_opt2 | 对*C*矩阵使用缓冲区 |
| Gemm_opt3 | 改进的packA(), packB() |
| Gemm_opt4 | 对K方向上不同阶段的计算使用不同KERNEL函数 |
| Gemm_opt5 | 针对预取操作的指令级优化 |

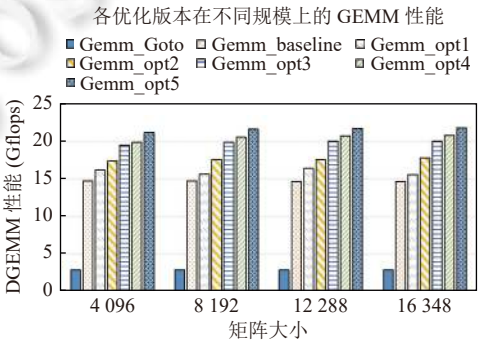


图 5 1621 处理器上单线程 DGEMM 性能测试结果

4.2.2 多线程实验结果

多线程 DGEMM 实验在两组数据规模上进行, 都采用 M, N, K 相同的方阵, 大小为 32 768, 分别在 1, 2, 4, 8, 16 个线程上进行测试. 结果见表 5. 从表中可以看出, 当线程数目小于等于 8 时, 扩展性效率超过 95%, CPU 效率超过 82%; 线程数目为 16 时, 可扩展性和 CPU 效率稍有下降. 我们分析, 16 线程时效率下降有两个可能的原因. 首先, 多线程 DGEMM 包含线程间同步的开销, 子函数计时结果 (表 6) 数据说明, 当线程数目达到 16 时, 同步开销已不能完全忽略. 其次, 使用 16 线程时, 线程间对访存带宽及三级缓存资源的使用会产生更多竞争, 导致 KERNEL 函数用时比同样规模在单线程上的运行时间略长.

表 5 多线程 DGEMM 扩展性测试结果 (输入规模 32 768)

| 核数 | 性能 (Gflops) | 加速比 | 扩展性效率 | CPU效率 (%) |
|---|---|---|---|---|
| 1 | 21.84 | 1 | 1 | 85.3 |
| 2 | 43.3 | 1.98 | 0.99 | 84.6 |
| 4 | 86.02 | 3.94 | 0.98 | 84.0 |
| 8 | 169.89 | 7.78 | 0.97 | 83.0 |
| 16 | 329.86 | 15.1 | 0.94 | 80.5 |

表 6 多线程 DGEMM 中计算每个分块用时统计 (s)

| 线程数 | 同步用时 | 计算函数用时 |
|---|---|---|
| 2 | 0.040 861 | 50.431 5 |
| 4 | 0.039 397 | 25.345 61 |
| 8 | 0.105 054 | 12.763 01 |
| 16 | 0.255 258 | 6.468 235 |

## 5 结 论

本文面向国产申威多核平台对稠密矩阵乘 DGEMM 单线程算法和多线程版的并行算法进行了重构与优化. 通过改变分块方案和打包方式等优化手段, 使程序运行最大化地减少额外开销, 同时又充分利用了数据局部性. 对 KERNEL 函数, 我们对函数在 K 方向迭代的不同阶段所读写的数据进行分析, 为每个阶段定制 KERNEL, 并对迭代中每个部分进行精准的数据预取. 使用我们的优化方法之后, 实验数据显示, 单线程版性能达到浮点峰值性能的 85%. 多线程版性能达到浮点峰值性能的 80%. 这些优化技术对于典型分层存储架构和现代微结构上的应用优化都有参考价值. 此外, 我们开发了代码生成器和自动调优脚本, 为优化参数搜索和高效调优提供了实用工具. 在未来工作中, 我们准备将对 GEMM 函数的优化应用到其他 BLAS 函数中, 并对小规模矩阵乘法优化进行研究. 另外, 我们将对各种优化方法进行提炼, 进一步完善代码生成器, 增加可移植性, 形成较完整的优化工具链.

**References:**


[1] Intel. Intel®-optimized math library for numerical computing. 2021. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html

[2] AMD. AMD optimizing CPU libraries. 2021. https://developer.amd.com/amd-aocl/

[3] NVIDIA. Basic linear algebra on NVIDIA GPUs. 2021. https://developer.nvidia.com/cublas

[4] AMD. RoCm documentation. 2021. https://rocmdocs.amd.com/en/latest/ROCm_Tools/rocblas.html

[5] Goto K, Van De Geijn R. On reducing TLB misses in matrix multiplication, FLAME working note 9. Technical Report, Austin: The University of Texas at Austin, Department of Computer Sciences, 2002.

[6] Goto K, van de Geijn R. High-performance implementation of the level-3 BLAS. ACM Trans. on Mathematical Software, 2008, 35(1): 4. [doi: 10.1145/1377603.1377607]

[7] Zhang XY, Kroeker M. OpenBLAS. 2022. http://www.openblas.net/

[8] Abdelfattah A, Keyes D, Ltaief H. KBLAS: An optimized library for dense matrix-vector multiplication on GPU accelerators. ACM Trans. on Mathematical Software, 2016, 42(3): Article 18. [doi: 10.1145/2818311]

[9] 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). [doi: 10.15888/j.cnki.csa.005456]

[10] Xie QC, Zhang YQ, Li Y, Pang RB, Wu ZL, Lu YQ, Gao PD. Package of the vector math library based on the Sunway processor. Ruan Jian Xue Bao/Journal of Software, 2014, 25: 70–79 (in Chinese with English abstract).

[11] Goto K, van de Geijn R. Anatomy of high-performance matrix multiplication. ACM Trans. on Mathematical Software, 2008, 34(3): 12. [doi: 10.1145/1356052.1356053]

[12] Lim R, Lee Y, Kim R, Choi J. An implementation of matrix-matrix multiplication on the Intel KNL processor with AVX-512. Cluster Computing, 2018, 21(4): 1785–1795. [doi: 10.1007/s10586-018-2810-y]

[13] Smith TM, van de Geijn R, Smelyanskiy M, Hammond JR, Van Zee FG. Anatomy of high-performance many-threaded matrix multiplication. In: Proc. of the 28th IEEE Int'l Parallel and Distributed Processing Symp. Phoenix: IEEE, 2014. 1049–1059. [doi: 10.1109/IPDPS.2014.110]

[14] Zhang M. Research on key issues of performance optimization in high performance computing based on the godson [Ph.D. Thesis]. Hefei: University of Science and Technology of China, 2017 (in Chinese with English abstract).

[15] Jiang LJ, Yang C, Ao YL, Yin WW, Ma WJ, Sun Q, Liu FF, Lin RF, Zhang P. Towards highly efficient DGEMM on the emerging SW26010 many-core processor. In: Proc. of the 46th Int'l Conf. on Parallel Processing. Bristol: IEEE, 2017. 422–431. [doi: 10.1109/ICPP.2017.51]

[16] Whaley RC, Petitet A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. Software: Practice and Experience, 2005, 35(2): 101–194.

[17] Chen TQ, Moreau T, Jiang ZH, Zheng LM, Yan E, Cowan M, Shen HC, Wang LY, Hu YW, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation (OSDI 2018). Carlsbad: USENIX Association, 2018. 579–594. [doi: 10.5555/3291168.3291211]

[18] Chen TQ, Zheng LM, Yan E, Jiang ZH, Moreau T, Ceze L, Guestrin C, Krishnamurthy A. Learning to optimize tensor programs. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems (NIPS 2018). Red Hook: Curran Associates Inc., 2018. 3393–3404. [doi: 10.5555/3327144.3327258]

[19] Chendu Sunway Technologies Co. Ltd. Introduction and specification of SW1621 processors. 2017 (in Chinese). http://www.swcpu.cn/show-190-254-1.html

[20] Gunnels JA, Henry GM, van de Geijn RA. A family of high-performance matrix multiplication algorithms. In: Proc. of the 2001 Int'l Conf. on Computational Sciences. San Francisco: Springer, 2001. 51–60. [doi: 10.1007/3-540-45545-0_15]

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

## 附中文参考文献:

[9] 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. [doi: 10.15888/j.cnki.csa.005456]

[10] 解庆春, 张云泉, 李焱, 逄仁波, 吴再龙, 鲁永泉, 高鹏东. 基于神威蓝光处理器的向量数学软件包. 软件学报, 2014, 25: 70–79.

[14] 张明. 龙芯平台上高性能计算的性能优化关键问题研究 [博士学位论文]. 合肥. 中国科学技术大学, 2017.

[19] 成都申威科技有限责任公司. 申威1621产品概述与产品规格. 2017. http://www.swcpu.cn/show-190-254-1.html

**闫昊**(1996－), 男, 硕士, 主要研究领域为高性能数值计算.

**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.

**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扩展数学库, 超级计算机评测软件.

**陈道琨**(1994－), 男, 博士, 主要研究领域为高性能计算, 异构并行, 稀疏矩阵相关的算法研究.