面向 ARMv8 64 位多核处理器的 QGEMM 设计与实现

姜 浩[1] 杜 琦[1] 郭 敏[1] 全 哲[2] 左 克[1] 王 锋[1] 杨灿群[1),3)]

[1)](国防科学技术大学计算机学院 长沙 410073)

[2)](湖南大学信息科学与工程学院 长沙 410082)

[3)](国防科学技术大学并行与分布处理重点实验室 长沙 410073)

摘 要 该文在 ARMv8 64 位多核处理器上基于 OpenBLAS 首次设计、实现并优化了四精度矩阵乘法(Quadruple precision General Matrix-Matrix Multiplication,QGEMM).由于浮点计算中不可避免地引入舍入误差,双精度矩阵乘法(DGEMM)在某些情况下不能给出令人满意的数值结果,因此需要高精度或多精度算法来实现更精确的计算.Double-double 算术是一种较为有效和广泛使用的手段.文中采用 double-double 数据格式构建结构体存储四精度浮点数据;基于 OpenBLAS 中的稠密矩阵计算的分块算法,增加四精度数据格式的相关的头文件和源文件,并用汇编代码撰写文中所提出的 QGEMM 的核心内核;利用无误差变换技术,调整并优化内核中的算法流程,避免规格化操作步骤造成的数据强制依赖关系;通过分析算法的数据依赖关系,设计寄存器的分配和轮转策略,优化指令调度顺序,开发指令级并行性,提高 QGEMM 的实际性能.根据具体算法使用混合乘加指令(FMA)的程度不同,文中采用了算法理论峰值性能这一概念,其有别于机器理论峰值的概念,能更好地评估文中所提出的 QGEMM 的实际效率.数值实验表明:文中通过汇编代码实现并优化的 QGEMM 性能最高达到 19.7 Gflops,效率为在 ARMv8 64 位多核处理器平台上 QGEMM 算法理论峰值性能的 82.1%,在满足数值结果精度要求的同时,其计算速度约是由 C 语言撰写的未优化的 QGEMM 和 MBLAS 中 QGEMM 的 5.8 倍,是编译器 GCC 实现的 long double 数据格式的 QGEMM 的 24 倍.同时数值实验还显示文中提出的 QGEMM 针对不同规模的矩阵具有较好的线程可扩展性.

关键词 ARMv8 64 位多核处理器;QGEMM;四精度;double-double 数据格式;long double 数据格式;OpenBLAS

中图法分类号 TP391 **DOI 号** 10.11897/SP.J.1016.2017.02018

Design and Implementation of QGEMM on ARMv8 64-bit Multi-Core Processor

JIANG Hao[1)] DU Qi[1)] GUO Min[1)] QUAN Zhe[2)] ZUO Ke[1)] WANG Feng[1)] YANG Can-Qun[1),3)]

[1)](*College of Computer, National University of Defense Technology, Changsha* 410073)

[2)](*College of Computer Science and Electronic Engineering, Hunan University, Changsha* 410082)

[3)](*Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha* 410073)

Abstract In this paper, we present the first design, implementation and optimization of quadruple precision matrix-matrix multiplication (QGEMM) based on OpenBLAS for ARMv8 64-bit multi-core processor. Sometimes, double precision matrix-matrix multiplication (DGEMM) can't give accurate results as expected owing to cancellation from round-off errors, therefore higher or multiple precision is required. The most efficient and widely used way is by using double-double arithmetic to achieve quadruple precision. The element of the designed QGEMM in this paper is stored as the structure, which consists of two floating-point numbers in double format corresponding to a double-double

收稿日期:2015-12-24;在线出版日期:2016-09-29.本课题得到国家"八六三"高技术研究发展计划项目基金(2012AA01A301)、国家自然科学基金项目(61402495,61303189,61602166,61170049,61402496)资助.**姜 浩**,男,1983 年生,博士,助理研究员,主要研究方向为高性能计算、舍入误差分析和数值计算.E-mail:haojiang@nudt.edu.cn.**杜 琦**,男,1986 年生,硕士,工程师,主要研究方向为并行计算和性能优化.**郭 敏**,女,1972 年生,硕士,副教授,主要研究方向为计算机软件.**全 哲**,男,1982 年生,博士,讲师,主要研究方向为人工智能、算法和编译器.**左 克**,男,1978 年生,博士,助理研究员,主要研究方向为网络和编译器.**王 锋**,男,1979 年生,博士,副研究员,主要研究方向为编译器、性能模型和计算机体系结构.**杨灿群**,男,1968 年生,博士,研究员,主要研究领域为高性能计算、异构计算和大尺度并行软件优化.

number. With GEMM blocking algorithm of OpenBLAS, we implement the QGEMM by adding some header files, source files and especially the inner kernel written in assembly. With error-free transformation, we optimize the algorithm flow in the inner kernel to avoid the renormalization step that sometimes is not necessary. By analyzing the data dependency, we design the register rotation and instruction scheduling to exploit instruction level parallelism. Considering that algorithms utilize fused multiply and add (FMA) instructions differently, we use the concept of algorithm's theoretical peak performance, which is different from that of machine's theoretical peak performance, to evaluate the efficiency of QGEMM better. Experimental results show that our QGEMM can perform up to 19.7 Gflops with the efficiency 82.1% of the algorithm's theoretical peak performance for ARMv8 64-bit multi-core processor. With the similar accuracy, our QGEMM runs 5.8 times faster than the un-optimized QGEMM based on OpenBLAS and the QGEMM in MBLAS, both of which utilize the double-double arithmetic to implement QGEMM and are written in C code. Our QGEMM also runs 24 times faster than the QGEMM implementation using GCC complier with long double format. The numerical tests show that our QGEMM has good scalability under varying thread counts across a range of matrix sizes evaluated.

Keywords　ARMv8 64-bit multi-core processor; QGEMM; quadruple precision; double-double format; long double format; OpenBLAS

1　引　言

浮点数计算具有舍入误差，在大规模的矩阵运算中，舍入误差的累积可能成为一个严重的问题. 在一些对舍入误差比较敏感的计算中，双精度有些时候并不能保证计算结果的精度，这时，四精度则是最直接最有效的解决途径[1]，所谓四精度浮点数简单来讲就是值具有双精度浮点数两倍精度的浮点数，即拥有 113 bit 有效精度位.

IEEE 754(2008)标准[2-3]提出了四精度浮点格式，但是包括 Intel、AMD 和 ARM 公司在内的大多数处理器生产设计厂商所研发生产的通用高性能处理器并没有对 IEEE-754(2008)标准的四精度浮点算术提供相应的硬件支持，仅通过编程语言和编译器在软件算法层面上来模拟实现四精度算术. 编程语言上对四精度支持较好的是 Fortran 语言，如 Intel Fortran 或 GNU Fortran 编译器定义四精度算术为 REAL×16 或 REAL(KIND=16)类型. 而 C/C++语言则通常将四精度定义为 long double 类型，但不同编译器对 long double 类型的定义各不一样，如 Microsoft Visual C++将 long double 等同于 double，GNU C 编译器 GCC 在 Intel 处理器平台将 long double 看成 80-bit 的扩展精度，IBM PowerPC 的 GCC 编译器和 Sun Studio 编译器则真正使用 long double 来表示四精度浮点数据.

目前，软件算法模拟高精度数值表示方式主要分为两大类：多数字模式(multiple-digit)和多部分模式(multiple-component). 多数字模式使用定制的类似于标准浮点数格式的一个数据类型来模拟实现高精度算术. 该模式的任意精度的浮点数运算主要应用任意精度的整数算法来实现，并不能有效利用当今高性能处理器所提供的浮点性能，所以运行速度较慢. 多部分模式使用若干个标准的浮点数据来表示一个高精度的数据，其数值为这几个浮点数据的和，每个组成部分拥有自己独立的指数和尾数. 这种高精度数据的有效位只能是工作精度有效位的整数倍，但是这类高精度数据之间的运算可以通过标准浮点操作模拟实现，能够充分利用高性能处理器所提供的浮点计算性能，计算速度相对于多数字模式有明显提高. 目前应用最为广泛的多部分模式高精度算法库是由美国劳伦斯伯克利国家实验室首席工程师 Bailey 等开发的 QD[4]①软件库，其支持双倍双精度(double-double)和四倍双精度(quad-double)数据格式，支持 C++和 Fortran90 语言，提供了基本的四则运算和三角函数、指数、对数等基本函数运算. 其中 double-double 格式近似拥有 106 bit 的有效精度位，略低于标准的四精度浮点数的 113 bit

① QD library. http://crd-legacy.lbl.gov/~dhbailey/mpdist, 2016,09,24

有效精度位. Double-double 格式数是一种应用较为广泛的近似模拟四精度浮点数的有效方式.

由于稠密线性代数在高性能计算领域发挥着重要作用,且很多计算问题的求解最终都可以转换为稠密性代数问题. 目前,在科学计算中使用比较广泛的开源 BLAS 库主要有基于手工汇编优化的 GotoBLAS[5-6]、OpenBLAS① 和基于自调优技术的 ATLAS[7] 等. 其中,OpenBLAS 由中国科学院计算技术研究所张云泉和中国科学院软件研究所张先轶等人[8] 在 GotoBLAS 2-1.13 BSD 基础上进一步开发和维护而产生的. 他们针对国产龙芯 3A CPU 处理器上提供了高性能的 BLAS 库实现,提出了代码自动生成方法 AUGEM[9],针对 X86 处理器所生成的代码性能优于 Intel 的 MKL 库和 AMD 的 ACML.

基础线性代数子程序集 BLAS 中稠密矩阵乘法 GEMM 的效率尤为重要,国内外学者对其展开了一系列有意义的优化工作. 中国科技大学朱海涛、李玲和中国科学院计算技术研究所陈云霁等人[10] 提出了一种用于通用处理器结构优化的矩阵乘法性能模型,用于预测矩阵乘时间和指导处理器体系结构优化. 中国科学院计算技术研究所崔慧敏、冯晓兵等人[11-13] 针对 DGEMM 研究了自动生成技术,取得了非常不错的效果. 中国科学院计算技术研究所的谭光明等人[14] 在 Fermi GPU 平台上,实现了优于 CUBLAS 的 DGEMM 性能. 此外,文献[15-16]研究了异构平台的 DGEMM 优化与实现.

随着计算机性能提升和科学应用对计算精度需求的增加,高精度和任意精度线性代数算法库得到了比较广泛的关注,一些科研工作人员针对稠密矩阵运算的高精度算法展开了研究. 日本理化研究所的 Maho 依托"京"超级计算机需求,基于 QD 和 GMP,MPFR 高精度软件库开发了高精度线性代数库 Mpack②,其针对 BLAS 实现了高精度的 MBLAS,但是其没有针对具体体系结构采用汇编代码优化,没能充分发挥计算平台的性能. 劳伦斯伯克利实验室的 Li 和 Demmel 等人[17] 引入 double-double 算法改善基本线性代数子程序集 BLAS,提出混合扩展精度的 XBLAS,但 XBLAS 不支持多线程并行. 日本筑波大学的 Takahashi 教授及其团队面向 GPU 设计了混合高精度的 BLAS 库[18].

上述文献中的多精度线性代数算法库中所实现的近似四精度稠密矩阵乘法(简称 QGEMM)主要是简单地用 double-double 格式数据的四则运算替换双精度浮点四则运算,针对具体平台没有特殊优化,效率往往不尽如人意,花费的运行时间较长. 且由于 double-double 格式数据要求两个 double 格式浮点数没有交叠,则输出数据的时候要求规格化的操作,这一步骤具有很强的数据依赖关系,往往会造成流水线的停顿.

近年来,在高性能计算领域计算性能功耗比、性能价格比一直备受关注. ARM 公司推出的 ARMv8 体系结构已经在高性能计算领域崭露头角. 如 NVIDIA 采用 ARM 的 IP 核构建了 Tegra 系统. AMD 基于 ARMv8 Cortex-A57 架构开发了代号西雅图的皓龙 A1100 系列芯片. 巴塞罗那超算中心构建了 Tibidabo 概念验证系统. 文献[19-20]面向 ARMv8 64 位多核处理器,研究了高效的 DGEMM 实现. 文献[21]在此基础上,采用补偿思想,利用无误差变换技术设计了高精度的 DGEMM 实现.

本文研究在 ARMv8 64 位多核处理器上设计、实现并优化四精度的一般矩阵乘法 QGEMM. 具体工作有:(1)基于多部分模式的高精度数据格式 double-double 构建结构体来近似并存储四精度浮点数据;(2)基于无误差变换技术,结合稠密矩阵乘法实现过程中的特殊分块计算步骤,调整优化计算步骤,避免规格化操作步骤造成的数据强制依赖关系;(3)采用 ARMv8 64 位访存指令、Cache 预取指令和 NEON 混合乘加向量计算指令等来手工撰写内核汇编代码;(4)优化向量寄存器的轮转使用和指令的调度.

此外,本文还展示了另外两种 QGEMM 的实现方式:(1)利用 GNU C 编译器 GCC 支持 long double 数据类型的特性,由编译器来实现四精度算术,该实现的输入和输出矩阵元素都为 long double 数据类型;(2)采用 C 代码分别撰写 double-double 数据格式下的四则运算,主要为加减法和乘法,并在此基础上实现 QGEMM. 考虑到不同算法含有混合乘加运算量的不同,为更好地评估算法的实际效率,本文提出一个新的算法峰值性能概念,即不同的算法在同一个计算平台具有不同的理论峰值性能. 数值实验表明,本文通过汇编代码优化实现的 QGEMM 性能达到 19.7 Gflops,效率为在 ARMv8 多核处理器平台上 QGEMM 算法理论计算峰值的 82.1%,其计算速度是未优化的 C 代码撰写内联函

① OpenBLAS. http://xianyi.github.com/OpenBLAS/, 2016, 05, 13

② The MPACK. http://mplapack.sourceforge.net/, 2015, 12, 29

数实现的 QGEMM 的 5.8 倍，是日本理化研究所的 MBLAS 中 QGEMM 的 5.8 倍，是 GCC 实现 long double 数据格式的 QGEMM 的 24 倍. 数值实验同时表明了本文提出的优化后的 QGEMM 具有较好的线程可扩展性.

本文第 2 节给出无误差变换和 double-double 数据格式的基本理论，以及计算平台和编译器的相关资料；第 3 节详细论述 QGEMM 的设计、实现和优化途径；第 4 节给出包括本文提出的 QGEMM 实现在内的 4 个 QGEMM 的性能和精度的比较分析；第 5 节对本文做了全面的总结.

2　相关背景简介

2.1 节简要介绍了 IEEE-754 浮点数标准，给出了标准四精度浮点数的定义，引出了 double-double 格式浮点数，并做了比较分析. 2.2 节介绍了无误差变换的定义，以及本文中涉及到的两个浮点数的加法和乘法的无误差变换算法，这些算法是构建 double-double 格式浮点数实现和四则运算的基础. 2.3 节介绍了 double-double 数据格式以及两个 double-double 格式数的加法、乘法和混合乘加算法. 2.4 节介绍了 ARMv8 处理器的体系结构特点和峰值计算性能，以及平台所安装的编译器等.

2.1　IEEE 浮点数标准

IEEE 二进制浮点数标准定义了如表 1 所示的几类浮点数格式，其存储格式为最高位为符号位 $s\in(0,1)$，次高位的 e bits 存储指数部分，最后剩下的 f bits 存储尾数部分. $L=1+e+f$ 为浮点数的位数长度. 规格化数(normal number)的第一位默认为 1，所以表 1 中各个精度的浮点数的精度位比尾数位多 1 位. 需要注意的是这里的规格化和本文针对 double-double 数据格式的规格化定义不同，这里的规格化数(normal number)对应于低规格化数(subnormal number)，是由浮点数格式中指数的值决定的，详见文献[2-3].

表 1　IEEE-754 二进制浮点数格式

位数	半精度 Binary16	单精度 Binary32	双精度 Binary64	四精度 Binary128
长度 L	16	32	64	128
符号 s	1	1	1	1
指数 e	5	8	11	15
尾数 f	10	23	52	112
精度 p	11	24	53	113

由表 1 可见，标准的四精度浮点数有 113 位有效精度位，但是绝大多数当前的通用高性能处理器并没有提供相应的硬件支持，仅通过算法来模拟实现，如 GMP，MPFR 等，由于是利用任意精度表示，采用了类似符号表示的整数计算来实现，效率普遍较低. 另外一种应用较为广泛的就是 double-double 数据格式，由表 1 可知其约有 106 位有效精度位，略低于标准四精度，但是由于可以充分利用处理器的浮点计算部件，double-double 格式数算是一种较为理想的四精度近似表示.

本文中，一般的加减乘除算术操作通常用{+，−，×，÷}来表示，用{⊕，⊖，⊗，⊘}来表示具有舍入误差操作的加减乘除运算. 我们用 $fl(a\circ b)$ 来表示 $a\circ b$ 的最佳浮点数近似，其中 $\circ\in\{+,-,\times,\div\}$ 用 $err(a\circ b)=a\circ b-fl(a\circ b)$ 表示舍入误差.

2.2　无误差变换

double-double 格式浮点数本质上是利用无误差变换技术设计并实现的，而无误差变换的思想由 Dekker、Kahan 和 Knuth 等学者[22-23]在 20 世纪六七十年代提出并进行了相关研究. 2005 年，Ogita，Rump 和 Oishi[24]正式提出这一概念. 定义 1 给出了无误差变换的定义.

定义 1[25]. 设 $\circ\in\{+,-\times\}$，a 和 b 是两个浮点数 $a,b\in F$，且有 $x=fl(a\circ b)\in F$，在没有上下溢出，舍入模式为就近舍入时，存在

$$a\circ b=x+y,$$

其中 x 表示计算结果最佳浮点数近似，$y=err(a\circ b)\in F$ 表示舍入误差. 将浮点数对 (a,b) 转化为浮点数对 (x,y)，且满足上式的过程即为一对浮点数加、减和乘法运算的无误差变换.

算法 1[23]. 浮点数 a 和 b 的加法的无误差变换，若满足条件 $|a|\geq|b|$，则

$$function[x,y]=QuickTwoSum(a,b)$$
$$x=a\oplus b$$
$$y=b\ominus(x\ominus a)$$

需要注意的是如果 $|a|<|b|$，那么总是得到 $y=0$. 而算法 2 可以计算任意情况下浮点数 a 和 b 加法的舍入误差项.

算法 2[22]. 浮点数 a 和 b 的加法的无误差变换

$$function\ [x,y]=TwoSum(a,b)$$
$$x=a\oplus b$$
$$z=x\ominus a$$
$$y=(a\ominus(x\ominus z))\oplus(b\ominus z)$$

当前多数计算平台支持混合乘加运算[26] FMA，可以用一条指令完成计算

$$FMA(a,b,c)=a\times b+c.$$

浮点数乘法的无误差变换通常为 TwoProd 算法[23]，其计算误差项花费巨大，需要将浮点数进行分割操作，由于篇幅所限，本文不详细列出. 同 TwoSum 算法相比，算法 3 可以利用 FMA 操作，用一条指令在一拍完成两个浮点数乘法的舍入误差计算，极大地节省了运算时间.

算法 3[3]. 使用 FMA 运算操作的浮点数 a 和 b 的乘法的无误差变换

$$function[x,y]=FMATwoProd(a,b)$$
$$x=a\otimes b$$
$$y=FMA(a,b,-x)$$

2.3 Double-double 格式数值算术

当数值计算需要近似四精度时，double-double 数据格式是最有效最常用的选择. 目前应用最为广泛的是 QD Library 中的 double-double 数据格式[4]. Double-double 格式数的加减法和乘法运算实现以无误差变换为基础. 定义 2 给出了 double-double 数的定义.

定义 2[25]. x_h 和 x_l 是两个浮点数，x 表示 x_h 和 x_l 的没有舍入操作的精确和，即 $x=x_h+x_l$，其中 x_h 和 x_l 没有交叠，满足

$$|x_l|\leq \mathrm{ulp}(x_h)/2.$$

如果限定 x_h 和 x_l 是 IEEE-754 标准中的双精度格式 double，则称浮点数对 (x_h,x_l) 为 double-double 格式数，即 x 为 double-double 数.

其中，ulp 函数的定义详见文献[3]. 本文中将这一强制的数据格式要求定义为规格化的步骤(有别于浮点规格化数)，其主要是利用 QuickTwoSum 算法来实现.

算法 4、算法 5 分别实现了 double-double 数据格式的加法和乘法，算法 5 采用了 FMA 运算操作.

算法 4[4]. double-double 格式数 $a=(a_h,a_l)$ 和 $b=(b_h,b_l)$ 的加法运算

$$function[r_h,r_l]=add\_dd\_dd(a_h,a_l,b_h,b_l)$$
$$[s_h,s_l]=TwoSum(a_h,b_h)$$
$$[t_h,t_l]=TwoSum(a_l,b_l)$$
$$s_l=s_l\oplus t_h$$
$$[t_h,s_l]=QuickTwoSum(s_h,s_l)$$
$$t_l=t_l\oplus s_l$$
$$[r_h,r_l]=QuickTwoSum(t_h,t_l)$$

算法 5[4]. 使用FMA运算操作的double-double格式数 $a=(a_h,a_l)$ 和 $b=(b_h,b_l)$ 的乘法运算

$$function[r_h,r_l]=FMAprod\_dd\_dd(a_h,a_l,b_h,b_l)$$
$$[t_h,t_l]=FMATwoProd(a_h,b_h)$$
$$t_l=a_h\otimes b_l\oplus a_l\otimes b_h\oplus t_l$$
$$[r_h,r_l]=QuickTwoSum(t_h,t_l)$$

下一步，考虑到稠密矩阵运算中包含大量的混合乘加运算，因此在算法 6 中给出了双倍工作精度的混合乘加运算 $(a_h,a_l)\times(b_h,b_l)+(c_h,c_l)$ 的实现方式.

算法 6. 双倍工作精度的混合乘加运算

$$function[r_h,r_l]=fma\_dd\_dd(a_h,a_l,b_h,b_l,c_h,c_l)$$
$$[t_h,t_l]=FMAprod\_dd\_dd(a_h,a_l,b_h,b_l)$$
$$[r_h,r_l]=add\_dd\_dd(t_h,t_l,c_h,c_l)$$

算法 6 直接调用了算法 4 和算法 5 来实现. 表 2 给出了上述 6 个基本算法的运算量和所需指令数.

表 2 本文涉及算法所需运算量

算法	加减法	乘法	FMA	指令总数	运算量
算法 1	3	0	0	3	3
算法 2	6	0	0	6	6
算法 3	0	1	1	2	3
算法 4	20	0	0	20	20
算法 5	5	3	1	9	10
算法 6	25	3	1	29	30

2.4 开发平台和编译器

开发平台为ARMv8 64位处理器，频率为 2.4GHz，具有8个CPU核，内存为16 GB(DDR3). ARMv8 64位处理器支持 FMA 运算操作，每一拍可以完成 1 个浮点加法和 1 个浮点乘法，则其双精度运算峰值为 2.4[GHz]×8[CPUCore]×(2[Flop]/1[Cycle])=38.4[GFlops]

每个 CPU 核有 32 个 128 位浮点向量寄存器，独享大小为 32 KB 的 L1 级数据 Cache 和大小 32 KB 的 L1 级指令 Cache；每两个 CPU 核共享大小为 256 KB 的 L2 级 Cache，组成一个模块，总共 4 个模块，模块之间通过高速互联连接；所有模块通过高速互联共享大小为 8 MB 的 L3 级 Cache；高速互联通过两个 DDR 内存控制器实现对内存的访问. L1 Cache、L2 Cache 和 L3 Cache 之间是 Inclusive 关系.

编译器为 GNU C 编译器 GCC，版本号为 4.8.2. 该版本提供-march=armv8-a 编译选项用于 ARMv8 64 位处理器平台的软件编译；支持 C/C++语言为表示四精度浮点数而定义的 long double 数据类型.

3 QGEMM 的设计、实现和优化

3.1 QGEMM 的分块算法实现

在高性能计算领域，稠密线性代数发挥着重要

作用，很多计算问题的求解最终转化为稠密线性代数问题，而稠密线性代数问题的核心即为矩阵乘法. 一般矩阵乘法 GEMM 计算 $\boldsymbol{C}=\alpha\boldsymbol{AB}+\beta\boldsymbol{C}$，其中，$\boldsymbol{C}$，$\boldsymbol{A}$，$\boldsymbol{B}$ 分别为 $M\times N$，$M\times K$，$K\times N$ 的矩阵.

OpenBLAS 的 GEMM 实现方法大致如图 1 所示. 为充分利用计算机系统的多层存储结构，提高处理器中 Cache 的命中率，减少流水线停顿，矩阵乘 GEMM 通过循环分块的方法充分开发数据的空间局部性和时间局部性. 最内层循环完成一个列向量乘行向量的内积操作，包括加载数据到寄存器、对寄存器中的数据进行乘加操作、把结果从寄存器存储回内存，这部分被称为内核. 内核是整个矩阵乘 GEMM 计算最重要的部分，直接影响整个矩阵乘的计算性能.

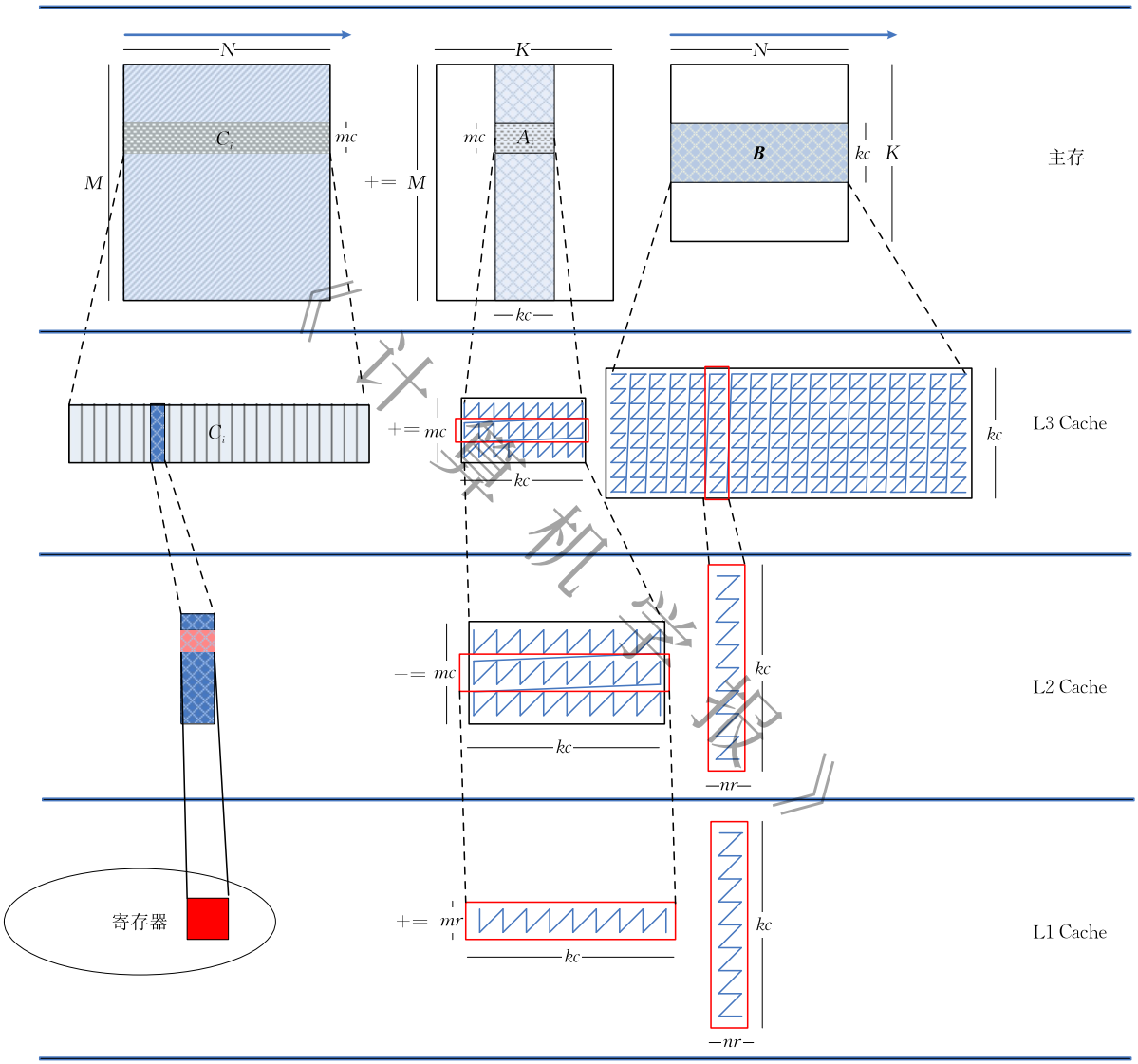

图 1　OpenBLAS 库的 GEMM 算法流程框架和存储位置

OpenBLAS 库目前还不支持四精度的相关运算，本文第一次基于 OpenBLAS 开源软件设计了 double-double 数据格式的结构体，并利用图 1 所示的矩阵分块算法实现了四精度稠密矩阵乘法 QGEMM. 结构体中由两个 double 格式的浮点数构成，满足定义 2 的要求. 输入和输出矩阵的元素数据类型都为 double-double 结构体，元素存储大小是双精度的两倍.

3.2　QGEMM 的内核设计

设计 QGEMM 的内核，首先要确定内核的大小，其主要由 ARMv8 多核处理器的物理寄存器数量和 SIMD 向量寄存器的宽度来决定. 浮点向量寄存器主要用来存储图 1 中最下面层的 $\boldsymbol{A}$ 矩阵（维度为 $mr\times 1$），$\boldsymbol{B}$ 矩阵（维度为 $1\times nr$）和 $\boldsymbol{C}$ 矩阵（维度为 $mr\times nr$），以及预取下一个循环的 $\boldsymbol{A}$ 和 $\boldsymbol{B}$. 根据 ARMv8 多核处理器的每个 CPU 核有 32 个 128 位

浮点向量寄存器，并结合四精度 double-double 的结构体存储要求，备选的内核大小只能是 2×4，4×2 和 2×2 这 3 种内核大小. 因为再大一点的 4×4 内核就需要所有的 32 个向量寄存器来存储 ***A***，***B*** 和 ***C*** 矩阵了，已经没有多余的向量寄存器来协助完成内部的四精度混合乘加运算的实现. 考虑到 2×2 内核没有充分利用向量寄存器的数量，而过小的内核可能导致计算访存比过低，不利于发挥性能，而 2×4 和 4×2 区别不是非常明显，故本文设计了 4×2 规模的 QGEMM 内核. 其利用从 ***v***24 到 ***v***31 共计 8 个向量寄存器来存储 ***C*** 矩阵，利用从 ***v***0 到 ***v***3 共 4 个向量寄存器存储 ***A*** 矩阵，利用 ***v***4 和 ***v***5 来存储 ***B*** 矩阵，并分别利用 ***v***6 到 ***v***10 共 6 个向量寄存器预取下一个循环所需要的 ***A*** 和 ***B*** 矩阵，循环展开为 2 次，具体详见图 2.

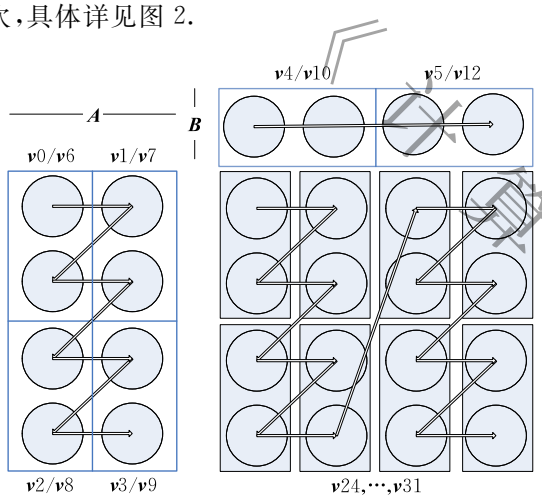

图 2 QGEMM 的 4×2 内核实现

整个内核由 4 个计算部分构成，如第一计算部分即为向量 ***v***0，***v***1 和向量 ***v***4 之间的计算得到向量 ***v***24，***v***25. 黑色实线表示数据的存储顺序，为更好地利用向量指令，本文没有简单地将一个 double-double 结构的两个浮点数存储到一个 128 bit 的向量寄存器中，而是针对 ***A***，***B*** 和 ***C*** 矩阵采用不同的存储方式，如图 3 所示，存储 ***A*** 矩阵的时候利用 ***v***0 向量寄存器来存储 double-double 格式数据的上半部分双精度数据，用 ***v***1 来存储下半部分的双精度数据. 而存储 ***B*** 矩阵则是用 ***v***4 存储一个 double-double 数据. 黑色表示 double-double 数据的高位部分，而灰色表示低位部分，这样的存储方式方便调用向量汇编指令运算.

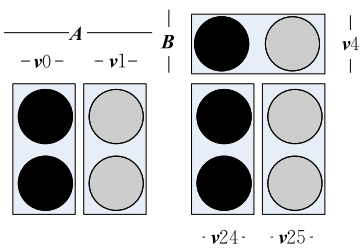

图 3 第一计算部分中寄存器的使用实例

结合上述论述则需要用来存储的向量寄存器共 20 个，剩余 12 个向量寄存器来完成内部的四精度混合乘加运算. 由于循环展开次数为 2，表 3 仅展示了第一次循环展开的寄存器的分配情况，其中 #1～#4 分别表示图 2 中 4×2 内核中的 4 个主要计算部分，每个计算部分完成 2 个 double-double 数和 1 个 double-double 数的混合乘加运算. 在每个计算部分中分配了 3 个向量寄存器协助完成算法的实现.

表 3 寄存器使用情况列表

计算部分	#1	#2	#3	#4
A	***v***0，***v***1	***v***0，***v***1	***v***2，***v***3	***v***2，***v***3
B	***v***4	***v***5	***v***4	***v***5
C	***v***24，***v***25	***v***26，***v***27	***v***28，***v***29	***v***30，***v***31
其他	***v***12～***v***14	***v***15～***v***17	***v***18～***v***20	***v***21～***v***23

3.3 QGEMM 的内核汇编实现

2.3 节的算法 6 实现了四精度 double-double 数据结构的混合乘加运算，但是其调用的算法 4 和算法 5 中含有规格化步骤，即 QuickTwoSum 算法. 这一步运算主要保证了算法输出的数据严格满足 double-double 的数据结构特点，即定义 2 中提到的 $|x_l| \leq \mathrm{ulp}(x_h)/2$. 由于这一规格化步骤引起的强制数据依赖关系，频繁的规格化步骤将严重降低算法的效率，而其带来的精度收益却不是非常明显，绝大多情况下最多仅仅提升一位有效精度位. 因此本文对算法 6 做了可容忍的简化处理，详见算法 7. 具体来讲主要包含：(1) 针对算法 6 所调用的算法 4 中 double-double 格式加法的低位计算部分，不采用无误差变换技术来计算，这样虽然可能损失一些精度，但不是很明显；(2) 将算法 6 所调用的算法 4 和算法 5 的规格化部分统一.

算法 7. 优化的双倍工作精度的混合乘加运算

$$function[r_h, r_l] = fma\_dd\_dd(a_h, a_l, b_h, b_l, c_h, c_l)$$
$$[t_h, t_l] = FMATwoProd(a_h, b_h)$$
$$[s_h, s_l] = TwoSum(t_h, c_h)$$
$$s_l = a_h \otimes b_l \oplus a_l \otimes b_h \oplus t_l \oplus s_l \oplus c_l$$
$$[r_h, r_l] = QuickTwoSum(s_h, s_l)$$

此外，根据图 1 所示，在最底层的计算中，需要经过 kc 次步骤的累加计算，并且每次累加后的数据存储到 ***C*** 矩阵相同的 8 个向量寄存器中，如果按照

算法 7 的计算步骤，那么每次累加计算后都要规格化(QuickTwoSum 算法)一次，这样效率依然较低，本文采取将规格化步骤放到 kc 次累加之后，即 kc 次累加运算后进行一步规格化，简单来讲就是算法 7 中的最后一步 QuickTwoSum 算法不执行，而在 kc 次调用算法 7 之后再执行 QuickTwoSum 算法，这样不仅仅减少了运算量，还能使代码的实际执行效率提高.

根据表 3 所示，在每个计算部分分配了 3 个向量寄存器来协助完成算法 7，但是还不足以完成算法的实现，本文根据算法 7 的特点，观察数据的相互依赖关系，详见图 4，其中 fmla 表示混合乘加，fmls 表示混合乘减($FMLS(a,b,c)=c-a\times b$)，fmul 表示乘法. 根据图 4，我们发现存储 ***A*** 矩阵(或 ***B*** 矩阵)的向量寄存器在完成相关的运算后完全可以释放出来，因为后续的绝大部分都是用来计算 ***C*** 矩阵的 double-double 数据的低精度部分，主要是误差累积的计算，其不需要 ***A*** 矩阵(和 ***B*** 矩阵)数据参与. 故可以调整指令顺序，比如将同矩阵 ***A*** 的数据相关计算指令提前，并在指令执行之后释放存储 ***A*** 的向量寄存器如 ***v***0，并用这个向量寄存器来协助后续算法实现.

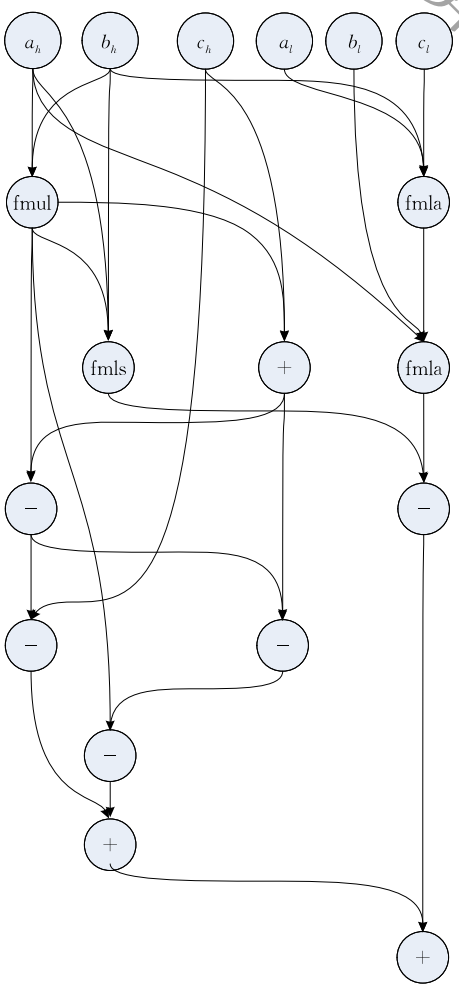

图 4　内核算法的数据依赖关系图

图 5 展示了优化后的汇编代码实例，其中■行表示的代码实现了算法 7 的第 1 行 FMATwoProd 算法；◆行表示的代码部分实现了第 2 行 TwoSum 算法；而●行表示的代码部分实现了第 3 行运算. 图 5 中没有展示访存指令，我们针对 ***A*** 矩阵采用 ld2 指令进行跳步访存，如图 2 所示，将两个 double-double 格式浮点数据的上半部分的 double 数放入一个 128 bit 向量寄存器，将下半部分的两个 double 数放入另一个向量寄存器. 而存储 ***B*** 矩阵则直接使用 ld1 指令将两个连续的 double-double 格式浮点数据存入两个 128 bit 的向量寄存器，每个 double-double 数存入一个向量寄存器. 统计得到每个计算部分的计算量和计算指令数分别为 15 flops 和 12.

■	fmul	v12.2d	v0.2d	v4.d[0]
■	orr	v13.16b	v12.16b	v12.16b
■	fmls	v13.2d	v0.2d	v4.d[0]
●	fmla	v25.2d	v1.2d	v4.d[0]
●	fmla	v25.2d	v0.2d	v4.d[1]
●	sub	v25.2d	v25.2d	v13.2d
◆	fadd	v0.2d	v12.2d	v24.2d
◆	fsub	v14.2d	v0.2d	v12.2d
◆	fsub	v13.2d	v0.2d	v14.2d
◆	fsub	v14.2d	v24.2d	v14.2d
◆	orr	v24.16b	v0.16b	v0.16b
◆	fsub	v13.2d	v12.2d	v13.2d
◆	fadd	v13.2d	v13.2d	v14.2d
●	fadd	v25.2d	v25.2d	v13.2d

图 5　QGEMM 的内核汇编实例

前文提到在每个计算部分分配独立的 3 个向量寄存器，同时使用 1 个重用的向量寄存器来协助完成具体算法实现. 那么由于一个循环共有 4 个计算部分的指令是没有相互依赖关系的，且使用了不同的中间寄存器，仅仅输入 ***A***，***B*** 和 ***C*** 矩阵数据相同，因此可以将指令进行混排，即 4 个计算部分同时执行，这样可以有效地利用指令的发射槽，提高算法的执行效率. 这里每个计算部分重用的向量寄存器不同，如在第一个循环中，4 个计算部分分别重用向量寄存器 ***v***0～***v***3 共 4 个向量寄存器. 该优化策略能有效地开发指令级并行性.

3.4　QGEMM 的分块参数确定

由上节的内核实现可知，QGEMM 的核心计算量较大，计算访存比远大于 DGEMM. 因此 QGEMM 算法实现过程中更容易隐藏访存开销. 根据图 1 所示的矩阵各个分块存储的 Cache 位置，结合各个

Cache 的大小,同时参考文献[22,23],我们给出了 QGEMM 的分块参数:

$$mr\times nr\times kc\times mc\times nc=4\times 2\times 768\times 8\times 578,$$

这些参数通过下述公式得出:

$$\begin{cases} kc\times nr\times 8\times 2\leq\dfrac{4-k1}{4}\times 32\times 1024 \\ (mr\times nr+mr\times 2)\times 8\times 2\leq\dfrac{k1}{4}\times 32\times 1024 \end{cases}\Rightarrow kc=768,$$

$$\begin{cases} 2\times mc\times kc\times 8\times 2\leq\dfrac{16-k2}{16}\times 256\times 1024 \\ 2\times kc\times nr\times 8\times 2\leq\dfrac{k2}{16}\times 256\times 1024 \end{cases}\Rightarrow mc=8,$$

$$\begin{cases} kc\times nc\times 8\times 2\leq\dfrac{16-k3}{16}\times 8\times 1024\times 1024 \\ 8\times mc\times kc\times 8\times 2\leq\dfrac{k3}{16}\times 8\times 1024\times 1024 \end{cases}\Rightarrow nc=578.$$

4 数值实验

本文基于 OpenBLAS 给出了面向通用多核处理器的高效 QGEMM 实现.表 4 给出了 ARMv8 64 位多核处理器测试平台详细配置.为了更好地评估所设计优化的 QGEMM 的正确性,本文实现了以 long double 数据类型结合编译器方式的四精度稠密矩阵乘法,并以此作为精度测试基准.这里需要指出 long double 数据类型的 QGEMM 针对 2×2 内核,2×4 内核,4×2 内核和 4×4 内核的实现运行时间相差不大.因此在测试过程中,选择 2×2 内核 long double 数据格式 QGEMM 实现.同时,为了评价本文所设计优化的 QGEMM 的优异性和可信性,本文用 C 代码编写了 double-double 格式数据的加减法和乘法运算,为了避免函数的调用开销,将其设置为内联函数,通过对不同内核版本的 QGEMM 的性能测试,最终选择了 4×4 内核来实现.本文还选取了日本理化研究所的开源软件 MBLAS 作为性能比较对象.综上,本文共有 4 个版本的 QGEMM 实现,详见表 5.

表 4 ARMv8 64 位多核处理器测试平台配置表

处理器	ARMv8 64 位处理器(2.4GHz,8 核)
内存	16GB(DDR3)
操作系统	Linux(Aarch64) 内核版本 3.8.0
编译器	GCC 4.8.2 支持-march=armv8-a
OpenBLAS	OpenBLAS_develop-r0.2.14

为了评价不同计算规模下 QGEMM 的计算结果精度与性能,我们选取 $M=N=K$ 从 128 到 2048,步长为 128 的 16 种维度矩阵进行测试.矩阵元素为[−1,1]区间的随机数,这个随机数是经过规格化处理的 double-double 格式数据.需要强调的是如果矩阵的元素为[0,1]区间的随机数,那么由于符号相同,都是正浮点数,数值计算结果精度要高.

表 5 4 个版本的 QGEMM 实现

QGEMM-v1	汇编代码内核实现的 double-double 数据格式的 QGEMM
QGEMM-v2	C 代码内核实现的 double-double 数据格式的 QGEMM
QGEMM-v3	MBLAS 实现的 double-double 数据格式的 QGEMM
QGEMM-v4	编译器是现代的 long double 格式 QGEMM

4.1 算法计算量和算法峰值性能分析

根据 2.4 节,可知计算平台 ARMv8 八核处理器的理论峰值为 38.4 Gflops,这是在满负荷运行混合乘加指令下才能达到的理论峰值.那么没有混合乘加运算的算法能达到理论性能峰值也仅仅是机器峰值性能的一半.

参考并修正文献[21]中式(19),根据表 2,如果采用算法 6 设计 QGEMM,那么每 29 cycles 完成 30 flops 的浮点运算,故其 QGEMM(QGEMM-v2 和 QGEMM-v3)可以达到的理论峰值估算为

$$2.4[\text{GHz}]\times 8[\text{CPUCore}]\times(30[\text{Flop}]/29[\text{Cycle}])\approx 19.862[\text{GFlops}]$$

而根据 3.3 节的分析和图 5 展示的汇编代码,其核心计算用 12 cycles 完成 15 flops 的浮点运算.故本文设计优化的 QGEMM(QGEMM-v1)可以达到的理论峰值估算为

$$2.4[\text{GHz}]\times 8[\text{CPUCore}]\times(15[\text{Flop}]/12[\text{Cycle}])\approx 24[\text{GFlops}]$$

相比于没有优化的 QGEMM 实现,FMA 指令占所有计算指令的比例增加导致优化后的 QGEMM 实现能达到的理论峰值性能上限提升了.

本文提出的 QGEMM-v1 实现的浮点计算量为

$$\sum_{jc=1}^{N/nc}\sum_{kl=1}^{K/kc}\sum_{ic=1}^{M/mc}\Big(\sum_{jr=1}^{nc/nr}\sum_{ir=1}^{mc/mr}\Big(\sum_{kl=1}^{kc}15\times mr\times nr+18\times mr\times nr\Big)+2\times mc\times nc\Big)\approx 15\times M\times N\times K.$$

类似的 QGEMM-v2 和 QGEMM-v3 的浮点计算量为 $30\times M\times N\times K$.优化后 QGEMM-v1 的运算量是优化前 QGEMM-v2 和 QGEMM-v3 的一半.

4.2 精度比较分析

本文在验证 QGEMM 的计算结果矩阵元素的精度时,以 long double 数据类型的 QGEMM 测试

结果为基准值. 在 ARMv8 64 位多核处理器平台，GCC 支持 long double 数据类型表示四精度浮点数. 设 long double 数据类型 QGEMM 计算结果矩阵元素为 $C1_i$，double-double 数据类型 QGEMM 计算结果矩阵元素为 $C2_i$，其中 $0 \leq i \leq M \times N$，将 $C2_i$ 中元素数据强制转化为 long double 类型，并计算相对误差. 如表 6 所示，结果矩阵元素 $C2_i$ 与 $C1_i$ 的最大相对误差随矩阵规模增加而有所增大，但最大相对误差小于 1.0E-25 数量级，即数值结果至少有 25 位是可靠的. 而相比而言，相同规模矩阵运行 DGEMM 一般平均相对误差维持在 1.0E-15 数量级，最大相对误差约为 1.0E-10 数量级.

表 6　QGEMM 计算结果的相对误差

(M=N=K)	QGEMM-v1		QGEMM-v2	
	最大相对误差	平均相对误差	最大相对误差	平均相对误差
256	9.66E-27	8.14E-31	1.66E-28	1.08E-31
512	1.17E-25	1.87E-30	1.96E-26	2.86E-31
768	7.07E-26	1.33E-30	3.49E-26	3.24E-31
1024	9.88E-25	2.59E-30	1.30E-25	4.74E-31
1280	5.08E-26	1.33E-30	1.58E-26	3.58E-31
1536	4.12E-25	1.75E-30	3.73E-26	4.56E-31
1792	2.67E-25	1.69E-30	1.15E-25	5.58E-31
2048	6.77E-25	1.79E-30	5.95E-25	8.93E-31

由于篇幅所限，表 6 仅列出了16个维度矩阵测试结果其中的 8 个. 通过对比 QGEMM-v1 和 QGEMM-v2，可以得出本文在优化过程中采用将规格化步骤部分省略和调整位置的策略是可行的，数值结果精度没有明显的缺失. 但可以节省很多的运算量，最重要的是避免了强制数据依赖关系，进而带来了更好的指令级并行性.

4.3　性能比较分析

上一节的精度实验证实了本文设计、实现并优化的 QGEMM 的数值结果精度满足要求，这一节将展示其性能的优异性. 首先我们得出在 8 线程情况下，计算 $M=N=K=2048$ 规模矩阵时，QGEMM-v1 较 QGEMM-v2 和 QGEMM-v3 加速约 5.8 倍，较 QGEMM-v4 加速约 24 倍，具体数据详见图 6. 其中 QGEMM-v2 是由 C 代码编写的 double-double 乘加算法的内联函数实现，虽然采用了分块算法实现，但是其性能依然和同样由 C 代码实现的基于 MBLAS 的 QGEMM-v3 相同. QGEMM-v4 由编译器 GNU C 编译器 GCC 支持的 long double 数据类型表示四精度浮点数. GCC 通过多数字模式来模拟四精度，实现过程由编译器完成. 具体来讲，GCC 的子目录下有/libgcc/soft-fp. 其中 quad.h 给出了 long double 数据类型四精度数的头文件，addtf3.c 和 multf3.c 执行了四精度的加法和乘法运算. 该目录同样提供了模拟单精度 single 和双精度 double 运算的程序. 由于利用整数计算来模拟浮点运算，编译器 GCC 实现的 QGEMM-v4 效率非常低.

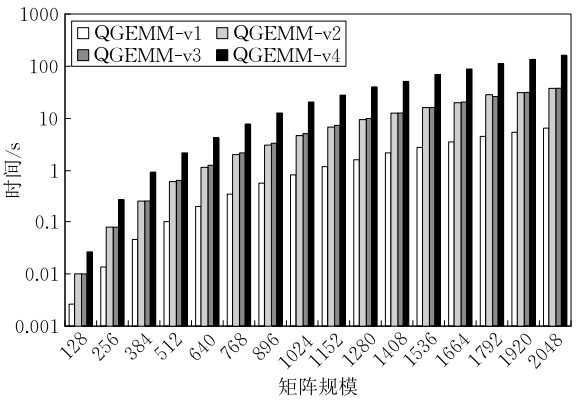

图 6　4 种 QGEMM 实现在 8 线程的执行时间

根据 4.1 节关于计算量的分析，以及计算平台峰值性能和算法峰值性能的讨论比较，图 7 展示了 QGEMM-v1，QGEMM-v2 和 QGEMM-v3 的计算性能. 其中，QGEMM-v1 的计算性能最高达到 19.7 Gflops，对应计算平台 38.4 Gflops 的峰值，效率约为 51.3%，对应算法的 24 Gflops 的理论峰值，效率为 82.1%. 而 QGEMM-v2 和 QGEMM-v3 的计算性能最高则仅为 6.8 Gflops，对应计算平台 38.4 Gflops 的峰值，效率约为 17.7%，对应算法的 19.862 Gflops 的理论峰值，效率为 34.2%. 我们发现 QGEMM-v1 的性能是 QGEMM-v2 和 QGEMM-v3 的 2.9 倍，而计算速度却是 5.8 倍，这是因为其计算量是 QGEMM-v2 和 QGEMM-v3 的一半. 由图 8 可见 QGEMM-v1 有良好的线程扩展性，在单线程情况下，其达到算法理论峰值性能的 83.1%，多线程情况效率降低很小，8 线程也能达到算法理论峰值性能的 82.1%，这主要是由 QGEMM 的算法特性决定的，即计算量远远大于访存量，访存不再是瓶颈. 同样算法内计算舍入

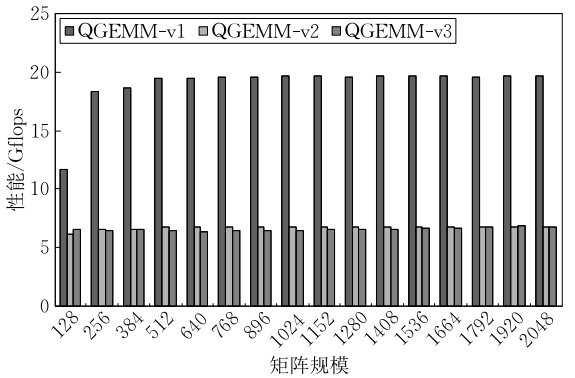

图 7　3 种 QGEMM 实现在 8 线程的性能

误差的过程不可避免地引入了复杂的数据依赖关系,可能这部分造成了计算流水的停顿,导致单线程的算法效率没能更高.

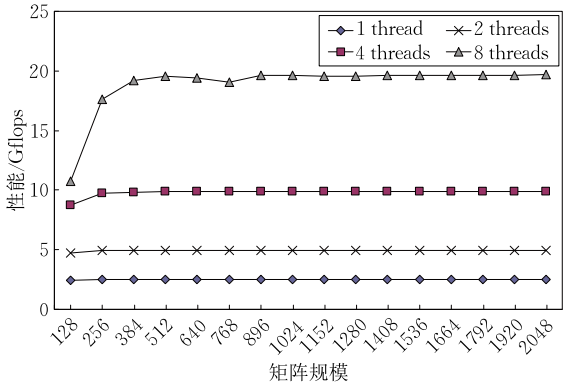

图 8 QGEMM-v1 的多线程扩展性

5 结 论

本文首次在 ARMv8 64 位多核处理器上基于 OpenBLAS 实现了四精度稠密矩阵乘法 QGEMM. 由于计算平台硬件上不支持四精度的运算,因此本文采用软件模拟的方式实现,为充分利用计算平台的浮点计算性能,本文选择 double-double 数据格式模拟四精度. 本文利用无误差变换技术,结合 OpenBLAS 中稠密矩阵乘法实现过程中的特殊分块计算步骤,采用计算指令顺序优化调度和寄存器轮转策略,回避不必要的规格化操作,增加指令级并行度,提升算法效率. 实验数值表明,本文设计实现并优化的 QGEMM 比优化前的版本快 5.8 倍,比 MBLAS 中的 QGEMM 快 5.8 倍,比编译器实现 long double 数据格式的 QGEMM 快 24 倍. 基于本文提出的计算平台上具体算法的理论计算峰值性能为标准,本文优化的 QGEMM 效率达到其理论算法峰值性能的 82.1%.

参 考 文 献

[1] Hasegawa H. Utilizing the quadruple-precision floating point arithmetic operation for the Krylov subspace methods//Proceedings of the 8th SIAM Conference on Applied Linear Algebra(LA'03). Williamsburg, USA, 2003

[2] IEEE. Standard for binary floating point arithmetic ANSI/IEEE standard754-2008. USA, 2008

[3] Muller J M, et al. Handbook of Floating Point Arithmetic. Boston, USA: Birkhauser Boston, 2009

[4] Hida Y, Li X S, Bailey D H. Algorithms for quad-double precision floating point arithmetic//Proceedings of the 15th Symposium on Computer Arithmetic(ARITH'01). San Diego, USA, 2001: 155-162

[5] Goto K, van de Geijn R. High-performance implementation of the level-3 BLAS. ACM Transactions on Mathematical Software, 2008, 35: 1-14

[6] Goto K, van de Geijn R. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software, 2008, 34: 1-25

[7] Whaley R C, Dongarra J J. Automatically tuned linear algebra software//Proceedings of the 12th International Conference on Supercomputing(ICS'98). Melbourne, Australia, 1998: 1-27

[8] Zhang X Y, Wang Q, Zhang Y Q. Model-driven level 3 BLAS performance optimization on Loongson 3A processor//Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems(ICPADS'12). Singapore, 2012: 684-691

[9] Wang Q, Zhang X Y, Zhang Y Q, Yi Q. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs//Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis(SC'13). Denver, USA, 2013: No. 25

[10] Zhu Hai-Tao, Li Ling, Chen Yun-Ji, Qian Cheng. Matrix multiplication performance model for optimizing general-purpose processor architecture. Journal of Chinese Computer Systems, 2012, 33(5): 981-986(in Chinese)
(朱海涛, 李玲, 陈云霁, 钱诚. 一种用于通用处理器结构优化的矩阵乘法性能模型. 小型微型计算机系统, 2012, 33(5): 981-986)

[11] Cui H M, Wang L, Fan D R, Feng X B. Automatic library generation for BLAS3 on GPUs//Proceedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS'11). Anchorage, USA, 2011: 255-265

[12] Cui H M, Yi Q, Xue J L, Feng X B. Layout-oblivious compiler optimization for matrix computations. ACM Transactions on Architecture and Code Optimization, 2013, 9(4): 35

[13] Yi Q, Wang Q, Cui H M. Specializing compiler optimizations through programmable composition for dense matrix computations//Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO'14). Cambridge, UK, 2014: 596-608

[14] Tan G M, Li L, Triechle S, et al. Fast implementation of DGEMM on Fermi GPU//Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis(SC'11). Seattle, USA, 2011: 35:1-35:11

[15] Li J J, Li X J, Tan G M, et al. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs//Proceedings of the 26th ACM International Conference on SuperComputing (ICS'12). Venice, Italy, 2012: 377-386

[16] Wang L N, Wu W, Xiao J X, Yang Y. BLASX: A high performance level-3 BLAS library for heterogeneous multi-GPU computing//Proceedings of the 30th ACM International Conference on SuperComputing(ICS'16). Istanbul, Turkey, 2016: No. 20

[17] Li X S, Demmel J W, Bailey D H, et al. Design, implementation and testing of extended and mixed precision BLAS. ACM Transactions on Mathematical Software, 2002, 28(2): 152-205

[18] Mukunoki D, Takahashi D. Implementation and evaluation of quadruple precision BLAS functions on GPUs//Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing (PARA'10). Reykjavík, Iceland, 2010: 249-259

[19] Jiang Hao, Wang Feng, Zuo Ke, et al. The implementation of DGEMM for 64-bit ARMv8 multi-core processors. Journal of Northeastern University Natural Science, 2014, 35(S1): 37-43(in Chinese)
(姜浩，王峰，左克等. 面向 ARMv8 64 位多核处理器 DGEMM 的实现与优化. 东北大学学报自然科学版，2014，35(增刊 1)：37-43)

[20] Wang F, Jiang H, Zu K, et al. Design and implementation of a highly efficient DGEMM for 64-bit ARMv8 multi-core processors//Proceedings of the 44th International Conference on Parallel Processing (ICPP'15). Beijing, China, 2015: 200-209

[21] Jiang H, Wang F, Li K, et al. Implementation of an accurate and efficient compensated DGEMM for 64-bit ARMv8 multi-core processors//Proceedings of the 21st International Conference on Parallel and Distributed Systems(ICPADS'15). Melbourne, Australia, 2015: 200-209

[22] Knuth D E. The Art of Computer Programming, Volume 2, Seminumerical Algorithms. MA, USA: Addison-Wesley, 1998

[23] Dekker T J. A floating-point technique for extending the available precision. Numerische Mathematik, 1971, 18(3): 224-242

[24] Ogita T, Rump S M, Oishi S. Accurate sum and dot product. SIAM Journal on Scientific Computing, 2005, 26: 1955-1988

[25] Jiang Hao. Study on Reliable Computing and Rounding Error Analysis in Floating-Point Arithmetic [Ph. D. dissertation]. National University of Defense Technology, Changsha, 2013 (in Chinese)
(姜浩. 高精度可靠性浮点计算及舍入误差分析研究[博士学位论文]. 国防科学技术大学，长沙，2013)

[26] Nievergelt Y. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. ACM Transactions on Mathematical Software, 2003, 29(1): 27-48

JIANG Hao, born in 1983, Ph. D., assistant professor. His research interests include high performance computing, rounding error analysis, numerical computation.

DU Qi, born in 1986, M. S., engineer. His research interests include parallel computing and performance optimization.

GUO Min, born in 1972, M. S., associate professor. Her research interests include software and algorithm.

QUAN Zhe, born in 1982, Ph. D., lecturer. His research interests include AI, algorithm and compiler.

ZUO Ke, born in 1978, Ph. D., assistant professor. His research interests include internet and complier.

WANG Feng, born in 1979, Ph. D., associate professor. His research interests include complier, performance model and computer architecture.

YANG Can-Qun, born in 1968. Ph. D., professor. His research interests include high performance computing, heterogeneous computing and large scale parallel software optimization.

Background

In high-performance computing, large-scale and long-time numerical calculations often produce inaccurate and invalidated results owing to cancellation from round-off errors. To deal with this problem, high precision computation is required. Since most modern processors only support up to double precision, higher precision operations are usually carried out via software emulation. A common software technique for achieving nearly quadruple precision is double-double arithmetic.

This article introduces the first implementation of quad-precision matrix-matrix multiplication (QGEMM) based on ARMv8 64-bit multi-core processor with OpenBLAS. The purpose of the project is providing theoretical and technical support to build the next generation of the efficient and credible numerical algebra algorithm library. As a basic and common research of high-performance scientific computing, the project will greatly enhance the practical application of high performance computers and promote the development of various fields, such as biology, ocean, atmosphere, and defense.

This research is partly supported by the National High Technology Research and Development Program of China (No. 2012AA01A301), the National Natural Science Foundation of China (61402495, 61303189, 61602166, 61170049, 61402496). Our research group has published several related research papers on journals and conferences, such as Journal of Computational and Applied Mathematics, Applied Mathematics and Computation, Computers & Mathematics with Applications, Applied Numerical Mathematics, Internal Conference on Parallel Processing, Internal Conference on Parallel and Distributed Systems, IEEE Symposium on Computer Arithmetic.