doi:10.11887/j.cn.202301006　　http://journal.nudt.edu.cn

多核数字信号处理器并行矩阵转置算法优化*

裴向东[1],王庆林[1,2],廖林玉[1,2],李荣春[1,2],梅松竹[1,2],刘　杰[1,2],庞征斌[1]
(1. 国防科技大学 计算机学院,湖南 长沙　410073;2. 国防科技大学 并行与分布处理国防科技重点实验室,湖南 长沙　410073)

摘　要:矩阵转置是矩阵运算的基本操作,广泛应用于信号处理、科学计算以及深度学习等各种领域。随着国防科技大学自主研制的飞腾异构多核数字信号处理器(digital signal processor,DSP)在各种领域中的推广应用,对高性能矩阵转置实现提出了强烈需求。针对飞腾异构多核 DSP 的体系结构特征与矩阵转置操作的特点,提出了一种适配不同数据位宽(8 B、4 B 以及 2 B)矩阵的并行矩阵转置算法 ftmMT。该算法基于 DSP 中向量处理单元的 Load/Store 部件实现了向量化,同时基于矩阵分块实现了多个 DSP 核的并行处理,通过隐式乒乓设计实现了片上向量化转置与片外访存的重叠以及访存性能的大幅提升。实验结果表明,ftmMT 能够显著加快矩阵转置操作,与 CPU 上的开源转置库 HPTT 相比,可获得高达 8.99 倍的性能加速。

关键词:多核 DSP;矩阵转置;并行算法;算法优化
中图分类号:TP391　　**文献标志码**:A　　**开放科学(资源服务)标识码(OSID)**:
文章编号:1001-2486(2023)01-057-10

Optimizing parallel matrix transpose algorithm on multi-core digital signal processors

PEI Xiangdong[1], *WANG Qinglin*[1,2], *LIAO Linyu*[1,2], *LI Rongchun*[1,2], *MEI Songzhu*[1,2], *LIU Jie*[1,2], *PANG Zhengbin*[1]
(1. College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China;
2. Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, China)

Abstract: Matrix transpose is one of the common matrix operations, which is widely employed in various fields such as signal processing, scientific computing, and deep learning. With the popularization of Phytium heterogeneous multi-core DSPs(digital signal processors) developed by National University of Defense Technology, there is a strong demand for high-performance matrix transpose implementations for Phytium multi-core DSPs. Based on the architecture of multi-core DSPs and the characteristic of matrix transpose operations, a parallel matrix transpose algorithm (called ftmMT) for matrices with different element bit widths (8 B, 4 B, and 2 B) was proposed. In ftmMT, the main optimizations include vectorization based on vector Load/Store functions, core-level parallelization based on matrix blocking, and overlapping between vectorization and memory access through implicit ping-pong methods. The experimental results show that ftmMT can significantly improve the performance of matrix transpose operations, and achieve a speedup of up to 8.99 times in comparison with the open-source transpose library HPTT running on CPU.

Keywords: multi-core digital signal processors; matrix transpose; parallel algorithm; algorithm optimization

矩阵转置是矩阵运算中最常见的一种操作,广泛应用于科学和工程领域[1-2],如信号处理、科学计算和深度学习[3-5]。作为重要的基础算子[6-7],矩阵转置的效率高低对应用的性能会产生直接的影响。特别是矩阵转置操作属于访存受限型运算,对访存密集型应用的影响将更大[8-9]。

面对各领域对高性能矩阵转置操作的需求,国内外学者对矩阵转置优化已经进行了大量的研究工作。远远等提出适应对称多处理器系统的两维均衡细粒度交织矩阵转置算法[10],通过提升矩阵在存储器上的读写效率来降低矩阵转置开销。Zekri 等提出了面向 Intel 处理器的矩阵转置优化算法[11],通过采用向量扩展指令 AVX 来加速矩阵转置运算。王琦等提出了面向 Intel KNL 融合处理器的并行矩阵转置优化[12]。Springer 等提出了面向张量转置的编译器 TTC[13],以离线方式自动产生高性能的 C++转置实现。TTC 效率虽高,但不能直接应用在运行时才能确定张量大小与转置需求的场景中,因此 Springer 等随后又提出了支持在线自动调优的张量转置库 HPTT[14],

* **收稿日期**:2022-07-09
基金项目:国家自然科学基金资助项目(62002365)
作者简介:裴向东(1985—),男,山西长治人,博士研究生,E-mail:18903588277@163.com;
王庆林(通信作者),男,贵州思南人,副研究员,博士,硕士生导师,E-mail:wangqinglin_thu@163.com

集成了自动调优、显式向量化和多线程并行等优化技术，在多种中央处理器（central processing unit，CPU）硬件架构上均展现了优异的性能，是当前面向 CPU 架构的流行张量转置开源库。肖汉等采用 OpenCL 编程模型实现了面向图形处理器（graphic processing unit，GPU）的并行矩阵转置算法优化，相比 CUDA 实现，具有更好的可移植性[15]。Hynninen 等面向 NVIDIA GPU 构建了张量转置函数库 cuTT[16]。Vedurada 等为 GPU 提出张量转置库 TTLG[17]，构建了性能预测模型来选择不同转置内核以及分块大小。高捷针对“神威”国产超算平台优化了张量转置库 SWTT[18]。

由于能源效率和功耗的限制，高性能计算的异构计算领域引入了低功耗嵌入式架构，如数字信号处理器[19]（digital signal processor，DSP）。与 CPU 和 GPU 相比，DSP 通常采用基于超长指令字（very long instruction word，VLIW）和顺序执行的向量核，集成了直接存储器存取（direct memory access，DMA）引擎用于数据访问[20-21]，从而已有的针对 CPU 和 GPU 的并行矩阵转置实现并不能直接适用于多核 DSP 架构。

FT-M7032 是国防科技大学自主研发的一款异构多核 DSP[19]，主频为 1.8 GHz 时双精度浮点峰值性能高达 5.53 Tflops/s，在信号处理、科学计算和深度学习等领域有巨大潜力[22]，对高性能矩阵转置实现提出了强烈需求。本文面向 FT-M7032 芯片的体系结构特征，研究高性能矩阵转置并行实现特性，提出了一种适配不同数据位宽（8 B、4 B 以及 2 B）矩阵的并行矩阵转置算法 ftmMT，基于 DSP 核中向量 Load/Store 单元支持的取模存储功能构建了向量化矩阵转置核心，通过矩阵分块、隐式乒乓以及多核并行的设计实现片上向量化转置与片外访存的重叠以及对片外存储带宽的高效利用。实验结果显示，ftmMT 获得了高达 75.95% 的 DDR 有效带宽利用率；与主流矩阵转置开源算法库 HPTT 相比，ftmMT 实现了高达 8.99 倍的性能加速。因此，本文工作对于推动 FT-M7032 在各领域的应用具有重要的意义。

1 矩阵转置定义与 FT-M7032 结构简介

本文的研究范围仅限于二维矩阵，故令输入矩阵为 $\boldsymbol{A}[M][N]$，转置输出矩阵为 $\boldsymbol{B}=\boldsymbol{A}^{\mathrm{T}}$，其中 M 和 N 表示矩阵的行数和列数。矩阵转置的具体计算如式（1）所示：

$$\boldsymbol{B}_{(m,n)}=\boldsymbol{A}_{(n,m)} \tag{1}$$

其中，$0\leqslant m<M$，$0\leqslant n<N$。

FT-M7032 是由一个多核 CPU 和 4 个 GPDSP 簇构成，其体系结构如图 1 所示。多核 CPU 是一个简化版本的 FT-2000plus 处理器[4-5]，由 16 个兼容 ARMv8 架构的 CPU 核构成，主要负责进程管理和通信，上面运行操作系统。4 个 GPDSP 簇负责提供最主要的计算性能，每个 GPDSP 簇由 8 个 DSP 核与全局共享内存（global shared memory，GSM）通过片上互联网络连接而成。GSM 大小为 6 MB，由单簇内的 8 个 DSP 核共享，可以作为数据或指令的片上存储空间使用。同时，CPU 与 GPDSP 共享内存空间。每个 GPDSP 簇所能访问的 DDR 理论带宽为 42.62 GB/s。每个 GPDSP 簇中的 DSP 核基于 VLIW 架构实现，包含指令调度单元（instruction fetch unit，IFU）、标量处理单元（scalar processing unit，SPU）、向量处理单元（vector processing unit，VPU）和 DMA 引擎等部分，FT-M7032 中 DSP 核的微体系结构如图 2 所示。

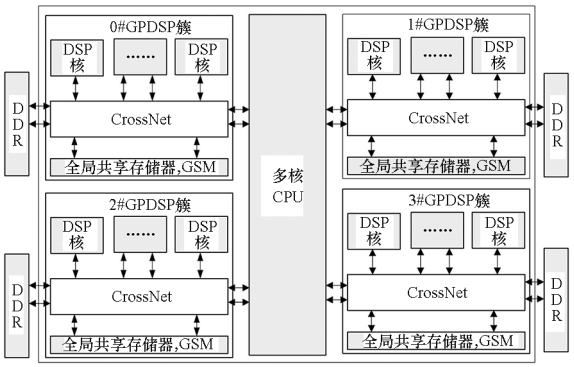

图 1 FT-M7032 体系结构

Fig. 1 Architecture of FT-M7032

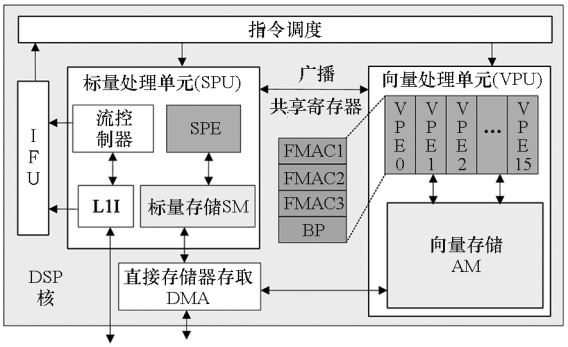

图 2 FT-M7032 中 DSP 核的微体系结构

Fig. 2 Micro-architecture of DSP cores in FT-M7032

SPU 用于指令流控制和标量计算，主要由标量处理部件（scalar processing elements，SPE）和 64 KB 标量存储（scalar memory，SM）空间组成，每周期最多能够处理 5 条标量指令。VPU 主要由 768 KB 的向量存储（vector memory，AM）空间

和 16 个向量处理部件(vector processing elements, VPEs)构成,为每个 DSP 核提供最主要的计算能力。每个 VPE 包含 64 个寄存器、3 个浮点乘累加(floating point multiply accumulator, FMAC)和1 个位操作(bit processing, BP)单元,一次可以处理 1 个8 B数据(如 FP64、Int64)、2 个 4 B 数据(如 FP32、Int32),或 4 个 2 B 数据(如 FP16、Int16)。16 个 VPE 以单指令多数据(single instruction multiple data,SIMD)方式运行,因而 8 B 数据类型的 SIMD 宽度为 16。AM 空间可以通过 VPU 中的两个向量 Load/Store 单元在每个周期向寄存器提供多达 512 B 的数据,从而 AM 与向量寄存器之间的访问带宽高达 921.6 GB/s。

由于功耗的限制,VPU 的多个 VPEs 之间不支持混洗功能,同时 DMA 部件也不支持矩阵转置功能。尽管如此,VPU 的 Load/Store 单元支持取模存储 VSTDW0M16/VSTDW1M16 指令,如图 3所示。执行完该指令后,可以将 4 个向量寄存器中的数据按地址模 16 的方式存储在 AM 空间中,即实现 4 ×16 矩阵的转置。

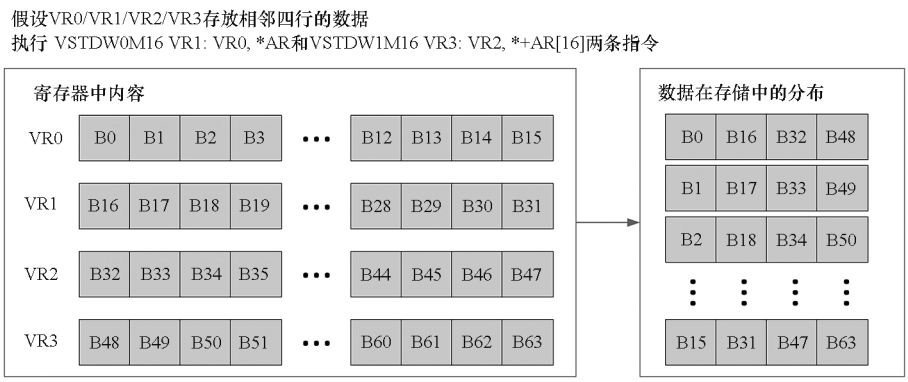

图 3　VPU 中模 16 的存储功能

Fig. 3　Store based on Module 16 in VPU

2　面向 FT-M7032 的矩阵转置实现分析

在 FT-M7032 上实现矩阵转置主要有四种方法:一是基于 FT-M7032 上多核 CPU 的实现,二是基于 DSP 纯 DMA 操作的实现,三是基于 DSP SPU 的实现,四是基于 DSP VPU 的实现。

基于 FT-M7032 上多核 CPU 的实现方法是直接调用已有面向 CPU 的开源矩阵转置库,如 HPTT[14]等。然而,由上一节内容可知,FT-M7032 的主要计算能力是由 GPDSP 簇提供,虽然多核 CPU 与 GPDSP 簇共享存储空间,但 GPDSP 簇与多核 CPU 之间进行交互时需要将多核 CPU 中 Cache 的内容进行作废或者写回 DDR 中,当多次交互时,交互开销较大。

基于 DSP 纯 DMA 操作的实现方法是通过将 DMA 操作的块大小设置为矩阵中单个数据的大小来实现转置。然而,FT-M7032 中 DMA 操作最小粒度为 8 B,这意味着该方法只适合于单个数据为 8 B 的类型,如 FP64、Int64 等。同时,DMA 的块大小直接影响 DMA 的访存效率,这意味着使用该方法不能充分发挥 FT-M7032 的访存带宽。

基于 DSP SPU 的实现方法是先将数据加载到 SPU 的 SM 空间,然后基于标量操作来实现矩阵转置运算。在 FT-M7032 的 SPU 中,SM 仅为 64 KB,使得在转置实现过程中需要频繁调用 DMA,DMA 操作的启动开销较大。与此同时,SPU 仅有一个标量 Load/Store 单元,每周期仅能向寄存器提供 16 B 的数据,使得基于 SPU 的片上转置核心在实现过程中性能受限。

基于 DSP VPU 的实现方法则是将数据加载到 VPU 的 AM 空间,然后基于向量操作来实现矩阵转置运算。在 FT-M7032 的 VPU 中,AM 空间大小为 768 KB,VPU 的两个 Load/Store 单元每周期可以提供 512 B 的数据。相比前面两种基于 DSP 的实现方法,可一定程度上有效克服片上存储带宽受限、频繁调用 DMA 以及访存效率较低等问题。

综上所述,基于 DSP VPU 的方法是面向 FT-M7032 的高性能矩阵转置实现的必然选择。但是 VPU 的多个 VPE 之间不支持混洗功能,使得基于 DSP VPU 的转置实现成为一个难题。本文将通过充分利用 VPU 中 Load/Store 单元支持的取模存储功能来解决这一问题。考虑到基于纯 DMA 操作和 SPU 的两种实现在理论上性能就远低于本文基于 VPU 的实现,故本文在后续实验部分仅

与基于多核 CPU 的实现进行性能比较。

3 并行矩阵转置算法与优化

本节先对本文提出的面向飞腾异构多核 DSP 的并行矩阵转置算法 ftmMT 进行整体介绍,然后对该算法的具体思路及实现过程做详细描述。

3.1 ftmMT 算法整体设计

如第 2 节所分析,基于 DSP VPU 的转置实现是面向 FT-M7032 的高性能矩阵转置实现的必然选择。基于此,本文提出了并行矩阵向量化转置算法 ftmMT,如算法 1 所示。ftmMT 算法充分结合了 FT-M7032 的体系结构特征和矩阵转置的计算访存特点,主要由三个步骤构成。

算法 1 并行矩阵转置算法 ftmMT

Alg. 1 Parallel matrix transpose algorithm (ftmMT)

输入:矩阵 $\boldsymbol{A}[M][N]$

输出:矩阵 $\boldsymbol{B}=\boldsymbol{A}^{\mathrm{T}}$

1. Step 1:调用 cache_flush_all( ) 函数将 CPU Cache 中的内容写回 DDR。
2. Step 2:调用 DSP 端函数__ MTrn ($\boldsymbol{A},\boldsymbol{B},M,N$) 完成矩阵转置运算。
3. Step 3:调用 cache_inv_all( ) 函数作废 CPU Cache 中的内容。
4. Function __ MTrn ($\boldsymbol{A},\boldsymbol{B},M,N$)
5. 计算分块大小 M_{b} 与 N_{b};
6. **for** $m=0:M_{\mathrm{b}}:M$ **do** in parallel
7. $m_{\mathrm{b}}=\min(M-m,M_{\mathrm{b}})$
8. $M_{\mathrm{a}}=(m_{\mathrm{b}}+h_{\mathrm{a}}-1)/h_{\mathrm{a}}\times h_{\mathrm{a}}$
9. **for** $n=0:N_{\mathrm{b}}:N$ **do** in parallel
10. $n_{\mathrm{b}}=\min(N-n,N_{\mathrm{b}})$
11. $N_{\mathrm{a}}=(n_{\mathrm{b}}+w_{\mathrm{a}}-1)/w_{\mathrm{a}}\times w_{\mathrm{a}}$
12. 调用 DMA 函数将 DDR 中 $\boldsymbol{A}_{(m,n)}[m_{\mathrm{b}}][n_{\mathrm{b}}]$ 传入 AM 空间,并扩展为 $\boldsymbol{A}_{\mathrm{am}}[M_{\mathrm{a}}][N_{\mathrm{a}}]$
13. trnKernel($\boldsymbol{A}_{\mathrm{am}},\boldsymbol{B}_{\mathrm{am}},M_{\mathrm{a}},N_{\mathrm{a}}$)
14. 调用 DMA 函数将 AM 空间 $\boldsymbol{B}_{\mathrm{am}}[N_{\mathrm{a}}][M_{\mathrm{a}}]$ 中 $n_{\mathrm{b}}\times m_{\mathrm{b}}$ 大小子矩阵存储回 DDR 中 $\boldsymbol{B}_{(n,m)}[n_{\mathrm{b}}][m_{\mathrm{b}}]$ 位置
15. **end for**
16. **end for**
17. Function trnKernel($\boldsymbol{A}_{\mathrm{am}},\boldsymbol{B}_{\mathrm{am}},M_{\mathrm{a}},N_{\mathrm{a}}$)
18. **for** $i=0:h_{\mathrm{a}}:M_{\mathrm{a}}$ **do**
19. **for** $j=0:w_{\mathrm{a}}:N_{\mathrm{a}}$ **do**
20. Vtkernel($\boldsymbol{A}_{\mathrm{am}},\boldsymbol{B}_{\mathrm{am}},M_{\mathrm{a}},N_{\mathrm{a}},i,j$)
21. **end for**
22. **end for**

步骤 1:将 CPU Cache 中的内容写回 DDR 中(Step 1)。

步骤 2:调用 DSP 端函数__ MTrn( ) 执行矩阵转置运算(Step 2)。

步骤 3:作废 CPU Cache 中的内容,完成矩阵转置运算(Step 3)。

DSP 端函数__ MTrn( ) 负责在一个 GPDSP 簇上完成矩阵的转置操作,其基本思想是将 $\boldsymbol{A}$ 矩阵划分为 $M_{\mathrm{b}}\times N_{\mathrm{b}}$ 大小的子矩阵,依次加载到 DSP 的 AM 空间并扩展成 $M_{\mathrm{a}}\times N_{\mathrm{a}}$ 大小(第 12 行),调用 trnKernel( ) 函数完成 $M_{\mathrm{a}}\times N_{\mathrm{a}}$ 子矩阵的转置运算(第 13 行),然后从 AM 空间读取 $N_{\mathrm{b}}\times M_{\mathrm{b}}$ 子矩阵存储回 $\boldsymbol{B}$ 矩阵中(第 14 行)。trnKernel( ) 函数负责将 AM 空间中 $M_{\mathrm{a}}\times N_{\mathrm{a}}$ 的矩阵进一步划分成 $h_{\mathrm{a}}\times w_{\mathrm{a}}$ 大小的子矩阵,然后依次调用向量化矩阵转置核心 Vtkernel( ) 基于 VPU 完成 $h_{\mathrm{a}}\times w_{\mathrm{a}}$ 子矩阵的转置。

对于 DSP 端的应用,可以直接调用 DSP 端函数__ MTrn( ),避免了 CPU 与 DSP 之间的反复交互开销。

总的来说,本文提出的并行矩阵转置算法 ftmMT 具有以下三个特点:

1)不依赖任何 CPU 开源转置函数库,也不依赖其他 DSP 矩阵转置函数库;

2)基于 AM 空间实现,增加了 DMA 每次传输的数据量,从而减少了 DMA 操作的总次数,大幅降低了 DMA 启动开销;

3)基于 DSP VPU 单元实现,能够有效利用 VPU 单元的 Load/Store 带宽。

3.2 向量化矩阵转置核心

Vtkernel( ) 是 DSP 上基于 VPU 来实现矩阵转置的关键核心函数。为充分利用 FT-M7032 的向量部件来提高矩阵转置的性能,以单个数据大小为 8 B、h_{a} 和 w_{a} 均为 16 为例,本文提出了如图 4所示的 Vtkernel( ) 的实现过程,一共分为 6 个步骤。

步骤 1:基于 VPU 的向量 Load 功能将 16 × 16 大小子矩阵中的 16 行分别读入 16 个对应的向量寄存器中。比如,第 0 ~ 3 行,分别读入向量寄存器 VR0/4/8/12 中;第 4 ~ 7 行,分别读入向量寄存器 VR1/5/9/13 中等。

步骤 2:对向量寄存器进行分组,将向量寄存器分为 VR0 ~ VR3、VR4 ~ VR7、VR8 ~ VR11 和 VR12 ~ VR15 四组,然后将分组后的寄存器组分别调用模 16 存储指令 VSTDW0M16/VSTDW1M16,第一次将数据存入 AM 临时空间,分别完成 4 行的转

置。比如,基于 VR0 ~ VR3,调用模 16 存储指令 VSTDW0M16/VSTDW1M16 完成了原 16 × 16 子矩阵中第 0、4、8、12 行组成的新子矩阵的转置。

步骤 3:基于 VPU 的向量 Load 功能将步骤 2 产生的 AM 临时空间中 16 × 16 大小临时矩阵重新读入 16 个对应的向量寄存器中,读入规则与步骤 1 完全一致。

步骤 4:再次将向量寄存器进行分组,并再次基于模 16 存储指令 VSTDW0M16/VSTDW1M16 将数据存入 AM 临时空间,分组规则与调用模 16 存储指令的规则与步骤 2 完全一致,从而完成了原 16 × 16 大小子矩阵的转置,转置结果存储在 AM 临时空间。

步骤 5:从 AM 临时空间中读入转置后的临时矩阵到向量寄存器中。

步骤 6:将向量寄存器中保存的转置后子矩阵存入 $M_a \times N_a$ 矩阵中对应位置,完成 $h_a \times w_a$ 大小子矩阵的转置。

对于单个数据大小分别为 4 B 和 2 B 的矩阵来说,处理流程大致相似,只需在步骤 1 与步骤 2 之间加入数据打包过程,即:根据最后的转置需求,将原始子矩阵中相邻两行的 2 个 4 B 数据和相邻四行的 4 个 2 B 数据分别合并为 1 个 8 B数据,分别产生新的 2 行和 4 行 8 B 数据存储于向量寄存器中。最后,基于所产生的新寄存器数据执行步骤 2 ~ 6,构建转置后的矩阵。以单个数据大小为 4 B 为例,设置 h_a 和 w_a 为 32,每两行数据分别进行高低位打包操作形成新

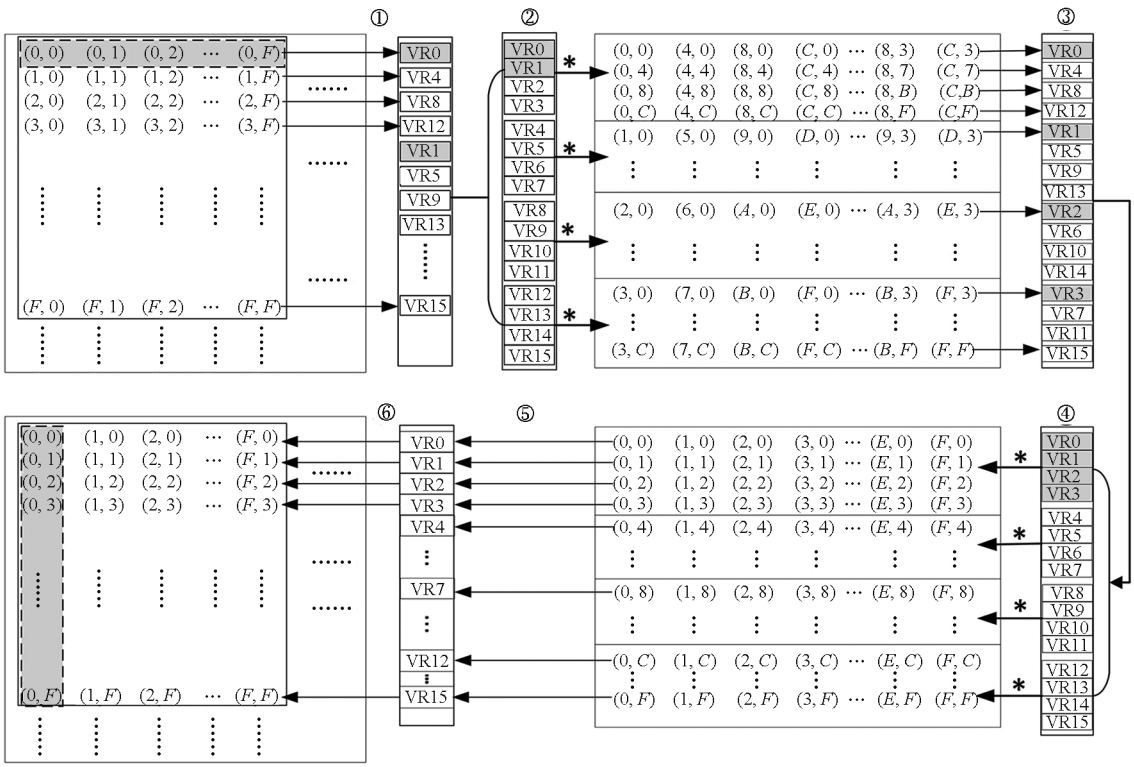

图 4　基于 DSP VPU 的向量化矩阵转置核心流程

Fig. 4　Flow chart of vectorized matrix transpose kernel based on VPU of DSP

的 32 行数据,如图 5 所示。随后将新 32 行数据以行号为基准进行奇偶分组,即奇数编号 16 行与偶数编号 16 行,随后分别执行步骤 2 ~ 6,即可完成 32 × 32 大小 4 B 矩阵的转置。对于单个数据大小为 2 B 的矩阵转置,设置 h_a 和 w_a 为 64,然后分别进行打包和转置。

3.3　分块设计

本文算法在设计中主要涉及 $h_a \times w_a$、$M_b \times N_b$ 两组分块参数的选择与评估。

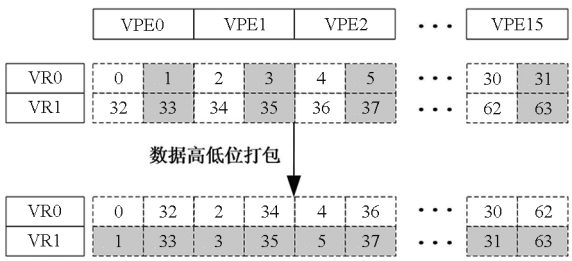

图 5　VPU 中高低位数据打包功能

Fig. 5　Packing of data in VPU

在矩阵转置过程中，$h_a \times w_a$ 表示 Vtkernel 一次处理的子矩阵大小，其主要涉及两个方面因素的影响：一是对于特定元素大小，向量化转置核心功能实现中所要求的最小矩阵大小。如第 3.2 节所示，对于 8 B、4 B 以及 2 B 的矩阵转置来说，$h_a \times w_a$ 最小要求为 16×16、32×32 以及 64×64。二是为充分利用 VPU 中两个向量 Load/Store 单元的处理能力，需要同时执行多个最小矩阵的转置，尽可能填充满流水线，实现 VPU 的最大化利用。在本文实现中，对于 8 B、4 B 以及 2 B 的矩阵转置，Vtkernel 分别一次完成 4 个、1 个以及 1 个最小矩阵的转置，从而 $h_a \times w_a$ 分别取为 32×32、32×32 以及64×64。

$M_b \times N_b$ 的选择涉及四个方面因素的影响：一是 M_b 和 N_b 应该尽可能分别是 h_a 和 w_a 的整数倍。二是受到 AM 可用空间大小的约束，即存储 $\boldsymbol{A}_{am}[M_a][N_a]$、$\boldsymbol{B}_{am}[N_a][M_a]$ 和 Vtkernel 实现中临时空间大小 $Temp_{am}$ 之和不能超过 AM 可用空间的大小。对于8 B、4 B 以及2 B 的 Vtkernel 实现，$Temp_{am}$ 分别为 8 KB、4 KB 以及8 KB。三是 DMA 访问 DDR 内存的性能受连续访问的块大小与调用次数的影响较大。从 DDR 读取矩阵 $\boldsymbol{A}$ 的子矩阵 $\boldsymbol{A}_{(m,n)}[M_b][N_b]$ 过程中，连续访问 DDR 的块大小为 N_b 个数据；将转置后的子矩阵 $\boldsymbol{B}_{am}[N_b][M_b]$ 存入 DDR 中 $\boldsymbol{B}$ 的过程中，块大小为 M_b 个数据。因此，在前两个因素的约束下，M_b 与 N_b 的取值理论上应尽可能接近，从而才能在转置实现过程中同时保证 DDR 读和写的性能。四是根据参数的实时调整，当输入矩阵 $\boldsymbol{A}$ 的宽度 N 小于 N_b 或高度 M 小于 M_b 时，将分别动态降低 N_b 或 M_b 的大小。同时，在 AM 可用空间约束下，随之增大 M_b 或 N_b 的大小。

3.4 多核并行与乒乓设计

为充分发挥多核 DSP 的并行性能，本文算法设计中将实现面向多个 DSP 核的线程级并行，具体为将矩阵 $\boldsymbol{A}$ 划分为多个 $M_b \times N_b$ 子矩阵，然后调用多个 DSP 核并行完成多个 $M_b \times N_b$ 子矩阵的转置，直到完成矩阵 $\boldsymbol{A}$ 中所有 $M_b \times N_b$ 子矩阵的转置。在算法实现上，即在 M 和 N 两个维度进行多核并行，如算法 1 中第 6 行和第 9 行所示。

乒乓算法是用来实现计算与数据传输重叠的经典方法，分为显式乒乓与隐式乒乓两种。显式乒乓是在单个 DSP 核内实现计算与数据传输重叠，需要将单个 DSP 核内如 AM 等片上空间平均划分为两部分，一部分用于数据传输（DMA 操作），一部分用于计算（trnKernel 函数）。隐式乒乓算法是在多个 DSP 核之间实现计算与数据传输的重叠，即一个 DSP 核的计算与另一个 DSP 核的数据传输重叠。在 FT-M7032 单簇 8 个 DSP 核并行计算时，隐式乒乓算法是一种天然存在的重叠形态，不需进行特殊设计。两种算法相比各有优缺点，如隐式乒乓算法中总存在 DSP 计算核在等待数据的传输，而显式乒乓算法虽然可以让单个 DSP 核不间断地进行计算，但每次 DMA 传输的数据量降为隐式算法的一半，总数据传输开销变大。鉴于矩阵转置属于访存密集型操作，访存性能或数据传输性能才是决定性能的关键。因此，ftmMT 采用隐式乒乓算法来实现 trnKernel 函数与 DMA 数据传输操作的重叠，通过最大化降低数据传输开销来实现总时间开销成本的最小化。

4 性能评估

在本节中，将全面评测本文所提算法 ftmMT 的性能。首先，测试转置矩阵位于片上 AM 空间时矩阵转置实现的性能，即 trnKernel 实现的性能。其次，测试不同 $M_b \times N_b$ 大小对于 ftmMT 性能的影响。最后，测试 ftmMT 在不同矩阵大小下的算法性能，并与运行在 FT-M7032 多核 CPU 上的开源算法 HPTT 进行性能比较。

在本节中将用两个指标来衡量矩阵转置的性能，一个是完成矩阵转置的时间 T，另一个是转置带宽 F，其计算如式(2)所示。

$$F = \frac{2 \times M \times N \times Q_{\text{Sizeof(a)}}}{T} \tag{2}$$

其中，$Q_{\text{Sizeof(a)}}$ 表示矩阵 $\boldsymbol{A}$ 或 $\boldsymbol{B}$ 中一个数据的字节数。本文算法主要支持8 B、4 B 以及2 B 大小数据的矩阵，在后续讨论中对应实现分别标记为 ftmMT－64/trnKernel－64、ftmMT－32/trnKernel－32 以及 ftmMT－16/trnKernel－16。考虑到这些实现所处理的基本块大小 $h_a \times w_a$ 不尽相同，实验中采用了不同的输入矩阵规模来进行对应实现的测试。

4.1 trnKernel 的性能评估

当 $M_a \times N_a$ 取不同大小时，处理不同数据大小的 trnKernel 的性能如图 6～8 所示。对于所有测试的子矩阵来说，时间开销均低于 10 μs。随着 $M_a \times N_a$ 大小的增加，转置带宽呈逐渐增加的趋势。对于 8 B 矩阵，当 $M_a \times N_a$ 为 192×224，获得了最高 145.45 GB/s 的矩阵转置带宽，如图 6 所示；对于 4 B 矩阵，当分块大小为 256×352 时，最高矩阵转置带宽为 98.12 GB/s，如图 7 所示；对

于 2 B 矩阵,当分块大小为 384 × 448 时,最高矩阵转置带宽为 102.77 GB/s,如图 8 所示。

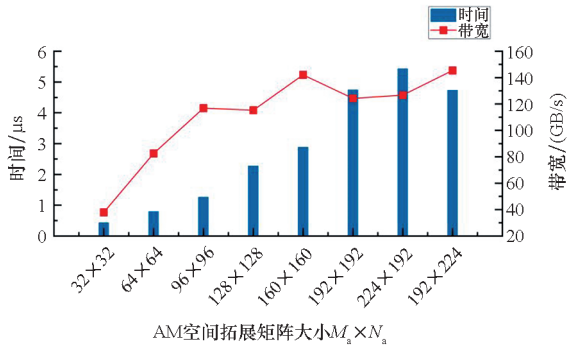

图 6　trnKernel – 64 实现的性能

Fig. 6　Performance of trnKernel – 64 implementation

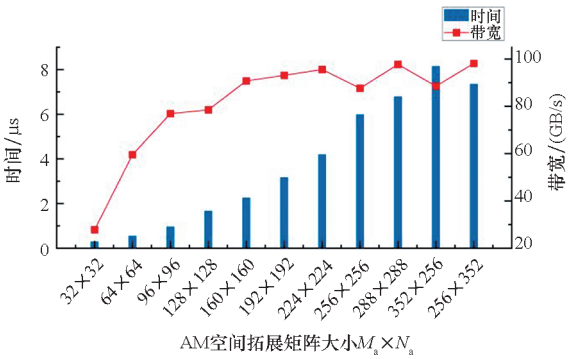

图 7　trnKernel – 32 实现的性能

Fig. 7　Performance of trnKernel – 32 implementation

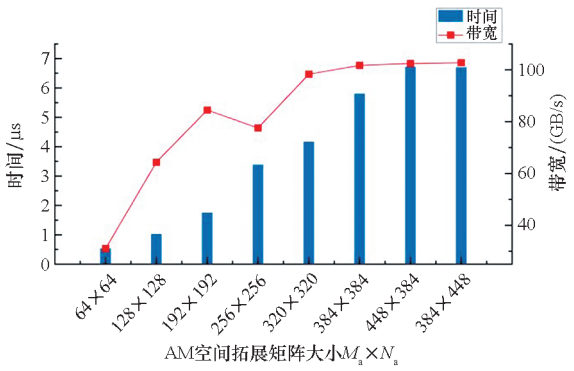

图 8　trnKernel – 16 实现的性能

Fig. 8　Performance of trnKernel – 16 implementation

4.2　不同分块大小下 ftmMT 的性能评估

如第 3.3 节分析所知,分块矩阵 $M_b \times N_b$ 受多个因素约束,同时也对最终矩阵转置性能有较大的影响。为评估不同分块大小的性能,本小节测试了 ftmMT 实现在 $M_b \times N_b$ 不同大小下的性能,实验结果分别如图 9 ~ 11 所示。其中,横坐标表示不同的分块大小,从左到右,M_b 逐渐增大,N_b 逐渐减小,即分块子矩阵从矮瘦矩阵逐渐演变为高瘦矩阵,但 M_b 和 N_b 的乘积尽可能接近 ftmMT 实现中 AM 空间所允许的最大值。本节测试中,输入矩阵 $\boldsymbol{A}$ 的大小 $M \times N$ 为固定值,对于 ftmMT – 64 和 ftmMT – 32,输入矩阵 $\boldsymbol{A}$ 均为 16 384 × 16 384 的方形矩阵;对于 ftmMT – 16,输入矩阵 $\boldsymbol{A}$ 的大小为 32 768 × 32 768。

对于 ftmMT – 64,如图 9 所示,当输入矩阵 $\boldsymbol{A}$ 为 16 384 × 16 384 的矩阵,分块大小为 192 × 224 时,矩阵转置性能最佳,其中耗时为 152.56 ms,转置带宽达到了 26.22 GB/s,对应的 DDR 有效带宽利用率为 61.61%。

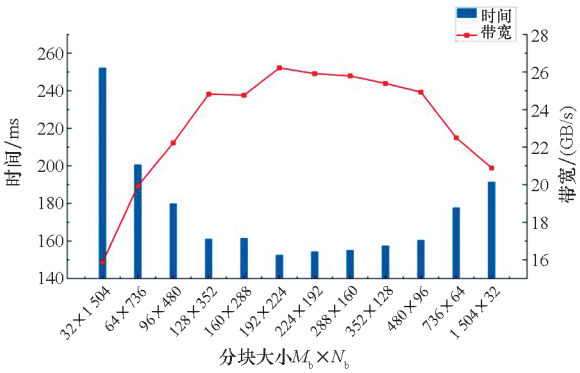

图 9　输入矩阵 $\boldsymbol{A}$ 为 16 384 × 16 384 的矩阵,不同分块大小下 ftmMT – 64 实现的性能

Fig. 9　Performance of ftmMT – 64 under different block sizes when the input matrix $\boldsymbol{A}$ is 16 384 × 16 384 matrix

对于 ftmMT – 32,如图 10 所示,当输入矩阵 $\boldsymbol{A}$ 为 16 384 × 16 384 的矩阵,分块大小为 256 × 352 时,矩阵转置性能最佳,其中耗时为 72.78 ms,转置带宽达到了 27.48 GB/s,相应的 DDR 有效带宽利用率为 65.17%。

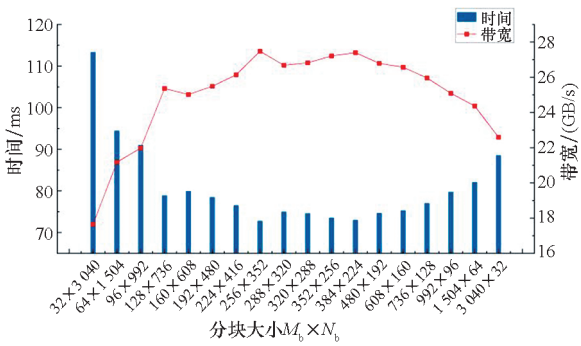

图 10　输入矩阵 $\boldsymbol{A}$ 为 16 384 × 16 384 的矩阵,不同分块大小下 ftmMT – 32 实现的性能

Fig. 10　Performance of ftmMT – 32 under different block sizes when the input matrix $\boldsymbol{A}$ is 16 384 × 16 384 matrix

对于 ftmMT – 16,如图 11 所示,当输入矩阵 $\boldsymbol{A}$ 为 32 768 × 32 768 的矩阵,分块大小为 448 × 384 和 576 × 320 时,矩阵转置性能最佳,其中耗时为 155.34 ms,转置带宽达到了 25.75 GB/s,相应

的 DDR 有效带宽利用率为 60.41%。

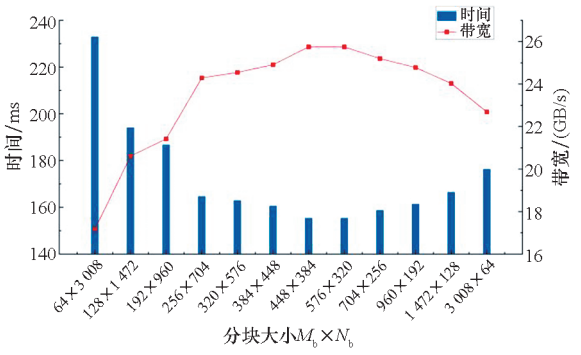

图 11　输入矩阵 $\boldsymbol{A}$ 为 32 768 × 32 768 的矩阵，不同分块大小下 ftmMT – 16 实现的性能

Fig. 11　Performance of ftmMT – 16 under different block sizes when the input matrix $\boldsymbol{A}$ is 32 768 × 32 768 matrix

通过本小节的评估，从图 9 ~ 11 呈现的实验结果可以看出，对于不同分块大小 $M_b \times N_b$，ftmMT – 64、ftmMT – 32 和 ftmMT – 16 所实现的性能均呈现中间高、两边低的特点，与第 3.3 小节的分析设计相符。

4.3　不同大小矩阵下 ftmMT 的性能评估

本小节对 ftmMT 处理不同大小矩阵的性能进行评估，并与运行在 FT-M7032 中多核 CPU 上的开源高性能张量转置库 HPTT 进行性能比较。本小节采用第 4.2 节中获得最佳性能的 $M_b \times N_b$ 大小作为 ftmMT 实现中的分块大小。同时，由于 HPTT 仅支持 8 B 和 4 B 大小数据矩阵的转置，因此也仅进行相关实现的比较。在 HPTT 测试中，线程数设置为 16，即使用 FT-M7032 中所有的 CPU 核来进行并行矩阵转置运算。

对于 ftmMT – 64，测试结果如表 1 所示。与 HPTT 相比，在输入矩阵小于 1 024 × 1 024 规模时，ftmMT – 64 的性能较低。主要原因是 ftmMT – 64 的时间开销里包含了刷 Cache 的开销（算法 1 中 Step 1 和 Step 3 的开销），该部分开销是相对固定的，属于 ftmMT 的固有开销，与输入的矩阵大小无关。当输入的矩阵规模较小时，刷 Cache 的开销在总时间开销中所占比例较大，从而使得 ftmMT 算法的性能低于 HPTT。当输入的矩阵规模变大时，DSP 端函数__MTrn()的时间开销增加，刷 Cache 开销所占比例也逐渐变小。当输入矩阵大于等于 1 024 × 1 024 时，ftmMT – 64 的性能均优于 HPTT。当输入矩阵规模为 4 096 × 4 096 时，ftmMT – 64 的耗时仅为 HPTT 的 15.14%，计算速度为 HPTT 的 6.61 倍。在 ftmMT – 64 的测试中，转置带宽最高为 30.98 GB/s，与 DDR 的理论带宽 42.62 GB/s 相比，对应 DDR 有效带宽利用率达到了 72.69%。除了刷 Cache 开销包含在 ftmMT 总开销内的原因外，影响 ftmMT 的 DDR 有效带宽利用率的另一个重要因素是 DDR 本身的实测带宽与理论值存在差距。

表 1　不同矩阵大小下 ftmMT – 64 与 HPTT 的性能

Tab. 1　Performance of ftmMT – 64 and HPTT implementations under different matrix sizes

矩阵 $H \times W$	ftmMT – 64		HPTT	
	时间/ms	带宽/(GB/s)	时间/ms	带宽/(GB/s)
64 × 64	0.51	0.12	0.03	2.43
128 × 126	0.60	0.41	0.04	6.69
256 × 256	0.67	1.45	0.08	11.60
512 × 512	0.72	5.41	0.34	11.54
1 024 × 1 024	1.07	14.61	1.37	11.37
2 048 × 2 048	2.44	25.62	10.99	5.69
4 096 × 4 096	8.07	30.98	53.30	4.69
8 192 × 8 192	35.76	27.96	212.60	4.70
16 384 × 16 384	152.55	26.22	924.62	4.33

对于 ftmMT – 32，测试结果如表 2 所示。当输入矩阵规模为 8 192 × 8 192 时，转置带宽达到了 32.37 GB/s，相应的 DDR 带宽利用率为 75.95%。与 HPTT 相比，当输入矩阵小于等于 1 024 × 1 024 时，ftmMT – 32 的性能较低，与 ftmMT – 64 情况一样，也是因为刷 Cache 开销占

表 2　不同矩阵大小下 ftmMT – 32 与 HPTT 的性能

Tab. 2　Performance of ftmMT – 32 and HPTT implementations under different matrix sizes

矩阵 $H \times W$	ftmMT – 32		HPTT	
	时间/ms	带宽/(GB/s)	时间/ms	带宽/(GB/s)
64 × 64	0.51	0.06	0.02	1.47
128 × 126	0.64	0.19	0.03	4.28
256 × 256	0.66	0.74	0.05	9.40
512 × 512	0.69	2.84	0.14	14.43
1 024 × 1 024	0.84	9.31	0.62	12.61
2 048 × 2 048	1.52	20.55	2.91	10.76
4 096 × 4 096	4.24	29.50	31.89	3.92
8 192 × 8 192	15.45	32.37	138.86	3.60
16 384 × 16 384	72.78	27.48	579.11	3.45

总执行时间的比率较大。当输入矩阵大于 1 024 × 1 024 时,ftmMT – 32 的性能也是均高于 HPTT。当输入矩阵规模为 8 192 × 8 192 时,ftmMT – 32 的耗时仅为 HPTT 的 11.13%,转置带宽加速比高达 8.99。

对于 ftmMT – 16,测试结果如表 3 所示。当输入矩阵规模为 16 384 × 16 384 时,ftmMT – 32 的转置带宽达到了 31.78 GB/s,DDR 的有效带宽利用率为 74.57%。

表 3　不同矩阵大小下 ftmMT – 16 的性能

Tab. 3　Performance of ftmMT – 16 implementation under different matrix sizes

矩阵 $H \times W$	时间/ms	带宽/(GB/s)
64 × 64	0.56	0.03
128 × 126	0.51	0.12
256 × 256	0.52	0.47
512 × 512	0.55	1.79
1 024 × 1 024	0.63	6.22
2 048 × 2 048	1.01	15.45
4 096 × 4 096	2.38	26.27
8 192 × 8 192	8.17	30.60
16 384 × 16 384	31.47	31.78
32 768 × 32 768	155.31	25.75

从上述实验结果可知,ftmMT – 64、ftmMT – 32 和 ftmMT – 16 均能有效发挥 DDR 存储器的性能,且 ftmMT – 64 与 ftmMT – 32 相对运行在多核 CPU 上的 HPTT 实现了较高的性能提升。

5　结论

本文针对飞腾异构多核 DSP 的体系结构特征与矩阵转置操作的特点,基于 DSP VPU 的实现方法,提出了一种适配不同数据位宽(8 B、4 B 以及 2 B)矩阵的并行矩阵转置算法 ftmMT。ftmMT 基于矩阵分块实现了多个 DSP 核的并行处理,通过隐式乒乓设计实现了片上向量化转置与片外访存的重叠,有效克服片外存储带宽受限、频繁调用 DMA 以及访存效率较低等问题。通过开展相关实验,证明了 ftmMT 能够显著加快多核 DSP 上的矩阵转置操作,DDR 的带宽利用率最高达到 75.95%;与 HPTT 相比,可获得高达 8.99 倍的性能加速。该项研究对于推动国产化多核 DSP 在科学计算和深度学习领域的应用具有重要的意义。

参考文献(References)

[1] CHOI J, DONGARRA J J, WALKER D W. Parallel matrix transpose algorithms on distributed memory concurrent computers[J]. Parallel Computing, 1995, 21(9): 1387 – 1405.

[2] GORAWSKI M, LOREK M. General in situ matrix transposition algorithm for massively parallel environments [C]// Proceedings of International Conference on Data Science and Advanced Analytics, 2014: 379 – 384.

[3] HUANG X D, WANG Q L, LU S Y, et al. Evaluating FFT-based algorithms for strided convolutions on ARMv8 architectures [J]. Performance Evaluation, 2021, 152: 102248.

[4] WANG Q L, LI D S, HUANG X D, et al. Optimizing FFT-based convolution on ARMv8 multi-core CPUs [C]// Proceedings of European Conference on Parallel Processing, 2020: 248 – 262.

[5] 王庆林, 李东升, 梅松竹, 等. 面向飞腾多核处理器的 Winograd 快速卷积算法优化[J]. 计算机研究与发展, 2020, 57(6): 1140 – 1151.
WANG Q L, LI D S, MEI S Z, et al. Optimizing winograd-based fast convolution algorithm on Phytium multi-core CPUs[J]. Journal of Computer Research and Development, 2020, 57(6): 1140 – 1151. (in Chinese)

[6] LU Q D, KRISHNAMOORTHY S, SADAYAPPAN P. Combining analytical and empirical approaches in tuning matrix transposition [C]// Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, 2006: 233 – 242.

[7] KOTLAR M, PUNT M, MILUTINOVIC V. Energy efficient implementation of tensor operations using dataflow paradigm for machine learning[J]. Advances in Computers, 2022, 126: 151 – 199.

[8] 张鹏. 面向异构众核平台的多任务流编程与性能优化技术研究[D]. 长沙: 国防科技大学, 2020.
ZHANG P. Optimizing streaming parallelism for heterogeneous many-core architectures[D]. Changsha: National University of Defense Technology, 2020. (in Chinese)

[9] 苏醒. 高性能稠密线性代数数学库关键技术研究[D]. 长沙: 国防科技大学, 2020.
SU X. Research on high performance dense linear algebra libraries[D]. Changsha: National University of Defense Technology, 2020. (in Chinese)

[10] 远远, 陈亮. 基于 SMP 的矩阵转置算法研究与实现[J]. 计算机工程与设计, 2016, 37(10): 2690 – 2694.
YUAN Y, CHEN L. Research and implementation of matrix transpose based on SMP system[J]. Computer Engineering and Design, 2016, 37(10): 2690 – 2694. (in Chinese)

[11] ZEKRI A S. Enhancing the matrix transpose operation using intel AVX instruction set extension[J]. International Journal of Computer Science and Information Technology, 2014, 6(3): 67 – 78.

[12] 王琦, 韩林, 高雨辰, 等. 基于矩阵转置优化的 Intel KNL 特性分析[J]. 计算机工程与设计, 2018, 39(5): 1358 – 1364, 1371.
WANG Q, HAN L, GAO Y C, et al. Parallel optimization on transposition of square matrices for Knights Landing processors[J]. Computer Engineering and Design, 2018, 39(5): 1358 – 1364, 1371. (in Chinese)

[13] SPRINGER P, HAMMOND J R, BIENTINESI P. TTC: a high-performance compiler for tensor transpositions [J]. ACM Transactions on Mathematical Software, 2017, 44(2): 1-21.

[14] SPRINGER P, SU T, BIENTINESI P. HPTT: a high-performance tensor transposition C++ library [C]// Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, 2017: 56-62.

[15] 肖汉, 李彩林, 李琦, 等. CPU+GPU 异构并行的矩阵转置算法研究[J]. 东北师大学报(自然科学版), 2019, 51(4): 70-77.
XIAO H, LI C L, LI Q, et al. Research on matrix transpose algorithm of CPU + GPU heterogeneous parallelism [J]. Journal of Northeast Normal University (Natural Science Edition), 2019, 51(4): 70-77. (in Chinese)

[16] HYNNINEN A P, LYAKH D I. cuTT: a high-performance tensor transpose library for CUDA compatible GPUs [EB/OL]. (2017-05-03)[2022-06-09]. https://arxiv.org/abs/1705.01598.

[17] VEDURADA J, SURESH A, RAJAM A S, et al. TTLG—an efficient tensor transposition library for GPUs [C]// Proceedings of 2018 IEEE International Parallel and Distributed Processing Symposium, 2018: 578-588.

[18] 高捷. 基于神威国产超算平台的张量库实现与优化[D]. 郑州: 战略支援部队信息工程大学, 2021.
GAO J. Implementation and optimization of tensor library based on Sunway domestic supercomputer platform [D]. Zhengzhou: PLA Strategic Support Force Information Engineering University, 2021. (in Chinese)

[19] YIN S F, WANG Q L, HAO R C, et al. Optimizing irregular-shaped matrix-matrix multiplication on multi-core DSPs[C]//Proceedings of IEEE International Conference on Cluster Computing, 2022: 451-461.

[20] 刘仲, 田希. 面向多核向量处理器的矩阵乘法向量化方法[J]. 计算机学报, 2018, 41(10): 2251-2264.
LIU Z, TIAN X. Vectorization of matrix multiplication for multi-core vector processors [J]. Chinese Journal of Computers, 2018, 41(10): 2251-2264. (in Chinese)

[21] 王占立. 面向 GPDSP 科学计算的高性能 DMA 传输方式的设计与实现[D]. 长沙: 国防科技大学, 2015.
WANG Z L. Design and implementation of high-performance DMA transmission modes for scientific computation on GPDSP[D]. Changsha: National University of Defense Technology, 2015. (in Chinese)

[22] YANG C, CHEN S M, ZHANG J, et al. A novel DSP architecture for scientific computing and deep learning[J]. IEEE Access, 2019, 7: 36413-36425.