基于 ARMv8 平台的多维 FFT 实现与优化研究

陈 暾[1),2)] 李志豪[1),2)] 贾海鹏[1)] 张云泉[1)]

[1)](中国科学院计算技术研究所计算机体系结构国家重点实验室 北京 100190)

[2)](中国科学院大学 北京 100190)

摘 要 FFT(快速傅里叶变换)是用于计算离散傅里叶变换(DFT)或其逆运算的快速算法,它广泛应用于工程、科学和数学计算.到目前为止,鲜有基于 ARM 平台的高性能 FFT 算法的实现和优化,然而,随着 ARMv8 处理器应用的日益广泛,研究 FFT 算法在 ARM 平台上高性能实现日益重要.该文在 ARMv8 平台上实现和优化了一个高性能的多维 FFT 算法库:PerfFFT,通过 FFT 蝶形网络优化、蝶形计算优化、蝶形自动生成、SIMD 优化、内存对齐、cache-aware 的分块算法和高效转置等优化方法的应用,显著提升了 FFT 算法的性能.实验结果表明,PerfFFT 相比目前应用最为广泛的开源 FFT 库 FFTW 实现了 10%~591%的性能提升,而相比 ARM 高性能商业库 ARM Performance Library 实现了 13%~44%的性能提升.

关键词 ARMv8;FFT 算法;FFTW;ARMPL;SIMD 优化;Cache 优化;矩阵分块

中图法分类号 TP393 **DOI 号** 10.11897/SP.J.1016.2019.02384

Multi-Dimensional FFT Implementation and Optimization on ARMv8 Platform

CHEN Tun[1),2)] LI Zhi-Hao[1),2)] JIA Hai-Peng[1)] ZHANG Yun-Quan[1)]

[1)](*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)

[2)](*University of Chinese Academy of Sciences, Beijing* 100190)

Abstract With the development of ARM architecture, especially the introduction of ARMv8 architecture, ARM's application fields are more and more extensive. Research on ARM architecture has become a hotspot. Therefore, it is important to build a complete ARM software ecosystem. It is of great research significance and practical value to study the implementation and optimization of Fast Fourier Transform (FFT) algorithm in ARMv8 platform. Its computing ability has been greatly improved and application area has become more extensive. FFT is a fast algorithm for calculating Discrete Fourier Transform (DFT) or its inverse operation. It is widely used in engineering, science and mathematics. So far, there is a little implementation and optimization of high-performance FFT algorithm based on ARM platform. We implement and optimize a high-performance multi-dimensional FFT library on the ARMv8 platform which is PerfFFT. It is optimized by FFT butterfly network optimization, butterfly optimization, butterfly auto-generation, SIMD optimization, assembly optimization, memory alignment, cache-aware blocking algorithm,

收稿日期:2017-12-01;在线出版日期:2018-06-18.本课题得到国家重点研发计划(2017YFB0202105,2016YFB0200803,2017YFB0202302)、国家自然科学基金青年基金(61602443)、国家自然科学基金重点基金(61272136)、国家自然科学基金创新群体(61521092)、广东省重大科技专项项目(2015B010108006)资助.**陈 暾**,博士研究生,主要研究方向为高性能计算、并行编程.E-mail:chentun@ict.ac.cn.**李志豪**(通信作者),博士研究生,主要研究方向为高性能计算、异构计算.**贾海鹏**,博士,高级工程师,中国计算机学会(CCF)会员,主要研究方向为高性能计算、众核编程方法、面向众核平台的关键优化技术研究.**张云泉**,博士,研究员,中国计算机学会(CCF)杰出会员,主要研究领域为高性能计算及并行数值软件、并行计算模型.

efficient matrix transposition and other optimization methods. These approaches greatly enhance the FFT algorithm performance. The results of experiments show that PerfFFT achieves a 10% to 591%, and 13% to 44% performance improvement compared to ARM high-performance commercial library (ARM Performance Library). Our main contributions are as follows: First, we propose a set of FFT algorithm implementation and optimization on ARMv8 platform, which not only improves the performance of FFT algorithm on ARMv8 platform, but also has practical Guiding significance for implementation of other algorithms on ARM platform. Second, we propose a set of FFT butterfly calculation code automatic generation scheme. A computational template is formed by abstracting and extracting typical computational patterns of butterfly calculations for different radix of the FFT. And on this basis, it can automatically generate high performance code of different radix FFT butterfly calculation. By analyzing the FFT's various radix butterfly calculation methods, we abstract the core computational model into a computational template library, thus realizing the automatic generation of FFT butterfly calculation. Specific steps are as follows: (1) Building an atomic calculation template library. (2) Building a hybrid computing template library. (3) The butterfly calculation of the radix-N is automa-tically generated. (4) SIMD optimization: Based on the calculation template library obtained above, a mapping of the calculation template to the SIMD optimized template library can be constructed. Specifically, a set of parameterized code templates are used to obtain a SIMD instruction sequence that matches the c code, and the high performance parameterized code template is combined to obtain a butterfly computing kernel of different radix. Finally, we implement a high-performance multi-dimensional FFT algorithm library: PerfFFT. Compared with FFTW and ARM Performance Library, PerfFFT is the highest performance multi-dimensional FFT library on ARMv8 platform.

Keywords　ARMv8; FFT algorithm; FFTW; ARMPL; SIMD optimization; Cache optimization; matrix block

1　引　言

FFT(Fast Fourier Transform)是用于计算离散傅立叶变换(Discrete Fourier Transform,DFT)或其逆运算的快速算法,是 IEEE 科学与工程计算期刊评选的 20 世纪十大算法之一,在工程、科学和数学领域的应用非常广泛[1]. 如在国际大科学工程——平方公里阵列射电望远镜(Square Kilometer Array,SKA)项目中,FFT 是数据处理的五大算法之一,其计算量占总计算量的 40%. 同时,随着 ARM 架构的发展,特别是 ARMv8 架构的推出,ARM 的应用领域越来越广泛,基于 ARM 架构的服务器的研究已经成为研究热点. 因此,构建完善的 ARM 软件生态越来越重要,研究 FFT 算法在 ARMv8 平台的实现与优化具有重要的研究意义和实用价值.

然而,截至目前鲜有 FFT 算法在 ARM 平台上的实现和优化工作. 为此,本文在 ARMv8 计算平台上实现和优化了 radix-2/3/4/5/7/8/9/11 的一维 FFT 算法,并在此基础上,通过 cache 感知的分块算法的应用,实现了一个高性能的多维 FFT 算法库:PerfFFT. 通过 FFT 蝶形网络优化、蝶形计算优化、蝶形自动生成、SIMD 优化、内存对齐、cache-aware 的分块算法和高效转置算法,本文实现的 FFT 算法库达到了非常高的性能. 实现结果表明:在 ARM Cortex A57 计算平台上,相对于 FFTW3.3.6(目前应用最为广泛的开源 FFT 库),本文实现的 FFT 算法库达到了 10%～591%的性能提升;相对于 ARM Performance Library①(ARM 高性能商业库),本文实现的 FFT 算法达到了 13%～44%的性能提升.

本文的主要贡献如下:

(1)本文提出了一整套 FFT 算法在 ARMv8 平

① ARM Performance Library. https://developer.arm.com/docs/101004/latest/5-fast-fourier-transforms-ffts

台上实现和优化的方案，不仅提升了 FFT 算法在 ARMv8 平台上的性能，而且对其它算法在 ARM 平台上的实现和优化都具有实际的指导意义.

(2)提出了一套 FFT 蝶形计算代码自动生成方案.通过对 FFT 不同基的蝶形计算的典型计算模式的抽象和提取，形成计算模板.并在此基础上，自动生成 FFT 不同基的蝶形计算高性能代码.

(3)本文实现了一个高性能的多维 FFT 算法库:PerfFFT.通过与 FFTW 和 ARM Performance Library 的性能对比表明 PerfFFT 是目前 ARM 平台上性能最高的多维 FFT 库.

本文第 1 节介绍相关工作;第 2 节介绍本文的背景知识;在第 3 节介绍 FFT 算法在 ARMv8 平台上的实现;第 4 节详细讨论 FFT 算法在 ARMv8 平台上的优化方法;第 5 节通过实验验证优化方法的效果，并结合优化方法做简要分析;最后是总结与下一步工作展望.

2 背景介绍

2.1 ARMv8 架构

ARMv8 是首款支持 64 位指令集的 ARM 处理器架构，包含了 32 bit 与 64 bit 的执行状态.ARMv8 在引入了 64 bit 寄存器执行能力的基础之上，向后兼容 ARMv7 架构的机制.

ARMv8 除提供了 31×64 bit 通用寄存器外，还提供了 32 个 128 bit 浮点寄存器(V0～V31)，如图 1 所示，其中标量指令浮点操作数存储在浮点寄存器的低地址，如 Q0、D0、S0、H0、B0 分别在浮点寄存器 V0 中占低 128 bit、64 bit、32 bit、16 bit、8 bit.向量指令浮点操作可以处理多个浮点操作数，如 V0.2D、V0.4S、V0.8H、V0.16B 分别处理浮点寄存器 V0 中的 2 个 64 bit、4 个 32 bit、8 个 16 bit、16 个 8 bit 浮点操作数.浮点寄存器(V0～V31)用于处理标量浮点指令操作数和所有用于 NEON 操作的向量操作数.在执行指令的过程中，一条指令就能处理多个操作数，由此可以提高指令执行效率，提升性能.在 SIMD 优化过程中，浮点寄存器扮演着十分关键的角色.

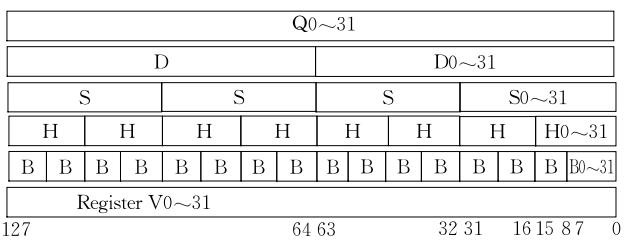


图 1 ARMv8 架构浮点寄存器图

2.2 FFT 算法介绍

FFT 算法的公式早在 1805 年就已被推导出来，在 1965 年 FFT 的基本思想得到了普及.美国著名数学家威廉·吉尔伯特·斯特朗在 1994 年把 FFT 描述为“我们一生中最重要的数值算法”，并被评为“20 世纪的十大算法”之一.FFT 是离散傅立叶变换(Discrete Fourier Transform，DFT) 的快速算法，而 DFT 是将信号时域上的采样变换到其频域上的采样.对于长度为 N 的复数序列 x，其中 $x=x_0,\cdots,x_{n-1}$，离散傅立叶变换(DFT)的计算公式为

$$X_k=\sum_{n=0}^{N-1}x_n\,\mathrm{e}^{-j2\pi k\frac{n}{N}},\ k=0,\cdots,N-1 \qquad (1)$$

库利-图基(Cooley-Tukey) FFT 算法是目前应用最广泛、最流行的 FFT 算法.利用式(1)给出的公式，将公式中的项在时域上进行重新分组，并将 $\mathrm{e}^{-j2\pi k\frac{n}{N}}$ 用 W_N^{nk} 进行替换，其中，替换后的 W_N^{nk} 被称之为“旋转因子”(twiddle factor)，亦称为“蝶形因子”.根据旋转因子在计算过程中出现的不同的位置，可以将 FFT 算法分为时域抽取(Decimation-In-Time，DIT)和频域抽取(Decimation-In-Frequency，DIF)两大类.时域抽取(DIT)的旋转因子出现在计算的输入端，而频域抽取(DIF)的旋转因子则出现在计算的输出端.且如果采用时域抽取(DIT)，数据输入是按照“位元翻转”(bit-reversed order)来进行排列，数据输出则是会依序排列;而如果采用频域抽取(DIF)，那么情况恰好相反，数据输入是依序排列，数据输出则是会按照“位元翻转”(bit-reversed order)来进行排列.如图 2 所示，以 radix-2 FFT 算法为例，介绍了库利-图基 FFT 算法的特性.从图中可以看出，蝶形网络可以抽象出三层:stage-section-butterfly，例如第一个 stage 中包含 4 个 sections，每一个 section 中包含一个蝶形;第二个 stage 中包含 2 个 sections，每个 section 包含两个蝶形.需要注意的是，如无特殊说明，本文后面的 FFT 算法是指库利-图基(Cooley-Tukey) FFT 算法.

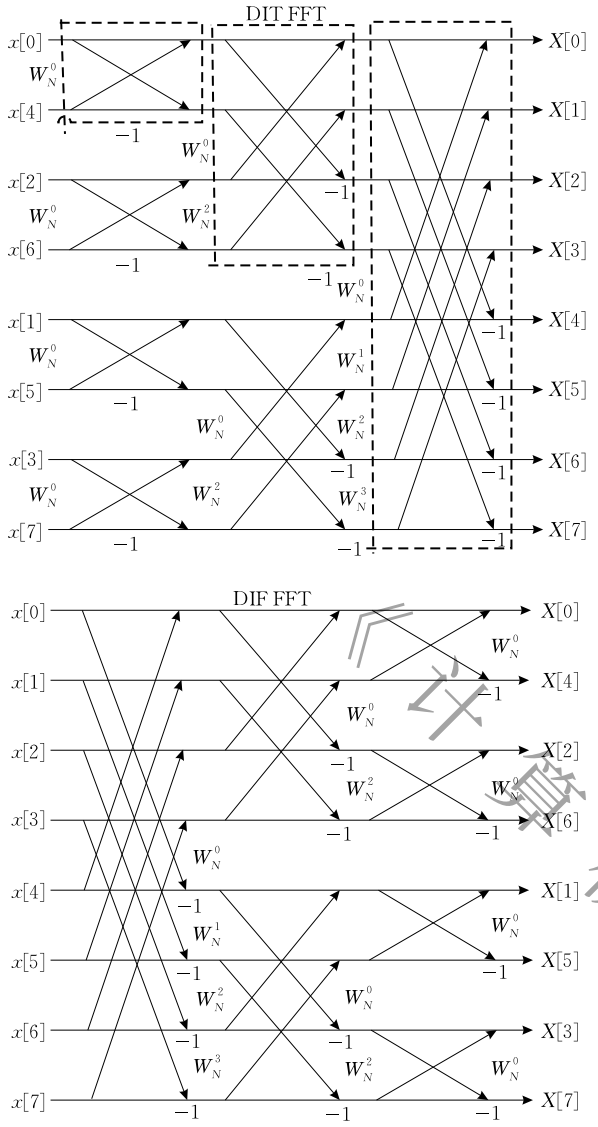


图 2　radix-2 FFT 时域抽取与频域抽取图[2]

3　相关工作

目前常用的 FFT 库有 FFTW[3]、PFFT[4]、MPFFT[5]、CUFFT①、PKUFFT[6] 等. 除 FFTW 外，算法库并没有针对 ARM 平台的实现和优化. FFTW(Fast Fourier Transform in the West)是 MIT 的 Frigo 和 Johnson 开发的自适应优化 FFT 软件包，同时支持共享存储多线程并行和 MPI 并行，其运算性能远远领先目前已有的其他 FFT 软件. FFTW 选择了过去 40 年的各种好的 FFT 算法，包括 Cooley-Tukey 算法、Prime-Factor 算法、Rader 算法、Split-radix 算法等[7-8].

(1) Split-radix FFT 算法

分裂基算法(Split-radix FFT Algorithm，SRA)由 Duhamel 和 Hollman 于 1984 提出. 分裂基算法是目前众多 FFT 算法中乘法和加法次数最少的算法，而且它具有良好的运算结构，以及较短的运算程序[9].

由此它被认为是一种实用并且高效的 FFT 算法. 分裂基算法的乘法运算复杂度接近理论上的最小值[10]. 基本思路就是对偶序列使用基-2FFT 算法，对奇序列使用基-4FFT 算法，可将 DFT 式(1)进行分裂基 FFT 变换分解为如式(2)所示.

$$X\left(k+\frac{N}{4}\right)=X_1\left(k+\frac{N}{4}\right)-jW_N^kX_2(k)+jW_N^{3k}X_3(k),$$

$$X\left(k+\frac{N}{2}\right)=X_1(k)-W_N^kX_2(k)-W_N^{3k}X_3(k),$$

$$X\left(k+\frac{3N}{4}\right)=X_1\left(k+\frac{N}{4}\right)+jW_N^kX_2(k)-jW_N^{3k}X_3(k) \tag{2}$$

以输入规模 N 为 16 的分裂基蝶形为例，其蝶形图如图 3 所示.

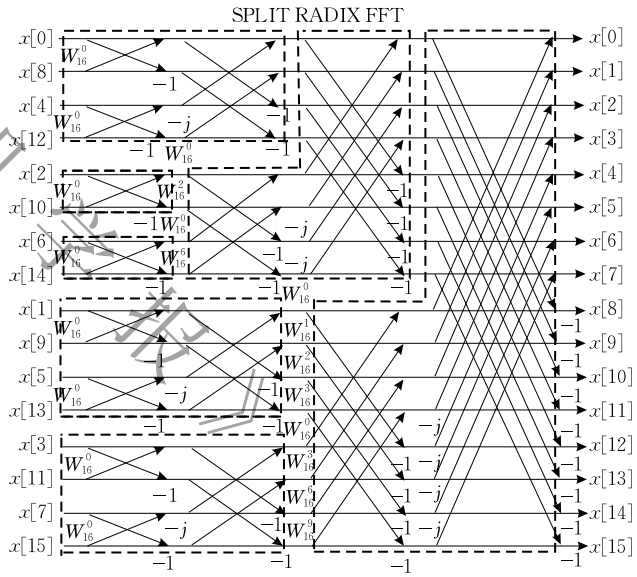


图 3　输入 N 为 16 的 Split-radix FFT 蝶形图

(2) Prime-Factor 算法

互质因子算法(Prime Factor Algorithm，PFA)最早由 Good 和 Thomas 提出. 它是把输入规模为 N 的离散傅里叶变换(DFT)分解为 $N_1\times N_2$ 大小的二维 DFT，其中 N_1 与 N_2 互质. 变成大小为 N_1 和 N_2 的 DFT 之后，可以继续递归使用 PFA，或选择其他 FFT 算法来计算[11]. 假设 $N=N_1\times N_2$，N_1 与 N_2 互质，然后把输入 n 和输出 k 对应到式：

$$n=(n_1\times N_2+n_2\times N_1)\bmod N,$$

① Nvidia CUFFT library. https://docs.nvidia.com/cuda/cufft

$$n_1\in 0\sim N_1-1,\ n_2\in 0\sim N_2-1,$$
$$k=(k_1\times N_2^{-1}\times N_2+k_2\times N_1^{-1}\times N_1)\bmod N,$$
$$k_1\in 0\sim N_1-1,\ k_2\in 0\sim N_2-1,$$

其中,N_1^{-1}表示 N_1在模 N_2下的反元素,即 N_1^{-1}为满足 $(N_1^{-1}\times N_1)\bmod N_2=1$ 的最小自然数. 对 DFT 式(1)进行 PFA 分解得到式(3):

$$e^{-\frac{2\pi i}{N}nk}=e^{-\frac{2\pi i}{N}(n_1\times N_2+n_2\times N_1)k}=e^{-\frac{2\pi i}{N_1}n_1k_1}e^{-\frac{2\pi i}{N_2}n_2k_2}.$$

即
$$X_{(k_1,k_2)}=\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}x_{(n_1,n_2)}W_{N_1}^{k_1n_1}W_{N_2}^{k_2n_2}\tag{3}$$

互质因子算法将一维的 DFT 问题转变为多维问题来进行计算. 相对于 Cooley-Tukey FFT 算法来说,Prime-Factor 算法虽然在浮点计算量上有所减少,但是因为其复杂的映射关系,这将导致更多的数据存取,因此该算法适用于数据存取开销小于浮点计算开销的平台架构.

(3) Rader's FFT 算法

Rader 算法是针对输入规模为质数 DFT 的快速算法,该算法的精髓在于将规模为 n 的 DFT 进行长度为 $n-1$ 的循环卷积来表示[12]. 假设 X_k 为输入规模为 N 的 DFT 如式(1),g^{-k}是 g^k的反元素,g^{-k}为满足$(g^{-k}\times g^k)\bmod N=1$ 的最小自然数,则 X_k可分解为式:

$$X_0=\sum_{n=0}^{N-1}x_n,$$
$$X_{g^{-k}}=x_0+\sum_{n=0}^{N-2}x_{g^n}e^{-\frac{2\pi i}{N}g^{-(k-n)}},\ k\in 0\sim N-2\tag{4}$$

式(4)可归纳为长度为 $N-1$ 的两个序列 a_n 和 $b_n(n=0,\cdots,N-2)$的循环卷积,其中 a_n和 b_n定义为 $a_n=x_{g^n}$、$b_n=e^{-\frac{2\pi i}{N}g^n}$,根据卷积定理得到式(5):

$$X_{g^{-k}}-x_0=DFT^{-1}(DFTa_n\times DFTb_n)\tag{5}$$

FFTW 从 FFTW 3.3.1 版本开始进行针对 ARM 平台的实现和优化,并实现了非常高的性能. FFTW 将作为本文研究工作的性能比较对象. 除 FFTW 外,ARM 公司也推出了针对 ARMv8 平台的高性能商业库:ARM Performance Library,里面也包含了高性能的 FFT 实现. 这个库也将作为本文研究工作的性能比较对象.

类似于本文提出的蝶形计算代码自动生成模板,基于模板的代码自动生成优化框架 AUGEM 在 BLAS(Basic Linear Algebra Subprograms,基础线性代数程序集)中得到很好的应用,它可以在多核 CPU 上为几种密集线性代数 DLA kernel(如 GEMM,GEMV,AXPY 和 DOT)自动生成完全优化的汇编代码. 以 DLA 内核的简单 C 实现为输入,它通过四个组件(优化的 C 内核生成器,模板识别器,模板优化器和汇编内核生成器)自动为输入代码生成高效的汇编 kernel[13]. AUGEM 构造自动生成的 GEMM 内核的平均性能在 Intel Sandy Bridge 处理器上相对于 Intel MKL,ATLAS 和 GotoBLAS 分别提高 1.4%,3.3%和 89.5%.

4 FFT 在 ARMv8 平台的实现

通常情况下,库利-图基(Cooley-Tukey) FFT 算法的实现涉及两步关键计算:旋转因子生成与蝶形计算. 以基 4 时域抽取(DIT)Cooley-Tukey FFT 算法为例. 当序列长度为 $N=4^n$,对 DFT 式(1)进行时域抽如图 4 所示.

$$\underbrace{\begin{bmatrix}X_k\\X_{k+N/4}\\X_{k+N/2}\\X_{k+3N/4}\end{bmatrix}}_{\text{输出数据}}=\underbrace{\begin{bmatrix}1&1&1&1\\1&-j&-1&j\\1&-1&1&-1\\1&j&-1&-j\end{bmatrix}}_{\text{蝶形计算公式}}\underbrace{\begin{bmatrix}W_N^0A_k\\W_N^kB_k\\W_N^{2k}C_k\\W_N^{3k}D_k\end{bmatrix}}_{\text{旋转因子 输入数据}}$$

图 4 基 4 时域抽取 FFT 公式

图中 $A_k=\sum_{n=0}^{N/4-1}x_{4n}W_N^{4nk}$、$B_k=\sum_{n=0}^{N/4-1}x_{4n+1}W_N^{4nk}$、$C_k=\sum_{n=0}^{N/4-1}x_{4n+2}W_N^{4nk}$、$D_k=\sum_{n=0}^{N/4-1}x_{4n+3}W_N^{4nk}$为每一级(stage)的输入数据.

由此可知 FFT 变换的输入数据首先需要乘以旋转因子,然后再进行对应 radix 的蝶形计算. 最后得到输出结果. 相对于每个基 radix 都有特有的旋转因子生成与蝶形计算公式. 本节将详细介绍这两部关键计算步骤.

4.1 一维 FFT 的实现

(1) 旋转因子(twiddles)的生成

由图 4 可知,FFT 变换中首先对输入数据乘以旋转因子 W_N^p,p 称为旋转因子的指数. 在 $N(N=radix^M)$点 DIT-FT 运算中,每级都有 $N/radix$ 个蝶形,这些蝶形分成 $fstride$ 份,每份平均分配 $mstride$ 个蝶形. 旋转因子的生成如图 5 所示,其中 $W_N^p=W_N^{i\times j\times fstride}$,$0\le i\le mstride-1$,$1\le j\le radix-1$.

```
N：输入数据规模.
radix：FFT 变换中蝶形计算的基.
M：FFT 变换的总级数，即 N=radix^M.
fstride：fstride=radix^(M-L)（FFT 变换第 L 级蝶形计算的 section 数目）.
mstride：mstride=radix^(L-1)（FFT 变换第 L 级每一个 section 中所包含的蝶形数目）.
twiddles：旋转因子.
1. For i from 0 to mstride-1
2.    For j from 1 to radix-1
3.       Phase←-2×π×fstride×j×i/N
4.       twiddles[mstride×(j-1)+i].r←cos(phase)
5.       twiddles[mstride×(j-1)+i].i←sin(phase)
6.    End for
7. End for
```

图 5　旋转因子生成伪代码

(2)蝶形计算

根据傅里叶变换式(1)可知，傅里叶变换的实质就是 DFT 矩阵与输入向量进行矩阵向量乘，可得其矩阵形式的表示式为 $y=DFT_n x$，其中 DFT_n 如式(6)所示.

$$DFT_n=\begin{bmatrix}1 & 1 & 1 & \cdots & 1\\ 1 & W_n^1 & W_n^2 & \cdots & W_n^{n-1}\\ 1 & W_n^2 & W_n^4 & \cdots & W_n^{2(n-1)}\\ \vdots & \vdots & \vdots & & \vdots\\ 1 & W_n^{n-1} & W_n^{2(n-1)} & \cdots & W_n^{(n-1)(n-1)}\end{bmatrix} \tag{6}$$

由 DFT_n 可表示基 $N(radix\text{-}N)$ 的蝶形计算公式(7)，此公式是蝶形计算基本公式，关于蝶形计算的优化将在 4.1 节中提到.

$$\begin{aligned}X(0)&=x_0+x_1+x_2+\cdots+x_{N-1}\\ X(1)&=x_0+W_N^1x_1+W_N^2x_2+\cdots+W_N^{n-1}x_{N-1}\\ X(2)&=x_0+W_N^2x_1+W_N^4x_2+\cdots+W_N^{2(n-1)}x_{N-1}\\ &\vdots\\ X(N-1)&=x_0+W_N^{N-1}x_1+W_N^{2(N-1)}x_2+\cdots+\\ &\quad W_N^{(N-1)(N-1)}x_{N-1}\end{aligned} \tag{7}$$

(3)实数 FFT 的 split 操作

在进行 FFT 的实数变换时，任何实数都可以看成虚部为零的复数. 以一维 R2C(Real to Complex) FFT 变换为例，输入数据为实数序列 x_n，可以将其认为是复数序列(x_n+j0)，再进行 FFT 变换. 但是这种做法是不可取的，一方面，把实数序列变成复数序列，存储器要增加一倍，这会需要更多的存储容量以及访存开销. 另一方面，在计算运行时，即使虚部为零，也要涉及虚部的运算，增加了计算开销.

一种合理的解决方法是利用复数 FFT 对实数 FFT 进行有效计算. 对长度为 N 的实数输入序列进行 R2C FFT 变换，首先采用长度为 $N/2$ 的复数 FFT 算法对输入序列进行处理，然后再进行 split 操作，得到实数 R2C FFT 变换后的复数序列. 其中 R2C 的 split 操作原理如下，若要计算实数序列 x_n 的离散傅里叶变换(1)：

$$\begin{aligned}X_k&=\sum_{n=0}^{N-1}(x_n+j0)W_N^{nk},\ k=0,1,\cdots,N-1\\ &=\sum_{n=0}^{N/2-1}x_{2n}W_N^{2nk}+W_N^k\sum_{n=0}^{N/2-1}x_{2n+1}W_N^{2nk}\\ &=\sum_{n=0}^{N/2-1}x_{2n}W_{N/2}^{nk}+W_N^k\sum_{n=0}^{N/2-1}x_{2n+1}W_{N/2}^{nk}\\ &=F_k+W_N^kG_k,\\ F_k&=\sum_{n=0}^{N/2-1}x_{2n}W_{N/2}^{nk},\ G_k=\sum_{n=0}^{N/2-1}x_{2n+1}W_{N/2}^{nk}\end{aligned} \tag{8}$$

只需计算出 F_k、G_k 代入到式(8).

而 F_k、G_k 的计算流程如下：

首先设 $y_n=x_{2n}+jx_{2n+1}$，$n=0,1,\cdots,\frac{N}{2}-1$，

对复数序列 y_n 做输入规模为 $N/2$ 的离散傅里叶变换，则有公式：

$$\begin{aligned}Y_k&=\sum_{n=0}^{N/2-1}y_nW_{N/2}^{nk}\\ &=\sum_{n=0}^{N/2-1}(x_{2n}+jx_{2n+1})W_{N/2}^{nk}\\ &=\sum_{n=0}^{N/2-1}x_{2n}W_{N/2}^{nk}+j\sum_{n=0}^{N/2-1}x_{2n+1}W_{N/2}^{nk}\\ &=F_k+jG_k\end{aligned} \tag{9}$$

因此 $F_k=\frac{1}{2}(Y_k+\overline{Y}_{\frac{N}{2}-k})$，$G_k=\frac{j}{2}(\overline{Y}_{\frac{N}{2}-k}-Y_k)$ (10)

综合式(8)、(10)即可求得实数序列 x_n 的离散傅里叶变换. 首先使用复数 FFT 算法计算出 Y_k，然后通过 split 操作式(10)由 Y_k 计算出 F_k 和 G_k，进而得到最终结果 X_k. 同时可得到

$$X_0.r=Y_0.r+Y_0.i,\ X_0.i=0,$$
$$X_{N/2}.r=Y_0.r-Y_0.i,\ X_{N/2}.i=0.$$

又因为除 X_0、$X_{N/2}$ 以外，X_k 与 $X_{k+N/2}$ 互为共轭，因此省去共轭部分数据，R2C FFT 的输出数据规模为 $N/2+1$[14].

4.2　二维 FFT 的实现

有限域 $0\le n_1\le N_1-1$，$0\le n_2\le N_2-1$ 上的二维序列 $f[n_1,n_2]$ 的离散傅里叶变换定义如下：

$$F[k_1,k_2]=\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}f[n_1,n_2]W_{N_2}^{k_2n_2}W_{N_1}^{k_1n_1},$$
$$0\le k_1\le N_1-1,\ 0\le k_2\le N_2-1 \tag{11}$$

对于离散傅里叶变换，它的内层叠加如式：

$$\sum_{n_2=0}^{N_2-1}f[n_1,n_2]W_{N_2}^{k_2n_2},\ 0\le n_1\le N_1-1 \tag{12}$$

是以 n_1 为参变量的一维 DFT. 若以 $F[n_1,k_2]$ 表示此叠加结果,则外层叠加如式:

$$F[k_1,k_2]=\sum_{n_1=0}^{N_1-1}F[n_1,n_2]W_{N_1}^{k_1n_1},\ 0\leq k_2\leq N_2-1\ (13)$$

又是以 k_2 为参变量的一维 DFT. 做 N_2 点的一维 FFT N_1 次,即可算出式(12);再做 N_1 点的一维 FFT N_2 次,即可算出式(13),从而完成 $N_1\times N_2$ 点序列的二维 DFT 计算.

总之,二维 FFT 的计算,就是在二维输入数据上先以行为维度,进行一维 FFT,然后再以列为维度,进行一维 FFT 计算. 其运算流程如图 6(a)所示.

根据二维 FFT 的定义和计算规则,本文不仅实现了复数 FFT 变换,而且实现了实数 FFT 变换,具体如下:

(1)二维 C2C(Complex to Complex) FFT 变换

由离散傅里叶变换式(11)可知,二维 C2C FFT 变换首先需要对数据行进行 C2C 一维 FFT 变换,然后对数据列进行 C2C 一维 FFT 变换;其中输入为 $n_1\times n_2$ 个复数,得到输出为 $n_1\times n_2$ 个复数. 运算流程如图 6(b)所示.

(2)二维 R2C(Real to Complex) FFT 变换

二维 R2C FFT 变换与之类似,先对数据行进行 R2C 一维 FFT 变换,然后对数据列进行 C2C 一维 FFT 变换,其中输入为 $n_1\times n_2$ 个实数,由于行向量在经过一维 R2C FFT 变换之后其输出数组位置在 $1\sim n_2/2$ 与 $n_2/2+2\sim n_2-1$ 的数据互为共轭复数,因此省去数组后 $n_2/2-2$ 个复数操作数,得到的输出为 $n_1\times(n_2/2+1)$ 个复数. 运算流程如图 6(c)所示.

(3)二维 C2R(Complex to Real) FFT 变换

二维 C2R FFT 变换为二维 R2C FFT 变换的逆变换,经过 R2C FFT 计算后的数据,可以通过 C2R FFT 变换返回其原始数据. 其运算过程为:先对输入数据的列做一维 C2C FFT 变换,再对行做一维 C2R FFT 变换,输入为 $n_1\times(n_2/2+1)$ 个复数,得到的输出为 $n_1\times n_2$ 个实数. 运算流程如图 6(d)所示.

(4)二维 R2R(Real to Real) FFT 变换

二维 R2R FFT 变换,先做行上一维 R2R FFT 变换,再做列上一维 R2R FFT 变换,由于一维 R2R 有正逆两种情况分别为 R2HC、HC2R 表示. 因此二维 R2R 排列后为四种情况,行列分别为:R2HC R2HC、R2HC HC2R、HC2R R2HC、HC2R HC2R. 其输入为 $n_1\times n_2$ 个实数,输出为 $n_1\times n_2$ 个实数. 运算流程如图 6(e)所示.

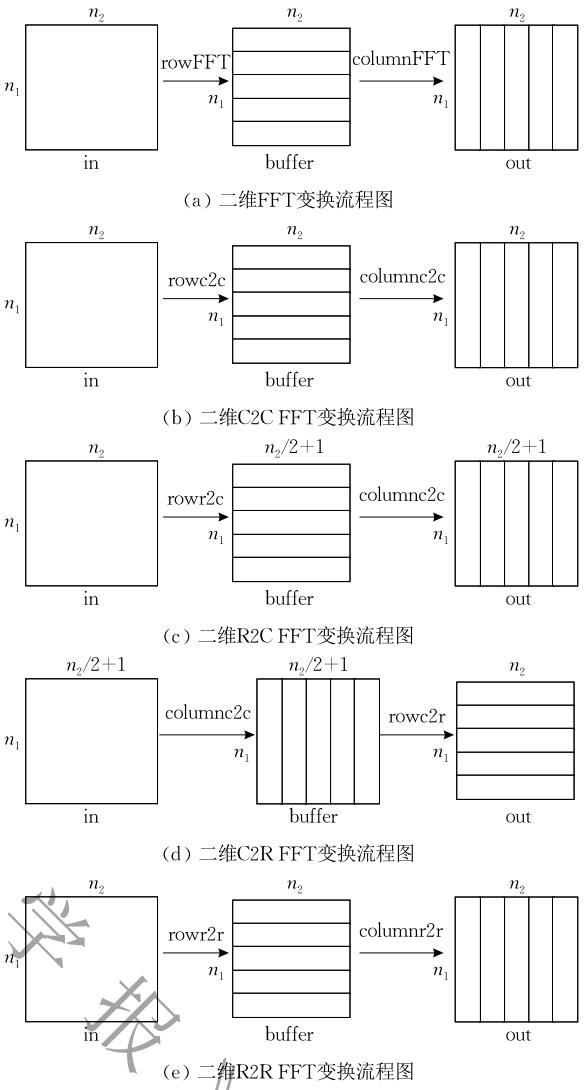


图 6 二维 FFT 变换流程图

4.3 三维 FFT 的实现

有限域 $0\leq n_1\leq N_1-1, 0\leq n_2\leq N_2-1, 0\leq n_3\leq N_3-1$ 上的三维序列 $f[n_1,n_2,n_3]$ 的离散傅里叶变换定义:

$$F[k_1,k_2,k_3]=\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}\sum_{n_3=0}^{N_3-1}f[n_1,n_2,n_3]W_{N_3}^{k_3n_3}W_{N_2}^{k_2n_2}W_{N_1}^{k_1n_1},$$

$$0\leq k_1\leq N_1-1,\ 0\leq k_2\leq N_2-1,\ 0\leq k_3\leq N_3-1\ (14)$$

三维 FFT 其基本思想类似于二维 FFT,它是以一维 FFT 和二维 FFT 为基础进行实现的. 可以概括为如下两个步骤,以 C2C FFT 变换为例具体实现如图 7 所示.

(1)针对 Z 个大小为 $X\times Y$ 的面进行二维 FFT 变换;

(2)在 Z 方向对 $X\times Y$ 个长度为 Z 的数据序列进行一维 FFT 变换.

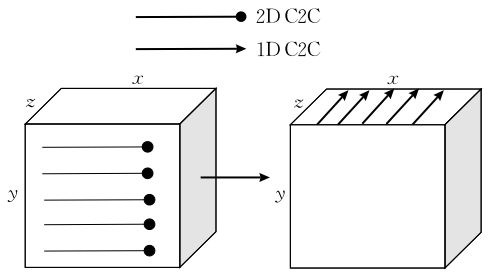


图 7　三维 C2C FFT 变换流程图

5　FFT 在 ARMv8 平台上的优化

5.1　一维 FFT 的优化

由上节讨论可知，多维 FFT 计算实质上是调用一维 FFT 计算. 因此，提升一维 FFT 的计算性能对于多维 FFT 计算性能来说至关重要.

5.1.1　蝶形网络优化

(1) 去除“位反转”的蝶形网络

上文所提到的时域抽取蝶形网络如图，在数据输入端要对输入序列进行位反转操作，这样一方面增加了一次内存存取开销，另一方面不利于混合基计算框架的搭建. 为了更好的把不同的基揉合进同一个高性能蝶形网络框架，本文引入两个步长控制器 *fstride* 和 *mstride*，*fstride* 用以指示蝶形网络中每一个层(stage)包含的 section 数目，而 *mstride* 则指示每一个 section 所包含的蝶形数目，进而统一蝶形网络的结构，使传统的时域抽取蝶形网络得到顺序输入，时域抽取蝶形网络如图 8 所示.

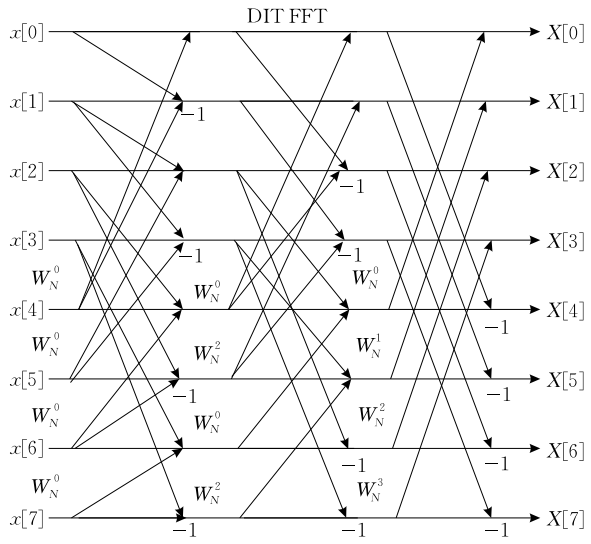


图 8　顺序输入时域抽取 FFT 蝶形图

① 去除位翻转操作，一方面减少内存访问开销，另一方面也增加网络框架的可扩展性.

② *fstride* 控制器可控制不同的基在每一个 stage 中包含的 section 数目，该控制器相较于“位反转”预操作，能更好地规范化蝶形网络，方便于统一蝶形网络以及访存优化.

③ *mstride* 用以指示每一个 section 里所包含的蝶形数目，方便进行 SIMD 指令优化上层控制.

因此，通过用控制器描述 FFT 蝶形网络的三级结构(stage-section-蝶形)，能够统一蝶形网络输入输出的访存行为，进而统一底层的优化方法.

(2) 第一级 FFT 计算单独优化

由图 2 FFT 变换算法可知，在进行一维 FFT 计算时，第一级蝶形计算的旋转因子为 $W_N^0=1$，使用该旋转因子做乘法运算时，并不改变运算结果. 因此，第一级 FFT 计算不需要乘以旋转因子，这是第一级计算与其它级计算不同的地方. 鉴于此，在本文的 FFT 实现中，将第一级 FFT 计算同其它级的 FFT 计算分离开，单独进行计算和优化. 这样可带来三方面的好处：首先，降低了内存访问开销，第一级计算不需要读取旋转因子；其次，降低了计算开销，第一级计算减少了对旋转因子的乘法开销；最后，在进行 SIMD 优化时能够区别这两个阶段分别进行指令优化：第一级计算结果的写入是不连续的，需要用 zip 指令进行数据重组，而其它级计算在采用 SIMD 优化时，不需要用 zip 指令，直接对计算结果进行 store 操作即可.

(3) 小规模优化

当进行 FFT 变化的规模较小(如输入规模为 3、5、7 等)时，不需要按照大规模 FFT 计算的方式进行计算，可直接根据 FFT 算法的定义进行计算即可. 在计算过程中，可直接采用宏定义的方式预设旋转因子的值，从而减少旋转因子的计算以提升程序性能.

此外，针对只有两层 stage 的 FFT 输入序列而言，同样可以进行小规模优化操作，比如说长度为 9，25，49 等规模. 这种情况可以采用如下方式进行简化：

① 针对于第一个 stage，由于其旋转因子均为复数(1，0)，因此可以省略对旋转因子的加载和运算，减少访存操作和冗余计算；

② 针对于第二个 stage，使用宏定义方式具体定义出所需的旋转因子，而无需在内存中存取数据.

5.1.2　蝶形计算优化

程序的主要执行为蝶形计算函数，本文针对蝶形计算公式，进行分析与优化. 此处以 radix-5 为例

进行说明.利用旋转因子的周期性 $W_N^{k+N}=W_N^k$,和对称性 $W_N^{k+N/2}=-W_N^k$ 将蝶形计算式(7)简化为如下形式:

$$X(0)=x_0+x_1+x_2+x_3+x_4$$
$$X(1)=x_0+W_5^1x_1+W_5^2x_2+W_5^{-2}x_3+W_5^{-1}x_4$$
$$X(2)=x_0+W_5^2x_1+W_5^{-1}x_2+W_5^1x_3+W_5^{-2}x_4$$
$$X(3)=x_0+W_5^{-2}x_1+W_5^1x_2+W_5^{-1}x_3+W_5^2x_4$$
$$X(4)=x_0+W_5^{-1}x_1+W_5^{-2}x_2+W_5^2x_3+W_5^1x_4 \quad (15)$$

又因为 W_5^k 与 W_5^{-k},其中 $k=0,1,2$,在复数坐标系中的图像关于 x 轴对称,也就是说它们有相同的实部和互为相反数的虚部,根据旋转因子的这种特性,对式(15)进行提取并合并同类项化简,最终得到式(16)所示的蝶形计算简化公式:

$$X(0)=x_0+(x_1+x_4)+(x_2+x_3),$$
$$X(1)=x_0+A-B,$$
$$X(2)=x_0+C+D,$$
$$X(3)=x_0+C-D,$$
$$X(4)=x_0+A+B,$$
$$A=(x_1+x_4)\times W_5^1.r+(x_2+x_3)\times W_5^2.r,$$
$$B=[(x_1-x_4)\times W_5^1.i+(x_2-x_3)\times W_5^2.i]\times(-j),$$
$$C=(x_1+x_4)\times W_5^2.r+(x_2+x_3)\times W_5^1.r,$$
$$D=[(x_1-x_4)\times W_5^2.i-(x_2-x_3)\times W_5^1.i]\times j \quad (16)$$

由式(16)可知,radix-5 的蝶形计算 kernel 中首先计算相同项 x_1+x_4、x_1-x_4、x_2+x_3、x_2-x_3,其次计算相同项是 A、B、C、D,最后可得到 kernel 的输出结果.这里通过挖掘等式中共同项,使其初始化后供各公式重复利用,相对于简化前的蝶形计算 kernel,减少了大量的浮点计算开销以及代码量.

5.1.3 基于模板的蝶形自动生成

FFT 对于不同的基(如 radix-2,3,4,5,7,11,13,…)具有不同的蝶形,需要对每种蝶形进行特定实现,理论上,有多少个质数,就需要实现多少种不同的基,其工作量极大,不可能完全手工实现;因此,在尽量减少手工优化的情况下,实现高效、可移植性高的 FFT 蝶形计算自动生成代码尤为重要.

在这种情况下,本文通过分析 FFT 各种基的蝶形计算方式,将其中的核心计算模式抽象为计算模板库,从而实现了 FFT 蝶形计算的自动生成.具体步骤如下:

(1)构建原子计算模板库

在蝶形计算自动生成方法中,首先根据 FFT 的定义,最小化蝶形的计算复杂度,如 5.1.2 节所述;然后对 FFT 各种基的蝶形计算方式进行统一和规范化;最后从 FFT 蝶形计算中抽象出 5 个核心计算序列,即原子计算模板,并进一步组成原子计算模板库.通过这 5 个原子计算模板的组合,就可以生成各种 FFT 各种基的蝶形.这 5 个原子计算模板如图 9 所示.

```
PERF_ATOM_ODD_ADDSUB(CMPLE, SUM, ORI)
{
    SUM.r=SUM.r+SUM.r;
    SUM.i=SUM.i+SUM.i;
    CMPLE.r=SUM.r-ORI.r;
    CMPLE.i=SUM.i-ORI.i;
}
```

```
PERF_ATOM_ADDSUB(OUT1, OUT2, IN1, IN2)
{
   OUT1.r=IN1.r+IN2.r;
   OUT1.i=IN1.i+IN2.i;
   OUT2.r=IN1.r-IN2.r
   OUT2.i=IN1.i-IN2.i;
}
```

```
PERF_CPX_ADD_NEG_I(Z1, Z2, A, B, S)
{
   B.r=B.r*S;
   B.i=B.i*S;
   Z1.r=A.r+B.i;
   Z1.i=A.i-B.r;
   Z2.r=A.r-B.i;
   Z2.i=A.i+B.r;
}
```

```
PERF_FORMER_ODD_WITH_F(OUT, IN1, IN2, TWR, TWI,
TMP, F, SUM)
{
   SUM.r=F.r+TWR*IN1.r;
   SUM.i=F.i+TWR*IN1.i;
   TMP.r=TWI*IN2.i;
   TMP.i=TWI*IN2.r;
   OUT.r=SUM.r-TMP.r;
   OUT.i=SUM.i+TMP.i;
}
```

```
PERF_FORMER_ODD_TW(OUT, IN1, IN2, TWR, TWI, TMP1,
TMP2, SUM)
{
   TMP1.r=TWR*IN1.r;
   TMP1.i=TWR*IN1.i;
   TMP2.r=TWI*IN2.i;
   TMP2.i=TWI*IN2.r;
   OUT.r=OUT.r+TMP1.r-TMP2.r;
   OUT.i=OUT.i+TMP1.i+TMP2.i;
   SUM.r=SUM.r+TMP1.r;
   SUM.i=SUM.i+TMP1.i;
}
```

图 9 原子计算模板

(2)构建混合计算模板库

以原子计算模板库为基础,从中选择一个或者多个原子计算模板,组装所需的蝶形计算混合模板,即为基 N 的蝶形计算模板.其中 2 的幂的蝶形计算模版相对于非 2 的幂具有它的独特性,由于 2 的幂蝶形计算中的旋转因子 twiddles 具有很强的对称性,因此在计算中通过合并同类项化简可以省去因乘以相同 twiddles 值而造成的冗余计算操作,得到优化后的 2 的幂蝶形计算模版,该模版减少了乘

twiddles 中多余的计算开销，是改进后的高性能蝶形计算模板.

① 2 的幂蝶形计算模板

针对于输入规模为 2 的幂的 FFT 变换，其计算由 radix-2，4，8，…，2^M($M\in N^+$)的混合基构成. 本文以 radix-2、radix-4 蝶形计算为例生成蝶形计算模板，则分别需要 1 个和 2 个原子模板进行组装即可. 原子模板分别为 PERF_ATOM_ADDSUB() 和 PERF_CPX_ADD_NEG_I()，两者具体的组合计算序列如图 10 所示.

2的幂

```
R2_MIX_ATOM_TEMPLATE(...)//基2
{
  PERF_ATOM_ADDSUB(Fout[0], Fout[1], Fin[0], Fin[1]);
}
```

```
R4_MIX_ATOM_TEMPLATE(...)//基4
{
  PERF_ATOM_ADDSUB(scratch[0], scratch[1], Fin[0], Fin[2]);
  PERF_ATOM_ADDSUB(scratch[2], scratch[3], Fin[1], Fin[3]);
  PERF_ATOM_ADDSUB(Fout[0], Fout[2], scratch[1], scratch[3]);
  PERF_CPX_ADD_NEG_I(Fout[0], Fout[2], scratch[1], scratch[3],
     TW4_1I)
}
```

图 10　2 的幂蝶形计算模板

② 非 2 的幂蝶形计算模板

针对于输入规模为非 2 的幂的 FFT 变换，其计算由 radix-3，5，…，L^M($L\in P$，$M\in N^+$)的混合基构成. 由于存在无穷个质数，本文以 radix-n 的蝶形计算为例生成蝶形计算模板. 则其混合计算模板将包含 3 种原子计算模板，分别为

PERF_FORMER_ODD_WITH_F()、
PERF_FORMER_ODD_TW()和
PERF_ATOM_ODD_ADDSUB().

由此三个原子计算模板组装出长度为 $m=\lfloor n/2\rfloor$ 的计算序列，即为 radix-n 的蝶形计算模板. 该计算序列的组合顺序为

1 个 PERF_FORMER_ODD_WITH_F()、
$m-2$ 个 PERF_FORMER_ODD_TW()和
1 个 PERF_ATOM_ODD_ADDSUB().

以基 7 和基 11 为例，其蝶形计算模板如图 11.

(3) 基 N 的蝶形计算自动生成

基于上一步得到的蝶形计算模板，构建基 N 的蝶形如图 12 所示.

① 针对于 2 的幂的 FFT 序列长度，直接调用代码生成部分的 radix-2/4 的蝶形网络序列.

② 针对非 2 的幂，设当前需要为 radix-n 生成蝶形计算序列，其所需调用蝶形计算模板的次数为 $m=\lfloor n/2\rfloor$ 次. 将生成的蝶形计算模板 kernels 嵌入到蝶形网络中，生成对应 FFT 分解方式的最终 FFT 代码.

非2的幂

```
R7_MIX_ATOM_TEMPLATE(...)//基7
{
 PERF_FORMER_ODD_WITH_F(Fout[1], scratch[0],
   scratch[1], TW7_1R_F, TW7_1I, scratch[10], in_0, sum);
 PERF_FORMER_ODD_TW(Fout[1], scratch[2],
   scratch[3], TW7_2R_F, TW7_2I, scratch[10], scratch[11], sum);
 PERF_FORMER_ODD_TW(Fout[1], scratch[4],
   scratch[5], TW7_3R_F, TW7_3I, scratch[10], scratch[11], sum);
 PERF_ATOM_ODD_ADDSUB(Fout[6], sum, Fout[1]);
}
```

```
R11_MIX_ATOM_TEMPLATE(...)//基11
{
 PERF_FORMER_ODD_WITH_F(Fout[1], scratch[0],
   scratch[1], TW11_1R_F, TW11_1I, scratch[10], in_0, sum);
 PERF_FORMER_ODD_TW(Fout[1], scratch[2],
   scratch[3], TW11_2R_F, TW11_2I, scratch[10], scratch[11], sum);
 PERF_FORMER_ODD_TW(Fout[1], scratch[4],
   scratch[5], TW11_3R_F, TW11_3I, scratch[10], scratch[11], sum);
 PERF_FORMER_ODD_TW(Fout[1], scratch[6],
   scratch[7], TW11_4R_F, TW11_4I, scratch[10], scratch[11], sum);
 PERF_FORMER_ODD_TW(Fout[1], scratch[8],
   scratch[9], TW11_5R_F, TW11_5I, scratch[10], scratch[11], sum);
 PERF_ATOM_ODD_ADDSUB(Fout[10], sum, Fout[1]);
}
```

图 11　非 2 的幂蝶形计算模版

Radix-7蝶形序列

```
FFT_R7_KERNEL(...)
{
   loading source data...
   R7_MIX_ATOM_TEMPLATE(scratch_out[1], scratch_out[6],
scratch, TW_TABLE, in_0, sum);
   R7_MIX_ATOM_TEMPLATE(scratch_out[2], scratch_out[5],
scratch, TW_TABLE, in_0, sum);
   R7_MIX_ATOM_TEMPLATE(scratch_out[3], scratch_out[4],
scratch, TW_TABLE, in_0, sum);
   storing output data...
}
```

Radix-11蝶形序列

```
FFT_R11_KERNEL(...)
{
   loading source data...
   R11_MIX_ATOM_TEMPLATE(scratch_out[1], scratch_out[10],
scratch, TW_TABLE, in_0, sum);
   R11_MIX_ATOM_TEMPLATE(scratch_out[2], scratch_out[9],
scratch, TW_TABLE, in_0, sum);
   R11_MIX_ATOM_TEMPLATE(scratch_out[3], scratch_out[8],
scratch, TW_TABLE, in_0, sum);
   R11_MIX_ATOM_TEMPLATE(scratch_out[4], scratch_out[7],
scratch, TW_TABLE, in_0, sum);
   R11_MIX_ATOM_TEMPLATE(scratch_out[5], scratch_out[6],
scratch, TW_TABLE, in_0, sum);
   storing output data...
}
```

图 12　基 N 蝶形计算自动生成图

5.1.4　SIMD 优化

(1) 汇编优化

ARMv8 平台提供了 32 个 128 bit 的浮点寄存器，每个浮点寄存器能够存储 4 个 32 bit 的单精度

浮点数.因此,在指令执行期间,一条指令能够处理4个数据,提高了代码的并行处理能力.同时,FFT计算具有两方面的特征:① 每个蝶形计算的内存访问是不连续的,这就导致了访存效率极低;② 每个蝶形计算是相互独立的,可并行处理多个蝶形计算.这样情况下,采用SIMD优化,一次完成4个蝶形计算,可以带来较大的性能提升.这主要得益于两个方面:一方面SIMD指令可以一次处理多个数据,有效开发程序的并行性;另一方面可以充分提高cache line的利用率,减少cache miss.

使用汇编对核心计算进行优化,可更加方便地控制寄存器的使用和指令的排布,可大幅提升算法性能.特别是在第一级FFT的计算和SIMD优化中,虽然数据的读取是连续的,但是FFT计算结果的写入是非连续的,此时直接采用汇编指令如zip1等可方便有效地对数据进行重组.同时,使用faddp指令可有效地处理向量寄存器内部数据的相加以实现复数乘法以及fmla/fmls乘加乘减指令,更好地挖掘性能.

基于上文得到的计算模板库,可构建计算模板到SIMD优化模板库的映射如图13所示.具体而言,本文使用一组参数化代码模版得到与c代码相匹配的SIMD指令序列,组合该高性能参数化代码模版得到不同基的蝶形计算kernel.模版具体实施如下:

① 寄存器分组(见下节描述);

② 根据ARMv8硬件平台特性,选择性能最高的指令;

③ 优化指令流水,避免流水线stall.

(2) 寄存器使用优化

ARMv8共提供了32个128 bit的浮点寄存器,这些寄存器能否充分利用,直接关系到FFT程序性能的提升,特别是在大基的情况下.

寄存器的使用优化的主要思想是根据功能对寄存器进行分组,并严格定义各组寄存器的使用规则.在FFT的蝶形实现中,将32个寄存器分成了四组:输入寄存器组、twiddles寄存器组、中间计算结果寄存器组、输出寄存器组.每组寄存器的使用都有严格的规范:

① 在小基的情况下,如radix-3, radix-4, radix-5等,寄存器足够使用.那么输入寄存器组只负责存储输入、twiddles寄存器组只负责存储各级的twiddles、中间计算结果寄存器组只负责存中间计算结果,输出寄存器组只负责存储最终的FFT计算结果.

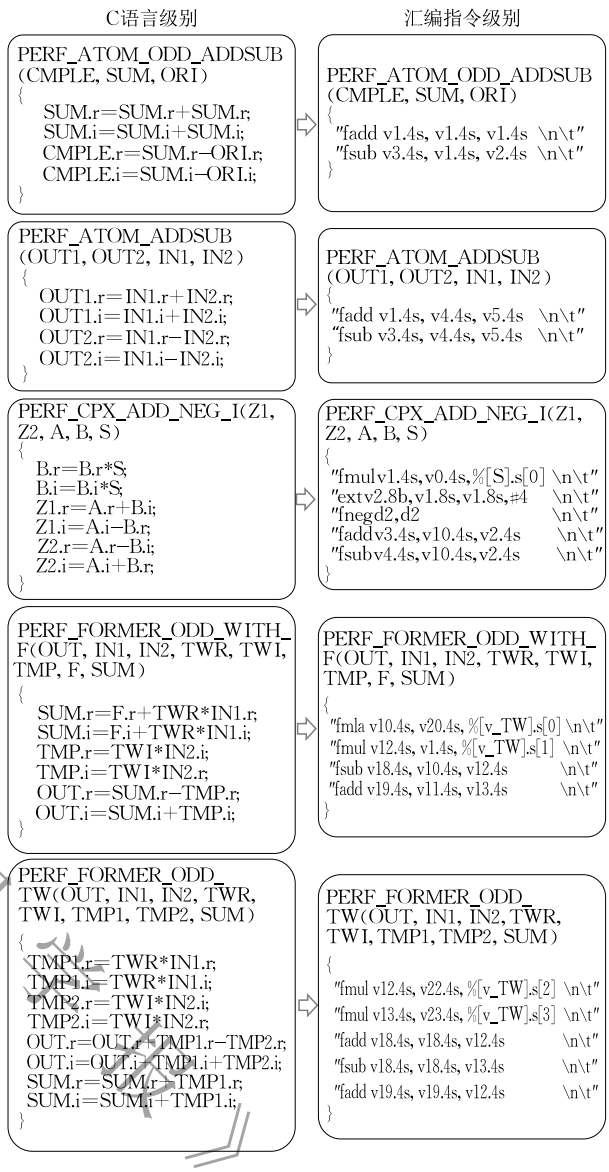


图13 计算模板SIMD优化

② 在大基的情况下,如radix-13,由于每个蝶形进行FFT变换的输入、输出和计算所使用的寄存器数量都增加,会出现寄存器不够使用的情况.此时需要寄存器复用:输入寄存器组与twiddles寄存器组在乘旋转因子运算后处于空闲状态,将输入寄存器组与twiddles寄存器组复用为中间结果寄存器组;同时该组寄存器的前后两次使用中间具有足够多的计算指令,能够最大程度上避免寄存器间的相互依赖.这样,在进行蝶形计算操作时寄存器得到最大化利用,最终避免了堆栈存取指令或内存存取指令的使用,减少了访存开销,提升了程序性能.

③ 在特别大基的情况下,如radix-19等,此时即使复用寄存器,寄存器依然不够用.则只能采用临时存入内存的方法.在这种情况下,要保证存入内存的数据以及相关寄存器和到下次使用,中间插入足

够多的计算指令.

通过寄存器的分组和使用优化，一方面消除了指令间的中间寄存器依赖，避免了流水线空泡；另一方面寄存器分组之后，可以抽象出优化模式并且统一优化方法，从而提升程序性能.

5.2　二维 FFT 的优化

5.2.1　二维 FFT 优化方法

在实现和优化完成一维 FFT 变换后，二维 FFT 的优化相对简单. 唯一需要面对的问题是：在对数据列进行 FFT 变换时，访存极不连续，这会造成性能的极大降低. 为此，二维 FFT 计算优化的核心是提升访存数据的连续性，本文使用的主要优化方法如下：

(1) Cache-aware 的分块方法

解决在对数据列进行 FFT 变换时，访存极不连续的初始方法就是：在对数据行进行 FFT 计算后，对计算结果矩阵进行转置，这样就把对数据列的 FFT 计算转换为对数据行的 FFT 计算，最后再进行一次转置操作，即可得到最终计算结果. 但这种方法有两次开销较大的全矩阵转换操作，而且对 cache 的利用率特别低，cache miss 非常高，降低了 FFT 计算的性能. 对此，本文提出了 cache-aware 的分块方法，其基本思想是，在对矩阵行进行完 FFT 计算后，对计算结果矩阵进行按列分块，保证该数据块始终位于 cache 中，然后对该数据块进行 FFT 计算，并将计算结果存放到结果矩阵中. 由于数据分块始终在 cache 中，可奖励 cache miss，从而提升计算性能. 具体为：首先，依据 L2 cache 的大小制定合适的分块策略，最大程度的重用 cache 中的数据. 在二维 FFT 计算过程中：首先，根据输入规模大小计算出分块列数 nb，将行计算结果矩阵分块为若干 $n_1 \times nb$ 大小的数据块，该数据块应始终位于 L2 cache；其次对该数据分块通过转置操作存储到中间 Buffer，并在该 Buffer 中进行 FFT 计算；最后将计算结果转置存储到最终结果矩阵中.

(2) 内存对齐

通过 cache-aware 分块方法的采用，二维 FFT 计算都转化为了对矩阵行的 FFT 计算，减少了内存访问的不连续程度，降低了 cache miss. 然而，在对矩阵行进行 FFT 变换时，由于输入矩阵规模的不确定，可能会造成矩阵从第二行开始出现内存不对齐的现象，这就会降低 cache line 利用率，从而降低程序性能. 同时，对于不对齐的向量化访存性能也会下降. 为此，本文在二维 FFT 的实现过程中，通过对中间 Buffer 每行加 Padding 的方式，实现了中间 Buffer 每行数据的对齐. 考虑到 ARM cache line 的大小为 64 Byte，因此我们将中间 Buffer 每行的数据都对齐到 64 个 Byte. 这样在进行 FFT 计算的 SIMD 优化时，每次向量化读取都是一个完整的 cache line，从而降低了访存开销. 通过在中间 Buffer 上的 Padding 操作，性能相比之前有大概 6%到 8%的提升.

(3) 高效的转置算法

虽然采用了 cache-aware 分块方法，但仍然需要对数据分块进行两次转置操作. 由于传统的两层 for 循环的转置方法性能仍然较低，对程序性能造成了一定的影响. 因此，本文首先通过循环展开，分配 8 个行首地址变量，每次循环将矩阵的 8 个行首地址分配给该变量，在计算中分别对首地址做 4 个操作数的偏移，取矩阵 4 列数据. 该算法将矩阵行展开 8 次，列展开 4 次，每次循环则对一个 8 行 4 列的矩阵子模块进行转置运算. 这样一方面增加了数据连续性，提高了 cache 利用率；另一方面，减少了循环变量的计算，降低了计算开销. 其次在转置运算中使用了 SIMD 指令进行优化，将每次循环展开的 4 个数据列，用单条 SIMD 指令进行处理，使得每次赋值运算并行化，提升了转置算法的性能.

5.2.2　二维 FFT 变换优化流程

(1) 二维 C2C(Complex to Complex) FFT 变换

二维 C2C 的 FFT 变换优化后的流程如图 14(a)所示，首先将输入矩阵数据在行上做一维 C2C FFT 变换，将结果存入已加 padding 的 buffer0 中，然后将数据按照 $nb \times n1$ 大小分块，接着转置存入 buffer1 中，以便接下来在列上进行一维 C2C FFT 变换，最后按照分块大小做 C2C FFT 变换转置存入输出矩阵中.

(2) 二维 R2C(Real to Complex) FFT 变换

二维 R2C FFT 变换首先在行上做一维 r2c FFT 变换，R2C FFT 变换包含 C2C 变换和 split 操作两步(split 用于实数 FFT 变换)，在做完 C2C FFT 变换后需要做一个 split 转换，因为 split 转换不需要涉及一维 FFT 变换的 kernel，因此转置操作可以在 split 数据存储时进行，二维 R2C 的 FFT 变换流程如图 14(b)所示.

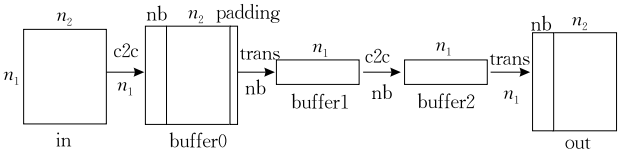
(3) 二维 C2R(Complex to Real)FFT 变换

二维 C2R FFT 变换是二维 R2C FFT 变换的逆变换，在优化过程中考虑到第一步是转置操作，因此 padding 加在最后一个 buffer 处. 二维 C2R 的 FFT 变换流程如图 14(c)所示.

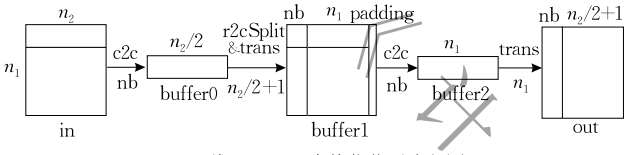
(4) 二维 R2R(Real to Real)FFT 变换

在二维 R2R FFT 变换优化时，考虑到一维

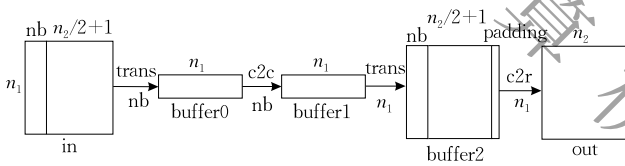
R2R FFT 的正逆 R2HC、HC2R 变换，因此二维 R2R FFT 变换排列后共有 4 种情况. 二维 R2HC R2HC FFT 变换优化后流程图如图 14(d)所示，二维 R2HC HC2R FFT 变换优化后流程图如图 14(e)所示，二维 HC2R R2HC FFT 变换优化后流程图如图 14(f)所示，二维 HC2R HC2R FFT 变换优化后流程图如图 14(g)所示，其中 R2HC 与 HC2R 互为正逆变换，而两者的主要区别为：R2HC 变换需要先做完 C2C 后再执行 split 操作，而 HC2R 则是先执行 split 操作，再做 C2C. 因此在执行 R2HC 时矩阵转置可以放在 split 操作中进行.



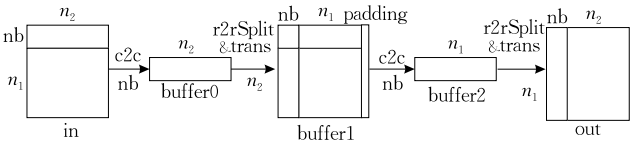
(a) 二维C2C FFT变换优化后流程图



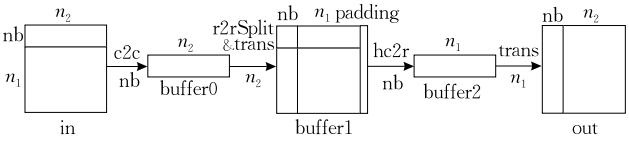
(b) 二维R2C FFT变换优化后流程图



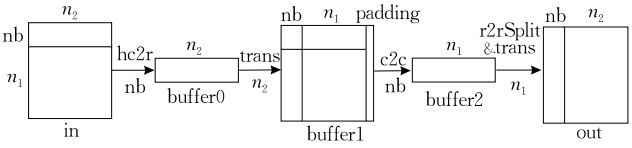
(c) 二维C2R FFT变换优化后流程图



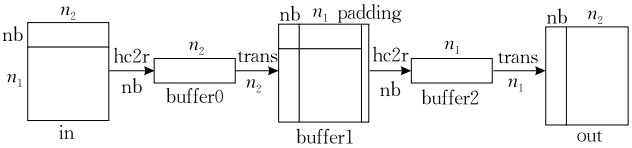
(d) 二维行r2hc列r2hc FFT变换优化后流程图



(e) 二维行r2hc列hc2r FFT变换优化后流程图



(f) 二维行hc2r列r2hc FFT变换优化后流程图



(g) 二维行hc2r列hc2r FFT变换优化后流程图

图 14 二维 FFT 变换优化后流程示意图

5.3 三维 FFT 的优化

三维 FFT 的优化其基本思想类似于二维 FFT 的优化，由于基础实现过程中，做 Z 方向一维 C2C FFT 时数据局部性极差，直接实现将会导致严重的访存延迟，为此 PerfFFT 采用 cache blocking 算法，开辟临时的数组，将进行一维 C2C FFT 运算的数据，转置到临时数组中，对临时数组中的数据进行一维 C2C FFT 操作，然后再把结果转置后写入结果数组. 由于过程中可以根据硬件的cache配置信息开启合适大小的临时数组，有效提升局部性，最终达到提升三维C2C FFT整体性能，其主要过程如图 15 所示.

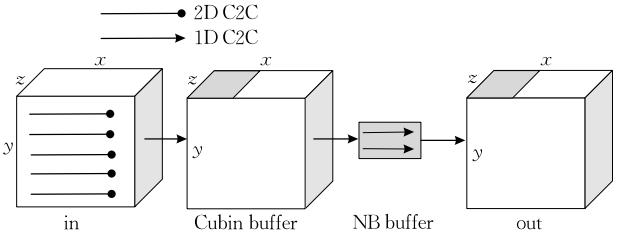


图 15 三维 C2C FFT 变换优化后流程示意图

6 性能评估

本节将对二维、三维 FFT 计算优化后的性能进行详细评估和说明.

6.1 测试环境搭建

(1) 硬件环境搭建

本文采用 ARM Cortex A57 CPU 作为性能测试平台. ARM Cortex A57 CPU 采用 ARMv8 架构. 本文的实验环境配置如表 1 所示.

表 1 实验环境配置

操作系统	Ubuntu 15.04 (GNU/Linux 3.19.0-rc4+aarch64)
CPU 型号	ARM CORTEX A57
内存容量	64 GB
L1 cache	32 KB
L2 cache	2 MB
cache line	64 Byte
时钟频率	2.1 GHz

(2) 软件环境搭建

本文的性能对比对象有两个：一是目前应用最为广泛的开源 FFT 算法库：FFTW3.3.6 版本；二是 ARM 公司推出的商业库 ARM Performance Library 2.2.0 版本，本文将其简称为 ARMPL-2.2.0. 本文实现的高性能 FFT 库为 PerfFFT.

本文对比的性能单位为 Gflops(Giga floating-point operation per second)即每秒所执行的浮点计算次数，计算复杂度为 $5\times x\times y\times z\times(\log x+\log y+\log z)$.

6.2　性能评估

6.2.1　单精度 FFT 性能评估

图 16、图 17 分别展示了单精度下，C2C、R2C、C2R、R2R 四种不同的 FFT 计算在二维与三维输入规模的性能图(ARMPL-2.2.0 暂不支持 R2R FFT 变换)，从中我们可以看出：无论是输入规模为 2 的幂还是输入规模为非 2 的幂的情况下，PerfFFT 的性能整体上高于 FFTW 和 ARMPL.

(1)二维单精度 FFT 性能比较

如图 16 所示，PerfFFT 相对于 FFTW 在二维单精度 C2C、R2C、C2R、R2R 2 的幂 FFT 变换优化结果中，分别实现了平均 216%、16%、11%、200%的加速比，相对于 ARMPL 在二维单精度 C2C、R2C、C2R 2 的幂 FFT 变换优化结果中，分别实现了平均 40%、44%、32%的加速比.

PerfFFT 相对于 FFTW 在二维单精度 C2C、R2C、C2R、R2R 非 2 的幂 FFT 变换优化结果中，分别实现了平均 10%、12%、22%、29%的加速比，相对于 ARMPL 在二维单精度 C2C、R2C、C2R 非 2 的幂 FFT 变换优化结果中，分别实现了平均 13%、17%、32%的加速比.

二维 FFT 变换性能主要影响因素一方面是由一维 FFT 变换决定，这部分是热点，占总性能值的 60%；另一方面是由两次矩阵转置运算决定. 由性能图走势可知：

首先二维单精度 FFT 性能由小规模输入数据向大规模性能呈现逐渐递减的趋势，当输入数据列返回至小规模，性能则又从图中极小值骤升至极大值. 其中主要的原因由于当输入规模偏小时，数据能够完全暂存在 2M 的 L2 cache 中，这样避免了 cache 的数据替换，增加了 cache 利用率，减少了访存开销，以致于小规模输入数据的性能相对于大规模偏高；

其次实数 FFT 变换的性能存在个别性能偏低的情况，这主要是因为实数 FFT 变换中的 split 操作每次循环需要对数组的首尾数据进行处理，在输入规模较大的情况下，访存数据不连续，导致 cache 中 cache line 数据不能一次存入待处理的有效数据，cache 得不到充分利用，因此在实数 FFT 变换时存在部分性能偏低的数据规模. 通过循环展开 4 次并且在 split 操作中的数据存取时一并进行转置操作，使得性能得到明显改善.

(2)三维单精度 FFT 性能比较

如图 17 所示，PerfFFT 相对于 FFTW 在三维单精度 R2C、C2R、R2R 2 的幂 FFT 变换优化结果中，分别实现了平均 36%、47%、529%的加速比.

PerfFFT 相对于 FFTW 在三维单精度 R2C、C2R、R2R 非 2 的幂 FFT 变换优化结果中，分别实现了平均 2%、7%、15%的加速比.

(3)单精度 FFT 性能分析

在单精度 FFT 变换的优化过程中，由于每一个单精浮点操作数占相对较少的空间 32 bit，因此在优化的过程中能够更有效地利用 ARMv8 的硬件特性进行优化，如每一个浮点寄存器能存储更多操作数、循环展开优化时展开规模更大等. 通过本文所提及的优化方法如 SIMD、循环展开等方法能够获得很不错的加速比效果.

在多维 FFT 变换优化过程中，采用 cache-aware 分块优化的方法，依据 L2 cache 存储容量 2 MB 的情况，将矩阵分块为 $n1\times 8$ 的规模大小，从而对每个数据块单独进行转置操作. 在转置过程中，本文根据分块规模，使用循环展开 8 次的方法，对每一次数据存取采用 SIMD 指令进行操作. 这样不仅最大程度地重用了 cache 中的数据、而且增加了计算的并行度.

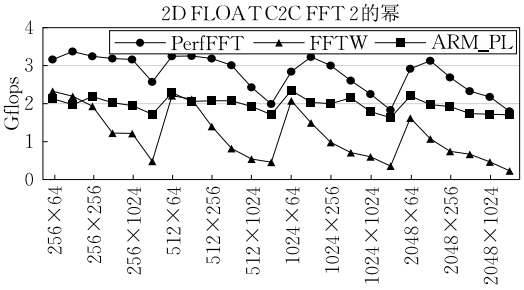
在实数 FFT 变换的优化过程中，由于实数 FFT 变换相对于复数 FFT 变换加入了用于数据重组的 split 操作，该操作是在核函数 kernel 外的实现，因此本文将 split 操作与其相邻的转置操作合并在一起处理，在 split 中数据的存储中直接实现转置，减少了一次矩阵转置. 考虑到 cache 的利用率，本文在 split 操作中使用 SIMD 指令的同时，将存储连续的数据放在一块进行 load、store 操作，有效利用了 cache 中的暂存数据.

在非 2 的幂 FFT 变换的优化过程中，考虑到矩阵的每一行的数据规模并不是按照 cache line 64 Byte 对齐，在 L1 cache 取数时则会取入冗余的数据，因此本文将 buffer 每行补上 padding，使每一行的数据对齐 64 Byte，高效地利用了 L1 cache.

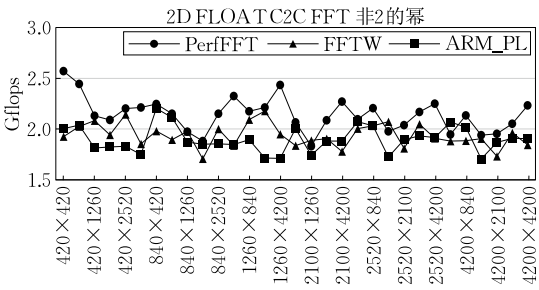
6.2.2　双精度 FFT 性能评估

(1)二维双精度 FFT 性能比较

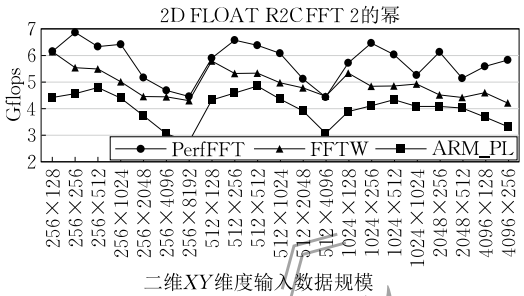
图 18 显示了双精度情况下，C2C FFT 计算的性能对比图，从中我们可以看出：当输入规模为 2 的幂的情况下，性能相于 FFTW 达到了 109%的加速比，但是和 ARMPL 相比不相上下，并没有达到特别理想的效果.



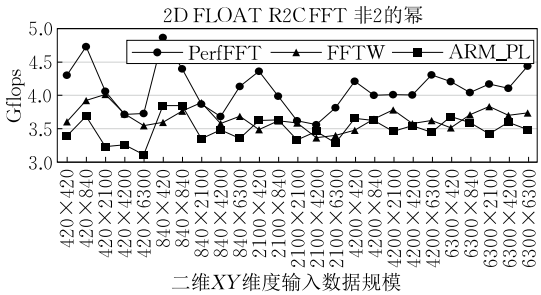
(a) 二维FLOAT C2C FFT 2的幂性能对比图



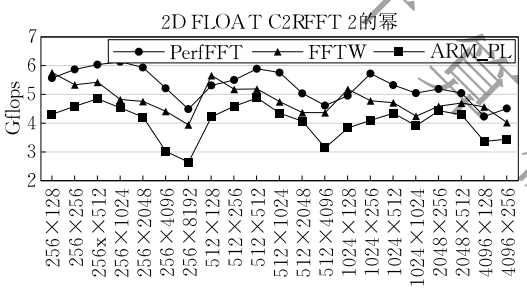
(b) 二维FLOAT C2C FFT非2的幂性能对比图



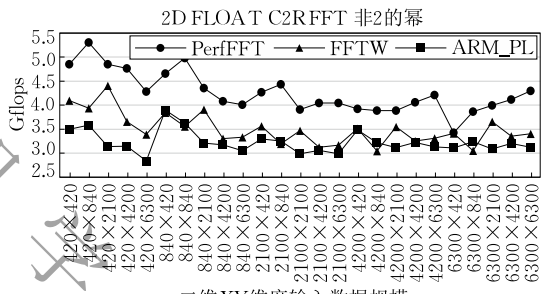
(c) 二维float r2c FFT 2的幂性能对比图



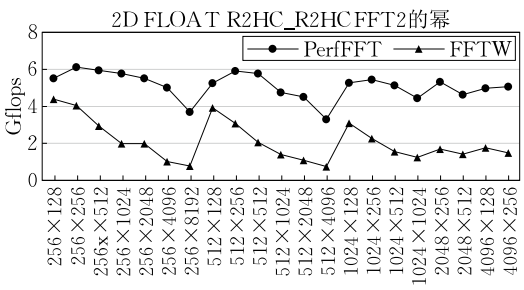
(d) 二维FLOAT R2C FFT非2的幂性能对比图



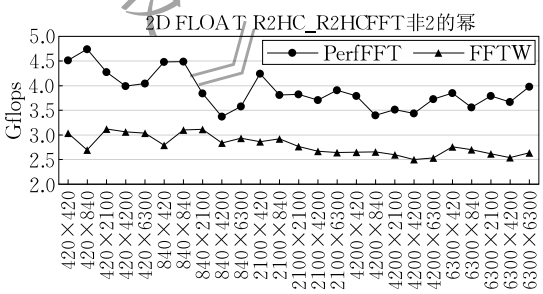
(e) 二维FLOAT C2R FFT 2的幂性能对比图



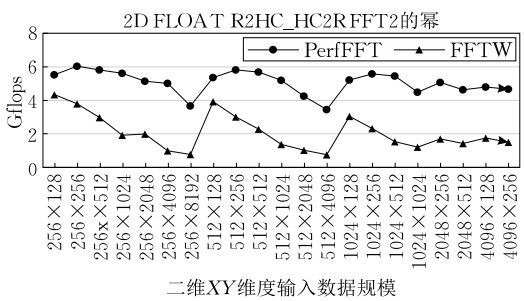
(f) 二维FLOAT C2R FFT非2的幂性能对比图



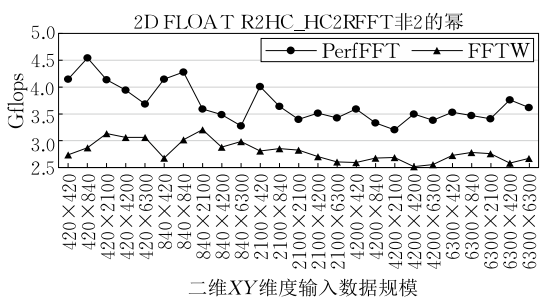
(g) 二维FLOAT R2HC_R2HC FFT 2的幂性能对比图



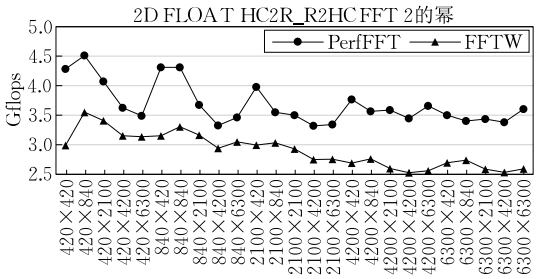
(h) 二维FLOAT R2HC_R2HC FFT非2的幂性能对比图



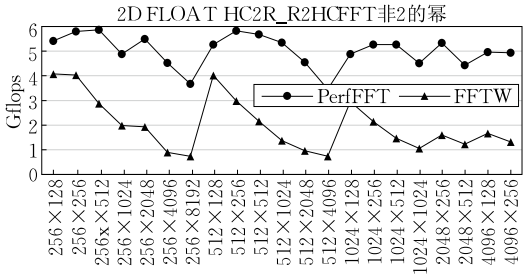
(i) 二维FLOAT R2HC_HC2R FFT 2的幂性能对比图



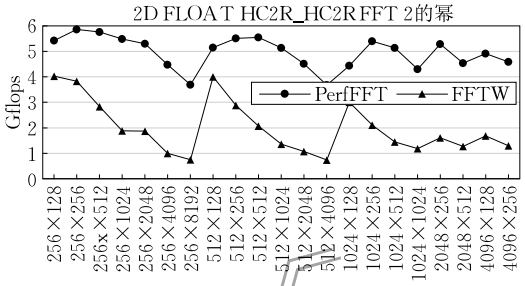
(j) 二维FLOAT R2HC_HC2R FFT非2的幂性能对比图



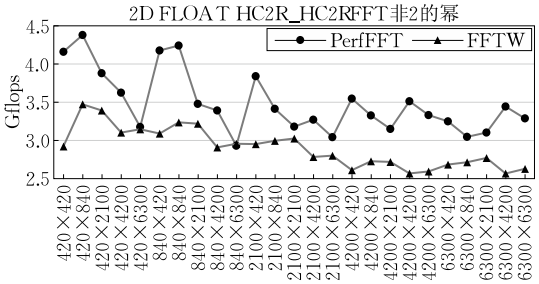
(k) 二维FLOAT HC2R_R2HC FFT 2的幂性能对比图



(l) 二维FLOAT HC2R_R2HC FFT非2的幂性能对比图

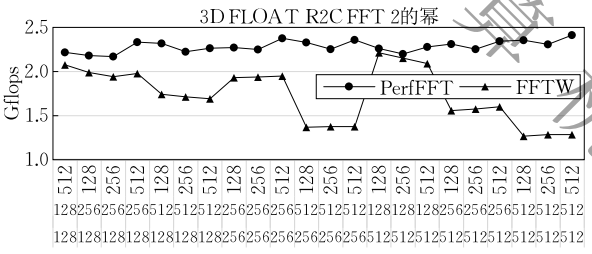


(m) 二维FLOAT HC2R_HC2R FFT 2的幂性能对比图

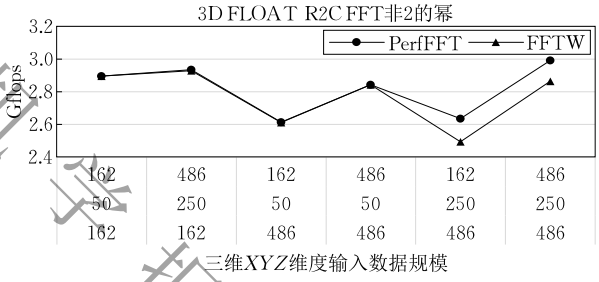


(n) 二维FLOAT HC2R_ HC2R FFT非2的幂性能对比图

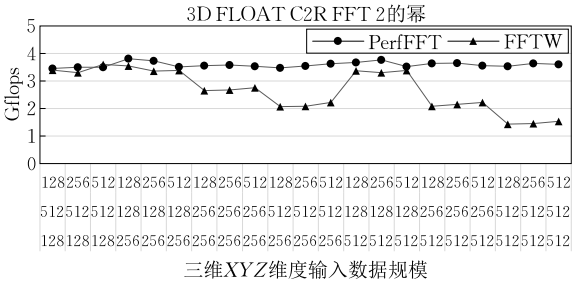
图 16　二维 FLOAT FFT 性能对比图



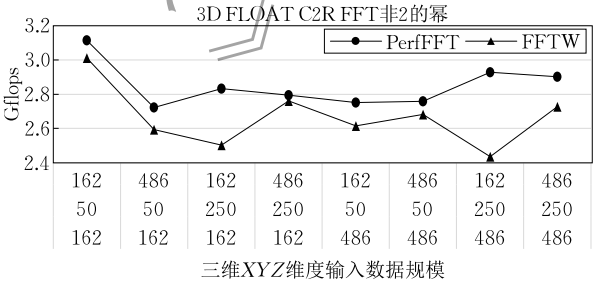
(a) 三维FLOAT R2C FFT 2的幂性能对比图



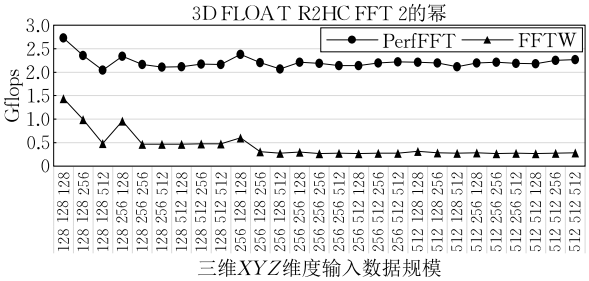
(b) 三维FLOAT R2C FFT 2的幂性能对比图



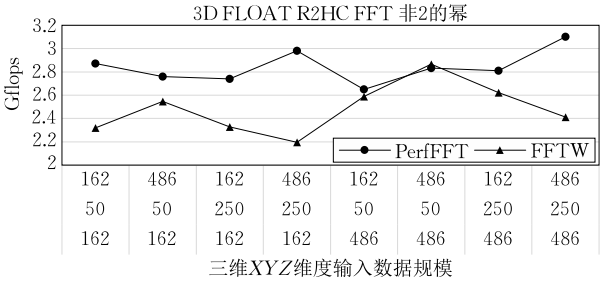
(c) 三维FLOAT C2R FFT 2的幂性能对比图



(d) 三维FLOAT C2R FFT 2的幂性能对比图

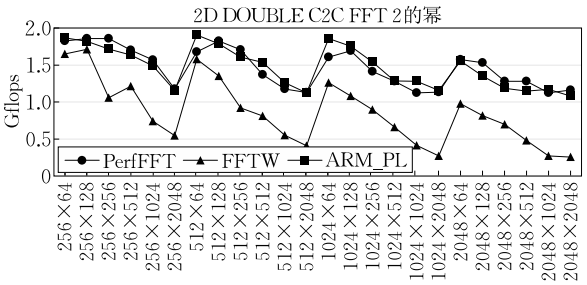


(e) 三维FLOAT R2HC FFT 2的幂性能对比图

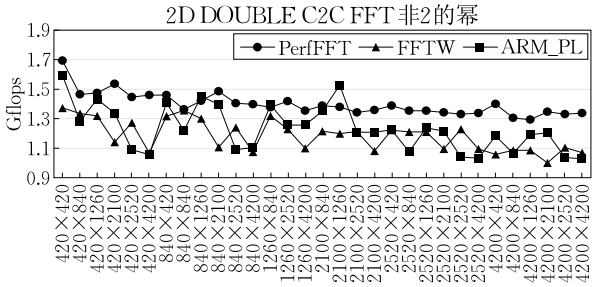


(f) 三维FLOAT R2HC FFT 2的幂性能对比图

图 17　三维 FLOAT FFT 性能对比图



(a) 二维DOUBLE C2C FFT 2的幂性能对比图



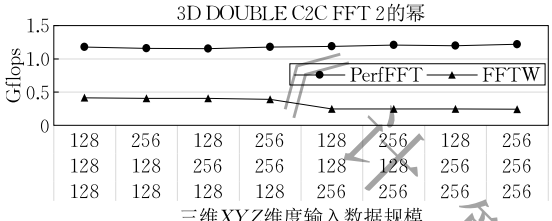
(b) 二维DOUBLE C2C FFT非2的幂性能对比图

图 18 二维 DOUBLE FFT 性能对比图

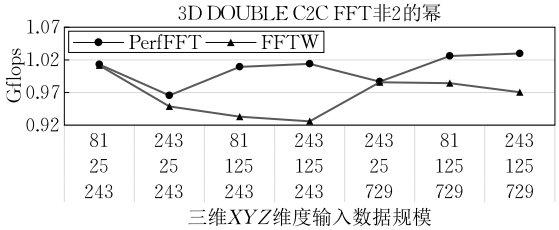
当输入规模为非 2 的幂的情况下,性能相对 FFTW 达到了 18%的加速比,相对于 ARMPL 达到了 14%的加速比.

(2) 三维双精度 FFT 性能比较

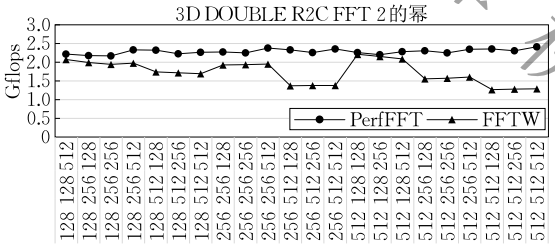
如图 19 所示,PerfFFT 相对于 FFTW 在三维双精度 C2C、R2C、C2R、R2R 2 的幂 FFT 变换优化



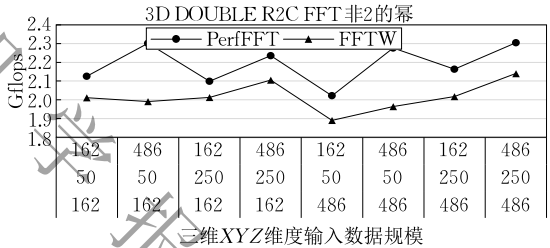
(a) 三维DOUBLE C2C FFT 2的幂性能对比图



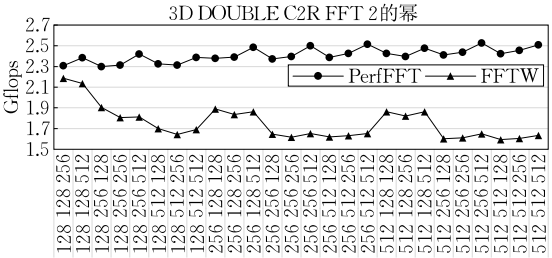
(b) 三维DOUBLE C2C FFT非2的幂性能对比



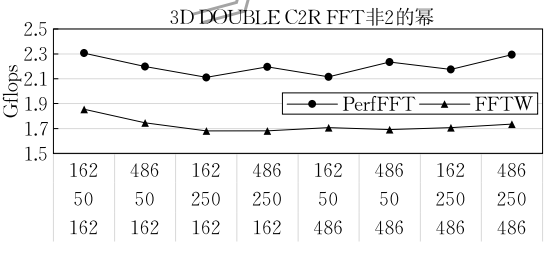
(c) 三维DOUBLE R2C FFT 2的幂性能对比图



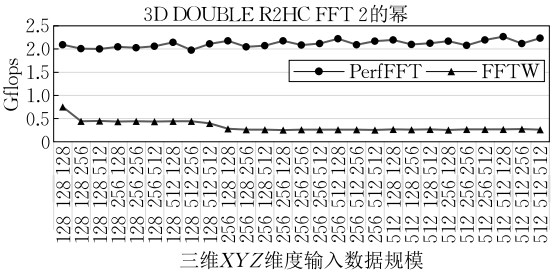
(d) 三维DOUBLE R2C FFT非2的幂性能对比图



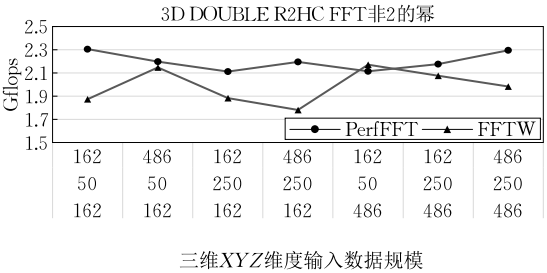
(e) 三维DOUBLE C2R FFT 2的幂性能对比图



(f) 三维DOUBLE C2R FFT非2的幂性能对比图



(g) 三维DOUBLE R2HC FFT 2的幂性能对比图



(h) 三维DOUBLE R2HC FFT非2的幂性能对比图

图 19 三维 DOUBLE FFT 性能对比图

结果中，分别实现了平均 289%、36%、38%、591%的加速比.

PerfFFT 相对于 FFTW 在三维双精度 C2C、R2C、C2R、R2R 非 2 的幂 FFT 变换优化结果中，分别实现了平均 4%、8%、27%、11%的加速比.

(3) 双精度 FFT 性能分析

由以上结果对比可知，绝大多数情况下，双精度的加速性能相对单精度的加速比有所降低(除几个 FFTW 没有实现好的三维 FFT 变换外)，主要原因有两个：

① SIMD 优化效果不明显. 双精度浮点的大小为 64bit，ARM 向量寄存器的长度为 128bit，这样每个向量寄存器一次只能处理 2 个双精度浮点数.

② ARM 有些双精度浮点数指令性能比较低，比如 zip、ld2 指令等，造成可优化的空间不大.

双精度实数 FFT 计算本文暂时没有实现，这将是本文未来的主要工作.

综上，PerfFFT 是当前针对 ARMv8 架构处理器性能最好的 FFT 库之一.

7　结束语

本文在实现多维 FFT 变换的基础上，提出了蝶形网络优化、蝶形计算优化、蝶形自动生成、SIMD 优化、内存对齐、cache-aware 的分块算法和高效转置算法等优化方法，实现了一个 ARMv8 平台上的高性能的多维 FFT 库：PerfFFT. 与现有的高性能 FFT 软件库 FFTW、ARMPL 相比，PerfFFT 性能都明显高于这两个库的性能. 这不仅实现了提升 ARMv8 平台 FFT 软件库性能的目标，而且为 ARMv8 平台上程序优化提供了新的思路. 未来的主要工作将从双精度 FFT 算法的实现和优化、三维 FFT 减少 TLB 缺失率优化，两个首要的入手点，最后，为 FFT 软件库搭建自适应框架也是未来一个重要的工作，我们着眼于手工优化和自适应优化技术相结合的方式，进一步提升 FFT 算法库的性能.

致　谢　感谢中国科学院计算技术研究所并行软件组黄珊、王霄、操庐宁、李晨荻的帮助！

参　考　文　献

[1] Li Yan, Zhang Yun-Quan. An automatic performance tuning framework for FFT on heterogenous platforms. Journal of Computer Research and Development, 2014, 51(3): 637-649 (in Chinese)
(李焱，张云泉. 异构平台上性能自适应 FFT 框架. 计算机研究与发展，2014，51(3)：637-649)

[2] Li Yan, Zhang Yun-Quan, Wang Ke, Zhao Mei-Chao. Implementation and optimization of the FFT using OpenCL on heterogeneous platforms. Journal of Computer Science, 2011, 38(8): 284-286(in Chinese)
(李焱，张云泉，王可，赵美超. 异构平台上基于 OpenCL 的 FFT 实现与优化. 计算机科学，2011，38(8)：284-286)

[3] Frigo M, Johnson S G. FFTW: An adaptive software architecture for the FFT//Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing. Seattle, USA, 1998: 1381-1384

[4] Pippig M. PFFT: An extension of FFTW to massively parallel architectures. SIAM Journal on Scientific Computing, 2013, 35(3): C213-C236

[5] Li Yan, Zhang Yunquan, Liu Yiqun, et al. MPFFT: An auto-tuning FFT library for OpenCL GPUs. Journal of Computer Science and Technology, 2013, 28(1): 90-105

[6] Chen Yifeng, Cui Xiang, Mei Hong. Large-scale FFT on GPU clusters//Proceedings of the 24th ACM International Conference on Supercomputing. Tsukuba, Japan, 2010: 315-324

[7] Frigo M, Johnson S G. The design and implementation of FFTW3. Proceedings of the IEEE, 2005, 93(2): 216-231

[8] Frigo M. A fast Fourier transform compiler//Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. Atlanta, USA, 1999, 34(5): 169-180

[9] Vetterli M, Duhamel P. Split-radix algorithms for length-p/sup m/ DFT's. IEEE Transactions on Acoustics, Speech, and Signal Processing, 1989, 37(1): 57-64

[10] Johnson S G, Frigo M. A modified split-radix FFT with fewer arithmetic operations. IEEE Transactions on Signal Processing, 2007, 55(1): 111-119

[11] Kolba D, Parks T W. A prime factor FFT algorithm using high-speed convolution. IEEE Transactions on Acoustics, Speech, and Signal Processing, 1977, 25(4): 281-294

[12] Rader C M. Discrete Fourier transforms when the number of data samples is prime. Proceedings of the IEEE, 1968, 56(6): 1107-1108

[13] Wang Qian, Zhang Xianyi, Zhang Yunquan, Yi Qing. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs//Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Denver, USA, 2013: 1-12

[14] Wang X, Jia H, Li Z, et al. Implementation and Optimization of Multi-dimensional Real FFT on ARMv8 Platform//Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing. Springer, Cham, Guangzhou, China, 2018: 338-353

CHEN Tun, Ph. D. candidate. His research interests include high performance computing and parallel programming.

LI Zhi-Hao, Ph. D. candidate. His research interests include high performance computing, hetenogenous computing.

JIA Hai-Peng, Ph. D., senior engineer. His research interests include high performance computing, the method of programing and optimization for many-core computing platform.

ZHANG Yun-Quan, Ph. D., professor. His research interests include high performance computing, parallel numerical software and parallel computing model.

Background

Fast Fourier Transform (FFT) is a fast algorithm used to calculate Discrete Fourier Transform (DFT) and its inverse operation. It is widely used in the field of engineering, science and mathematics. Currently, there are FFT library of FFTW, PFFT, MPFFT, CUFFT, PKUFFT and so on. It is only FFTW operate on ARM platform implementation and optimization. It supports both shared memory multi-thread parallelism and MPI parallelism, and its computational performance is far ahead of other existing FFT software. FFTW has been implemented and optimized for the ARM platform since the FFTW version 3.3.1, and has achieved very high performance. In addition to FFTW, ARM has also introduced a high-performance commercial library for the ARMv8 platform: ARM Performance Library.

Our main contributions are as follows: First, we propose a set of FFT algorithm implementation and optimization on ARMv8 platform, which not only improves the performance of FFT algorithm on ARMv8 platform, but also has practical Guiding significance for implementation of other algorithms on ARM platform. Second, we propose a set of FFT butterfly calculation code automatic generation scheme. A computational template is formed by abstracting and extracting typical computational patterns of butterfly calculations for different radix of the FFT. And on this basis, automatically generate FFT different radix butterfly calculation high performance code. Similar to the butterfly calculation code automatically generated template proposed in this paper, the template-based code automatic generation optimization framework AUGEM is well applied in BLAS (Basic Linear Algebra Subprograms), which can be used on multi-core CPUs. Several dense linear algebraic DLA kernels (such as GEMM, GEMV, AXPY, and DOT) automatically generate fully optimized assembly code. Using the simple C implementation of the DLA core as input, it automatically generates an efficient assembly kernel for the input code through four components (optimized C kernel generator, template recognizer, template optimizer and assembly kernel generator). The average performance of the automatically generated GEMM core in the AUGEM architecture was 1.4%, 3.3%, and 89.5% higher on Intel Sandy Bridge processors than Intel MKL, ATLAS and GotoBLAS, respectively. Finally, we implement a high-performance multi-dimensional FFT algorithm library: PerfFFT. Compared with the performance of FFTW and ARM Performance Library, PerfFFT is the highest performance multi-dimensional FFT library on ARM platform.

This work is supported by the National Key R&D Plan of China (Nos. 2017YFB0202105, 2016YFB0200803, 2017YFB0202302), the National Nature Fund Youth Fund of China (No. 61602443), the National Natural Science Fund Key Fund of China (No. 61272136), the National Natural Science Foundation of Innovation Group of China (No. 61521092), the Guangdong Province Major Science and Technology Projects (No. 2015B010108006).