计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.190900179

一种偶数基 Cooley-Tukey FFT 高性能实现方法

龚彤艳[1,2]　张广婷[2]　贾海鹏[2]　袁　良[2]
1 贵州财经大学信息学院　贵阳 550025
2 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
(gongtongyan@ict.ac.cn)

摘　要　快速傅里叶变换(Fast Fourier Transform,FFT)是最重要的基础算法之一,在科学计算、信号处理、图像处理等领域都有着广泛的应用。随着这些应用领域对实时性需求的进一步提高,FFT 算法面临着越来越高的性能要求。在现有的 FFT 算法库中,FFT 算法的求解速度和计算精度受到一定程度的限制,而且也少有研究者对偶数基 Cooley-Tukey FFT 的高性能实现提出相应的优化策略并对技术进行深入研究。基于此,文中提出了一套针对偶数基的 Cooley-Tukey FFT 的优化策略和方法。首先构建一个 SIMD(Single Instruction Multiple Data)友好、支持混合基的蝶形网络,然后根据偶数基旋转因子特性最大限度地降低蝶形计算的复杂度,接着通过 SIMD 汇编优化、汇编指令重排及选择、寄存器分配策略制定、高性能矩阵转置算法等方法来优化应用,最后实现一个高性能的 FFT 算法库。目前,最流行、应用最广的 FFT 有 FFTW 和 Intel MKL。实验结果表明,在 X86 计算平台上,新提出的这套针对偶数基 Cooley-Tukey FFT 的技术所实现的 FFT 算法库的性能全面优于 MKL 和 FFTW。所提出的这套高性能算法优化和实现技术体系,可推广到除偶数基以外的其他基的实现和优化上,为进一步的研究开发工作奠定一定的基础,进而突破 FFT 算法在硬件平台上的性能瓶颈,实现一套针对特定平台的高性能 FFT 算法库。

关键词:快速傅里叶变换算法;偶数基;蝶形计算优化;蝶形网络优化;SIMD 汇编优化;高性能 FFT 库
中图法分类号　TP311.52

High-performance Implementation Method for Even Basis of Cooley-Tukey FFT

GONG Tong-yan[1,2],ZHANG Guang-ting[2],JIA Hai-peng[2] and YUAN Liang[2]
1 School of Information,Guizhou University of Finance and Economics,Guiyang 550025,China
2 State Key Laboratory of Computer Architecture,Institute of Computing Technology,Chinese Academy of Sciences,Beijing 100190,China

Abstract　Fast Fourier transform (FFT) is one of the most important basic algorithms,which is widely used in scientific calculation,signal processing,image processing and other fields. With the further improvement of real-time requirements in these application fields,fast Fourier transform algorithms are facing higher and higher performance requirements. In the existing FFT algorithm library,the solution speed and calculation accuracy of FFT algorithm are limited to a certain extent,and few researchers put forward corresponding optimization strategies and conducted in-depth research on the implementation of cooley-tukey fast Fourier transform based on even Numbers. Based on this,this paper put forward a set of for even basis of optimization strategy and method for Colley-Turkey fast Fourier transform. Firstly,a friendly butterfly network supporting SIMD mixed is constructed. Secondly,according to the even base rotation factor characteristics,the complexity of the butterfly calculation is reduced to a maximum degree. Thirdly,through the SIMD assembly optimization,assembly instruction rearrangement and selection,register allocation strategy and high performance matrix transpose algorithm method,the application is optimized . Finally a high performance FFT algorithm library is achieved. Currently,the most popular and widely used FFT are FFTW and Intel MKL. Experimental results show that on X86 computing platform,the performance of FFT library based on cooley-tukey FFT is better than MKL and FFTW. The high performance algorithm is put forward by the new optimization method and implementation technology system,which can be generalized to other except the even base based on the realization and optimization of a certain basis for further research and development work,to break through the FFT algorithm performance bottlenecks in the hardware platform,to achieve a high performance FFT algorithms library for a specific platform.

到稿日期:2019-08-21　返修日期:2019-10-25　本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家重点研发计划(2018YFC0809306);国家自然科学基金青年科学基金(61602443);国家自然科学基金重点项目(61432018)
This work was supported by the National Key Research and Development Program of China (2018YFC0809306),Young Scientists Fund of the National Natural Science Foundation of China (61602442) and Key Program of the National Natural Science Foundation of China(61432018).
通信作者:张广婷(zhangguangting@ict.ac.cn)

Keywords Fast Fourier transform algorithm, Even basis, Butterfly calculation optimization, Butterfly network optimization, SIMD assembly optimization, High performance FFT library

1 引言

快速傅里叶变换[1-2]是离散傅里叶变换(Discrete Fourier Transform,DFT)[3]及其逆变换的快速计算方法。1994 年,美国数学家 Strang 称 FFT 为"我们一生中最重要的数值算法"[4-5]。FFT 还被 IEEE 期刊 *Computing in Science & Engineering* 列入 20 世纪十大算法之一[6]。快速傅里叶变换被广泛应用于科学、数学、工程等领域,是最重要的核心基础算法之一。各领域的高速发展以及对实时性能要求的不断提高,使得 FFT 的性能面临着更高的要求,所以 FFT 算法的实现和优化具有非常重要的研究价值。

Cooley-Tukey FFT 算法是应用最广泛的 FFT 算法,本文将研究该算法的实现和优化策略。由于 Cooley-Tukey FFT 算法存在蝶形网络结构复杂、蝶形种类繁多、算法计算量大、内存访问频繁等问题,因此 FFT 算法的高性能实现面临着严峻的挑战。其中,Cooley-Tukey FFT 算法基的选择对算法性能有着至关重要的影响,虽然已有很多工作对 2 的幂的基和质数基都有了很好的实现和优化,但是鲜有学者对偶数基(如基 6 和基 10)的实现和优化进行了较好的研究。偶数基的实现和应用可从 3 个方面提升算法性能:1)可减小蝶形网络的级数,降低访存开销,如对于基 6,可将基 2 和基 3 两级网络合并为一级;2)偶数基的旋转因子具有更好的对称性,可进一步减少算法的总体计算量;3)给性能调优提供了更大的优化空间。

对此,本文通过 SIMD 汇编优化、汇编指令重排及选择、寄存器分配策略制定、高性能矩阵转置算法等方法优化应用,实现了一个高性能的 FFT 算法库。实验结果表明:在 X86 Intel(R) Xeon(R) CPU 计算平台上,本文提出的 FFT 算法较 FFTW3.3.8 库(目前最新、应用最广的 FFTW 库)和 Intel® Math Kernel Library (Intel® MKL)库,分别将性能提升了 10%~190%和 10%~33%。

本文的主要贡献如下:1)提出了一套针对偶数基的 Cooley-Tukey FFT 算法的高性能实现策略和方法;2)设计了一种针对 FFT 偶数基的计算和优化模式,最大限度地降低了蝶形计算的复杂度;3)实现了一个高性能的 FFT 算法库。

本文第 2 节介绍了相关工作;第 3 节从算法原理的角度出发,介绍了 DFT 原理和旋转因子的相关特性;第 4 节分别从蝶形网络优化、蝶形计算优化、蝶形汇编优化方面讲述了 Cooley-Tukey FFT 的优化方式;第 5 节介绍了本文的测试环境并给出了实验的性能分析;最后总结全文并对未来工作和计划进行说明。

2 相关工作

目前,FFT 的研究工作通常以算法库的形式展现。常用的 FFT 算法库主要有 FFTW[7],MKL[8-9],CUFFT 和 ALGLIB。

2.1 FFTW

FFTW(Fastest Fourier Transform in the West)算法库由麻省理工学院的 Frigo 和 Johnson 开发[10-11],其开发的高性能 FFT 算法库,可高效计算任意规模、任意维度的离散傅里叶变换。同时,FFTW 支持包括 C2C,C2R,R2C,R2R 在内的所有实数和复数变换类型。FFTW 库中除了包含 Cooley-Tukey 算法外,还包含一些其他算法,如素因子(Prime Factor)算法[12]、分裂基(Split Radix)算法[13-14]、向量基(Vector Radix)算法[15]等。

FFTW 采用性能自适应优化策略,使得算法能够更好地适应硬件平台,从而最大限度地提高性能。FFTW 算法库的最大优势就是性能可移植性强,即可移植到任何具有 C 编译器的平台。

2.2 MKL

MKL(Math Kernel Library)是一个用于科学、工程和金融应用程序的优化数学例程库。核心数学函数包括 BLAS[16]、LAPACK[17]、ScaLAPACK[18]、稀疏求解器、快速傅里叶变换和矢量数学等[19]。该库支持 Intel 处理器,可以用于 Windows 系统、Linux 系统以及 macOS 操作系统。Intel 公司在 2003 年 5 月 9 日启动了 MKL,并将其命名为 blas.lib[20]。Intel C++编译器生成的 MKL 和其他程序一起通过函数多版本化的技术来提高性能。针对许多 x86 指令集扩展编译,在运行主函数时,使用 CPUID 选择最适合当前 CPU 版本的指令。但是,只要主功能检测到非英特尔 CPU,不管 CPU 声明要支持的指令是什么,它总是会在第一时间选择最基本(最慢)的功能,这使得该系统自 2009 年以来就有了"cripple AMD"的绰号[21]。截止到 2019 年,MKL 依旧是 Windows 系统上许多预编译的 Mathematical 应用程序的选择,但是,在具有等效指令集的 AMD CPU 上,其性能仍然大大落后[22]。

MKL-FFT 支持一维到多维、复数到复数(C2C)、实数到复数(R2C)、复数到实数(C2R)的任意长度的各种快速傅里叶变换。通过与 MKL 提供的接口进行链接,可以轻松地将开源 FFTW 编写的应用程序移植到 MKL。使用 Intel MKL 的 FFT 求解程序时,需要通过在链接行添加-lm 参数链接数学系统库。

2.3 CUFFT

NVIDIA CUDA[23]快速傅里叶变换库(CUFFT)提供 GPU 加速的 FFT 实现。CUFFT 是基于著名的 Cooley-Tukey 和 Bluestein 算法的基础库,用于建立跨学科的商业和学术应用,如计算物理、分子动力学、量子化学、地震和医学成像。它支持批量转换和优化精度算法,被广泛用于构建基于深度学习的计算机视觉应用。其主要特点有:复数和实数数据类型的 1D,2D,3D 变换;多 GPU R2C 和 C2R 支持;支持跨 16 个 GPU 的大型 FFT 模型,有效容量为 512 GB;单精度和双精度变换;执行多个转换的批处理;灵活的输入和输出数据

布局，类似于 FFTW 的“高级接口”。它通过允许各个元素和数组维度之间的任意跨步，来灵活地实现数据布局等。

2.4 ALGLIB

ALGLIB[24]是一个跨平台的数据分析和数据处理库，具有实数/复数 FFT 实现的 C++和 C# 库。当变换长度 N 为复合数时，应用 Cooley-Tukey 算法将初始变换序列转换为对应于 N 的素因子变换短序列。短序列变换须用特殊式计算。

3 Cooley-Tukey FFT 算法

在介绍 Cooley-Tukey FFT 算法之前，先对 DFT 进行简单介绍。DFT 是可逆的线性变换，其定义如下。

对于 $x(n)$点序列，其离散傅里叶变换定义为：

$$X^F(k)=\sum_{n=0}^{N-1}x(n)W_N^{kn} \tag{1}$$

其中，$k=0,1,\cdots,N-1$ 为 N 个 DFT 系数；$n=0,1,\cdots,N-1$；$W_N=\exp(-j2\pi/N)$是 1 的 N 次根；$X^F=k$ 为第 k 个 DFT 系数，$k=0,1,\cdots,N-1$；$j=\sqrt{-1}$。式(1)中，W_N^{kn} 是 DFT 的系数，称为旋转因子(twiddles)。

$$W_N^{kn}=\exp\left[\left(\frac{-j2\pi}{N}\right)kn\right]$$

式(1)以求和形式表示的 DFT，也可以用向量矩阵相乘的形式表达：

$$\begin{bmatrix}X^F(0)\\X^F(1)\\\vdots\\X^F(k)\\\vdots\\X^F(N-1)\end{bmatrix}=\begin{bmatrix}(W_N^{kn})\\(k,n=0,1,\cdots,N-1)\end{bmatrix}\begin{bmatrix}x(0)\\x(1)\\\vdots\\x(n)\\\vdots\\x(N-1)\end{bmatrix}$$

输出序列　　DFT 矩阵　　输入序列　　(2)

其中，DFT 矩阵如式(3)所示：

$$\begin{bmatrix}W_N^0 & W_N^0 & W_N^0 & \cdots & W_N^0\\W_N^0 & W_N^1 & W_N^2 & \cdots & W_N^{N-1}\\W_N^0 & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)}\\\vdots & \vdots & \vdots & & \vdots\\W_N^0 & W_N^k & W_N^{2k} & \cdots & W_N^{k(N-1)}\\\vdots & \vdots & \vdots & & \vdots\\W_N^0 & W_N^{(N-1)} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)(N-1)}\end{bmatrix} \tag{3}$$

由离散傅里叶变换的数学式(1)可知，其旋转因子$W_N^{kn}=\exp[(-j2\pi/N)kn]$，再结合欧拉公式 $e^{\pm j\theta}=\cos\theta\pm j\sin\theta$，可以得出旋转因子的重要性质。

(1)周期性：W_N 具有周期 N，即 $W_N^{k+N}=W_N^k$。

(2)对称性：$W_N^{k+N/2}=-W_N^k$。

(3)若 m 是 N 的约数，则 $W_N^{mkn}=W_{N/m}^{kn}$。

与此同时，旋转因子还有一些特殊值性质，即 $W_N^0=1$ 和 $W_N^{N/2}=-1$。

从离散傅里叶变换的定义式出发，如果计算傅里叶变换，则运算量高达 $O(n^2)$次。为了简化 DFT，降低算法的时间复杂度，利用旋转因子的周期性、对称性、可约性，将其时间复杂度降低到 $O(n\log n)$的快速傅里叶变换应运而生。截止目前，FFT 已经发展了多种成熟的算法，其中 Cooley-Tukey FFT 算法最为成熟，应用最广泛。因此，本文对 Cooley-Tukey FFT 算法进行研究。

Cooley-Tukey FFT 算法的核心思想是分而治之，即将 N(假设 $N=2^m$，且 m 为整数)点序列按照分治思想分解为两个点序列(一个为偶数点序列，另一个为奇数点系列)，之后通过计算这两个 $N/2$ 点序列的 FFT 得到原来 N 点序列的 FFT。分治算法每一级的分解可以节省大部分算术运算，直到最后一级分解为 2 点序列为止。

定义式(4)来表示快速傅里叶变换：

$$X^F(k)=\sum_{n=0}^{N-1}x(n)W_N^{kn}=G^F(k)+W_N^kH^F(k) \tag{4}$$

其中，$k=0,1\cdots,N/2-1$；$G^F(k)$和 $H^F(k)$分别表示 $x(n)$的 N 点 DFT 序列 $X^F(k)$的偶数序列和奇数序列。

由旋转因子的周期性($W_N^{k+N}=W_N^k$)可知，$X^F(k)$的周期为 N，$X^F(k)=X^F(k+N)$；$G^F(k)$和 $H^F(k)$的周期都为 $N/2$；再结合旋转因子的特殊值性质($W_N^{N/2}=-1$ 和 $W_N^{k+N/2}=-W_N^k$)，可对式(3)、式(4)进一步化简得：

$$X^F(k+N/2)=G^F(k)-W_N^kH^F(k) \tag{5}$$

式(5)可以用蝶形图来表示，如图 1 所示。

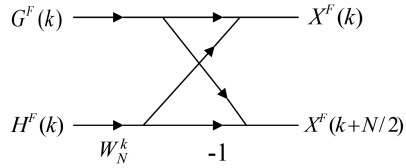


图 1　Cooley-Tukey FFT 蝶形图

Fig. 1　Cooley-Tukey FFT butterfly diagram

采用分而治之的思想对 N 点 DFT 序列进行拆分，直至产生 2 点序列为止，采用旋转因子的特性将这 2 点序列化简到最优时，其数据的流程图(图 1)呈现蝶形，将其称为蝶形图。

综上，本节首先从 DFT 的定义出发，说明了其算法复杂度高的属性，利用旋转因子的性质提出了降低算法复杂度的 FFT 算法；其次阐述了 Cooley-Tukey FFT 算法分而治之的核心思想；最后贯彻 Cooley-Tukey FFT 算法的核心思想，最简化算法后，用数据流图表示了算法中蝶形的产生过程。

4 Cooley-Tukey FFT 优化

4.1 蝶形网络优化

在引出新型的蝶形网络前，先简要介绍了传统蝶形网络中 stage-section-butterfly 的概念。

以图 2 为例，该蝶形网络是一个三级(stage)蝶形网络；红色框、蓝色框、黑色框分别表示蝶形网络的第一级、第二级、第三级。第一级中有 4 个红色框，每个红色框代表一个 section，每个 section 中有 1 个 butterfly，总共有 4 个 section；第二级中的蓝色框代表 section，总共有 2 个 section，且每个 section 都有两个 butterfly；第三级中的黑色框代表总共有一个 section，每个 section 中有 4 个 butterfly。

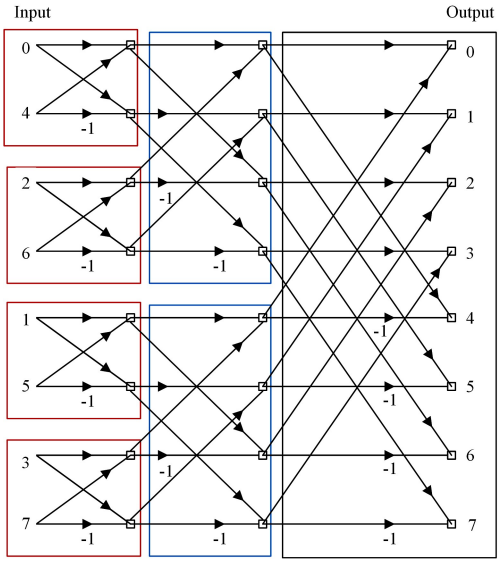


图 2 传统蝶形网络($N=8$)(电子版为彩色)

Fig. 2 Traditional butterfly network ($N=8$)

从图 2 中可以看出,输入数据存在“位反转”操作,这不仅增加了对全部数据的读写操作,而且不利于混合基的建立,对 SIMD 更不友好。鉴于此,本文引用了新型的蝶形网络——Stockham 蝶形网络,如图 3 所示。

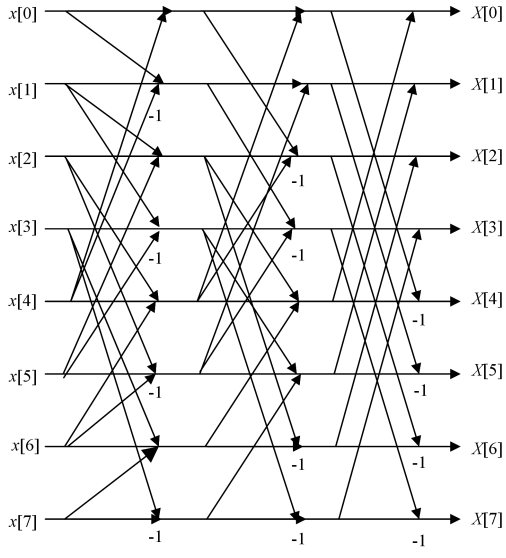


图 3 Stockham 蝶形网络

Fig. 3 Stockham butterfly network

新型蝶形网络 Stockham 并没有改变 FFT 本身的定义以及蝶形网络的生成方式,而是通过重新计算所需数据的地址来实现。相对于传统网络,Stockham 蝶形网络本身具有 3 方面的优势。

(1)消除“位反转”操作

输入数据和输出数据都是自然序列。蝶形输入消除了“位反转”置换操作,减少了内存顺序调整的操作,从而有利于提升 FFT 算法的性能。

(2)SIMD 友好

SIMD 是指一条指令作用在多个数据上面,本文使用 AVX2 指令集进行优化。其优化思路是同时对同一个 section 内的蝶形使用 SIMD 优化技术进行处理,即蝶形网络每级 stage 的每个 section 在汇编实现过程中可同时计算 4 个 butterfly。基于此,我们可以完美地使用 SMID 汇编实现蝶形网络。

图 4 中对传统计算蝶形的方式和 SIMD 优化计算蝶形的方式进行了对比。可以看出,传统计算方式一次取 1 个数,蝶形计算一次只能计算一个蝶形;SIMD 优化之后的计算中,由于连续 4 个蝶形的各个输入数据和输出数据在内存中都是连续存储的,因此可以一次性将 4 个蝶形的每个分量分别取到对应的向量寄存器中,并完成 4 个蝶形的计算。SIMD 优化在减少代码体积的同时,增加了处理器的吞吐量,实现了数据并行,大大提高了算法的计算性能。

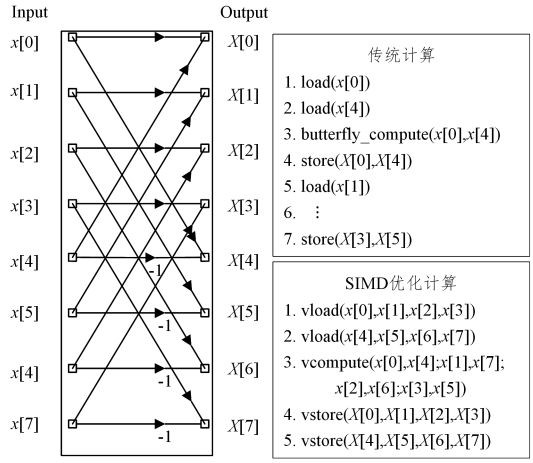


图 4 SIMD 优化

Fig. 4 SIMD optimization

(3)完美支持混合基

由于消除了“位反转”置换操作,因此不同 radix 的蝶形网络可以完美地融合在一起,从而实现了对混合基的完美支持。

本文实现的 Stockham 蝶形网络主要从两个方面对蝶形网络进行了优化:第一级 FFT 网络计算单独优化和小规模单独优化。

(1)第一级 FFT 网络计算单独优化

第一级 FFT 网络存在两个特性:

1)第一级蝶形网络对应的旋转因子为 W_N^0,其值为 1,因此,第一级网络计算可以去除输入数据和旋转因子相乘的操作,使其不仅降低了运算开销,而且减少了访存操作。

2)对于第一级蝶形计算的结果,每个蝶形的分量在内存中是连续存储的,并不适合使用 SIMD 指令直接存储,需要先对计算结果进行转置。因此,第一级 FFT 网络计算可以单独优化。

(2)小规模单独优化

虽然可以用通用规模来计算相对较小规模(如输入数据规模为 6,10,16 等)的数据,但是由于输入数据规模较小,此时的输入数据和旋转因子可存储在 Cache 中,且蝶形网络结构非常清晰,因此可以直接利用 FFT 的各定义和性质来编写对应的 FFT 程序,无须调用大规模的 FFT 函数来计算。经实验,当蝶形网络不超过三级时,都可以单独优化,且性能也获得了较大的提升。

4.2 蝶形算法优化

由第 3 节可知，离散傅里叶变换的实质是 DFT 矩阵与输入向量进行矩阵向量乘。蝶形网络中 radix＝N 的蝶形计算的实质是数据规模为 N 的 DFT 计算。那么，基 N(radix-N)的蝶形计算式可表示为：

$$\begin{cases} X(0)=x_0+x_1+\cdots+x_{N-1} \\ X(1)=x_0+W_N^1x_1+\cdots+W_N^{N-1}x_{N-1} \\ X(2)=x_0+W_N^2x_1+\cdots+W_N^{2(N-1)}x_{N-1} \\ \quad\vdots \\ X(N-1)=x_0+W_N^{N-1}x_1+\cdots+W_N^{(N-1)(N-1)}x_{N-1} \end{cases} \tag{6}$$

下面从蝶形计算的角度来分析旋转因子在复平面上的分布。对于复数来说，以实部为横坐标轴、虚部为纵坐标轴组成的虚拟二维坐标轴被称为复坐标轴。以 radix-6 为例，可看出其均匀分布在以原点为单位的复平面上，如图 5 所示。

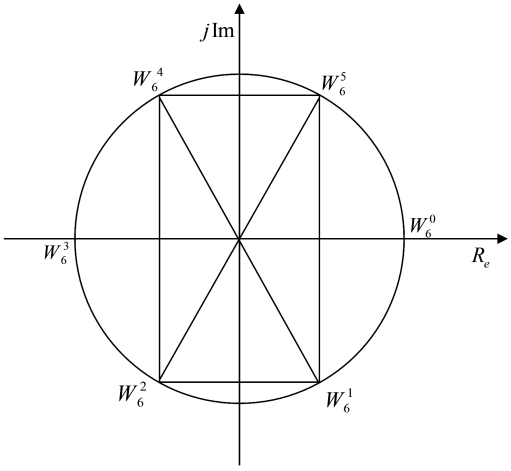


图 5 radix-6 旋转因子复平面分布图

Fig. 5 Radix-6 twiddles complex plane distribution

依据 twiddles 的周期性($W_N^{k+n}=W_n^k$)和对称性($W_N^{k+N/2}=-W_n^k$)对 radix-6 的 DFT 矩阵进行优化，优化后的 radix-6 DFT 矩阵如式(7)所示：

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W_6^1 & W_6^2 & -1 & W_6^{-2} & W_6^{-1} \\ 1 & W_6^2 & W_6^{-2} & 1 & W_6^2 & W_6^{-2} \\ 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & W_6^{-2} & W_6^2 & 1 & W_6^{-2} & W_6^2 \\ 1 & W_6^{-1} & W_6^{-2} & -1 & W_6^2 & W_6^1 \end{bmatrix} \tag{7}$$

因为 W_6^1 和 W_6^2 虚部相同而实部相反，W_6^1 和 W_6^4 实部和虚部都相反，W_6^1 和 W_6^5 实部相同而虚部相反，所以 W_6^2，W_6^4，W_6^5 可用 W_6^1 表示，进而可得出优化后的蝶形计算式：

$$\begin{cases} X(0)=x_0+x_1+x_2+x_3+x_4+x_5 \\ X(1)=x_0+W_6^1x_1+W_6^2x_2-x_3+W_6^{-2}x_4+W_6^{-1}x_5 \\ X(2)=x_0+W_6^2x_1+W_6^{-2}x_2+x_3+W_6^2x_4+W_6^{-2}x_5 \\ X(3)=x_0-x_1+x_2-x_3+x_4-x_5 \\ X(4)=x_0+W_6^{-2}x_1+W_6^2x_2+x_3+W_6^{-2}x_4+W_6^2x_5 \\ X(5)=x_0+W_6^{-1}x_1+W_6^{-2}x_2-x_3+W_6^2x_4+W_6^1x_5 \end{cases} \tag{8}$$

化简式(8)：

$$\begin{cases} X(0)=(x_0+x_3)+[(x_1+x_5)+(x_2+x_4)] \\ X(3)=(x_0-x_3)+[(x_1+x_5)-(x_2+x_4)] \\ X(1)=(x_0-x_3)+W_6^1.r*[(x_1+x_5)-(x_2+x_4)]+ \\ \qquad W_6^1.i*[(x_1-x_5)+(x_2-x_4)] \\ X(5)=(x_0-x_3)+W_6^1.r*[(x_1+x_5)-(x_2+x_4)]- \\ \qquad W_6^1.i*[(x_1-x_5)+(x_2-x_4)] \\ X(2)=(x_0+x_3)-W_6^1.r*[(x_1+x_5)+(x_2+x_4)]+ \\ \qquad W_6^1.i*[(x_1-x_5)-(x_2-x_4)] \\ X(4)=(x_0+x_3)-W_6^1.r*[(x_1+x_5)+(x_2+x_4)]- \\ \qquad W_6^1.i*[(x_1-x_5)-(x_2-x_4)] \end{cases} \tag{9}$$

令：

$S_1=(x_1+x_5)+(x_2+x_4)$

$S_2=(x_1+x_5)-(x_2+x_4)$

$S_3=(x_1-x_5)+(x_2-x_4)$

$S_4=(x_1-x_5)-(x_2-x_4)$

得出最后的简化式为：

$$\begin{cases} X(0)=(x_0+x_3)+S_1 \\ X(3)=(x_0-x_3)+S_2 \\ X(1)=(x_0-x_3)+W_6^1.r*S_2+W_6^1.i*S_3 \\ X(5)=(x_0-x_3)+W_6^1.r*S_2-W_6^1.i*S_3 \\ X(2)=(x_0+x_3)-W_6^1.r*S_1+W_6^1.i*S_4 \\ X(4)=(x_0+x_3)-W_6^1.r*S_1-W_6^1.i*S_4 \end{cases} \tag{10}$$

简化过程中，通过提取公共项，并重复利用初始化过的公共项，可以减少大量的浮点计算开销和代码开销。此优化方式最大限度地减少了冗余操作，降低了算法的时间复杂度，故可将利用了旋转因子性质的优化方式推广到 radix-10，得出 radix-10 的 DFT 优化矩阵(11)和 radix-10 的最优计算式(12)。

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W_{10}^1 & W_{10}^2 & W_{10}^3 & W_{10}^4 & -1 & W_{10}^{-4} & W_{10}^{-3} & W_{10}^{-2} & W_{10}^{-1} \\ 1 & W_{10}^2 & W_{10}^4 & W_{10}^{-4} & W_{10}^{-2} & 1 & W_{10}^2 & W_{10}^4 & W_{10}^{-4} & W_{10}^{-2} \\ 1 & W_{10}^3 & W_{10}^{-4} & W_{10}^{-1} & W_{10}^2 & -1 & W_{10}^{-2} & W_{10}^1 & W_{10}^4 & W_{10}^{-3} \\ 1 & W_{10}^4 & W_{10}^{-2} & W_{10}^2 & W_{10}^{-4} & 1 & W_{10}^4 & W_{10}^{-2} & W_{10}^2 & W_{10}^{-4} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & W_{10}^{-4} & W_{10}^2 & W_{10}^{-2} & W_{10}^4 & 1 & W_{10}^{-4} & W_{10}^2 & W_{10}^{-2} & W_{10}^4 \\ 1 & W_{10}^{-3} & W_{10}^4 & W_{10}^1 & W_{10}^{-2} & -1 & W_{10}^2 & W_{10}^{-1} & W_{10}^{-4} & W_{10}^3 \\ 1 & W_{10}^{-2} & W_{10}^{-4} & W_{10}^4 & W_{10}^2 & 1 & W_{10}^{-2} & W_{10}^{-4} & W_{10}^4 & W_{10}^2 \\ 1 & W_{10}^{-1} & W_{10}^{-2} & W_{10}^{-3} & W_{10}^{-4} & -1 & W_{10}^4 & W_{10}^3 & W_{10}^2 & W_{10}^1 \end{bmatrix} \tag{11}$$

简化后的 radix-10 计算式为：

$$\begin{cases} X(0)=(x_0+x_5)+S_1+S_5 \\ X(5)=(x_0-x_5)+S_2+S_6 \\ X(1)=(x_0-x_5)+W_{10}^1.r*S_2+W_{10}^1.i*S_3+ \\ \quad W_{10}^2.r*S_6+W_{10}^2.i*S_7 \\ X(9)=(x_0-x_5)+W_{10}^1.r*S_2-W_{10}^1.i*S_3+ \\ \quad W_{10}^2.r*S_6-W_{10}^2.i*S_7 \\ X(2)=(x_0+x_5)-W_{10}^1.r*S_5+W_{10}^1.i*S_8+ \\ \quad W_{10}^2.r*S_1+W_{10}^2.i*S_4 \\ X(8)=(x_0+x_5)-W_{10}^1.r*S_5-W_{10}^1.i*S_8+ \\ \quad W_{10}^2.r*S_1-W_{10}^2.i*S_4 \\ X(3)=(x_0-x_5)-W_{10}^1.r*S_6-W_{10}^1.i*S_7- \\ \quad W_{10}^2.r*S_2+W_{10}^2.i*S_3 \\ X(7)=(x_0-x_5)-W_{10}^1.r*S_6+W_{10}^1.i*S_7- \\ \quad W_{10}^2.r*S_2-W_{10}^2.i*S_3 \\ X(4)=(x_0+x_5)-W_{10}^1.r*S_1+W_{10}^1.i*S_4+ \\ \quad W_{10}^2.r*S_5-W_{10}^2.i*S_8 \\ X(6)=(x_0+x_5)-W_{10}^1.r*S_1-W_{10}^1.i*S_4+ \\ \quad W_{10}^2.r*S_5+W_{10}^2.i*S_8 \end{cases} \tag{12}$$

其中：

$$S_1=(x_1+x_9)+(x_4+x_6)$$
$$S_2=(x_1+x_9)-(x_4+x_6)$$
$$S_3=(x_1-x_9)+(x_4-x_6)$$
$$S_4=(x_1-x_9)-(x_4-x_6)$$
$$S_5=(x_2+x_8)+(x_3+x_7)$$
$$S_6=(x_2+x_8)-(x_3+x_7)$$
$$S_7=(x_2-x_8)+(x_3-x_7)$$
$$S_8=(x_2-x_8)-(x_3-x_7)$$

至此，radix-6 和 radix-10 的 DFT 矩阵和计算式为最简，即从算法计算的角度将算法优化到最佳状态。

4.3 蝶形汇编优化

(1)指令选择与指令重排

指令选择优化时，选择延迟低、吞吐量高的指令，如选择 vfnmadd231ps 和 vfmadd231ps 这样的乘加指令，以在减少对源操作数频繁累加的基础上，减少寄存器的占用。

指令重排是指相邻两条指令没有依赖关系，避免流水线空泡。本文使用的硬件架构中有 8 个程序调用发射端口，指令系统为寄存器-寄存器型，在寄存器中进行蝶形的计算。

顺序指令和重排指令实现的 radix-6 蝶形计算的对比图 6 直观地展示了两种方式的思想。指令重排后的运算结果和顺序执行的结果是一致的，但性能却得到了很大的提升。

理论上，进行蝶形运算时，按照顺序先从内存中取 6 个数(in0－in5)放入相应的浮点寄存器，之后按照 radix-6 优化后的蝶形算法式(10)依次计算出(x_0+x_3)，(x_0-x_3)，(x_1+x_5)，(x_1-x_5)，(x_2+x_4)和(x_2-x_4)。但是，由于硬件架构中的运算单元没有被充分利用，因此会出现流水线空泡的现象。在保证指令之间没有依赖关系的情况下，radix-6 的蝶形计算顺序可以调整为先从内存取数 in0 和 in3 到寄存器中，接着对取出的数进行(x_0+x_3)和(x_0-x_3)操作；以此类推，先取数 in1 和 in5，计算(x_1+x_5)和(x_1-x_5)；取数 in2 和 in4，计算(x_2+x_4)和(x_2-x_4)。

同理，存数也可以用指令重排。首先计算式(10)中的 S_1，S_2，S_3，S_4；然后计算 $X(0)$和 $X(3)$，$X(1)$和 $X(5)$，$X(2)$和 $X(4)$；最后按照 $X(0)-X(5)$的自然序列，使用指令 vmovups 将结果存入内存中。

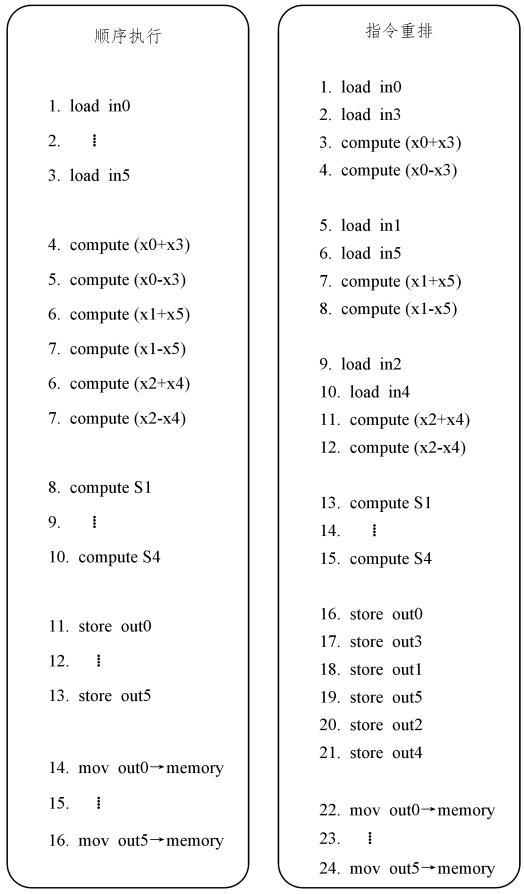


图 6 顺序执行与指令重排的对比

Fig. 6 Comparison of sequential execution and instruction rearrangement

(2)存储转置

并不是所有的计算输出结果都是顺序存储在寄存器或者内存中。下面以 4 * 4 矩阵转置为例，结合指令来具体阐述存储转置。

图 7 中具体显示了 4 * 4 矩阵在转置过程中，数据在寄存器中的存储状态。从图中可以看出，转置前数据的存储不是连续的，而是有间隔地存储在寄存器中。为了使寄存器中的数据能连续、无间隔地储存，就需要对这些数据进行转置操作。转置的思想是首先取数放入暂存，然后在暂存中对矩阵进行转置，最后将转置后的矩阵存入内存。

由于存储数据的寄存器是按照从右到左，依次是低位、高位、低位、高位的顺序存储数据，因此在转置过程中低位和高位所用到的指令是不同的。我们使用指令 vunpcklpd 将 out0 和 out1 的低位取出放入 S_0 中，同样，使用指令 vunpckhpd 将 out0 和 out1 的高位取出放入 S_2 中；对 out2 和 out3 进行同样的操作。最后，使用重排指令 vperm2f128 将数据重新排列，

使之顺序存储在寄存器中，之后将排列好的矩阵存入内存中。

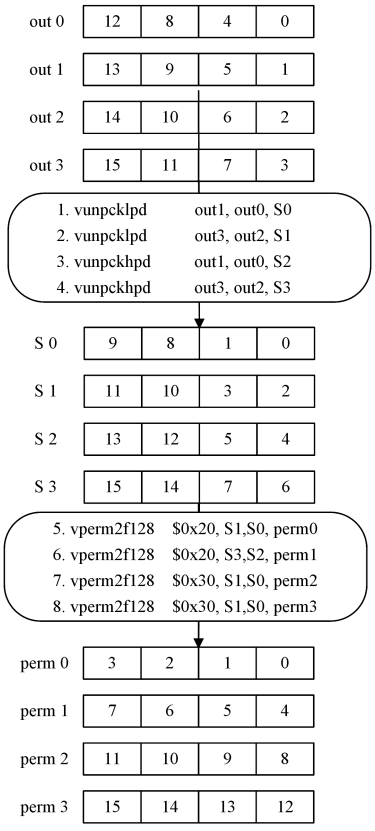


图 7 4 * 4 矩阵的转置流程

Fig. 7 Transposition process of 4 * 4 matrix

(3)register 的使用

寄存器是非常重要和稀缺的片上存储资源，合理对寄存器进行使用和分配对性能影响至关重要。在高级语言编写的程序中，一般由编译器负责寄存器的分配。但在本文的实现中，为提高算法性能，蝶形计算 kernel 都采用手工汇编编写。为此，需要制定合理的寄存器分配策略，从而最大限度地利用寄存器资源，提升程序性能[25-26]。

本文采用的硬件架构中有 16 个长度为 256 bit 的浮点寄存器(XMM0-XMM15)。每个 YMM 寄存器有高位和低位之分，所使用的架构具有兼容性，寄存器的第 128 位又可以当作 128 位的浮点寄存器(XMM0-XMM15)。以 radix-6 实现 kernel 为例，来说明寄存器的使用情况。

Radix-6 的 kernel 主要是由 4 组数据来进行计算实现，分别是输入数据 IN、旋转因子 TW、暂存 Scratch、输出数据 OUT。按照每组数据的规模来分配寄存器，其中输入数据需要 6 个 YMM；旋转因子 TW 的实部和虚部各需 5 个 YMM，所以旋转因子 TW 总共需要 10 个 YMM；暂存 Scratch 需要 10 个 YMM；输出数据需要 6 个寄存器。

综上，总共需要 32 个浮点寄存器，而 X86 平台只提供 16 个浮点寄存器，这样会出现寄存器不够用的情况。因此，寄存器的复用就显得尤为重要。

首先，给输入数据 in0－in5 分配浮点寄存器 YMM0－YMM5；给旋转因子 TW 的实部 re1－re5 和虚部 im1－im5 分配寄存器 YMM6－YMM15。

其次，由于在 kernel 的实现过程中是先将取出的输入数据 IN 和从寄存器中取出的旋转因子的值相乘后放入原 IN 使用的寄存器中，因此旋转因子所占用的寄存器就会释放掉里面的值，这时，我们可以对暂存 Scratch0－Scratch9 复用寄存器 YMM6－YMM15 来进行 butterfly 的计算实现。

最后，经过 butterfly 的计算，输入数据 IN 所占用的寄存器全部释放掉，输出数据 OUT 就可以复用寄存器 YMM0－YMM5。

5 实验分析

5.1 测试环境

(1)硬件环境搭建

本文采用 Intel(R) Xeon(R) CPU E5-2670 v3 作为性能测试平台。Intel(R) Xeon(R) CPU 使用的是 X86 架构。本文的测试环境配置如表 1 所列。

表 1 实验的硬件环境配置

Table 1 Experimental hardware environment configuration

操作系统	Ubuntu 14.04.5 LTS
L1 cache 大小/kB	32
寄存器长度/bit	256
寄存器数量	16
编译器版本	gcc version 4.8.4

(2)软件环境搭建

目前，应用最广的 FFT 算法库有 FFTW 和 Intel® Math Kernel Library (Intel® MKL)。对此，本实验采用最新 FFTW 的版本库 FFTW3.3.8 和 Intel? MKL 作为高性能 FFT 的对比库。本文将实现的 Cooley-Tukey FFT 高性能算法库命名为 AutoFFT。

5.2 性能分析

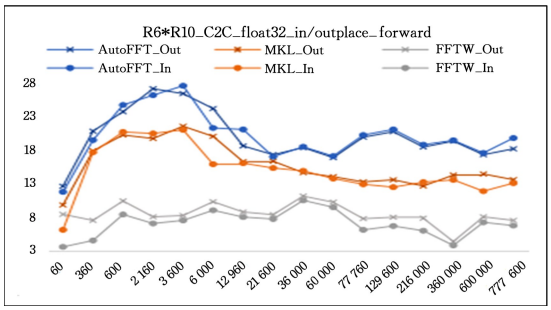
本文的数据测试规模为 $6^m * 10^n$。测试维度为一维，测试序列的输入和输出都是复数序列。本实验主要是将 AutoFFT 与目前流行的 MKL 库和 FFTW 库进行比较。图 8 中，横坐标表示实验的数据规模；纵坐标表示实验性能，其单位为 Gflops(Giga Floating-point Operation per Second)，即每秒所执行的浮点计算次数；性能走势中，蓝色线代表 AutoFFT 性能，橘色线表示 MKL 性能，灰色线表示 FFTW 性能。观察各个库的性能走势可以看出，新实现的算法库的性能整体高于目前流行的其他两个算法库。

(1)单精度浮点 FLOAT 性能的对比

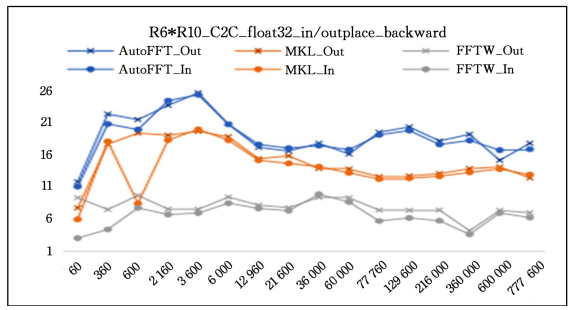
如图 8(a)所示，AutoFFT 相比于 MKL 的最大加速比为 90%，最小加速比为 10%，平均加速比为 38%；相比于 FFTW 的最大加速比为 418%，最小加速比为 76%，平均加速比为 206%。从图 8(b)看出，AutoFFT 相比于 MKL 的最大加速比为 138%，最小加速比为 13%，平均加速比为 40%；相比于 FFTW 的最大加速比为 416%，最小加速比为 77%，平均加速比为 209%。

比较 AutoFFT，MKL 和 FFTW 的性能走势可以看出，三者在性能曲线走势方面大体一致。从整体来看，AutoFFT 的性能曲线走势最高，其次是 MKL，FFTW 的性能最低。由图

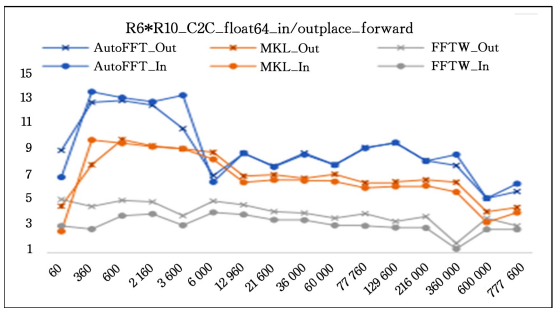
8(a)和图 8(b)可知,当输入数据规模在 60～3 600 时,AutoFFT 的性能呈逐步递增的趋势,主要原因是当输入数据的规模较小时,数据能够存储在 Cache 中,增加 Cache 命中率的同时,利用时间局部性和空间局部性来提高性能,减少访存开销。在这种情况下,输入小规模数据的性能高于输入大规模数据的性能。当输入数据规模在 6000～777600 时,算法的整体性能趋于稳定的走势。AutoFFT 取得高性能的原因有以下几点。



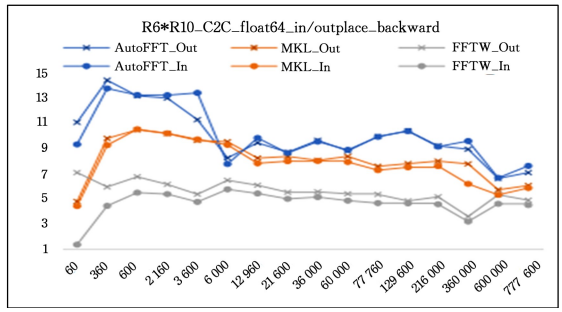
(a)1D C2C FLOAT 正变换性能对比



(b)1D C2C FLOAT 逆变换性能对比



(c)1D C2C DOUBLE 正变换性能对比



(d)1D C2C DOUBLE 逆变换性能对比

图 8 1D C2C FFT 性能的对比(电子版为彩色)

Fig. 8 Comparison of 1D C2C FFT performance

1)消除“位反转”操作

减少了对全部数据的读写操作,拓宽了网络架构的可扩展性,方便了混合基的搭建,节约了内存开销,性能得到了一定程度的提高。

2)蝶形计算式简化到最优

最优的蝶形计算式通过较小体积容量的代码来实现,使得冗余循环的计算操作大幅减少,处理器内部的运算单元被有效利用,从而大幅提升了算法的性能。

3)SIMD 友好

算法实现中,Stockham 蝶形网络与 SIMD 相结合,采用算法设计化一为多的方法,使得各处理器能同时执行;将小问题组合解决,以提高性能并减少任务开销。相邻两条指令间接触依赖,减少了代码体积。关键是使用多处理器执行并发计算,增加了处理器的吞吐量,从而实现算法在数据和空间上的并行性,加速了算法性能的提升。

4)采用精简指令和指令重排方法

AVX2 提供的精简指令将数据宽度扩展到 256 位;支持浮点乘法累积、向量-向量移位、跨距访存、可变移位等操作。采用与算法相对应的精简指令,对提升算法性能的重要性和影响不言而喻。为了避免流水线空泡,当相邻指令间不存在依赖关系时,利用指令重排来提升流水线并行工作的效率。

5)数据的存储转置

矩阵的存储转置不仅增加了数据的连续性,也提高了 Cache 的利用率,减少了访存次数,算法的性能无疑得到了提升。

6)寄存器复用

依据蝶形运算中数据的组成部分,在保证每一组数据有与之相吻合的寄存器数目,且每组数据所用寄存器之间不会出现依赖关系的情况下,片上稀缺资源寄存器的合理分配策略尤为重要。寄存器复用的方法最大限度地利用了寄存器资源,程序性能得到了进一步的提升。

(2)双精度浮点 DOUBLE 性能的对比

如图 8(c)所示,AutoFFT 相比于 MKL 的最大加速比为 81%,最小加速比为 0.8%,平均加速比为 28%;相比于 FFTW 的最大加速比为 200%,最小加速比为 35%,平均加速比为 106%。从图 8(d)看出,AutoFFT 相比于 MKL 的最大加速比为 61%,最小加速比为 0.83%,平均加速比为 21%;相比于 FFTW 的最大加速比为 143%,最小加速比为 25%,平均加速比为 76%。

比较 AutoFFT,MKL 和 FFTW 的性能走势可以看出,三者在性能曲线走势方面大体一致。从整体上看,AutoFFT 的性能曲线走势最高,其次是 MKL,FFTW 的性能最低。但是,在图 8(c)和图 8(d)中出现了异常点,即蓝色性能点在橘色性能点之下。出现这一现象的原因主要有以下方面。

1)Cache 命中率低

相对于 MKL,当数据规模为 6 000 时,Cache 命中率低,从而增加了访存开销且延迟变高,降低了算法在该点的性能。

2)数据预取

未做数据的预取,使得大量的无用数据被取回,导致带宽浪费;预取时间不恰当,数据在未使用前就被替换或者是要使

用时还没有取回,这些都可能会对算法的性能产生影响。

结束语 本文在原有 FFT 的基础上突破了 FFT 算法在硬件平台上的性能瓶颈,从算法优化和汇编优化两方面着手,形成了 FFT 算法高性能实现和优化的技术体系,实现了 X86 平台上的高性能 FFT 库,其性能整体高于目前被广泛应用的 MKL 库和 FFTW 库。未来将主要在现有基础上突破 FFT 算法更多的性能瓶颈,实现一套技术方法体系更完整、实现和优化更成熟的高性能 FFT 算法库。

参 考 文 献

[1] COOLEY J W,TUKEY J W. An algorithm for the machine calculation of complex Fourier series[J]. Mathematics of Computation,1965,19(90):297-301.

[2] COCHRAN W T,COOLEY J W,FAVIN D L,et al. What is the fast Fourier transform? [J]. Proceedings of the IEEE,1967,55(10):1664-1674.

[3] DUHAMEL P,VETTERLI M. Fast Fourier transforms:a tutorial review and a state of the art[J]. Signal Processing,1990,19(4):259-299.

[4] STRANG,GILBERT. "Wavelets"[J]. American Scientist,1994,2(3):250-255.

[5] KENT R D,READ C. Acoustic Analysis of Speech[M]. Singular Publishing Group,2002.

[6] DONGARRA J,SULLIVAN F. Guest editors' introduction:The top 10 algorithms[J]. Computing in Science & Engineering,2000,2(1):22-23.

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

[8] Intel® Math Kernel Library Release Notes and New Features [EB/OL]. (2017-09-10) [2019-09-19]. https://software.intel.com/en-us/articles/intel-math-kernel-library-release-notes-and-new-features.

[9] Developer Reference for Intel® Math Kernel Library - C [EB/OL]. (2019-09-10) [2019-09-10]. https://software.intel.com/zhcn/download/developer-reference-for-intel-math-kernel-library-c.

[10] FRIGO M,JOHNSON S G. The Design and Implementation of FFTW3[C]// Proceedings of the IEEE. 2005,216-231.

[11] JOHNSON S G,FRIGO M. Implementing FFTs in practice[J]. https://scholar.google.com/scholar?q=Implementing+FFTs+in+practice&hl=zh-CN&as_sdt=0&as_vis=1&oi=scholart.

[12] MOHAMED K,ALI F H H M,ARIFFIN S,et al. An Improved AES S-box Based on Fibonacci Numbers and Prime Factor[J]. IJ Network Security,2018,20(6):1206-1214.

[13] JOHNSON S G,FRIGO M. A modified split-radix FFT with fewer arithmetic operations[J]. Signal Processing,2007,55(1):111-119.

[14] DUHAMEL P,HOLLMANN H. Split radixFFT algorithm[J]. Electronics letters,1984,20(1):14-16.

[15] BADAR S,DANDEKAR D R. High speed FFT processor design using radix? 4 pipelined architecture[C]// 2015 International Conference on Industrial Instrumentation and Control (ICIC). IEEE,2015:1050-1055.

[16] NUGTEREN C. Clblast:A tuned opencl BLAS library[C]// Proceedings of the International Workshop on OpenCL. ACM,2018:5.

[17] NAKATA M. Basics and Practice of Linear Algebra Calculation Library BLAS and LAPACK[M]// The Art of High Performance Computing for Computational Science. Singapore:Springer,2019:83-112.

[18] BUJANOVIĆ Z,DRMAČ Z. New robust ScaLAPACK routine for computing the QR factorization with column pivoting[J]. arXiv:1910.05623,2019.

[19] MULTIPLICATION M,SICARD-RAMÍREZ A. Intel R G Math Kernel Library[J]. Universidad EAFIT,2018,2018:3-21.

[20] INTEL R. Math kernel library[J/OL]. https://scholar.google.com.hk/scholar?hl=zh-CN&as_sdt=0%2C5&as_yhi=2009&q=Intel+Math+Kernel+Library&btnG=.

[21] FOG A. Intel's "cripple AMD" function[EB/QL]. (2009-12-30) [2019-11-3]. https://www.agner.org/optimize/blog/read.php?i=49#49.

[22] YANG M R. MKL has bad performances on an AMD cpu[EB/QL]. (2019-02-02)[2019-11-03]. https://sites.google.com/a/uci.edu/mingru-yang/programming/mkl-has-bad-performance-on-an-amd-cpu.

[23] The NVIDIA CUDA Fast Fourier Transform library [EB/OL]. (2019-08-14) [2019-08-14]. https://developer.nvidia.com/cufft.

[24] ALGLIB User Guide online [EB/OL]. (2019-02-21) [2019-02-21]. http://www.alglib.net/fasttransforms/fft.php.

[25] EISL J,GRIMMER M,SIMON D,et al. Trace-based Register Allocation in a JIT Compiler[C]// Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform:Virtual Machines,Languages,and Tools. ACM,2016:14.

[26] EISL J,LEOPOLDSEDER D,MÖSSENBÖCK H. Parallel trace register allocation[C]// Conference on Managed Languages & Runtimes (ManLang'18). 2018.

GONG Tong-yan, born in 1992, postgraduate, not member of China Computer Federation (CCF). Her main research interests include parallel processing and high-performance computing.

ZHANG Guang-ting, born in 1987, postgraduate, is member of China Computer Federation (CCF). Her main research interests include parallel algorithms and big data.