doi:10.11887/j.cn.201502015 http://journal.nudt.edu.cn

使用融合乘加加速快速傅里叶变换计算的向量化方法*

刘 仲,陈海燕,向宏卫
(国防科技大学 计算机学院,湖南 长沙 410073)

摘 要:融合乘加指令加速快速傅里叶变换计算的向量化方法,通过变换快速傅里叶变换的蝶形单元运算流程,将传统计算方式中独立的乘法和加法操作组合成次数更少的融合乘加操作,使得时间抽取法基2快速傅里叶变换算法的蝶形单元计算的实数浮点操作由原来的10次乘(加)操作减少到6次融合乘加操作,时间抽取法基4快速傅里叶变换算法的蝶形单元计算的实数浮点操作由原来的34次乘(加)操作减少到24次融合乘加操作;优化了蝶形因子的向量访问,减少存储开销。实验结果表明,提出的方法能够显著加速快速傅里叶变换的计算,取得高效的计算性能和效率。

关键词:快速傅里叶变换;融合乘加;向量化;向量处理器
中图分类号:P391.4 **文献标志码**:A **文章编号**:1001-2486(2015)02-072-07

Vectorization of accelerating fast Fourier transform computation based on fused multiply-add instruction

LIU Zhong, *CHEN Haiyan*, *XIANG Hongwei*
(College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: A vectorization of accelerating fast Fourier transform computation based on fused multiply-add instruction was presented. Separate multiplication and addition operations in conventional computation were manipulated into less fused multiply-add operations by transforming process of fast Fourier transform butterfly computation, which decreased the real floating-point operations of radix-2 decimation in time fast Fourier transform butterfly computation from 10 multiplication (addition) operations to 6 multiply-add operations and decreased the real floating-point operations of radix-4 decimation in time fast Fourier transform butterfly computation from 34 multiplication (addition) operations to 24 multiply-add operations. Vector data access on twiddle factors was optimized to reduce memory cost. Experimental results show that the presented method can greatly accelerate fast Fourier transform computation and achieve efficient performance and efficiency.

Key words: fast Fourier transform; fused multiply-add; vectorization; vector processor

现有的许多处理器体系结构提供融合乘加(Fused Multiply-Add, FMA)指令,如Intel的Itanium,IBM的Power处理器等。给定3个输入操作数a,b,c,使用一条FMA指令即可实现y=±a±(b×c)中的任何一个操作。FMA指令对于数值计算亦具有重要的意义。因为,在软件上,一条FMA指令在执行时间上和一条乘法或加法指令几乎一样快;在硬件上,FMA单元通常比分开的乘法器和加法器耗费少;在计算上,FMA指令通过减少舍入操作可以提高数值计算精度。在提供FMA指令的处理器平台上,对于有些数值计算应用,转换到FMA指令是比较直接的,如普通的矩阵和向量乘法、矩阵和矩阵乘法以及成对出现的乘法和加法操作计算。但是,对更多的其他应用就没有那么简单了,需要平衡加法和乘法操作,改造原有的计算方法或流程以适合FMA指令,才能充分发挥FMA指令的作用,加速计算的性能。

向量处理器在单个芯片上集成多个向量处理单元(Vector Processing Element, VPE),每个VPE包含相同的MAC,ALU,BP等多个功能部件,能够在降低面积和功耗的情况下大幅提高整芯片的计算能力,适合处理高密集运算的任务,如矩阵计算、FFT运算、滤波运算等。然而,现有的大量程序和算法是基于单核处理器设计的,很多高密集运算的任务由于算法本身的特性,向量化处理困难,如何针对向量处理器的多运算部件的向量计算的体系结构特点,充分开发各个层次的并行性,高效地向量化这些算法是当前面临的主要困难[1-2]。

* **收稿日期**:2014-06-12
基金项目:国家自然科学基金资助项目(61133007,61472432)
作者简介:刘仲(1971—),男,湖南邵东人,副研究员,博士,E-mail:zhongliu@nudt.edu.cn

快速傅里叶变换(Fast Fourier Transform, FFT)作为时域和频域的基本变换工具,在现代信号处理系统领域应用广泛,如雷达、声呐、地震信号分析、频谱分析、语音识别、3G 通信和图像处理等。FFT 能显著减少离散傅里叶变换(Discrete Fourier Transform,DFT)计算的复杂度,对于实时性要求很高的信号处理应用来说,DFT 计算效率越高,信号处理的实时性就越好,因此 FFT 算法的优化方法一直是国内外的研究热点。Pease[3] 采用数学工具 Kronecker 积描述 FFT 算法,方便将 FFT 算法高效地映射到并行计算机上;Linzer[4],Goedecker[5] 和 Karner[6] 等针对 FMA 结构的处理器,分别研究了如何利用 FMA 指令优化 FFT 计算;Voronenko[7] 依据有向无环图和具有一定结构的矩阵因子化算法,提出一种将离散傅里叶变换、离散余弦变换等线性变换转化为 FMA 算法的一般方法;FFTW(Fastest Fourier Transform in the West)[8] 是通用 CPU 平台上广泛使用的 FFT 数学库,移植性好,可以为任意长度和维数的 FFT 自动生成一种最有效的实现方式;Loberiras[9] 等针对 GPU 提出小点数 FFT 的优化技术,受限于纹理存储器的大小,大点数 FFT 的性能不理想;Li[10] 等针对 GPU 提出一种 FFT 自适应优化框架,当点数大于 1K 时,获得接近 CUDA 快速傅里叶变换(CUDA Fast Fourier Transform,CUFFT)的 90% 性能;文献[11]针对 GPU 的大点数 FFT 优化方法达到 CUFFT 性能的 2.1 倍。

在传统的 FFT 计算中,蝶形单元的最终计算往往转化为实数的乘法和加法操作,如时域抽取基 2 FFT 算法的一个蝶形单元计算需要 4 次实数乘法和 6 次实数加法,即需要 10 次实数乘(加)操作;时域抽取基 4 FFT 算法的一个蝶形单元计算需要 12 次实数乘法和 22 次实数加法,即需要 34 次实数乘(加)操作。显然,FFT 计算中的乘、加操作是不平衡的,不能直接利用 FMA 指令来实现蝶形单元的计算,所以传统的 FFT 计算方法不能有效发挥具有 FMA 指令的处理器的计算效率。刘仲等针对 FMA 结构的向量处理器,提出一种基于 FMA 加速 FFT 计算的向量化方法,通过变换 FFT 的蝶形单元运算流程,将传统计算方式中的独立的乘法和加法操作组合成次数更少的 FMA 操作,使得计算一个 DIT 基 2 FFT 算法的蝶形单元所需要的浮点操作减少到 6 次 FMA 操作,计算一个 DIT 基 4 FFT 算法的蝶形单元所需要的浮点操作减少到 24 次 FMA 操作。实验结果表明,提出的方法能够显著加速 FFT 的计算性能。

1　向量处理器 Matrix 的体系结构

如图 1 所示,Matrix 是一款面向高密度计算应用的高性能浮点向量处理器,内核是超长指令字(Very Long Instruction Word, VLIW)体系结构,包括标量处理部件(Scalar Processing Unit, SPU)和向量处理部件(Vector Processing Unit, VPU)。SPU 负责标量任务计算和流控,SPU 和 VPU 可通过共享寄存器交换数据。Matrix 每时钟周期发射 11 条指令,包括 5 条标量指令和 6 条向量指令。指令派发单元对执行包进行识别,并将其中的指令派发到相应的功能单元中执行。VPU 负责向量计算,包括 16 个向量处理单元(Vector Processing Element, VPE),每个 VPE 含一个局部寄存器文件,以及 3 个浮点乘加单元(Float Multiply and Accumulate, FMAC)、1 个 BP 和 2 个 L/S 共 6 个并行功能部件,3 个 FMAC 均支持 FMA 指令。局部寄存器文件包含 64 个 64 位寄存器,所有 VPE 的同一编号的局部寄存器在逻辑上又组成一个 1024 位的向量寄存器。功能部件支持定点和浮点操作,向量指令在各个 VPE 上同时独立运行。向量数据访问单元支持向量数据的 Load/Store,提供大容量阵列向量存储器(Array Memory, AM),每周期同时支持 2 个 Load/Store 指令。

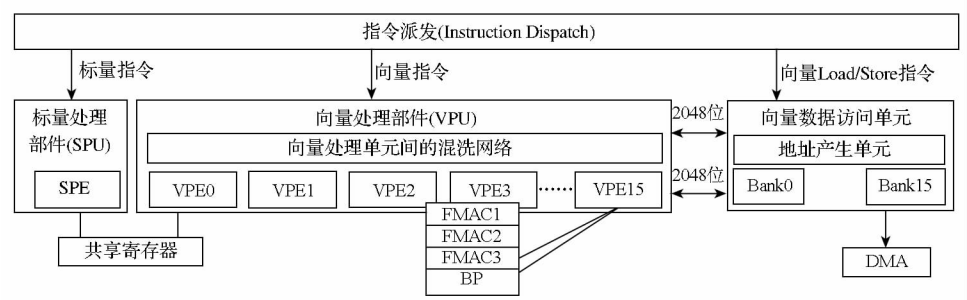

图 1　Matrix 的体系结构

Fig. 1 Architecture of Matrix

2 基于 FMA 的 FFT 向量化优化方法

序列 $x(n)(n=0,\cdots,N-1)$ 的离散傅里叶变换 $X(k)(k=0,\cdots,N-1)$ 定义为：

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{kn}\quad(k=0,\cdots,N-1)$$

其中 $W_N^{kn}=\mathrm{e}^{-\mathrm{j}(2\pi/N)kn}$ 是旋转因子。

依据离散傅里叶变换公式计算的 DFT 复数乘加运算的计算复杂度是 N^2。当 N 比较大时，运算量增长得非常快，使得直接采用 DFT 计算方法难以满足很多实际应用的需求。1965 年 Cooley 和 Turkey 提出一种快速傅里叶变换算法[12]，计算复杂度由原来的 $O(N^2)$ 降到 $O(N\log_2 N)$，可显著地减少运算量。FFT 算法分为时间抽取法（Decimation In Time，DIT）和频率抽取法（Decimation In Frequency，DIF）两种，现以 DIT 方法为例阐述 FFT 的向量化方法。

2.1 传统的 FFT 蝶形单元计算方法

2.1.1 DIT 基 2FFT 的蝶形单元计算方法

当 N 是 2 的整数次方时，DIT 基 2FFT 将输入数据序列 $x(n)$ 进行奇、偶分组：

$$\begin{aligned}X(k)&=\sum_{n=0}^{N-1}x(n)W_N^{kn}=\sum_{n\text{为偶数}}x(n)W_N^{kn}+\sum_{n\text{为奇数}}x(n)W_N^{kn}\\&=\sum_{l=0}^{N/2-1}x(2l)W_N^{k2l}+\sum_{l=0}^{N/2-1}x(2l+1)W_N^{k(2l+1)}\end{aligned}$$

由旋转因子的周期性特性易知：

$$W_N^{2kl}=W_{N/2}^{kl},\ W_N^{k+N/2}=-W_N^k,\ W_N^{k+N}=W_N^k$$

令 $a(l)=x(2l)$，$b(l)=x(2l+1)$，则序列 $X(k)$ 划分为 2 个长度为 $N/2$ 的子序列：

$$\begin{cases}X(k)=A(k)+W_N^kB(k)\\X(k+\dfrac{N}{2})=A(k)+W_N^{k+\frac{N}{2}}B(k)=A(k)-W_N^kB(k)\end{cases}\tag{1}$$

图 2 是 DIT 基 2FFT 的蝶形单元运算流程图，DIT 基 2FFT 的每次蝶形单元运算需要 1 次复数乘法，2 次复数加法，转变实数计算即为 4 次实数乘法和 6 次实数加法，即需要 10 次实数乘（加）操作。

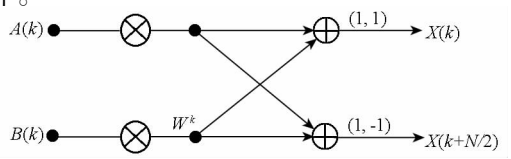

图 2 DIT 基 2 FFT 的蝶形单元运算流程图

Fig. 2 Radix－2 DIT FFT butterfly diagram

2.1.2 DIT 基 4FFT 的蝶形单元计算方法

当 N 是 4 的整数次方时，DIT 基 4FFT 将输入数据序列 $x(n)$ 按模 4 后的余数分组：

$$\begin{aligned}X(k)&=\sum_{n=0}^{N-1}x(n)W_N^{kn}\\&=\sum_{l=0}^{N/4-1}x(4l)W_N^{k4l}+\sum_{l=0}^{N/4-1}x(4l+1)W_N^{k(4l+1)}+\\&\quad\sum_{l=0}^{N/4-1}x(4l+2)W_N^{k(4l+2)}+\sum_{l=0}^{N/4-1}x(4l+3)W_N^{k(4l+3)}\\&=\sum_{l=0}^{N/4-1}x(4l)W_{N/4}^{kl}+W_N^k\sum_{l=0}^{N/4-1}x(4l+1)W_{N/4}^{kl}+\\&\quad W_N^{2k}\sum_{l=0}^{N/4-1}x(4l+2)W_{N/4}^{kl}+W_N^{3k}\sum_{l=0}^{N/4-1}x(4l+3)W_{N/4}^{kl}\end{aligned}$$

由旋转因子的周期性特性易知：

$$W_N^{4kl}=W_{N/4}^{kl},\ W_N^{k+N/4}=-\mathrm{j}W_N^k,$$

$$W_N^{k+2N/4}=-W_N^k,\ W_N^{k+3N/4}=\mathrm{j}W_N^k,\ W_N^{k+N}=W_N^k$$

令 $a(l)=x(4l)$，$b(l)=x(4l+1)$，$c(l)=x(4l+2)$，$d(l)=x(4l+3)$，则序列 $X(k)$ 划分为 4 个长度为 $N/4$ 的子序列：

$$\begin{cases}X(k)=A(k)+W_N^kB(k)+W_N^{2k}C(k)+W_N^{3k}D(k)\\X(k+\dfrac{N}{4})=A(k)-\mathrm{j}W_N^kB(k)-W_N^{2k}C(k)+\mathrm{j}W_N^{3k}D(k)\\X(k+\dfrac{2N}{4})=A(k)-W_N^kB(k)+W_N^{2k}C(k)-W_N^{3k}D(k)\\X(k+\dfrac{3N}{4})=A(k)+\mathrm{j}W_N^kB(k)-W_N^{2k}C(k)-\mathrm{j}W_N^{3k}D(k)\end{cases}\tag{2}$$

图 3 是 DIT 基 4FFT 的蝶形单元运算流程图，DIT 基 4FFT 的每次蝶形单元运算需要 3 次复数乘法，8 次复数加法，转变实数计算即为 12 次实数乘法和 22 次实数加法，即需要 34 次实数乘（加）操作。

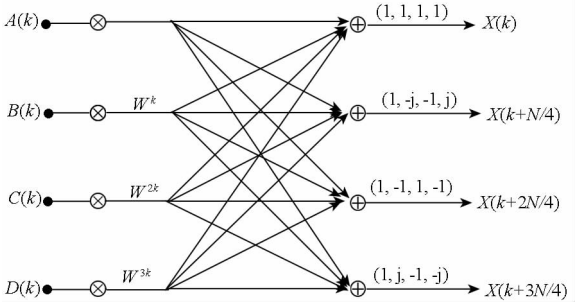

图 3 DIT 基 4 FFT 的蝶形单元运算流程图

Fig. 3 Radix－4 DIT FFT butterfly diagram

2.2 FMA 优化的 FFT 蝶形单元运算

2.2.1 FMA 优化的 DIT 基 2FFT 蝶形单元计算

假定 DIT 基 2FFT 的一个蝶形单元的两个输入分别为 A 和 B，输出分别为 Y_1 和 Y_2，蝶形因子为 W，下标 r 和 i 分别表示复数的实部和虚部，根

据式(1),则有:

$$\begin{bmatrix} Y_{1\mathrm{i}} \\ Y_{1\mathrm{r}} \\ Y_{2\mathrm{i}} \\ Y_{2\mathrm{r}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & W_{\mathrm{r}} & -W_{\mathrm{i}} \\ 0 & 1 & W_{\mathrm{i}} & W_{\mathrm{r}} \\ 1 & 0 & -W_{\mathrm{r}} & W_{\mathrm{i}} \\ 0 & 1 & -W_{\mathrm{i}} & -W_{\mathrm{r}} \end{bmatrix} \begin{bmatrix} A_{\mathrm{r}} \\ A_{\mathrm{i}} \\ B_{\mathrm{r}} \\ B_{\mathrm{i}} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & W_{\mathrm{r}} & 0 \\ 0 & 1 & 0 & W_{\mathrm{r}} \\ 1 & 0 & -W_{\mathrm{r}} & 0 \\ 0 & 1 & 0 & -W_{\mathrm{r}} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -W_{\mathrm{i}}/W_{\mathrm{r}} \\ 0 & 0 & W_{\mathrm{i}}/W_{\mathrm{r}} & 1 \end{bmatrix} \begin{bmatrix} A_{\mathrm{r}} \\ A_{\mathrm{i}} \\ B_{\mathrm{r}} \\ B_{\mathrm{i}} \end{bmatrix} \tag{3}$$

根据式(3),DIT 基 2 FFT 的蝶形单元运算可以分解成两步完成:

1)计算中间结果 $\bar{A}$,$\bar{B}$:

$$\begin{bmatrix} \bar{A}_{\mathrm{r}} \\ \bar{A}_{\mathrm{i}} \\ \bar{B}_{\mathrm{r}} \\ \bar{B}_{\mathrm{i}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -W_{\mathrm{i}}/W_{\mathrm{r}} \\ 0 & 0 & W_{\mathrm{i}}/W_{\mathrm{r}} & 1 \end{bmatrix} \begin{bmatrix} A_{\mathrm{r}} \\ A_{\mathrm{i}} \\ B_{\mathrm{r}} \\ B_{\mathrm{i}} \end{bmatrix}$$

$$\Rightarrow \begin{cases} \bar{A}_{\mathrm{r}} = A_{\mathrm{r}} \\ \bar{A}_{\mathrm{i}} = A_{\mathrm{i}} \\ \bar{B}_{\mathrm{r}} = B_{\mathrm{r}} - B_{\mathrm{i}} \times (W_{\mathrm{i}}/W_{\mathrm{r}}) \\ \bar{B}_{\mathrm{i}} = B_{\mathrm{i}} + B_{\mathrm{r}} \times (W_{\mathrm{i}}/W_{\mathrm{r}}) \end{cases}$$

2)计算输出结果 Y_1 和 Y_2:

$$\begin{bmatrix} Y_{1\mathrm{i}} \\ Y_{1\mathrm{r}} \\ Y_{2\mathrm{i}} \\ Y_{2\mathrm{r}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & W_{\mathrm{r}} & 0 \\ 0 & 1 & 0 & W_{\mathrm{r}} \\ 1 & 0 & -W_{\mathrm{r}} & 0 \\ 0 & 1 & 0 & -W_{\mathrm{r}} \end{bmatrix} \begin{bmatrix} \bar{A}_{\mathrm{r}} \\ \bar{A}_{\mathrm{i}} \\ \bar{B}_{\mathrm{r}} \\ \bar{B}_{\mathrm{i}} \end{bmatrix}$$

$$\Rightarrow \begin{cases} Y_{1\mathrm{r}} = \bar{A}_{\mathrm{r}} + W_{\mathrm{r}} \times \bar{B}_{\mathrm{r}} \\ Y_{1\mathrm{i}} = \bar{A}_{\mathrm{i}} + W_{\mathrm{r}} \times \bar{B}_{\mathrm{i}} \\ Y_{2\mathrm{r}} = \bar{A}_{\mathrm{r}} - W_{\mathrm{r}} \times \bar{B}_{\mathrm{r}} \\ Y_{2\mathrm{i}} = \bar{A}_{\mathrm{i}} - W_{\mathrm{r}} \times \bar{B}_{\mathrm{i}} \end{cases}$$

其中,步骤 1 的计算由 2 条 FMA 指令完成,步骤 2 的计算由 4 条 FMA 指令完成。从而计算一个 DIT 基 2 FFT 蝶形单元仅需要 6 条 FMA 指令操作,相比传统的计算方式减少了 4 次。在传统的计算方法下,计算 N 点的 DIT 基 2 FFT 需要的浮点操作次数为 $5N\log_2 N$;FMA 优化后需要的浮点操作次数减少到 $3N\log_2 N$,减少了 40%。

2.2.2　FMA 优化的 DIT 基 4FFT 蝶形单元计算

假定 DIT 基 4 FFT 的一个蝶形单元的 4 个输入分别为 A,B,C,D,输出分别为 Y_1,Y_2,Y_3,Y_4,3 个蝶形因子分别为 W_1,W_2,W_3,下标 r 和 i 分别表示复数的实部和虚部,根据式(2),则有:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & W_1 & 0 & 0 \\ 0 & 0 & W_2 & 0 \\ 0 & 0 & 0 & W_3 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & W_1 & 0 \\ 0 & 1 & 0 & -\mathrm{i}W_1 \\ 1 & 0 & -W_1 & 0 \\ 0 & 1 & 0 & \mathrm{i}W_1 \end{bmatrix} \begin{bmatrix} 1 & W_2 & 0 & 0 \\ 1 & -W_2 & 0 & 0 \\ 0 & 0 & 1 & W_2 \\ 0 & 0 & 1 & -W_2 \end{bmatrix} \begin{bmatrix} A \\ C \\ B \\ D \end{bmatrix} \tag{4}$$

根据式(4),DIT 基 4 FFT 的蝶形单元运算可以分解成两步完成:

1)计算中间结果 $\bar{A}$,$\bar{B}$,$\bar{C}$,$\bar{D}$:

$$\begin{bmatrix} \bar{A} \\ \bar{B} \\ \bar{C} \\ \bar{D} \end{bmatrix} = \begin{bmatrix} 1 & W_2 & 0 & 0 \\ 1 & -W_2 & 0 & 0 \\ 0 & 0 & 1 & W_2 \\ 0 & 0 & 1 & -W_2 \end{bmatrix} \begin{bmatrix} A \\ C \\ B \\ D \end{bmatrix}$$

$$\Rightarrow \begin{cases} \bar{A}_{\mathrm{r}} = A_{\mathrm{r}} + W_{2\mathrm{r}} \times [C_{\mathrm{r}} - C_{\mathrm{i}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \\ \bar{A}_{\mathrm{i}} = A_{\mathrm{i}} + W_{2\mathrm{r}} \times [C_{\mathrm{i}} + C_{\mathrm{r}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \\ \bar{B}_{\mathrm{r}} = A_{\mathrm{r}} - W_{2\mathrm{r}} \times [C_{\mathrm{r}} - C_{\mathrm{i}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \\ \bar{B}_{\mathrm{i}} = A_{\mathrm{i}} - W_{2\mathrm{r}} \times [C_{\mathrm{i}} + C_{\mathrm{r}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \\ \bar{C}_{\mathrm{r}} = B_{\mathrm{r}} + W_{2\mathrm{r}} \times [D_{\mathrm{r}} - D_{\mathrm{i}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \\ \bar{C}_{\mathrm{i}} = B_{\mathrm{i}} + W_{2\mathrm{r}} \times [D_{\mathrm{i}} + D_{\mathrm{r}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \\ \bar{D}_{\mathrm{r}} = B_{\mathrm{r}} - W_{2\mathrm{r}} \times [D_{\mathrm{r}} - D_{\mathrm{i}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \\ \bar{D}_{\mathrm{i}} = B_{\mathrm{i}} - W_{2\mathrm{r}} \times [D_{\mathrm{i}} + D_{\mathrm{r}} \times (W_{2\mathrm{i}}/W_{2\mathrm{r}})] \end{cases}$$

2)计算输出结果 Y_1,Y_2,Y_3,Y_4:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & W_1 & 0 \\ 0 & 1 & 0 & -\mathrm{i}W_1 \\ 1 & 0 & -W_1 & 0 \\ 0 & 1 & 0 & \mathrm{i}W_1 \end{bmatrix} \begin{bmatrix} \bar{A} \\ \bar{B} \\ \bar{C} \\ \bar{D} \end{bmatrix}$$

$$\Rightarrow \begin{cases} Y_{1\mathrm{r}} = \overline{A_{\mathrm{r}}} + W_{1\mathrm{r}} \times [\bar{C}_{\mathrm{r}} - \bar{C}_{\mathrm{i}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \\ Y_{1\mathrm{i}} = \overline{A_{\mathrm{i}}} + W_{1\mathrm{r}} \times [\bar{C}_{\mathrm{i}} + \bar{C}_{\mathrm{r}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \\ Y_{2\mathrm{r}} = \bar{B}_{\mathrm{r}} - W_{1\mathrm{r}} \times [\bar{D}_{\mathrm{i}} + \bar{D}_{\mathrm{r}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \\ Y_{2\mathrm{i}} = \bar{B}_{\mathrm{i}} + W_{1\mathrm{r}} \times [\bar{D}_{\mathrm{r}} - \bar{D}_{\mathrm{i}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \\ Y_{3\mathrm{r}} = \overline{A_{\mathrm{r}}} - W_{1\mathrm{r}} \times [\bar{C}_{\mathrm{r}} - \bar{C}_{\mathrm{i}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \\ Y_{3\mathrm{i}} = \overline{A_{\mathrm{i}}} - W_{1\mathrm{r}} \times [\bar{C}_{\mathrm{i}} + \bar{C}_{\mathrm{r}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \\ Y_{4\mathrm{r}} = \bar{B}_{\mathrm{r}} + W_{1\mathrm{r}} \times [\bar{D}_{\mathrm{i}} + \bar{D}_{\mathrm{r}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \\ Y_{4\mathrm{i}} = \bar{B}_{\mathrm{i}} - W_{1\mathrm{r}} \times [\bar{D}_{\mathrm{r}} - \bar{D}_{\mathrm{i}} \times (W_{1\mathrm{i}}/W_{1\mathrm{r}})] \end{cases}$$

其中,步骤 1 的计算由 12 条 FMA 指令完成,步骤 2 的计算由 12 条 FMA 指令完成。从而计算一个 DIT 基 4FFT 蝶形单元仅需要 24 条 FMA 指令操作,相比传统的计算方式减少了 10 次。在传统的计算方法下,计算 N 点的 DIT 基 4 FFT 需要的浮点操作次数为 $8.5N\log_4 N$;FMA 优化后需要的浮点操作次数减少到 $6N\log_4 N$,减少了 29.4%。

2.3　混合基 4 和基 2 的 FFT 计算方法

如图 4 所示,对于任意的 $N=2^n$,可采用混合

基 4 和基 2 的 FFT 方法加速 FFT 的计算。若 n 是偶数,令 $n=2m$,则 $N=2^{2m}=4^m$;若 n 是奇数,令 $n=2m+1$,则 $N=2^{2m+1}=4^m\times 2$。因此,N 点的 FFT 计算可转化为 m 级基 4FFT 或者 m 级基 4FFT 和最后一级的基 2 FFT 计算。

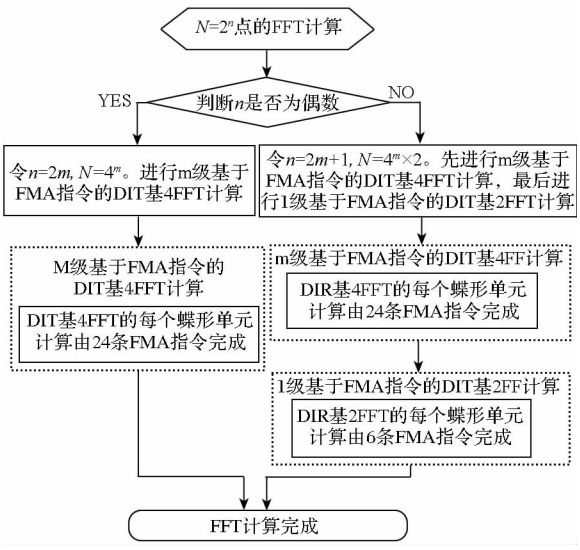

图 4 混合基 4 和基 2 的 FFT 计算流程

Fig. 4 FFT computation diagram of mixed radix - 2/4

2.4 旋转因子向量访问的优化

传统的基 4 FFT 计算方法中,每个蝶形单元运算需要乘以 3 个不同的旋转因子:W_N^i,W_N^{2i} 和 W_N^{3i}。FMA 优化后的蝶形单元运算只需计算 2 个不同旋转因子:W_N^i 和 W_N^{2i},并且预先计算出旋转因子虚部/实部(W_i/W_r)的值,然后把旋转因子按实部(W_r)、虚部/实部(W_i/W_r)交叉存储放置。这样,能减少旋转因子存储的个数。由于向量处理器只能连续存取向量数据,所以每级 FFT 运算的不同旋转因子都要单独放置。

同样,DIT 基 2 FFT 算法需要预先计算旋转因子 W_N^i 和虚部/实部(W_i/W_r)的值,并且按实部(W_r)、虚部/实部(W_i/W_r)交叉的方式存放。

以 1024 点的 DIT 基 4 FFT 为例,算法由 5 级蝶形运算实现,且每一级不同旋转因子所需的数目分别为:1 个、4 个、16 个、64 个和 256 个。由此可知,第 1 级和第 2 级的不同旋转因子的数目少于 VPE 的个数,并且第 1 级需要乘的旋转因子的值为 1,所以可以省略。为了提高运算的并行度,可以把第 2 级所需的旋转因子进行冗余存放 4 次。

在计算表 1 的旋转因子 W_{1024}^i 和 W_{1024}^{2i} 时,同时把每个旋转因子的虚部/实部(W_i/W_r)的值计算出来。并且,把旋转因子 W_{1024}^i 和 W_{1024}^{2i} 按它们的实部、虚部/实部交叉放置。

表 1 1024 点 DIT 基 4 FFT 的旋转因子

Tab. 1 Twiddle factors of 1024-point radix-4 DIT FFT

级数	系数 W_{1024}^i	系数 W_{1024}^{2i}	系数个数	运算次数
2	$i=0,64,128,192$	$i=0,64,128,192$	4×4	64
3	$i=0,16,\cdots,240$	$i=0,16,\cdots,240$	16	16
4	$i=0,4,\cdots,252$	$i=0,4,\cdots,252$	64	4
5	$i=0,1,\cdots,255$	$i=0,1,\cdots,255$	256	1

3 性能测试与分析

在向量处理器 Matrix 上对不同点数的 FFT 计算性能进行了测试(称 MatrixFFT),并与 Intel,AMD,IBM 平台上的 MKL,ACML,ESSL 和 FFTW 3.0算法库的性能进行比较。实验平台中,Matrix 的主频为 1GHz(双精度峰值性能为 96GFLOPS),MatrixFFT 的测试结果为 RTL 级仿真环境下的测试性能数据;比对实验的 CPU 中,Intel 选择主频 3.0GHz Intel Xeon Core Duo (Woodcrest)(峰值性能为 12GFLOPS);AMD 选择主频 2.2 GHz Dual Core AMD Opteron(峰值性能为 4.4GFLOPS);IBM 选择主频 2GHz PowerPC 970(峰值性能为 10GFLOPS),其性能数据选自 FFTW 网站报告的评测数据[8]。

图 5 和图 6 分别给出了在 Matrix 上测试的不同点数的基 2 和基 4FFT 的单精度和双精度计算性能。从图 5、图 6 中可以看出,在点数较小时,FFT 的性能较低,随着点数的增大,FFT 的计算性能显著提高,16K 点的单精度基 2 和基 4FFT 性能分别达到 74GFLOPS 和 98GFLOPS,计算效率分别

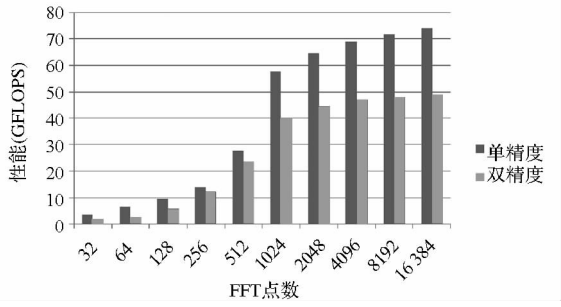

图 5 基 2FFT 的计算性能

Fig. 5 Performance of radix - 2 FFT

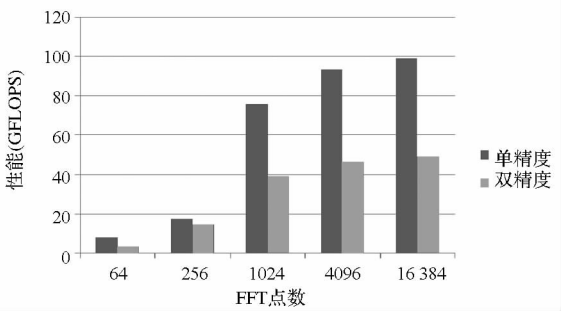

图 6 基 4FFT 的计算性能

Fig. 6 Performance of radix - 4 FFT

为 38.5% 和 51.04%。16K 点的双精度基 2 和基 4FFT 性能分别达到 48.83GFLOPS 和 49.05GFLOPS,计算效率分别为 50.08% 和 51.09%。

图 7 对比了不同点数的双精度 FFT 在 FMA 优化前后的计算性能,从图 7 中可以看出,无论是 DIT 基 2FFT,还是 DIT 基 4FFT,经过 FMA 优化后,FFT 的计算性能均显著提高,其中 DIT 基 2FFT 的计算性能平均提高 30.4%,DIT 基 4FFT 的计算性能平均提高 30.3%。

表 2 和表 3 分别从计算性能和效率两方面给出了不同点数的双精度 FFT 分别在 Matrix,Intel,AMD,IBM 平台上的 MKL,ACML,ESSL 和 FFTW3.0 算法库测试结果。

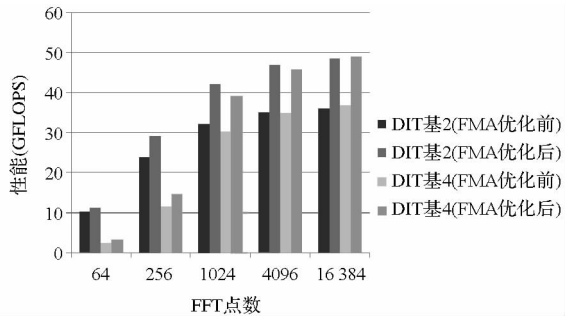

图 7　FMA 优化的 FFT 性能对比

Fig. 7　Performance comparison of FMA optimized FFT

表 2　不同处理器平台下 FFT 性能(MFLOPS)

Tab. 2　FFT performance of different processors(MFLOPS)

点数	32	64	128	256	512	1024	2048	4096	8192	16 384
MatrixFFT	1720	2591	5929	12 411	24 125	40 899	45 629	48 282	49 326	50 002
Intel – MKL	5210	6091	6661	6771	7100	6567	6366	5722	5919	5883
Intel – FFTW3	6952	8908	4916	5973	5987	6086	5417	5111	4326	4508
AMD – ACML	1034	1462	1402	2060	1838	2037	1831	1476	1235	1209
AMD – FFTW3	2381	2566	2026	2141	2137	2178	2057	1847	1462	1319
PowerPC – ESSL	2795	3287	2119	2645	3389	3420	3128	3120	2890	1257

表 3　不同处理器平台下 FFT 计算效率

Tab. 3　FFT efficiency of different processors

点数	32	64	128	256	512	1024	2048	4096	8192	16 384
MatrixFFT	1.75%	2.64%	6.03%	12.63%	24.54%	41.60%	46.42%	49.11%	50.18%	50.86%
Intel – MKL	42.40%	49.57%	54.21%	55.10%	57.78%	53.44%	51.81%	46.57%	48.17%	47.88%
Intel – FFTW3	56.58%	72.49%	40.01%	48.61%	48.72%	49.53%	44.08%	41.59%	35.21%	36.69%
AMD – ACML	22.95%	32.45%	31.12%	45.72%	40.79%	45.21%	40.64%	32.76%	27.41%	26.83%
AMD – FFTW3	52.85%	56.95%	44.97%	47.52%	47.43%	48.34%	45.65%	40.99%	32.45%	29.27%
PowerPC – ESSL	27.29%	32.10%	20.69%	25.83%	33.10%	33.40%	30.55%	30.47%	28.22%	12.28%

从绝对计算性能上看,MatrixFFT 的性能远超其他几种算法库,主要原因是 Matrix 的峰值性能远超对比 CPU 的峰值性能;另一方面也体现出提出的基于 FMA 优化的 FFT 计算效率较高,能够充分发挥 Matrix 的计算性能。这一点从表 3 可以看出,在点数较小时,MatrixFFT 的计算效率较低,这是因为 Matrix 是向量处理器,点数较小时,软件流水中的循环填充和排空的开销在整个计算中的占比较高,影响计算效率。在点数超过 1024 点以后,MatrixFFT 的计算效率显著提高,在 4K 点以后,计算效率都是最高的,其中 16K 点效率达 50.86%,而对比的其他算法库效率分别是 47.88%,36.69%,26.83%,29.27%,12.28%。

4　结论

本文针对 FMA 结构的向量处理器,提出 FMA 加速 FFT 计算的向量化方法。通过优化和重组 FFT 算法的蝶形单元运算流程,将原本不平衡的乘法和加法操作组合成融合乘加操作,利用 FMA 指令减少 FFT 计算的浮点操作指令次数,进而提高 FFT 算法的计算性能和向量处理器的计算效率,提高硬件资源的利用率,加速 FFT 算法的计算性能。

参考文献(References)

[1]　刘仲, 陈跃跃, 陈海燕. 支持任意系数长度和数据类型的 FIR 滤波器向量化方法[J]. 电子学报, 2013, 41(2): 346 – 351.

LIU Zhong, CHEN Yueyue, CHEN Haiyan. A vectorization of

FIR filter supporting any length and data types of coefficients[J]. Acta Electronica Sinica, 2013, 41(2): 346 – 351. (in Chinese)

[2] 刘仲, 邢彬朝, 陈跃跃. 一种面向多核处理器的高效并行PCA – SIFT算法[J]. 国防科技大学学报, 2012, 34(4): 103 – 107.
LIU Zhong, XING Binchao, CHEN Yueyue. An efficient parallel PCA-SIFT algorithm for multi-core processor [J]. Journal of National University of Defense Technology, 2012, 34(4): 103 – 107. (in Chinese)

[3] Pease M C. An adaptation of the fast Fourier transform for parallel processing[J]. Journal of the ACM, 1968, 15(2): 252 – 264.

[4] Linzer E N, Feig E. Implementation of efficient FFT algorithms on fused multiply-add architectures[J]. IEEE Transactions on Signal Processing, 1993, 41(1): 93 – 107.

[5] Goedecker S. Fast radix 2, 3, 4, and 5 kernels for fast Fourier transformations on computers with overlapping multiply-add instructions[J]. SIAM Journal on Scientific Computing, 1997, 18(6): 1605 – 1611.

[6] Karner H, Auer M, Ueberhuber C W. Multiply-add optimized FFT kernels[J]. Mathematical Models and Methods in Applied Sciences, 2001, 11(1): 105 – 117.

[7] Voronenko Y, Puschel M. Mechanical derivation of fused multiply-add algorithms for linear transforms [J]. IEEE Transactions on Signal Processing, 2007, 55 (9): 4458 – 4473.

[8] Frigo M, Johnson S G. BenchFFT[EB/OL]. [2014 – 03 – 15]. http://www.fftw.org/benchfft/.

[9] Lobeiras J, Amor M, Doallo R. Influence of memory access patterns to small-scale FFT performance [J]. Journal of Supercomputing, 2013, 64(1):120 – 131.

[10] Li Y, Zhang Y Q, Liu Y Q, et al. MPFFT: An auto-tuning FFT library for OpenCL GPUs [J]. Journal of Computer Science and Technology, 2013, 28(1): 90 – 105.

[11] 何涛, 朱岱寅. 大点数一维FFT的GPU设计实现[J]. 计算机工程与科学, 2013, 35(11): 34 – 41.
HE Tao, ZHU Daiyin. Design and implementation of large-point 1D FFT on GPU[J]. Computer Engineering & Science, 2013, 35(11): 34 – 41.

[12] Cooley J W, Turkey J W. An algorithm for the machine calculation of complex Fourier series [J]. Mathematics of Computation, 1965, 19: 297 – 301.