

南京大学 本科毕业论文

基于 Intel Pentium III/4 个人电脑的 脉冲多普勒超声血流成像系统的 高速谱计算

李康 98129045

电子科学与工程系 98 级

指导老师：李太宝 特聘教授

版 本	1.0
修订号	406
日 期	2002 年 6 月 27 日

摘要

本论文通过对以快速傅里叶变换 (FFT) 为核心的脉冲多普勒血流成像系统谱分析器在主流的 Intel® Pentium® III/4 个人电脑 (PC) 平台上的优化实现的研究, 初步探讨了医学超声成像系统 (软件部分) 在 PC 系统上实现的可行性, 以及为了充分利用现代微处理器运算能力, 提高程序速度所需的程序设计技术。在此基础上, 一个针对 Intel® Pentium® III/4 高度优化的 FFT 软件包 “SFFT” 编写完成。经过严格的评测发现, SFFT 在精度方面完全达到了设计要求, 而它的速度则大大超过了 FFTW 等国际知名的却未特别优化的 FFT 软件包。对于 256 点及以下的短序列 FFT, SFFT 的速度达到甚至超过了 Intel 公司为 Pentium 系列微处理器特别设计的 Math Kernel Library 和 Signal Processing Library, 可能是 Pentium III/4 平台上的目前世界上最快的 FFT 运算程序。医学超声成像系统软件的其他部分也可以使用类似的技术, 在现代 PC 平台上优化实现。

关键字: 脉冲多普勒, 高速谱计算, 并行运算, FFT, SIMD, Split-Radix。

Abstract

This thesis considers an optimized implementation of the Fast Fourier Transform (FFT) algorithm on Intel Pentium III/4 personal computers. Since FFT is the core component of the spectrum calculator of Pulsed-Doppler blood streaming imaging systems, we thus discussed the feasibility of implementing a medical ultrasound imaging system (software aspect) on personal computers, and the optimization techniques involved to exert the full power of recent microprocessors. Through a series of experiments and trial-and-fails, a highly optimized FFT library named "SFFT" was elaborated, which is completely handcoded with inline assembly and optimized using Short Vector SIMD instructions, such as MMX and SSE, of Pentium III/4 processors. We discovered after strict tests and measurements that SFFT not only attained the intended precision, but its speed is up to 2.7 times faster than FFTW - a world-famous but unoptimized FFT package. We also observed that for FFTs below 256 points, SFFT is probably the fastest FFT library around the world for its speed is even faster than the FFT routines of Intel Math Kernel Library (MKL) and Signal Processing Library (SPL), which are specially optimized for IA-32 processors, including Pentium III and 4. We then derived that the whole medical ultrasound imaging system can be optimized and speeded up by similar techniques on recent PC platforms.

Keywords: Pulsed-Doppler, high-speed spectrum calculation, parallel computing, FFT, SIMD, Split-Radix.

致谢

感谢李太宝教授对我的指导和支持，在他领导的南京大学声学所医学超声工程实验室里，我度过了忙碌、充实而愉快的一年。李教授放弃了海外优厚的生活，抱着报国的雄心回到祖国大地，他的爱国热情令我肃然起敬；而他在工作中的投入、忘我和敬业精神也为我们树立了很好的榜样，给予我莫大的鼓舞和动力。

特别感谢冯若教授，他已在耄耋之年，依然精神矍铄，为科研事业而来回奔波，日夜忙碌。他所编写的《超声手册》博大精深，图文并茂，理论翔实，为我踏入医学超声研究领域指点迷津。

感谢王耀俊教授和许坚毅教授，虽然接触不多，但他们严谨治学的作风和坚实的理论功底令人钦佩，使我终身难忘。感谢高怀教授，虽然他已不再是我们的邻居，但他的广博的知识面和一流的口才给我留下了深刻的印象。

感谢实验室的所有同学和朋友，包括杨敏、丁波、朱晓华、洪晶、孙敏、高常清、曾小政、伍九汇、张樯等等，与他们交谈是一件惬意的事，更让我发现了自己的很多错误和不足。

感谢我的父母和家人，他们多年来为我的学业和生活给予了莫大的鼓励和支持，使我受益匪浅。尤其难忘爸爸妈妈在百忙中送来的勉励的话语、奶奶和姨奶奶做的蒸鲈鱼和老鸡汤、张爷爷买的西瓜和说的风趣的话，还有张宏叔叔以及远在他乡的小平姑姑、张宁姑姑给我的指点和支持。

感谢实验室的 Pentium 4 电脑和爸爸妈妈为我买的 DELL®笔记本电脑，它们是我完成大部分实验工作的办公平台，它们的存在使我的工作效率倍增。

目录

摘要.....	i
关键字.....	i
Abstract.....	ii
Keywords.....	ii
致谢.....	iii
目录.....	v
插图目录.....	vii
表格目录.....	viii
第 1 章 引言	1
第 2 章 背景知识	2
2.1 医学超声多普勒技术.....	2
2.2 超声多普勒信号	2
2.3 多普勒信号处理.....	3
2.3.1 多普勒信号的解调	4
2.3.2 多普勒血流信号的速度谱	4
2.4 脉冲多普勒系统.....	5
2.5 离散傅里叶变换.....	7
2.6 快速傅里叶变换.....	8
2.6.1 库利-图基分解	8
2.6.2 Radix-2 FFT 算法.....	10
2.6.3 Radix-4 FFT 算法.....	11
2.6.4 Split-Radix FFT 算法.....	12
2.6.5 其它 FFT 算法	14
2.6.5.1 质因子 (Prime-Factor) FFT 算法.....	15
2.6.5.2 Winograd FFT 算法 (WFTA)	16
2.7 小结	16
第 3 章 基于 PC 平台的医学超声成像系统	17
3.1 机遇和动力	17
3.2 主要实现难点	18
3.3 PC 技术发展趋势.....	19
3.4 Intel Pentium III/4 的 SIMD 技术.....	22
3.4.1 我的系统支持 SSE 吗?	23

3.4.1.1 判断处理器对 SSE 的支持	23
3.4.1.2 判断操作系统对 SSE 的支持	25
3.4.2 SSE 指令集精粹	27
3.4.3 MMX 指令精粹	30
3.5 小结	31
第 4 章 FFT 谱分析器的优化与实现	33
4.1 算法选择	33
4.2 设计思想	34
4.2.1 优化设计策略	34
4.2.2 程序结构设计	37
4.3 汇编语言 vs. C/C++	40
4.4 小代码内核的优化实现	41
4.4.1 4 点 FFT 内核的实现	41
4.4.2 8 点 FFT 内核的实现	45
4.4.3 16 点 FFT 内核的实现	48
4.4.4 IFFT 内核的优化实现	49
4.5 通用内核函数的实现	49
4.6 接口函数的优化实现	50
4.6.1 固定点数接口函数的实现	51
4.6.2 通用接口函数的实现	52
4.7 位逆序函数的优化实现	54
4.8 软件包的封装与调用	56
4.9 小结	56
第 5 章 SFFT 软件包性能测试与评价	57
5.1 精度测评	57
5.2 速度测评	58
5.2.1 测评方法	58
5.2.1.1 时钟源的选择	59
5.2.1.2 提高测量精度的策略	60
5.2.2 测评结果	60
5.2.3 误差分析	64
5.3 应用测评	64
5.4 结论	66
参考文献	ix

插图目录

图 1	脉冲多普勒系统波形图	6
图 2	Radix-2 DIF 算法蝶形运算图.....	11
图 3	Split-Radix FFT 的 L 字蝶形图.....	14
图 4	基于 PC 的脉冲多普勒超声血流成像系统	18
图 5	四数据 SIMD 并行运算示意图	20
图 6	组合运算和标量运算示意图	23
图 7	CPUID 指令在 EDX 寄存器中返回的信息格式.....	24
图 8	对 Numerical Recipes FFT 的改进和执行时间比较	37
图 9	128 点 Split-Radix 分解流程图	39
图 10	4 点 Split-Radix FFT 蝶形运算图.....	42
图 11	4 点 Split-Radix FFT 实、虚部数据流向图.....	42
图 12	4 点 Split-Radix FFT: 并行加法的两种方案.....	43
图 13	4 点 Split-Radix FFT: 数值求反的两种方案.....	44
图 14	4 点 Split-Radix FFT: 第二组蝶形运算.....	45
图 15	8 点 Split-Radix FFT 蝶形运算图.....	46
图 16	4 点 Split-Radix FFT 实、虚部数据流向图.....	47
图 17	16 点 Split-Radix FFT 蝶形运算图.....	48
图 18	128 点 Split-Radix FFT 的分解树.....	51
图 19	SFFT 静态链接库版速度测评 (MSC++编译)	62
图 20	SFFT 静态链接库版速度测评 (Intel C++编译)	62
图 21	SFFT 动态链接库版速度测评 (MSC++编译)	63
图 22	SFFT 动态链接库版速度测评 (Intel C++编译)	63
图 23	超声成像系统 (UIS) 主界面	65
图 24	SFFT 应用评测结果—锯齿波频谱	66
图 25	SFFT 应用评测结果—方波频谱	66

表格目录

表 1	MMX/SSE/SSE2/3DNow! 指令集比较 (1)	21
表 2	MMX/SSE/SSE2/3DNow! 指令集比较 (2)	21
表 3	MMX/SSE/SSE2/3DNow! 指令集比较 (3)	22
表 4	常用的 SSE 指令说明	30
表 5	常用的 MMX 指令说明	31
表 6	Radix-2、Radix-4 和 Radix-2/4 复数乘法数量比较	34
表 7	对 Numerical Recipes FFT 的改进和执行时间比较	36
表 8	4 点 FFT 内核指令数统计	45
表 9	8 点 FFT 内核指令数统计	47
表 10	16 点 FFT 内核指令数统计	49
表 11	SFFT 精度测评结果	58
表 12	Windows 系统中可用时钟源精度对比	59

第1章 引言

医学超声成像和 CT、MRI 及核医学成像是现代四大医学影像技术。和其它成像技术相比，医学超声成像具有实时性好、无损伤、无痛苦、低成本和应用范围广等独特的优点，因此，医学超声成像设备的世界市场份额近年来一直呈现上升态势，并且其增长率超过了医学成像设备的平均增长率。

与 IT 技术和微电子技术结合是医学超声成像目前的研究热点之一，全数字化和超小型化成为医学超声成像设备未来的发展方向。随着“芯片即系统(System on a Chip)”技术的成熟，在不久的将来，今天看似庞然大物的彩超系统将可以缩小得装进口袋里，只需一个小小的接插件就可以和个人电脑(PC)互联，实时显示诊断的图像。另一方面，随着 PC 功能的不断强大和速度的突飞猛进，越来越多以往需要专用硬件实现的处理功能现在可移植到 PC 机上，利用软件实现，以发挥软件的灵活性。

然而，充分发挥现代 PC 机的运算能力，高效地实现医学超声信号的实时处理和显示并非一件易事。尽管得益于微处理器速度的迅速提高，目前 PC 机的性能得到了长足的进步，但由传统的程序设计方法编制的软件往往并不能完全发挥现代 PC 系统的运算能力，从而浪费了大量宝贵资源。

本文将借助对以快速傅里叶变换(FFT)为核心的脉冲多普勒超声血流成像系统谱分析器在主流 Intel Pentium III/4 个人电脑平台上的优化实现的讨论，提出了如何通过使用 SIMD (Single Instruction, Multiple Data) 指令集和改变程序结构，从而充分利用现代微处理器并行运算能力的程序设计原理和方法，并给出了软件实例和测评结果。

本文分为 5 章，文字编排由浅入深，由概念到具体。从下一章开始，我们将依次介绍脉冲多普勒成像系统和快速傅里叶变换(FFT)的基础概念、基于 PC 平台的医学超声成像系统的实现难点和 PC 系统的最新技术，以及 FFT 谱分析器的优化实现和测评等内容。我们将从这里开始，踏上设计基于 Intel Pentium III/4 处理器的世界最快的 FFT 谱分析器的旅程。

第2章 背景知识

2.1 医学超声多普勒技术

1842 年，奥地利物理学家 C. J. Doppler 发现，当振动源与接收器在连续媒质中作相对运动时，接收器所接收到的频率不等于振动源所辐射的频率，其频率之差与它们的相对运动方向和速度有关，这种由于相对运动而引起频移的现象称为多普勒（Doppler）效应。^[1]

多普勒效应自发现至今，已被广泛应用于各门学科和领域。医学超声多普勒技术的研究始于 50 年代，并在 60 年代开始了相关仪器的开发和临床应用。超声多普勒技术在医学临床上主要用于测量血流，根据频移信号可以了解血液的流速、流量以及血流方向等信息，及早发现血流异常与病变，进行心血管疾病的诊断。多普勒技术也可应用于测量其它生物组织或器官的运动速度等参量，用于测量血压，进行听诊、胎儿监护和组织运动成像等。

2.2 超声多普勒信号

若传声媒质为静止的，则声源和接收器之间存在相对运动时就会产生多普勒频移。若声源静止，接收器以速度 v_1 运动，声源辐射周期为 f_0 的声波，此时接收器接收一个周期为 f' 的信号，即：

$$f' = \frac{c + v_1 \cos \theta_1}{\lambda_0} = \frac{c + v_1 \cos \theta_1}{c} f_0 \quad (1)$$

其中， λ_0 为声波波长， c 为媒质中的声速。此时的多普勒频移 f_d 为：

$$f_d = f' - f_0 = \frac{v_1}{c} \cos \theta_1 \cdot f_0 \quad (2)$$

相反，若声源运动（速度 v_2 ）而接收器静止，则接收器所接收到的信号频率 f'' 为：

$$f'' = \frac{c}{c - v_2 \cos \theta_2} f_0 \quad (3)$$

通常满足 $v_2 \ll c$ ，所以，于是多普勒频移 f_d 为：

$$f_d = f'' - f_0 = \frac{v_2 \cos \theta_2}{c - v_2 \cos \theta_2} = \frac{v_2}{c} \cos \theta_2 \cdot f_0 \quad (4)$$

在医学超声脉冲多普勒仪器中常使用反射式探头，这时，发射源和接收器位于被测物体（血流）的同一侧。设被测血流以速度 v 运动，与声束呈 θ 夹角，声源的发射功率为 f_0 ，按式(1)，运动物体接收到的频率为：

$$f' = \frac{c + v \cos \theta}{c} f_0 \quad (5)$$

根据惠更斯原理，声波被运动物体反射（或散射），此反射（或散射）的声波可认为是一个新的声源（即运动着的物体）发出的，它以速度 v 运动着，发出的声波被静止的换能器接收，按照式(3)，接收到的声波频率为：

$$f'' = \frac{c}{c - v \cos \theta} f' \quad (6)$$

将式(5)代入(6)，得：

$$f'' = \frac{c + v \cos \theta}{c - v \cos \theta} f_0 \quad (7)$$

因为条件 $c \gg v$ 总可满足，所以多普勒频移 f_d 为：

$$f_d = f'' - f_0 = \frac{2v}{c} \cdot f_0 \cos \theta \quad (8)$$

上式就是医学超声多普勒技术中常用的多普勒频移公式。当速度矢量与声束方向夹角 θ 满足 $0^\circ \leq \theta \leq 90^\circ$ 时， $f_d > 0$ ；反之，当 $90^\circ \leq \theta \leq 180^\circ$ 时， $f_d < 0$ ，此时的回波频率低于发射频率。为方便起见，常将式(8)改写为角频率形式，即：

$$\Omega = \frac{2v}{c} \omega_0 \cos \theta \quad (9)$$

2.3 多普勒信号处理

由上一节我们可以知道，对于匀速运动的脏器，其多普勒频移表达式如式(9)所示。由于血流中存在大量的红细胞，它们的速度具有较大的差异，致使接收到的散射信号中具有多种多普勒频移分量，它们构成了一种窄带谱信号。于是换能器接收到的信号可表达为：

$$V = V_1 + V_2 = a \cos \omega_0 t + \sum_i b_i \cos(\omega_0 + \Omega_i)t \quad (10)$$

式中 V_1 为非运动脏器反射的信号； V_2 为血流散射信号，其中忽略了各频移分量的初相。利用三角变换公式，上式可化简为：

$$\begin{aligned} V &= (a + \sum_i b_i \cos \Omega_i t) \cos \omega_0 t - (\sum_i b_i \sin \Omega_i t) \sin \omega_0 t \\ &= A(\Omega t) \cos[\omega_0 t + \varphi(\Omega t)] \end{aligned} \quad (11)$$

其中, $A(\Omega t)$ 、 $\varphi(\Omega t)$ 分别为信号振幅和相角, 它们的表达式如下:

$$A(\Omega t) = a \left[1 + \sum_i m_i^2 + 2 \sum_i m_i \cos \Omega_i t + 2 \sum_i \sum_{k \neq i}^{(i \neq k)} m_i m_k \cos \Omega_i t \cos \Omega_k t + 2 \sum_i \sum_{k \neq i}^{(i \neq k)} m_i m_k \sin \Omega_i t \sin \Omega_k t \right]^{\frac{1}{2}} \quad (12)$$

$$\varphi(\Omega t) = \arctan \left(\frac{\sum_i m_i \sin \Omega_i t}{1 + \sum_i m_i \cos \Omega_i t} \right) \quad (13)$$

其中, m_i 为调制度, 并有:

$$m_i = \frac{b_i}{a} \quad (14)$$

由式(11) ~ (14)可见, 接收信号 V 的振幅和相位均受多普勒频移成分的非线性调制。

2.3.1 多普勒信号的解调

从接收信号 V 中提取频移信息的过程称为解调, 理想的解调应取得正比于 $\sum_i b_i \cos \Omega_i t$ 的输出。由简单的推导可以得出, 乘法器解调方法是理论上的最优的。

这种方法用参考信号 $c \cos(\omega_0 t + \theta)$ 与式(10)所示的接收信号相乘, 再用低通滤波器滤除高频分量, 可得输出为:

$$V_0 = \frac{1}{2} \eta a c \left[\cos \theta + \sum_i m_i \cos(\Omega_i t - \theta) \right] \quad (15)$$

可见这种解调方法可得到与 $\sum_i b_i \cos(\Omega_i t - \theta)$ 成正比的输出, 没有引起失真, 是一种理想的解调方法。

2.3.2 多普勒血流信号的速度谱

对多普勒血流信号作快速傅里叶变换 (FFT), 形成时间—频率谱, 进行动态功率谱分析 (即频域分析), 可以获得血流的方向、速度等血流动力学参数。以速度谱图方式显示的多普勒血流仪具有广泛的临床实用价值。

根据式(11), 经放大后的回波信号可表示为:

$$X(t) = A(t) \cos[\omega_0 t + \varphi(t)] \quad (16)$$

对 $X(t)$ 作正交分解，即分别与 $2 \cos \omega_0 t$ 和 $-2 \sin \omega_0 t$ 相乘后作低通滤波，得到：

$$\begin{aligned} V_I &= A(t) \cdot \cos \varphi(t) \\ V_Q &= A(t) \cdot \sin \varphi(t) \end{aligned} \quad (17)$$

把正交分量 V_I 和 V_Q 合成复包络信号：

$$V(t) = V_I(t) + jV_Q(t) \quad (18)$$

再对 $V(t)$ 作功率谱分析就可获得速度谱。在屏幕上显示这种图谱时，横坐标代表时间，纵坐标代表频率或速度，功率谱密度用灰阶表示。在图中，位于零速度线以上的明亮区域代表流向探头的正向流，而位于零刻度线以下的亮区显示流离探头的反向流。亮区愈亮，表示相应速度的红细胞愈多。

2.4 脉冲多普勒系统

实用的医学超声多普勒成像系统通常分为连续波（CW）和脉冲波（PW）两种。连续波系统采用收发分离式换能器，分别连续不断的发射和接收超声波；脉冲波系统则采用收发公用的换能器，它进行间断式的脉冲发射，并作延迟门控式选通接收。

人体血流中有大量的运动着的血细胞，对入射声波能产生丰富的多普勒回波信号。连续多普勒系统所测得的信号是声波传输路径上所有运动目标信号的混合，而脉冲多普勒系统可以检测出人体某指定深度处的血流信息，因此在临床中被普遍应用。

为了说明脉冲多普勒系统的工作原理，我们先假想一个简单的模型。假设换能器向前发射超声波，遇到界面 A 而产生相应的回波。如果界面 A 是固定的，则它的回波与参考信号之间有固定的相位关系，经乘法器解调滤波和采样保持后将输出一个固定的直流电压，其大小与两信号的相位差有关；而如果界面 A 是运动的，其回波信号和参考信号之间的相位差将随着界面的移动而变化，因此经乘法器解调滤波后的输出电压也将随着界面的移动不断变化。当界面的运动速度较低，而脉冲重复频率（PRF）又较高时，前后两次回波之间的相位差就较小，相应的输出电压值的变化也较小，经采样保持后可得到一条与界面运动速度相关的正弦曲线，这就是用脉冲多普勒系统检测到的运动界面 A 的多普勒频

移信号。

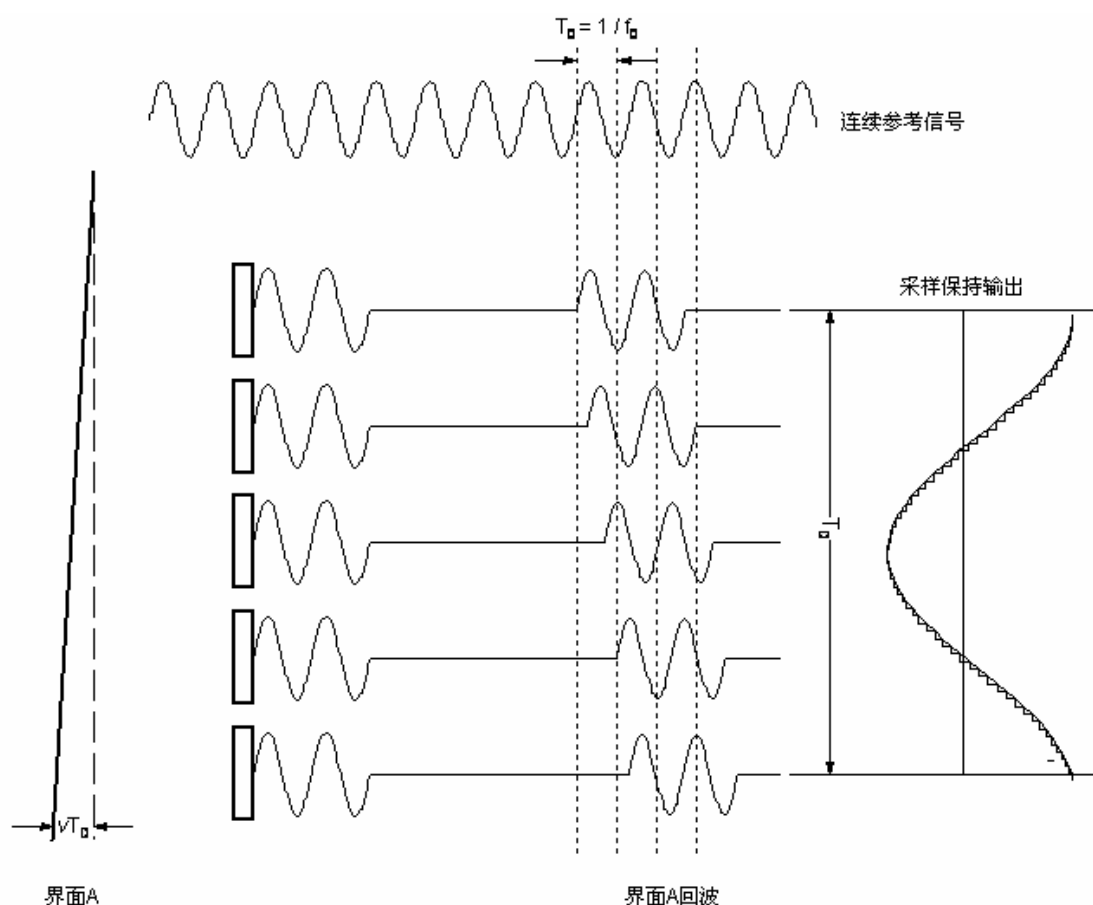


图 1 脉冲多普勒系统波形图

设载波频率为 f_0 ，周期为 $T_0 = 1/f_0$ ；多普勒频移信号的频率为 f_D ，周期为 $T_D = 1/f_D$ ；界面 A 的运动速度恒定为 v ，超声波的声速为 c 。从图 1 可以看出，在脉冲回波信号与参考信号之间的相位差从 0° (t_1 时刻) 变化到 360° (t_5 时刻) 的过程中，输出的频移信号曲线形成了一个完整的周期，此时界面 A 移位了 vT_D ，因此，界面 A 在 t_5 时刻的回波信号要比 t_1 时刻延迟 $2vT_0/c$ ，且有：

$$\frac{2vT_D}{c} = T_0 \quad (19)$$

$$f_D = \frac{2v}{c} \cdot f_0 \quad (20)$$

式(20)为界面运动方向和超声束的方向一致时多普勒频移的计算公式。设界面运动速度 v 的方向和超声束的方向之间的夹角为 θ ，则有：

$$f_D = \frac{2v \cos \theta}{c} \cdot f_0 \quad (21)$$

式(21)为计算多普勒频移信号的一般公式，形式上和连续多普勒频移的计算是一样的；所不同的是，由于在脉冲多普勒系统中，频移信号是通过频率为 PRF 的

采样脉冲对输入信号进行采样保持后获得的，可以检测出的最高频移只能是超声波脉冲重复频率（PRF）的一半，即：

$$f_{D\max} = \frac{1}{2} PRF \quad (22)$$

脉冲多普勒系统可测量的最高流速也因此被限制了，为：

$$v_{\max} \leq \frac{c}{2} \cdot \frac{f_{D\max}}{\cos \theta} = \frac{c}{4} \cdot \frac{PRF}{f_0 \cos \theta} \quad (23)$$

2.5 离散傅里叶变换

离散傅里叶变换（DFT）及其反变换（IDFT）在一段离散信号序列的时域表达 $x(n)$ 和频域表达 $X(k)$ 间建立了线性变换关系。这一离散序列通常是对一个连续信号 $f(x)$ 进行均匀抽样而得到的。假设 $x(n)$ 为一个 N 点的序列，则它的离散傅里叶变换也为一个 N 点的序列，即：

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) e^{-i2\pi nk/N} \\ &= \sum_{n=0}^{N-1} x(n) W_N^{nk}, \quad k = 0, 1, \dots, N-1 \end{aligned} \quad (24)$$

同时，我们有离散傅里叶反变换的表达式：

$$\begin{aligned} x(n) &= \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{i2\pi nk/N} \\ &= \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}, \quad n = 0, 1, \dots, N-1 \end{aligned} \quad (25)$$

其中， W_N 代表：

$$\begin{aligned} W_N &= e^{-i2\pi/N} \\ &= \cos\left(\frac{2\pi}{N}\right) - i \sin\left(\frac{2\pi}{N}\right) \end{aligned} \quad (26)$$

它又被称作是“ N 次单位根”，这是因为 $(W_N)^N = e^{-i2\pi} = 1$ ； W_N 的另一个重要的性质是它是周期性的，对于任意正数 m ，有 $(W_N)^n = (W_N)^{n+mN}$ 。

离散傅里叶变换是数字信号处理的重要方法，然而，它庞大的计算量曾让研究者们望而却步。从式(24)和式(25)可以看出，在 DFT 的 N 个输出中，每个元素都是 N 个 $x(n)(W_N)^{nk}$ 的代数和，这样的计算中包含 $N(N-1)$ 个复加和 N^2 个复乘，因此，直接按定义计算 DFT 大约需要 $2N^2$ 次复数运算，或者说，计算复杂度是

$O(N^2)$ ，这对于实际而言是不可取的。

2.6 快速傅里叶变换

快速傅里叶变换（FFT）最初往往特指库利（Cooley）和图基（Tukey）于 1965 年发表的论文中介绍的离散傅里叶变换快速算法，在此之前，几乎所有的 DFT 都是采用耗时的 $O(N^2)$ 算法。后来，快速傅里叶变换（FFT）逐渐演变为离散傅里叶变换一系列快速算法的统称。FFT 是多普勒血流谱分析的核心，也是多普勒成像等系统中使用最为普遍的信号处理方法。

DFT 算法之所以可以得以简化的原因源于式(24)中“ N 次单位根” W_N 的内在特性，包括前面所述的周期性、对称性等等，现归纳如下：

$$1. \quad W_N^n = W_N^{n+mN}, \quad m \in \mathbb{Z} \quad (27)$$

$$2. \quad W_N^{mk} = W_{N/m}^k, \quad m=1, 2, \dots, N \quad (28)$$

$$3. \quad W_N^{n+N/2} = -W_N^n \quad (29)$$

$$4. \quad W_N^0 = W_N^N = 1, \quad W_N^{\frac{N}{2}} = -1, \quad W_N^{\frac{N}{4}} = -i, \quad W_N^{\frac{3N}{4}} = i$$

其中， i 代表虚数单位， $i = \sqrt{-1}$ 。

2.6.1 库利—图基分解

库利—图基（Cooley-Tukey）FFT 算法的精髓在于所谓的库利—图基分解（Cooley-Tukey Decomposition）^[5]。设输入序列长度为 N ，且 N 可分解为两非互素因子 N_1 和 N_2 之积，即 $N = N_1 \times N_2$ ， $(N_1, N_2) > 1$ （括号“ $()$ ”代表最大公约数），则可将输入的一维序列 $x(n)$ 映射成二维的形式 $\tilde{x}(n_1, n_2)$ ^[11]：

$$\tilde{x}(n_1, n_2) = \begin{bmatrix} \tilde{x}(0,0) & \tilde{x}(0,1) & \cdots & \tilde{x}(0, N_2-1) \\ \tilde{x}(1,0) & \tilde{x}(1,1) & & \\ \vdots & & \ddots & \\ \tilde{x}(N_1-1,0) & & & \tilde{x}(N_1-1, N_2-1) \end{bmatrix} \quad (30)$$

相应地，输出序列 $X(k)$ 也可被映射为 $\tilde{X}(k_1, k_2)$ ：

$$\tilde{X}(k_1, k_2) = \begin{bmatrix} \tilde{X}(0,0) & \tilde{X}(0,1) & \cdots & \tilde{X}(0, N_2-1) \\ \tilde{X}(1,0) & \tilde{X}(1,1) & & \\ \vdots & & \ddots & \\ \tilde{X}(N_1-1,0) & & & \tilde{X}(N_1-1, N_2-1) \end{bmatrix} \quad (31)$$

如式(30)和式(31)所示，设二维形式的宽为 N_2 ，高为 N_1 ，则有：

$$\begin{aligned} n &= n_1 \cdot N_2 + n_2 \\ k &= k_1 + k_2 \cdot N_1 \end{aligned} \quad (32)$$

这时，式(24)可写为二维的形式：

$$\tilde{X}(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_N^{N_2 n_1 k_1} W_N^{N_2 n_1 N_1 k_2} W_N^{n_2 k_1} W_N^{n_2 N_1 k_2} \quad (33)$$

根据 W_N 的性质有：

$$1. \quad \forall n_1, k_2 \in \mathbb{Z}, \quad W_N^{N_2 n_1 N_1 k_2} = W_N^{N n_1 k_2} = 1 \quad (34)$$

$$2. \quad \forall n_1, k_1 \in \mathbb{Z}, \quad W_N^{N_2 n_1 k_1} = W_{N_1}^{n_1 k_1} \quad (35)$$

$$3. \quad \forall n_2, k_2 \in \mathbb{Z}, \quad W_N^{n_2 N_1 k_2} = W_{N_2}^{n_2 k_2} \quad (36)$$

将它们代入式(33)，可得：

$$\begin{aligned} \tilde{X}(k_1, k_2) &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2} \\ &= \sum_{n_1=0}^{N_1-1} \left[\left(\sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_{N_2}^{n_2 k_2} \right) \cdot W_N^{n_2 k_1} \right] \cdot W_{N_1}^{n_1 k_1} \end{aligned} \quad (37)$$

这样一来，对于输入长度为 N 的离散序列 $x(n)$ ，在二维形式 $\tilde{x}(n_1, n_2)$ 表示下，其

DFT 可分为以下三个步骤计算，如式(38)所示：

1. 计算式(30)中 N_1 个 N_2 点的行向量的 DFT；
2. 将第 1 步的 $N_1 \times N_2 = N$ 个计算结果分别乘上适当的 $W_N^{n_2 k_1}$ 因子，此因子又被称为“旋转因子”；
3. 计算第 2 步计算结果中 N_2 个 N_1 点的列向量的 DFT。

$$\tilde{X}(k_1, k_2) = \underbrace{\sum_{n_1=0}^{N_1-1} \left[\underbrace{\sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_{N_2}^{n_2 k_2}}_{N_2 \text{点FFT}} \cdot \underbrace{W_N^{n_2 k_1}}_{\text{旋转因子}} \right]}_{N_1 \text{点FFT}} \cdot W_{N_1}^{n_1 k_1} \quad (38)$$

这个式子就是所谓的库利—图基分解公式。根据式(32)给出的映射法则，上式也

可以写为一维的形式，即：

$$X(k_1 + N_1 k_2) = \sum_{n_1=0}^{N_1-1} \left[\left(\sum_{n_2=0}^{N_2-1} x(N_2 n_1 + n_2) W_{N_2}^{n_2 k_2} \right) \cdot W_N^{n_2 k_1} \right] \cdot W_{N_1}^{n_1 k_1} \quad (39)$$

这个公式具有递归性质，其中长为 N_1 和 N_2 的序列可以按照此法继续分解为更短的序列，构成分级的运算结构，直到每一序列都短至可以快捷计算为止。

当输入序列的点数 N 可表示为 $N = p^m$ (m 为正整数， p 通常为较小的素数) 时，库利—图基分解公式具有特殊的形式：

$$\tilde{X}(k_1, k_2) = \sum_{n_1=0}^{p^r-1} \left[\left(\sum_{n_2=0}^{p^s-1} \tilde{x}(n_1, n_2) W_{p^s}^{n_2 k_2} \right) \cdot W_{p^m}^{n_2 k_1} \right] \cdot W_{p^r}^{n_1 k_1} \quad (40)$$

式中 $r + s = m$ 。这一特例中 $p = 2$ 是最常使用的情况，也是很多流行的 FFT 算法的适用范围。根据实际需要，本文将讨论集中于此种情况上。

2.6.2 Radix-2 FFT 算法

Radix-2 FFT，即基为 2 的快速傅里叶变换算法是结构最为简单和最为流行的 FFT 算法，这也是库利和图基最早提出的一种 FFT 算法。

观察式(40)，令 $p = 2$ ， $r = 1$ ， $s = m - 1$ ，则我们得到 Radix-2 按频率抽取 (DIF) 算法的表达式：

$$\begin{aligned} \tilde{X}(k_1, k_2) &= \sum_{n_1=0}^1 \left[\left(\sum_{n_2=0}^{2^{m-1}-1} \tilde{x}(n_1, n_2) W_{2^{m-1}}^{n_2 k_2} \right) \cdot W_{2^m}^{n_2 k_1} \right] \cdot W_2^{n_1 k_1} \\ &= \sum_{n_1=0}^1 \left[\left(\sum_{n_2=0}^{\frac{N}{2}-1} \tilde{x}(n_1, n_2) W_{\frac{N}{2}}^{n_2 k_2} \right) \cdot W_N^{n_2 k_1} \right] \cdot W_2^{n_1 k_1} \end{aligned} \quad (41)$$

回到一维的表达形式，我们有：

$$X(2k_2) = \sum_{n_2=0}^{\frac{N}{2}-1} \left[x(n_2) + x\left(n_2 + \frac{N}{2}\right) \right] \cdot W_{\frac{N}{2}}^{n_2 k_2} \quad (42)$$

$$X(2k_2 + 1) = \left\{ \sum_{n_2=0}^{\frac{N}{2}-1} \left[x(n_2) - x\left(n_2 + \frac{N}{2}\right) \right] \cdot W_{\frac{N}{2}}^{n_2 k_2} \right\} \cdot W_N^{n_2} \quad (43)$$

式(42)和(43)分别为输出频域序列偶数项和奇数项的表达式。运用式(32)，可以将奇数项和偶数项写入一个式子，如下所示：

$$\begin{aligned}
 X(k) &= \sum_{n=0}^{\frac{N}{2}-1} \left[x(n) + (-1)^k \cdot x\left(n + \frac{N}{2}\right) \right] \cdot W_N^{kn} \\
 &= \sum_{n=0}^{\frac{N}{2}-1} x(n) \cdot W_N^{kn} + \sum_{n=\frac{N}{2}}^N x(n) \cdot W_N^{kn}
 \end{aligned} \tag{44}$$

可以看出，Radix-2 的 DIF 算法每一级将输入序列对半分为 $0 \sim N/2-1$ 和 $N/2 \sim N-1$ 点两个部分，是一种逐步细分的（Divide-and-Conquer）算法，它的运算主要集中于第一级分解上。

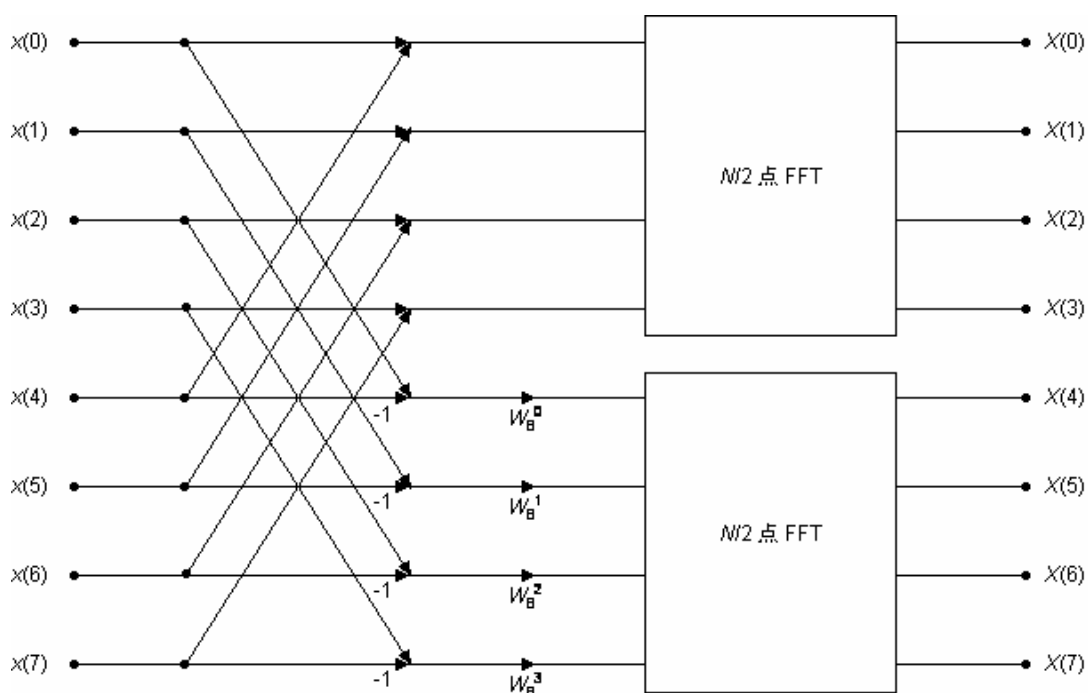


图 2 Radix-2 DIF 算法蝶形运算图

类似地，若令式(40)中的 $p = 2$, $r = m - 1$, $s = 1$ ，则可以得到 Radix-2 的按时间抽取（DIT）算法，它的运算复杂度与按频率抽取算法是一致的，只是其运算主要集中于最后一级分解上；另外，DIT 算法为输入序列逆位序，输出序列正序；DIF 算法恰恰相反，它对正序的输入序列进行运算，输出序列逆位序。

2.6.3 Radix-4 FFT 算法

Radix-4 FFT 算法与 Radix-2 算法类似，是一种具有完美递归结构的算法，且前者比后者运算过程中的复乘和复加数目更少，计算复杂度有所降低。同时，4 点序列也是可以无需乘法进行 FFT 运算的最长序列长度。

令式(40)中的 $p = 4$, $r = 1$, $s = m - 1$, 可以得到 Radix-4 DIF 算法的库利—图基分解表达式, 这也是一种逐步细分的算法, 所不同的是, Radix-4 算法每一级将输入序列分解成四个部分, 表达为:

$$X(k) = \sum_{n=0}^{\frac{N}{4}-1} x(n) \cdot W_N^{kn} + \sum_{n=\frac{N}{4}}^{\frac{N}{2}-1} x(n) \cdot W_N^{kn} + \sum_{n=\frac{3N}{4}}^{\frac{N}{2}-1} x(n) \cdot W_N^{kn} + \sum_{n=\frac{3N}{4}}^{N-1} x(n) \cdot W_N^{kn} \quad (45)$$

与 Radix-2 算法相比, Radix-4 FFT 算法约需要相同数量的复数加法和 75% 左右的复数乘法, 计算量有所降低; 需要付出的代价是运算结构明显复杂了。

2.6.4 Split-Radix FFT 算法

1989 年, Matin Vetterli 和 Pierre Duhamel 二人在他们的论文^[15]中提出, 对于长度为 p^m 的离散序列, 使用 p 和 p^2 两种长度的基对输入序列进行分解, 得到的 Radix- p/p^2 FFT 算法的计算复杂度将比 Radix- p 和 Radix- p^2 算法均要更低, 他们将这种新的算法命名为 Split-Radix FFT, 或简称 SR-FFT。

在此基础上, 对于我们关心的 $p = 2$ 的情况, Radix-2/4 算法将比上面所提的 Radix-2 和 Radix-4 算法更为便捷。Matin Vetterli 和 Pierre Duhamel 同时证明, Radix-2/4 算法对于长度为 2、4、8 和 16 输入序列都是最优的, 因此, 若需要对 Radix-2/4 算法进行改进, 只能够通过编写更为快捷的 32 点 FFT, 构成 Radix-2/32 算法来实现^[15]。

下面着重就成熟的 Radix-2/4 算法进行讨论。从前两节的内容我们知道, Radix-2 算法每一级将长度为 N 的输入序列 $x(n)$ 对称地分为两部分, 即 $x(0) \dots x(N/2 - 1)$ 和 $x(N/2) \dots x(N - 1)$; Radix-4 算法则将输入序列等分为四部分, 即 $x(0) \dots x(N/4 - 1)$ 、 $x(N/4) \dots x(N/2 - 1)$ 、 $x(N/2) \dots x(3N/4 - 1)$ 和 $x(3N/4) \dots x(N - 1)$ 。

Radix-2/4 FFT 将这两种算法作了适当的折衷, 它每一级将输入序列分为三个部分, 即 $x(0) \dots x(N/2 - 1)$ 、 $x(N/2) \dots x(3N/4 - 1)$ 和 $x(3N/4) \dots x(N - 1)$, 对第一部分施以 Radix-2 算法, 对后两段序列则使用 Radix-4 算法进行运算, 得到的三部分结果乘上特定的旋转因子后, 分别在按此法继续分解, 如此对原始序列进行不断细化, 直到完全分解为长度为 2 点和 4 点序列为止。根据这种规律, 可以知道输出序列 $X(k)$ 相应地被划分为 $X(2k_2)$ 、 $X(4k_2' + 1)$ 和 $X(4k_2' + 3)$ 三部分, 其中

$k_2 = 0, 1, \dots, N/2 - 1, \quad k_2' = 0, 1, \dots, N/4 - 1。$

按照式(42)所示, Radix-2/4 FFT 输出序列的偶数项 $X(2k_2)$ 可以表达为:

$$X(2k_2) = \sum_{n=0}^{\frac{N}{2}-1} \left[x(n) + x\left(n + \frac{N}{2}\right) \right] \cdot W_N^{nk_2}, \quad (46)$$

$$k_2 = 0, 1, \dots, \frac{N}{2} - 1$$

而 $X(4k_2'+1)$ 和 $X(4k_2'+3)$ 项可用下面的方法求得; 根据库利-图基分解公式(39), 令 $N_1 = 4, N_2 = N/4$, 可以得到:

$$X(4k_2 + 1) = \sum_{n_1=0}^3 \left[\left(\sum_{n_2=0}^{\frac{N}{4}-1} x(n_2 + \frac{N}{4}n_1) W_{\frac{N}{4}}^{n_2k_2} \right) \cdot W_N^{n_2} \right] \cdot W_4^{n_1} \quad (47)$$

$$= \sum_{n_2=0}^{\frac{N}{4}-1} [x_R(n_2) - i \cdot x_I(n_2)] \cdot W_{\frac{N}{4}}^{n_2k_2} \cdot W_N^{n_2}$$

以及:

$$X(4k_2 + 3) = \sum_{n_1=0}^3 \left[\left(\sum_{n_2=0}^{\frac{N}{4}-1} x(n_2 + \frac{N}{4}n_1) W_{\frac{N}{4}}^{n_2k_2} \right) \cdot W_N^{3n_2} \right] \cdot W_4^{3n_1} \quad (48)$$

$$= \sum_{n_2=0}^{\frac{N}{4}-1} [x_R(n_2) + i \cdot x_I(n_2)] \cdot W_{\frac{N}{4}}^{n_2k_2} \cdot W_N^{3n_2}$$

其中:

$$x_R(n_2) = x(n_2) - x\left(n_2 + \frac{N}{2}\right) \quad (49)$$

$$x_I(n_2) = x\left(n_2 + \frac{N}{4}\right) - x\left(n_2 + \frac{3N}{4}\right)$$

由式(46)、(47)和(48)构成了 Split-Radix 2/4 FFT 算法所有输出项的表达。将这些表达式转换成蝶形运算图, 就构成了 Split-Radix FFT 算法颇具特色的“L”字蝶形, 如图 3 所示。

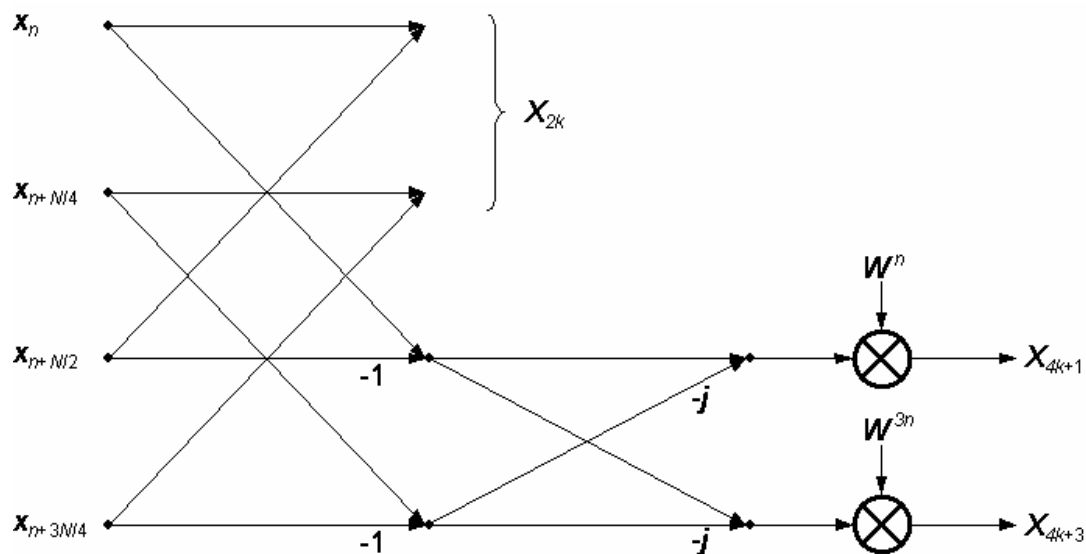


图 3 Split-Radix FFT 的 L 字蝶形图

2.6.5 其它 FFT 算法

除了库利—图基分解，还有其它的分解方法将输入的一维序列映射到二维形式下，从而构成多种不同的 FFT 算法。事实上，这些将一维序列 $x(n)$ 应设为二维序列 $\tilde{x}(n_1, n_2)$ 的更通用的映射表达式为（Burrus, 1977）：

$$n = An_1 + Bn_2 \bmod N \quad (50)$$

类似的，将 $X(k)$ 映射到 $\tilde{X}(k_1, k_2)$ 的映射表达式为：

$$k = Ck_1 + Dk_2 \bmod N \quad (51)$$

其中， A 、 B 、 C 和 D 均为整数常量。

将(50)和(51)代入式(24)，可得到以下的形式：

$$\begin{aligned} \tilde{X}(k_1, k_2) &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_N^{(An_1+Bn_2)(Ck_1+Dk_2)} \\ &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_N^{An_1Ck_1} W_N^{An_1Dk_2} W_N^{Bn_2Ck_1} W_N^{Bn_2Dk_2} \end{aligned} \quad (52)$$

式(50)和(51)中的求模运算符 $\bmod$ 在(52)中被省略，这是因为“ N 次单位根” W_N 的值是以 N 为周期循环的。

2.6.5.1 质因子（Prime-Factor）FFT 算法

与库利—图基分解类似，假设输入序列长度为 N ， N 可分解为两因子 N_1 和 N_2 之积。如果 N_1 和 N_2 是互素的，即 $N = N_1 \times N_2$ ， $(N_1, N_2) = 1$ （括号“ $()$ ”代表最大公约数），则可以采用所谓的质因子（Prime-Factor）FFT 算法。这种算法的优点在于，在其分解表达式中可以完全消除旋转因子，从而使运算复杂度进一步降低。

但是这种简化也是需要付出代价的，质因子算法的映射公式并不直观，并且需要使用孙子剩余定理（CRT）^[11]。Burrus（1985）和 Blahut（1985）指出了使用 CRT 建立质因子 FFT 映射的细节，一个常见的例子如下：

$$A = \alpha N_2, \quad B = \beta N_1, \quad C = \gamma N_2, \quad D = \delta N_1 \quad (53)$$

其中， α 、 β 、 γ 和 δ 均为整数常量。

这样，式(52)变为：

$$\tilde{X}(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_N^{\alpha N_2 n_1 \gamma N_2 k_1} W_N^{\alpha N_2 n_1 \delta N_1 k_2} W_N^{\beta N_1 n_2 \gamma N_2 k_1} W_N^{\beta N_1 n_2 \delta N_1 k_2} \quad (54)$$

运用性质 $W_N^{N_2} = W_{N_1}$ 和 $W_N^{N_1} = W_{N_2}$ ，由于 N_1 和 N_2 互素，故有 $W_N^{\alpha N_2 n_1 \gamma N_2 k_1} = W_{N_1}^{\alpha N_2 n_1 \gamma k_1}$ 和

$W_N^{\beta N_1 n_2 \delta N_1 k_2} = W_{N_2}^{\beta N_1 n_2 \delta k_2}$ ；运用性质 $W_N^N = 1$ ，则有 $W_N^{\alpha N_2 n_1 \delta N_1 k_2} = W_N^{\beta N_1 n_2 \gamma N_2 k_1} = 1$ 。因此式

(54)可以化简为：

$$\begin{aligned} \tilde{X}(k_1, k_2) &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_{N_1}^{\alpha N_2 n_1 \gamma k_1} \times 1 \times 1 \times W_{N_2}^{\beta N_1 n_2 \delta k_2} \\ &= \sum_{n_1=0}^{N_1-1} \left[\sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_{N_2}^{\beta N_1 n_2 \delta k_2} \right] W_{N_1}^{\alpha \gamma N_2 n_1 k_1} \end{aligned} \quad (55)$$

上式就是质因子 FFT 的分解公式，与式(37)比较可以看出，(55)式中完全没有旋转因子。下面的重复更加清楚地示意了这一点：

$$\tilde{X}(k_1, k_2) = \underbrace{\sum_{n_1=0}^{N_1-1} \left[\underbrace{\sum_{n_2=0}^{N_2-1} \tilde{x}(n_1, n_2) W_{N_2}^{\beta N_1 n_2 \delta k_2}}_{N_2 \text{ 点 FFT}} \right]}_{N_1 \text{ 点 FFT}} \cdot W_{N_1}^{\alpha \gamma N_2 n_1 k_1} \quad (56)$$

质因子算法对输入输出序列由一维向二维的映射涉及较为繁复的运算，包括求模（mod）运算，以及对原始序列非连续的数据读写，这一点在计算机上实现

时将消耗较多的时间。

2.6.5.2 Winograd FFT 算法 (WFTA)

WFTA 是质因子算法的一种, 它的运算核心采用高效的卷积算法, Blahut(1985)全面地阐述了 WFTA 算法的推导和结论。从运算所需要的复数乘法数量的角度, WFTA 是运算量较低, 十分快捷的一种算法; 对于输入长度为 N 的序列, 它仅需要 $O(N)$ 数量级的复乘。但同时, 这种算法需要比前面所述的几种算法更多的复加, 并且其结构和数据流程也是所有 FFT 算法中最为复杂和不规则的之一。

2.7 小结

在这一章的前 4 节, 我们简要讲述了医学超声多普勒技术, 尤其是脉冲多普勒技术的原理, 介绍了多普勒信号处理和谱分析的原理和方法。在后几节中, 我们重点讨论了离散傅里叶变换 (DFT) 和快速傅里叶变换 (FFT) 的历史、理论推导和常用的算法。从这些基础知识我们了解到, FFT 是脉冲多普勒系统谱分析器的核心; FFT 有多种算法分支, 库利—图基算法由于其结构简单, 实现方便, 是其中使用较为普遍的分支, 这种算法基于著名的“库利—图基分解”; Split-Radix FFT 是库利—图基算法中较为高效的一种。

第3章 基于 PC 平台的医学超声成像系统

3.1 机遇和动力

当今的世纪是一个科技迅猛发展的时代，各领域成果的不断推陈出新使人应接不暇。作为 20 世纪的代表性、革命性科技，计算机科学与技术几十年来迅猛发展，为人类生活、生产和娱乐带来了翻天覆地的变化，“摩尔定律 (Moore's Law)¹”已成为家喻户晓的名词，而这个发展和变化的势头在 21 世纪依然锐势不减。

个人电脑，即 PC，是计算机科学与技术发展的主导。几十年来，它以微处理器技术的变革为核心，总线技术和数据存储技术的成熟为依托，毫不动摇地恪守着“摩尔定律”，经历着一轮又一轮的更新换代，速度不断翻番。

目前，个人电脑已走进寻常百姓的家庭。在成本不断降低的同时，PC 的性能也直逼甚至超越了传统的工业计算机和专用工作站，这使得以往许多需要借助于这些专业平台进行的工作可以移植到 PC 系统上。

PC 系统速度、性能的提升也为医学超声成像系统的设计和改良提供了新的研究机遇。以往，医学成像系统对实时性的苛刻要求致使这些系统只能依赖于精心选择的多处理器工作站，或通过专用硬件实现。然而时至今日，通用 PC 系统的性能已能完全胜任这一工作。由于诸多的优越性，一股使用通用 PC 平台实现医学超声成像系统的研究热流开始流向世界各地。

使用 PC 实现的医学成像系统主要有以下优点：

- 体积小巧。借助现代 PC 系统的性能优势，我们可以把越来越多的处理环节放入软件中实现，这使得整个系统硬件简单，可集成在 ASIC 中，精致小巧；
- 灵活性强。由于更多的使用软件代替硬件完成处理功能，这使得系统开发灵活，便于扩展和升级，并适用于各种通用的 PC 系统；
- 成本低廉。软件开发成本低于硬件，硬件的减少使系统整体成本大大降低；

¹ 在 20 世纪 60 年代中叶，Intel 公司主席 Board Gordon Moore 总结出规律：“The number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years.”。

- 利于普及。PC 已进入千家万户，加之低廉的成本有助于吸引用户。

3.2 主要实现难点

在 PC 平台上实现医学超声成像系统的优势在于可利用现代微处理器的强大运算功能，实现更多的信号变换、处理和显示工作，而无需硬件的干预。一个基于 PC 的脉冲多普勒超声血流成像系统的结构框图如下所示：

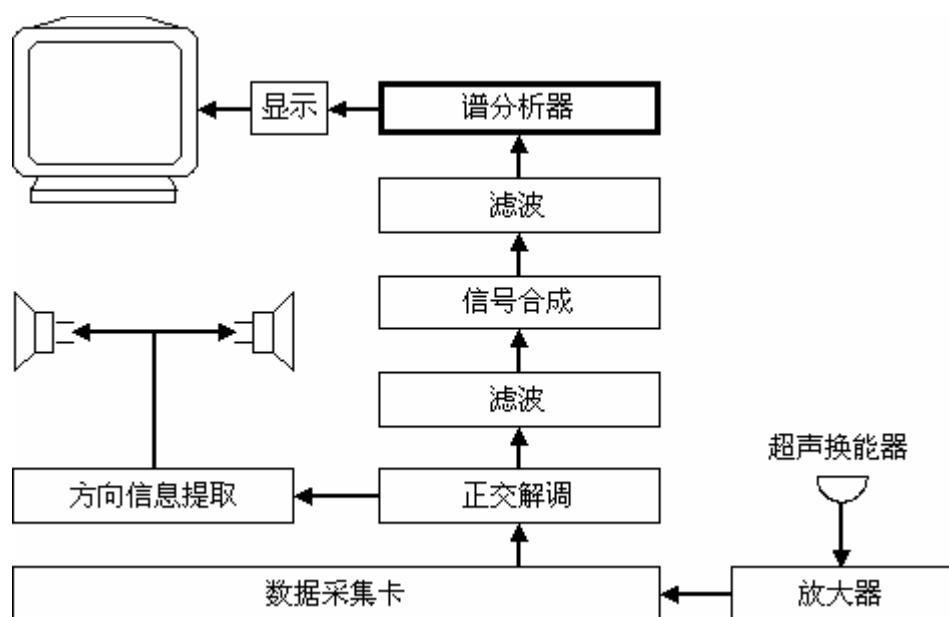


图 4 基于 PC 的脉冲多普勒超声血流成像系统

通常一个医学超声成像系统需要使用软件实现的运算包括：

- 数字信号的解调、滤波、谱分析、特征信息识别和提取等；
- 图像的灰度变换、增强、坐标变换、三维建模等；
- 其它智能化的分析和处理。

尽管现代 PC 微处理器的性能已十分强大，但对于这些实时性要求较高的处理和计算，仍需使用诸多技巧，以使运算速度加快。尤其是对于一些基本且重要的功能，例如信号的谱分析、灰度变换和显示等，更需要格外优化，使它们占用尽可能少的计算时间，以将微处理器的绝大部分运算能力预留给更复杂和高级的处理功能。

另一方面，我们将会看到，虽然现代通用微处理器的功能愈发强大，但使用传

统的编程方法却根本无法发挥它们的全部性能。如何运用新的编程技术，充分利用现代微处理器的新特性和发掘它们的速度潜力，成为下几章我们主要讨论的问题。在第4章中，我们将以具有FFT谱分析器在目前主流的Intel Pentium III/4 PC平台上的优化实现为例，讲解这一议题。

3.3 PC 技术发展趋势

微处理器技术的飞速发展造就了今日PC技术的腾飞。自从20世纪60年代集成电路的发明时起，微处理器速度一直遵照着“摩尔定律”，速度成倍提高。

时至今日，微处理器技术的发展形成两种潮流和趋势：一方面，适用于兼容PC系统的通用型微处理器（General Purpose Microprocessor）从字长、速度和功能三条路线不断提升和强化；另一方面，瞄准便携式设备和嵌入式系统的中小规模微处理器则不断朝着低功耗、低成本和专用化方向发展。本文主要关心的是基于通用型微处理器的PC技术。

目前PC机上主流的通用微处理器为32位，并很快将向64位过渡。对于相同位宽的微处理器，它的速度以时钟频率和每时钟周期平均执行的指令数（IPC）两大主要因素为指标，其中时钟频率占据主导地位。截至本文完稿时止，以Intel Pentium 4为代表的32位主流微处理器的主频已远远突破2GHz，而与微处理器相配合的最新芯片组则将系统总线速度提升到了400~533MHz，配合高速的Rambus内存（RDRAM）技术，系统内存带宽被前所未有的提高到了3.2GB/s，使PC系统的整体性能得到极大改善。

与此同时，现代信息技术的发展将我们带入了多媒体和网络时代，当今的PC机无不带有丰富且强大的多媒体和网络功能。衡量现代PC系统性能的指标主要有三^[2]：整数运算能力、浮点数运算能力和多媒体性能。为了提高这三方面的指标，现代通用微处理器相继融合了诸多更强大的运算和处理功能，这些功能以SIMD指令集的增加为代表。

SIMD是Single Instruction, Multiple Data（单指令流，多数据流）²的英文缩写，它是一种并行处理技术^[3]，使用这种技术可用单一的计算机指令对多个连续数据

² 与SIMD相对应的技术还有MIMD（Multiple Instruction, Multiple Data），它通过多处理器同时运行指令对数据进行并行处理。

进行并发操作，从而使处理速度成倍提高。一个简单的示意图如图 5 所示：

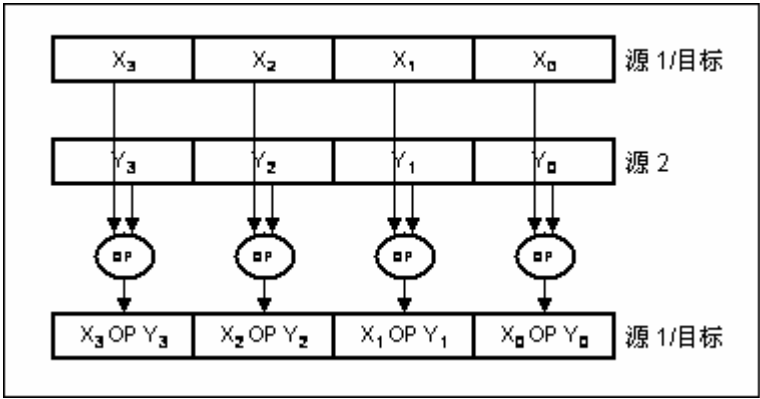


图 5 四数据 SIMD 并行运算示意图

图中的“OP”代表一个算术或逻辑运算符，如“加”、“减”、“乘”、“除”、“与”、“或”、“非”等等；X₀...X₃是源 1 / 目标寄存器中的值，Y₀...Y₃是源 2 寄存器的值，它们同时经过 OP 运算后，得到的结果存入源 1 / 目标寄存器。上述例子的运算是通过一条 SIMD 指令完成的，它同时对 4 个数据进行操作。

在通用微处理器中加入 SIMD 指令集最初是为了满足多媒体技术（包括音、视频信号的编、解码，网络流媒体播放等）和 3D 实时动画应用的要求。这一新特性的引入取得了巨大的成功，使得多媒体计算机的实时处理能力大幅提升。现在，我们也可以将这一新特性用于科学计算以及其它应用领域，同样可以取得显著的效果。

目前主流的通用微处理器主要有 Intel、AMD 等生产厂家，Intel 系列微处理器的 SIMD 指令集包括 MMX(Multimedia Extensions)、SSE(Streaming SIMD Extensions)和 SSE2(Streaming SIMD Extensions 2)，AMD 处理器则在在支持 MMX 指令集的同时，以自己的 3DNow!指令集与 Intel 的 SSE 和 SSE2 相抗衡。表 1 到表 3 罗列了这 4 种指令集的特点，并比较了它们的异同。

指令集	指令数	适用范围
MMX	57	• Pentium Processor with MMX™ technology • Intel Pentium II Processor (and Above) • Intel Celeron Processor • AMD K6-2/III Processors • AMD Athlon Processor

SSE	70	- Intel Pentium III/4 Processors (and Above) - Intel Celeron Processor - Intel Xeon Processor
SSE2	144	- Intel Pentium 4 Processor (and Above) - Intel Xeon Processor
3DNow!	21	- AMD K6-2/III Processors - AMD Athlon Processor
Extensions to 3DNow! and MMX	24	AMD Athlon Processor

表 1 MMX/SSE/SSE2/3DNow! 指令集比较 (1)

指令集	寄存器名称	寄存器位数	说明
MMX	MM0, MM1, ..., MM7	64	映射到 8 个 80 位 x86 FPU 寄存器
SSE	XMM0, XMM1, ..., XMM7	128	新增的 8 个寄存器
SSE2	XMM0, XMM1, ..., XMM7	128	
3DNow!	MM0, MM1, ..., MM7	64	复用 MMX 寄存器

表 2 MMX/SSE/SSE2/3DNow! 指令集比较 (2)

指令集	新增特性
MMX	- 新增 8 个 64 位 MMX 寄存器³，每寄存器可同时容纳 8 个字节 (Byte)、4 个字 (Word)、2 个双字 (Double-word) 或 1 个四字 (Quad-word)； - 支持的数据类型包括 8 位、16 位、32 位和 64 位整型数； - 指令功能涉及数据传输、整数的算术 / 逻辑 / 比较 / 移位运算、数据类型转换等。
SSE	新增 8 个 128 位 XMM 寄存器，每寄存器可同时容纳 16 个字节、8 个字、4 个双字、2 个四字或 1 个双四字 (Double Quad-word)；

³ 在硬件上这 8 个寄存器被映射到原有的 x87 FPU 的 8 个 80 位寄存器 (ST(0), ST(1), ..., ST(7)) 上，因此，在进行运算时，MMX 指令集和 x87 浮点数运算指令集不能混合使用，详情请参阅[18]。

	• 支持的数据类型包括单精度浮点数（32 位）和 8 位、16 位、32 位及 64 位整型数； • 指令功能涉及数据传输、单精度浮点数算术 / 逻辑 / 比较运算、数据类型转换、增强的整数运算和高速缓存控制等。
SSE2	• 在 SSE 指令集的基础上增加对双精度浮点数（64 位）和 128 位整数的支持； • 在 SSE 指令集的基础上增加新的数据传输和增强的数据类型转换指令，增加双精度浮点数算术 / 逻辑 / 比较运算指令以及整数的算术 / 逻辑 / 比较 / 移位运算等指令。
3DNow!	• 功能包括单精度浮点数的算术 / 比较和数据类型转换等。

表 3 MMX/SSE/SSE2/3DNow! 指令集比较（3）

各处理器厂家的 SIMD 指令集在本质上作用是一致的。由于 Intel Pentium III/4 系列微处理器在 PC 中使用更为广泛，在本文中主要针对适用于这类处理器的 MMX、SSE 和 SSE2 指令集进行讨论。

3.4 Intel Pentium III/4 的 SIMD 技术

Intel 将其生产的一系列适用于通用 PC 平台的 32 位微处理器归为 IA-32 处理器家族，其中 IA 代表 Intel Architecture，即英特尔架构。

Intel 生产的第一款具备 SIMD 运算能力的微处理器是 1997 年问世的 Pentium Processor (200, 166 MHz) with MMX Technology（带有 MMX 技术的奔腾处理器）。它运用 MMX 技术，可同时对 MMX 寄存器中的数据进行 2 条 32 位、4 条 16 位或 8 条 8 位整数运算，数倍地提高了多媒体应用中的整数运算速度。由于 MMX 技术的巨大成功，在随后的 Intel 处理器中，MMX 已成为标准的配置，很多软件公司在编制软件时，都默认 MMX 指令集的存在，并加以使用。

SSE 指令集是第一套面向浮点运算的 SIMD 指令集，它最初出现在 Intel Pentium III 系列微处理器中。浮点运算是二维、三维动画和实时显示中频繁使用的运算，可以说，没有 SSE 指令集的支持，众多炫目的 3D 游戏和精彩的网络实时流媒体将难以展现其今日的风采。SSE 指令包含 57 条主要针对单精度浮点运算的指令，而随着 Intel Pentium 4 处理器而引入的 SSE2 指令集则增加了对双精度浮点运

算的支持，从此 Intel 微处理器的运算能力得到了全面提升。

MMX、SSE 和 SSE2 中的运算指令均包含两类：组合（Packed）运算指令和标量（Scalar）运算指令。相应地，参与组合运算的数据称为组合数据，参与标量运算的则称为标量数据。只有组合运算指令才是真正意义上的 SIMD 指令。这两种指令的工作方式示意图如图 6 所示。

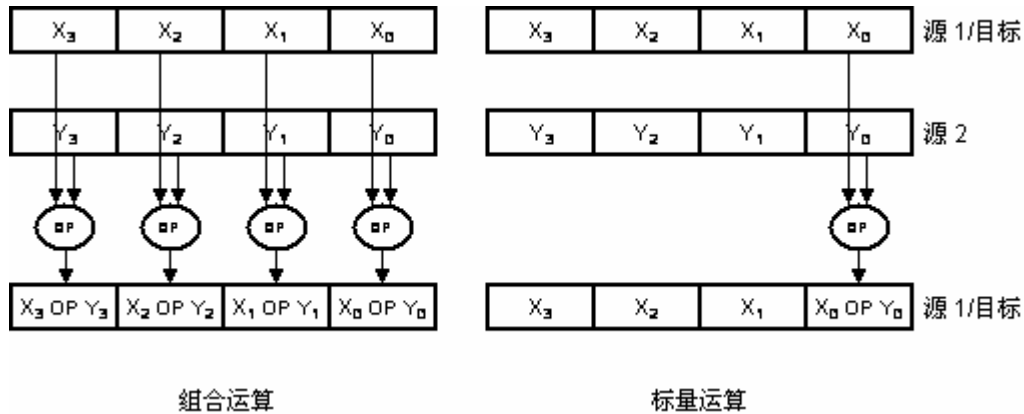


图 6 组合运算和标量运算示意图

从第 4 章的讨论我们将会知道，FFT 谱分析器使用的将主要是单精度浮点数运算，并且考虑到此谱分析器对主流微处理器（Pentium III/4）的兼容性，我们将以使用 SSE 指令集为主，MMX 技术为辅，优化我们的程序。下面几节的介绍可使我们对 SSE 指令集有一个提纲挈领式的了解。

3.4.1 我的系统支持 SSE 吗？

虽然大致上看 Pentium III 及以后的 Intel 微处理器都是支持 SSE 技术的，但使用编程确认 SSE 的存在还是一种较为保险和妥善的做法。判断 PC 系统对 SSE 的支持分为两个方面，一是判断微处理器是否支持 SSE，二是确认操作系统提供执行 SSE 指令集的环境。下面简要给出判断这两个条件的方法，更详细的信息请参阅[17]。

3.4.1.1 判断处理器对 SSE 的支持

要判断微处理器对 SSE 技术的支持，可以使用 Intel 处理器的 CPUID 指令，该指令根据 EAX 寄存器中放置的参数不同，在 EAX、EBX、ECX 和 EDX 寄存器中返回

微处理器的品牌、特性或型号等信息。

若 EAX 的初始值为 1，CPUID 将在 EDX 寄存器中返回处理器所支持的功能信息，其格式如图 7 所示。

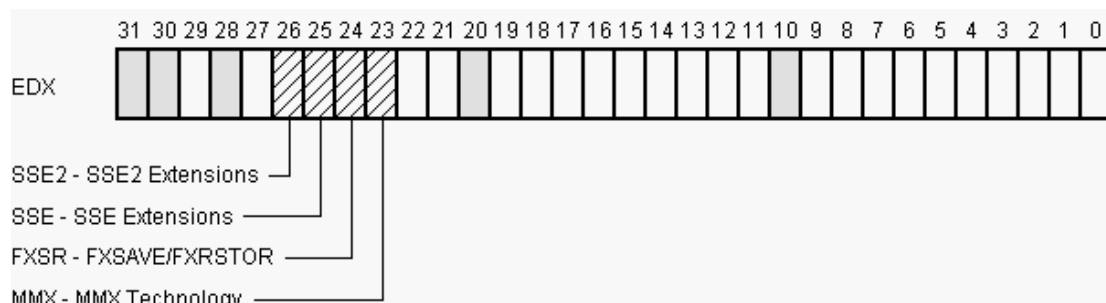


图 7 CPUID 指令在 EDX 寄存器中返回的信息格式

图中有斜线阴影标出的位是我们感兴趣的位，灰色的位是保留位，其它的位标志了其它功能信息，我们并不关心，这里省略不写。这样，我们就可以根据检查 EDX 的第 25 位是否为“1”来判断处理器对 SSE 的支持，如下面的代码所示：

```
//
// -----
//  NAME: sse_hw_support
//
//  DESC: Determines whether the processor has SSE support.
//
//  Return values:
//  * 1 - The processor supports SSE extensions.
//  * 0 - The processor does not support SSE extensions.
//  * -1 - The processor does not support CPUID instruction.
//  * -2 - The processor is not an Intel processor.
//  * -3 - Unexpected exception occurred.
// -----
//
int sse_hw_support(void)
{
    int    result;
    char   brand[12];

    /*
     * STEP1: Determine whether the processor has CPUID,
     * and it is "GenuineIntel".
     */
    __try
```

```

{
    __asm
    {
        mov     eax, 0
        cpuid
        mov     brand, ebx        // ebx contains "Genu"
        mov     brand + 4, edx    // edx contains "ineI"
        mov     brand + 8, ecx    // ecx contains "ntel"
    }
}

__except(EXCEPTION_EXECUTE_HANDLER)
{
    if (_exception_code() == STATUS_ILLEGAL_INSTRUCTION)
        return -1;
    return -3;
}

if (!strcmp(brand, "GenuineIntel", 12)) return -2;

/*
 * STEP2: Determine whether the processor supports SSE extensions.
 */
__asm
{
    mov     eax, 1
    cpuid
    xor     eax, eax              // If bit 25 of edx is 1,
    test    edx, 02000000h       // set eax to 1,
    cmovnz  eax, 1               // otherwise eax is 0.
    mov     result, eax
}

return result;
}

```

3.4.1.2 判断操作系统对 SSE 的支持

微处理器为了执行 SSE 指令，引入了一系列新的状态标志，操作系统必须具有存取和管理这些状态信息的能力。一般而言，当下面两个条件满足时，表示一个操作系统可以保证 SSE 指令的顺利执行：

- 位于控制寄存器 0 (CR0) 的浮点计算仿真标志位无效，即 CR0.EM=0;

- 位于控制寄存器 4 (CR4) 的 OSFXSR 位有效, 即 CR4.OSFXSR=1。
- 其中, 第一个条件是 SSE 和 MMX 指令顺利执行的前提。而第二个条件又取决于以下两点:
- 处理器支持 FXSAVE 和 FXRSTOR 指令, 可用 CPUID 指令 (EAX=1) 返回处理器的功能标志, 如图 7 所示, 其 FXSR 位 (第 24 位) 应为 1;
 - 操作系统支持使用 FXSAVE 和 FXRSTOR 指令存取 SSE 状态信息。

但是, 由于控制寄存器 (CR0~CR4) 只能由操作系统在处理器特权级 0 (Ring 0) 上存取, 我们不能通过直接读取控制寄存器的状态位来获得所需信息, 必须另辟蹊径。使用 Windows 的结构化异常机制可以方便的完成这一点。这实际上是一种试探的方法: 我们试着在程序中执行一条 SSE 指令, 若所在的操作系统不支持执行此指令, 则会抛出一条 STATUS_ILLEGAL_INSTRUCTION 异常, 这一异常可被程序捕获, 从而判断系统对 SSE 的支持。一个示范性的函数如下, 注意此函数应该在确认处理器支持 SSE 指令集的基础上调用。

```
//
// -----
//  NAME: sse_os_support
//
//  DESC: Determines whether the operating system has SSE support.
//
//      Return values:
//      * 1 - The processor supports SSE extensions.
//      * 0 - The processor does not support SSE extensions.
//      * -1 - Unexpected exception occurred.
// -----
//
int sse_os_support(void)
{
    /*
     * Execute an instruction using SSE to see if support exists.
     */
    __try
    {
        __asm xorps xmm0, xmm0
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        if (_exception_code() == STATUS_ILLEGAL_INSTRUCTION)
        {
            return 0;
        }
    }
}
```

```

    }
    return -1;
}

return 1;
}

```

3.4.2 SSE 指令集精粹

Intel 的 SSE（Streaming SIMD Extensions）指令集按功能可分为六大类：

- 数据传输指令
- 算术运算指令
- 逻辑运算指令
- 数值比较指令
- 数据类型转换指令

这里我们挑选一些常用的指令进行简要介绍。

指令	说明	
MOVAPS ⁴	格式	MOVAPS xmm1, xmm2/m128 ⁵ MOVAPS xmm2/m128, xmm1
	功能	将 128 位（16 字节）的组合数据从 16 字节对齐的内存空间传送到 XMM 寄存器，或将其从寄存器传送到内存，或在寄存器间传送。
MOVUPS ⁶	格式	MOVUPS xmm1, xmm2/m128 MOVUPS xmm2/m128, xmm1
	功能	将 128 位（16 字节）组合数据从非 16 字节对齐的内存空间传送到 XMM 寄存器；或将其从寄存器传送到内存，或在寄存器间传送。
MOVSS	格式	MOVSS xmm1, xmm2/m32 MOVSS xmm2/m32, xmm1
	功能	将 32 位（4 字节）的标量数据从内存传送到 XMM 寄存

⁴ SSE 指令有统一的后缀约定。“PS”代表组合单精度数据（Packed Single），“SS”代表标量单精度数据（Scalar Single）。

⁵ 这里的 `xmm1` 和 `xmm2` 代表任意两个 XMM 寄存器，`m128` 代表 128 位（16 字节）的内存空间，类似地，`m64` 和 `m32` 分别代表 64 位（8 字节）和 32 位（4 字节）的内存空间。

⁶ 对非 16 字节对齐的内存空间读写速度比 16 字节对齐的空间慢，一般应尽量使用 16 字节对齐的地址和 `MOVAPS` 指令。用 `MOVAPS` 指令对非 16 字节对齐的内存进行读写将导致一般性保护（GP）错误。

		器低 32 位；或将其从寄存器传送到内存，或在寄存器间传送。
MOVLPS	格式	MOVLPS xmm1, xmm2/m64 MOVLPS xmm2/m64, xmm1
	功能	将 64 位（8 字节）的组合数据从内存传送到 XMM 寄存器低 64 位；或将其从寄存器传送到内存，或在寄存器间传送。
MOVHPS	格式	MOVHPS xmm1, xmm2/m64 MOVHPS xmm2/m64, xmm1
	功能	将 64 位（8 字节）的组合数据从内存传送到 XMM 寄存器高 64 位；或将其从寄存器传送到内存，或在寄存器间传送。
MOVLHPS	格式	MOVLHPS xmm1, xmm2
	功能	将 XMM 寄存器低 64 位（8 字节）组合数据移动到高 64 位，原高 64 位数据被覆盖，低 64 位不变。
MOVHLPS	格式	MOVHLPS xmm1, xmm2
	功能	将 XMM 寄存器高 64 位（8 字节）组合数据移动到低 64 位，原低 64 位数据被覆盖，高 64 位不变。
ADDPS	格式	ADDPS xmm1, xmm2/m128
	功能	将 xmm1 寄存器中的 4 个单精度浮点组合数据与 xmm2 或 16 字节对齐的内存空间中的 4 个单精度浮点数相加，结果存入 xmm1 寄存器。
ADDSS	格式	ADDSS xmm1, xmm2/m32
	功能	将 xmm1 寄存器低 32 位中的单精度浮点标量数据与 xmm2 低 32 位或 16 字节对齐的内存空间中的 1 个单精度浮点数相加，结果存入 xmm1 寄存器低 32 位。
SUBPS	格式	SUBPS xmm1, xmm2/m128
	功能	将 xmm1 寄存器中的 4 个单精度浮点组合数据（被减数）与 xmm2 或 16 字节对齐的内存空间中的 4 个单精度浮点数（减数）相减，结果存入 xmm1 寄存器。
SUBSS	格式	SUBSS xmm1, xmm2/m32
	功能	将 xmm1 寄存器低 32 位中的单精度浮点标量数据与 xmm2 低 32 位或 16 字节对齐的内存空间中的 1 个单精度浮点数相减，结果存入 xmm1 寄存器低 32 位。
MULPS	格式	MULPS xmm1, xmm2/m128

	功能	将 <code>xmm1</code> 寄存器中的 4 个单精度浮点组合数据与 <code>xmm2</code> 或 16 字节对齐的内存空间中的 4 个单精度浮点数相乘，结果存入 <code>xmm1</code> 寄存器。
MULSS	格式	<code>MULSS xmm1, xmm2/m32</code>
	功能	将 <code>xmm1</code> 寄存器低 32 位中的单精度浮点标量数据与 <code>xmm2</code> 低 32 位或 16 字节对齐的内存空间中的 1 个单精度浮点数相乘，结果存入 <code>xmm1</code> 寄存器低 32 位。
DIVPS	格式	<code>DIVPS xmm1, xmm2/m128</code>
	功能	将 <code>xmm1</code> 寄存器中的 4 个单精度浮点组合数据除以 <code>xmm2</code> 或 16 字节对齐的内存空间中的 4 个单精度浮点数，结果存入 <code>xmm1</code> 寄存器。
DIVSS	格式	<code>DIVSS xmm1, xmm2/m32</code>
	功能	将 <code>xmm1</code> 寄存器低 32 位中的单精度浮点标量数据除以 <code>xmm2</code> 低 32 位或 16 字节对齐的内存空间中的 1 个单精度浮点数，结果存入 <code>xmm1</code> 寄存器低 32 位。
ANDPS	格式	<code>ANDPS xmm1, xmm2/m128</code>
	功能	将 <code>xmm1</code> 寄存器中的 4 个单精度浮点组合数据与 <code>xmm2</code> 或 16 字节对齐的内存空间中的 4 个单精度浮点数相与，结果存入 <code>xmm1</code> 寄存器。
ORPS	格式	<code>ORPS xmm1, xmm2/m128</code>
	功能	将 <code>xmm1</code> 寄存器中的 4 个单精度浮点组合数据与 <code>xmm2</code> 或 16 字节对齐的内存空间中的 4 个单精度浮点数相或，结果存入 <code>xmm1</code> 寄存器。
XORPS	格式	<code>XORPS xmm1, xmm2/m128</code>
	功能	将 <code>xmm1</code> 寄存器中的 4 个单精度浮点组合数据与 <code>xmm2</code> 或 16 字节对齐的内存空间中的 4 个单精度浮点数相异或，结果存入 <code>xmm1</code> 寄存器。
SHUFPS	格式	<code>SHUFPS xmm1, xmm2/m128, imm8⁷</code>
	功能	寄存器内数据重排指令。根据 <code>imm8</code> 低 4 位从 <code>xmm1</code> 中选取 2 个单精度浮点数置入 <code>xmm1</code> 的低 64 位；根据 <code>imm8</code> 高 4 位从 <code>xmm2</code> 中选取 2 个单精度浮点数置入 <code>xmm1</code> 的高 64 位。具体方法如下： <code>case imm8[1:0] of</code>

⁷ 这里的 `imm8` 代表 8 位立即数。

		00: $\text{xmm1}[31:0] \leftarrow \text{xmm1}[31:0]$ 01: $\text{xmm1}[31:0] \leftarrow \text{xmm1}[63:32]$ 10: $\text{xmm1}[31:0] \leftarrow \text{xmm1}[95:64]$ 11: $\text{xmm1}[31:0] \leftarrow \text{xmm1}[127:96]$ case imm8[3:2] of 00: $\text{xmm1}[63:32] \leftarrow \text{xmm1}[31:0]$ 01: $\text{xmm1}[63:32] \leftarrow \text{xmm1}[63:32]$ 10: $\text{xmm1}[63:32] \leftarrow \text{xmm1}[95:64]$ 11: $\text{xmm1}[63:32] \leftarrow \text{xmm1}[127:96]$ case imm8[5:4] of 00: $\text{xmm1}[95:64] \leftarrow \text{xmm2}[31:0]$ 01: $\text{xmm1}[95:64] \leftarrow \text{xmm2}[63:32]$ 10: $\text{xmm1}[95:64] \leftarrow \text{xmm2}[95:64]$ 11: $\text{xmm1}[95:64] \leftarrow \text{xmm2}[127:96]$ case imm8[7:6] of 00: $\text{xmm1}[127:96] \leftarrow \text{xmm2}[31:0]$ 01: $\text{xmm1}[127:96] \leftarrow \text{xmm2}[63:32]$ 10: $\text{xmm1}[127:96] \leftarrow \text{xmm2}[95:64]$ 11: $\text{xmm1}[127:96] \leftarrow \text{xmm2}[127:96]$
UNPCKLPS	格式	UNPCKLPS xmm1, xmm2/m128
	功能	寄存器内数据重排指令。将 xmm1 第 31~0 和第 63~32 位数据分别交错地置入 xmm1 第 95~64 位和第 31~0 位；将 xmm2 第 31~0 和 63~32 位数据分别交错地置入 xmm1 的第 63~32 位和 127~96 位。
UNPCKHPS	格式	UNPCKHPS xmm1, xmm2/m128
	功能	寄存器内数据重排指令。将 xmm1 第 95~64 和第 127~96 位数据分别交错地置入 xmm1 第 95~64 位和第 31~0 位；将 xmm2 第 95~64 和 127~96 位数据分别交错地置入 xmm1 的第 63~32 位和 127~96 位。

表 4 常用的 SSE 指令说明

3.4.3 MMX 指令精粹

这一节中就将就少量的、我们将会使用的 MMX 指令进行简要说明，其余的指令可以参阅[18]。

指令	说明
----	----

MOVQ	格式	MOVQ mm, r/m32 MOVQ r/m32, mm MOVQ xmm, r/m32 MOVQ r/m32, xmm
	功能	将 32 位（4 字节）数据从 MMX 寄存器传送到内存、通用寄存器或 XMM 寄存器，或由内存、通用寄存器或 XMM 寄存器传送到 MMX 寄存器。当源寄存器为 XMM 寄存器时，该寄存器低 32 位数据被传送到目标寄存器；当 XMM 寄存器作为目标寄存器时，数据被传送到该寄存器低 32 位，而其高 96 位被清零；当 MMX 寄存器作为目标寄存器时，数据被传送到该寄存器低 32 位，而其高 32 位被清零。
EMMS	格式	MOVQ mm1, mm2/m64 ⁸ MOVQ mm1/m64, mm2 MOVQ xmm1, xmm2/m64 MOVQ xmm2/m64, xmm1
	功能	将 64 位（8 字节）数据从 MMX 寄存器传送到内存或 XMM 寄存器，或由内存或 XMM 寄存器传送到 MMX 寄存器，或在 MMX 寄存器间传送。当源寄存器为 XMM 寄存器时，该寄存器低 64 位数据被传送到目标寄存器；当 XMM 寄存器作为目标寄存器时，数据被传送到该寄存器低 64 位，而其高 64 位被清零。
MOVQ	格式	EMMS
	功能	清空 MMX 状态，重置 FPU 标志字，使 FPU 寄存器可重新被正常用于浮点数计算。若程序在使用完 MMX 指令后需要进行正常的浮点数计算，必须执行此指令。

表 5 常用的 MMX 指令说明

3.5 小结

这一章介绍了 PC 技术的发展趋势、在 PC 系统上实现医学超声成像系统的可行性和主要困难以及进行此项研究的机遇和动力。然后，我们着重讨论了主流 PC 系统的最新技术，尤其是 Intel Pentium III/4 最新引入的 SIMD 指令集，包括

⁸ 这里的 mm1 和 mm2 指的是任意两个 MMX 寄存器，m64 指 64 位（8 字节）的内存空间。

MMX、SSE 和 SSE2 等，这些将是阅读本文下一章的重要知识基础。

第4章 FFT 谱分析器的优化与实现

4.1 算法选择

在第 2 章中，我们已经介绍了各种 FFT 算法的推导和公式。对于多普勒超声血流成像系统谱计算的应用场合，我们可以总结出如下特点：

1. FFT 序列通常在 64 (2^6) 到 256 (2^8) 点之间，通常为 128 (2^7) 点，属于短时傅里叶变换 (Short-Time Fourier Transform, STFT)；
2. 为了生成流动的声谱图，对计算的实时性要求较高，FFT 运算重复频率快；
3. 用于频谱计算的 FFT 精度要求并不很高，采用单精度浮点数⁹即可达到要求。

这些特点使我们可以在所有适用于 2^m 长度的 FFT 算法中择优选。对于当今最为流行的高主频和具备向量化指令处理能力的 Intel Pentium III/4 计算机架构，我们在选择算法时通常有如下的原则：

1. 数据存取尽量连续，并且在同一时刻，对连续的数据可进行尽可能多相同的运算；这两个特点保证了算法便于向量化；
2. 占用尽量少的存储空间，可以进行“即位运算”，尽可能减少临时变量，减少内存存取，节省有限的内存带宽；
3. 涉及的浮点运算尽可能少，由于 Pentium 系列处理器对浮点乘法和加法的运算速度几乎是一致的，因此总运算量较小的算法较占优势。

根据以上的原则权衡，Split-Radix FFT 算法运算量较小¹⁰，同时又具有库利—图基算法所共有的数据流程简单和便于向量化等特点，是在 Intel Pentium III/4 处理器 PC 上实现的较为理想的算法。

算法	所需复数乘法数量
Radix-2	$\frac{1}{2} N \log_2(N)$
Radix-4	$\frac{3}{8} N \log_2(N)$

⁹ 32 位 Windows 系统中的单精度浮点数范围为 3.4×10^{-38} 到 3.4×10^{38} ，精确到小数点后第 7 位。

¹⁰ 根据文献[11]和[16]，Radix-2、Radix-4 和 Radix-2/4 算法复杂度比较如表 6 所示。

Radix-2/4	$\frac{1}{3}N \log_2(N)$
-----------	--------------------------

表 6 Radix-2、Radix-4 和 Radix-2/4 复数乘法数量比较

4.2 设计思想

在选择好适当的算法后，我们开始着手设计 Split-Radix FFT 在 Intel Pentium III/4 平台上的实现方案。

4.2.1 优化设计策略

通过对[6]、[8]和[9]等文献的研究发现，当今较为成功和成熟的 FFT（以及其它的 DSP 变换）程序为了达到高效率，已摒弃了试图用较短的通用代码实现全部运算的做法；这样尽管程序紧凑，指令较少，但往往具有几个极为致命的弱点：

1. 循环结构复杂；
2. 重复冗余运算多；
3. 内存存取不易控制。

这些弱点使得这样的实现尽管看来结构简明，但在现代微处理器架构下却远远达不到高效的目的，这是因为：

1. 现代通用微处理器通常采用分支预测（Branch Prediction）技术，它使得处理器可以预先估测程序的分支方向，从而提前执行分支后的代码。复杂的循环结构会频繁的使用条件跳转指令（jnz、jc 等），这增加了分支预测的复杂度，并增加了预测错误的机率。对于普遍采用深度流水线（Pipelining）设计的现代通用微处理器¹¹，分支预测错误后将涉及从流水线中恢复执行的操作，这将付出很大的代价，影响处理性能。
2. 传统的算法程序常热衷于以较少的语句实现较多的功能，这使程序中包含大量的冗余计算，结果往往效率不高。经验表明，高级语言代码较少并不意味着程序更为高效；甚至在汇编语言或机器语言的级别，也并不是指令越少，程序执行得越快。之所以这样，其一在于微处理器执行各指令的延迟时间或者是指令周期不同；其二，由于分支预测等技术的存在，指令的选择和先后

¹¹ 如 Intel Pentium 4 的 NetBurst 微结构采用 20 级的“超流水线技术（Hyper Pipelined Technology）”，以适应较高的主频；同时，Pentium 4 引入了“先进动态执行（Advanced Dynamic Execution）”技术，使分支预测精度较上一代微处理器提高了 33%^[9]，用以抵消流水线深度带来的负面影响。

次序等因素都会影响到程序的整体表现；最后，寄存器使用、内存存取等因素也将极大影响程序的执行速度，这是与微处理器系统架构参数密切相关的，这些参数包括系统高速缓存（Cache）的大小、内存带宽、总线速度等。

下面以最经典的算法著作《Numerical Recipes in C》^[12]中给出的 Radix-2 FFT 例程为例说明以上的两点。针对《Numerical Recipes in C》附带的源代码（版本 2.02）中给出的一维 Radix-2 FFT 函数 `four1`（简称为 NR-FFT），我们进行如下的改进：

1. 第一步，对于我们所关心的短时傅里叶变换（STFT）特例（我们选取 4、8、16、32、64、128 和 256 点的情况），将 `four1` 函数原始程序中的循环全部展开，按点数分别保存为新的不含循环的函数“`four1_o1_%d`”，函数名中的“o1”代表第 1 步优化（Optimization），“%d”代表 FFT 的点数，例如 4、8、……、256 等。
2. 第二步，在第 1 步优化的基础上，我们去除程序中计算位逆序索引和 W_N 因子的语句。因为我们只考虑 4、8、16、……、256 点的特例，对于每一种情况，由于位逆序而需要对原始序列进行的重排，以及各 W_N 因子的值都已经固定。这样一来，我们只需要保留对原始序列特定位置的元素进行交换重排的代码，以及利用已知数值的 W_N 因子计算变换结果的代码，程序因此又得到进一步优化。我们将新生成的函数命名为“`four1_o2_%d`”，函数名中的“o2”代表第 2 步优化。
3. 第三步，由第 2 步优化的结果我们看到，程序对 $W_N = 1$ 和 $W_N = i$ 的情况也进行乘法计算，这些运算显然是多余的；此外，对于 $\text{Re}(W_N) = -\text{Im}(W_N)$ 的情况，由于数值的对称性，也可以剔除某些乘法。至此，通过清理这些冗余计算，我们可以进一步减少各函数的指令数量。我们将新生成的函数命名为“`four1_o3_%d`”，函数名中的“o3”代表第 3 步优化。

为了比较以上三步改进对 `four1` 函数执行速度的影响，我们设计了一个测试程序。在程序中，我们用上述函数对不同长度的输入序列进行 FFT 运算，比较它们的执行时间。为了使时间测量更为精确，程序中将每个变换重复 10^6 次，并将输入序列的值全设为“0”¹²，测量执行的总时间，得到的结果如表 7：

点数	NR-FFT	NR-FFT(O1)	NR-FFT(O2)	NR-FFT(O3)
4	1228884200	604465772	100643416	69167420

¹² 由于零序列的 FFT 输出仍然是零序列，这一步避免了非零序列经多次 FFT 运算后数值溢出的问题。

	(100.00%)	(49.19%)	(8.19%)	(5.63%)
8	1920537304 (100.00%)	945588936 (49.24%)	220250460 (11.47%)	197850872 (10.30%)
16	3784360932 (100.00%)	1609690000 (42.54%)	531509612 (14.04%)	501241016 (13.25%)
32	5153208256 (100.00%)	2885039496 (55.99%)	1304194628 (25.31%)	1271703996 (24.68%)
64	10421146992 (100.00%)	6480604560 (62.19%)	4301090120 (41.27%)	4256171528 (40.84%)
128	18131828272 (100.00%)	16175496064 (89.21%)	12216455544 (67.38%)	11539749040 (63.64%)
256	41403850992 (100.00%)	37064333788 (89.52%)	30386998268 (73.39%)	27707820516 (66.92%)

表 7 对 Numerical Recipes FFT 的改进和执行时间比较

表中, NR-FFT 代表《Numerical Recipes in C》中给出的例程, 而 NR-FFT(O1)、NR-FFT(O2)、NR-FFT(O3)分别表示前面所述的三个改进版本; 表中的数据是相应的函数被执行 10^6 次所耗费的时钟周期数, 我们将这些数据绘制成曲线, 可以清楚的看到前述三步优化带来的速度提高, 如图 8 所示。

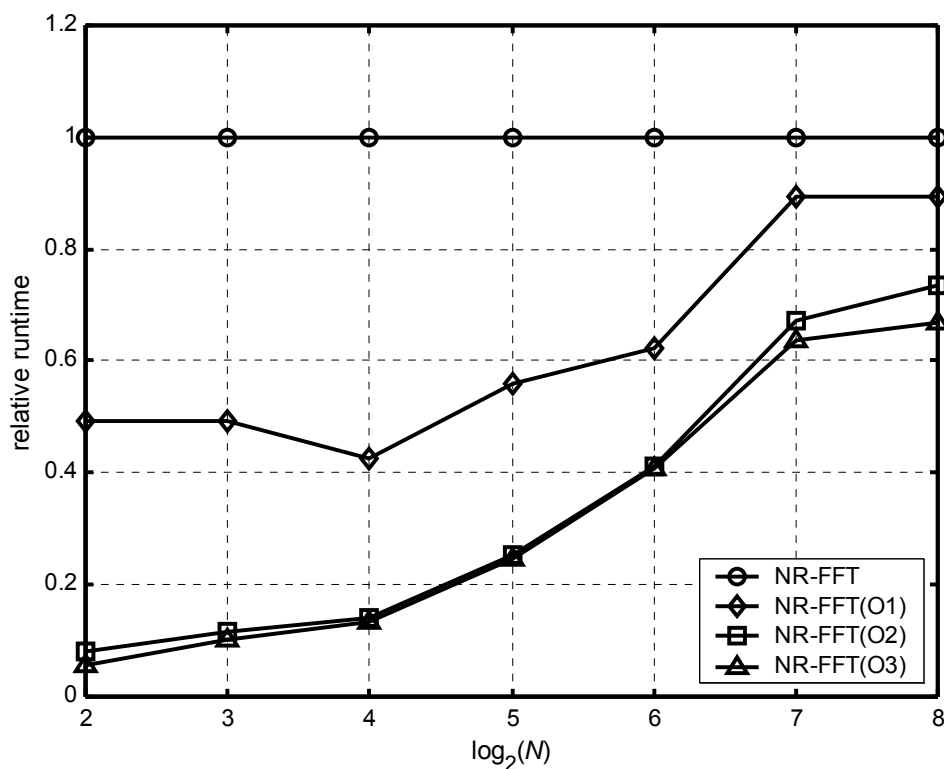


图 8 对 Numerical Recipes FFT 的改进和执行时间比较

通过以上的比较，我们可以得出以下的结论：

1. 代码中的循环结构会给程序执行带来额外的 CPU 时间支出，尤其对短序列的 FFT 影响更为明显；经循环展开后的代码性能可以获得较大的提高，但副作用是代码长度明显增加，若使用中 / 高级编程语言（如 C/C++），则相应的编译时间也会明显增加。而且，一个函数的代码太长对程序的执行效率也会有影响，因为代码太长会超出微处理器高速缓存的容纳能力，因此并不总是循环展开得越多，程序执行速度越快。
2. 对于特定点数序列的 FFT，我们可以通过有针对性的优化而得到非常高效的代码。这是由于点数一旦确定，则相应的逆位序过程和 W_N 因子的值也就固定了，我们可以通过查表等方法迅速计算出结果，从而省略了大量计算。

以上二点结论构成了我们设计高效的短时傅里叶变换软件包的基础。其中第 2 点结论尤其重要，因为由此我们可以通过对某些短序列（如 2、4、8 和 16 点）FFT 进行特殊的优化，构成不含循环且极为高效的运算内核，而更长序列的 FFT 则通过逐步细化（Divide-and-Conquer）的方法分解为短序列 FFT，通过调用这些高效的内核进行计算。

4.2.2 程序结构设计

从上一节的结论中我们已经得出了高效的 FFT 算法程序结构设计的思路。事实上，这些思路已经被一些研究人员所采纳，并得到了很大的成功。例如目前国际上公认的最为成功的通用 FFT 软件包 FFTW（The Fastest Fourier Transform in the West, <http://www.fftw.org/>）^{[8][9]}就是一个很好的案例。

在 FFTW 软件包中，一个特定长度序列的 FFT 并不是通过固定的一段程序计算的，FFTW 库内部包含很多计算不同长度短序列（短至 2 点，长至 64 点）FFT 变换的代码段，称为“小代码（Codelet）”，它们构成了 FFTW 的运算内核。这些“小代码”实际就是一些高度优化的程序片段，它们是由作者经过无数次的实验，综合比较各种算法的优劣而编写出的，并可通过一种特殊的编译器^{13[8]}根据指定的长度自动生成。同时，这些 Codelets 还分为两种类型，即 Normal Codelets 和 Twiddle Codelets^[10]，Normal Codelets 计算特定长度的 FFT，而

¹³ 这一编译器包含在 FFTW 软件包中，名为“genfft”。

Twiddle Codelet 除了进行 Normal Codelets 所进行的计算外，还将结果乘上适当的旋转因子（Twiddle Factors）。

FFTW 在计算一个任意长度序列的 FFT 时，首先将序列分解为多个较短的序列。有时，一个序列可能包含多种不同的分解方案，程序会自动找到最便捷的方案。然后，对较短序列的 FFT 的计算就可以通过调用内核“小代码”完成。下面举例简单说明这一过程。假设输入序列长度为 128，对它的计算包含以下步骤：

```
Divide-and-Conquer(128, 4);  
Divide-and-Conquer(32, 8);  
Solve(4);
```

首先，程序发现 128 点序列可被分解为四个长度为 32 的子序列，并调用 4 点 Twiddle Codelets 将各子问题的计算结果相接合。然后，程序发现所划分的 32 点子序列可以继续划分为八个 4 点子问题，从而可以调用 4 点的 Normal Codelets 解决，同时调用 8 点的 Twiddle Codelets 接合子问题的计算结果。

FFTW 为了找到序列的最佳分解方案，还应用了所谓“动态编程（Dynamic Programming）”的概念，生成具有自适应（Adaptive）能力的软件包，可根据运行平台的不同，自动选择速度最快的分解方法和算法组合。对这些问题的讨论超出了本文的范围，因为我们目前仅针对一种计算机平台，即主流的 Intel Pentium III/4 架构平台，并假定我们所得到的最优结果对同一系列的计算机平台都是适用的。

结合所选择的 Radix-2/4 算法，下面陈述了我们所选择的程序结构设计方案：

1. 采用逐步细分（Divide-and-Conquer）的 Split-Radix 算法（参见 2.6.4 节的说明），输入序列正序，输出时需要逆位序；
2. 使用针对 Intel Pentium III/4 向量化指令集高度优化的 FFT“小代码”内核；根据实际需要，我们采用了 4 点、8 点和 16 点三种内核；FFT 和 IFFT 分别有各自的一套内核；更长序列的细化过程使用一个通用函数完成，以减少编码量，该函数含有循环结构； W_N 系数预先计算好并存入查找表中。
3. 对 4、8、16、32、64、128、256、512、1024 和 2048 点的 FFT 分别有各自的接口函数，该函数输入（/ 出）序列长度固定，使用直列代码，无循环结构，采用程序自动生成（4.6 节中将详细介绍）；入口函数根据 Split-Radix 算法规则，以 $1/2, 1/4, 1/4$ 的方式分解输入序列，调用内核函数完成运算；FFT 和 IFFT 分别有各自的一套入口函数；另有一通用的接口函数，根据序

列长度和正逆变换等参数，调用相应的入口函数完成运算。

4. 将该 FFT 计算程序封装成独立的软件包，采用 C 语言静态链接库 (*.lib) 或 Windows 动态链接库 DLL (Dynamic Link Library) 的形式，输出一系列接口函数，以供我们的医学超声多普勒血流成像系统主程序调用。

我们将此 FFT 运算软件包命名为“SFFT”。对于刚才举过的 128 点序列的例子，使用 SFFT 的计算过程如下图所示（假设采用 8 点和 4 点两种内核）：

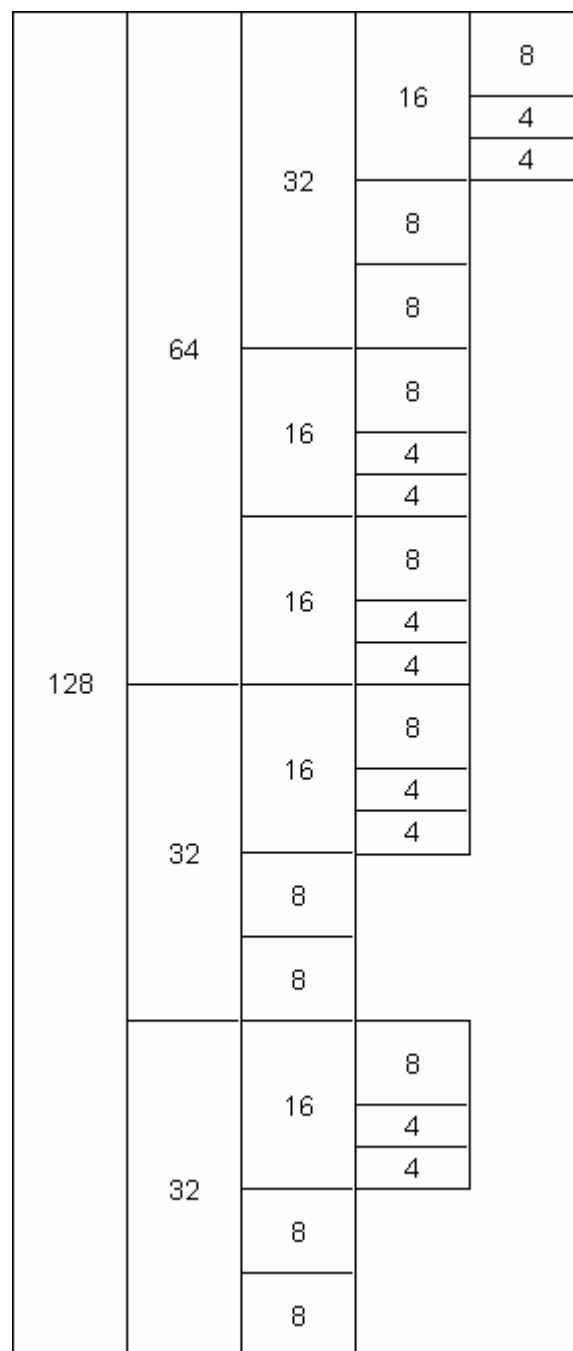


图 9 128 点 Split-Radix 分解流程图

4.3 汇编语言 vs. C/C++

我们选择了嵌入式汇编语言（Inline Assembly）来编写 SFFT，之所以作出这样的选择，其原因有以下几点：

1. 汇编语言无疑是众多编程语言中速度最快、最贴近硬件的语言，它可以使我们更加深入计算机的底层，观察不同的指令选择和编排对运行速度的影响；
2. 汇编语言可以使我们以更深的层次透视 FFT 算法，理解 FFT 运算中完整的数据流程。同时，也方便我们对所编写的代码进行统计分析，例如可获知浮点乘法、加法指令的数量、寄存器内部重排指令的数量、内存读写指令的数量以及有无循环结构、有无使用临时变量等，这些信息容许我们在未来建立分析程序性能的统计模型；
3. 汇编语言不受任何高级语言编译器的限制，因此可以尽可能的发掘主流微处理器，如 Intel Pentium III/4 中的最新功能和最大潜力；汇编语言中可以包含许多高级语言不支持的特性，例如使用 Intel MMX、SSE 或 SSE2 指令集优化代码，这些是高级语言编译器所做不到的¹⁴；
4. 嵌入式汇编是镶嵌在 C/C++ 语言函数代码中的汇编语言，使用 “__asm” 关键字标识，这是使我们既可以领略到汇编语言的速度优势，又可以与编辑和调用 C/C++ 语言函数一样方便地调用它们。

下面举几个简单的例子说明内嵌式汇编的使用方法。一个最简单的例子如下所示，它计算 $num \times 2^{power}$ 的值：

```
int power2( int num, int power )
{
    __asm
    {
        mov eax, num    ; Get first argument
        mov ecx, power  ; Get second argument
        shl eax, cl      ; EAX = EAX * ( 2 to the power of CL )
    }
    /* Return with result in EAX */
}
```

¹⁴ 事实上，Intel 公司出品的 C++ Compiler 5.0/6.0 已经支持使用 MMX/SSE/SSE2 等 SIMD 指令集优化 C/C++ 代码。但是这些编译器不能自动发现传统 C/C++ 语言源程序中的并行性，而需要依照 SIMD 指令集的习惯，手工编写具有并行性质的 C/C++ 源代码，编译器方可成功译出 SIMD 指令优化的代码。参阅[7]。

```
}
```

根据 C/C++ 语言的规范，函数一般使用 `EAX` 或 `EDX:EAX` 寄存器保存返回值。其中，32 位的返回值存入 `EAX` 寄存器中，而 64 位的返回值则将低 32 位存入 `EAX`，高 32 位存入 `EDX`。因此，以上的代码最后将计算结果存储在 `EAX` 中，而没有显式地书写 `return` 语句。

另外，C/C++ 语言使用 `esp` 作为堆栈指针（Stack Pointer），`ebp` 作为框架指针（Frame Pointer），这两个寄存器的值一般在嵌入式汇编中是不宜改变的，否则程序堆栈和指示入口参数位置的基地址指针将会错乱，得到错误的入口参数的地址和返回地址，导致程序不能正确执行。在 4.4 节中我们将看到，通过采用特殊的办法保存这些寄存器，我们还是可以在嵌入式汇编程序中使用它们的。若想得到更多的关于嵌入式汇编语言的帮助，可以参阅 MSDN^[22]。

4.4 小代码内核的优化实现

4.4.1 4 点 FFT 内核的实现

4 点 FFT Codelet 是 SFFT 最小的内核函数，它计算指定偏移地址处连续 4 个数据的 FFT 变换。4 点 Split-Radix FFT 的蝶形运算流程如图 10 所示：

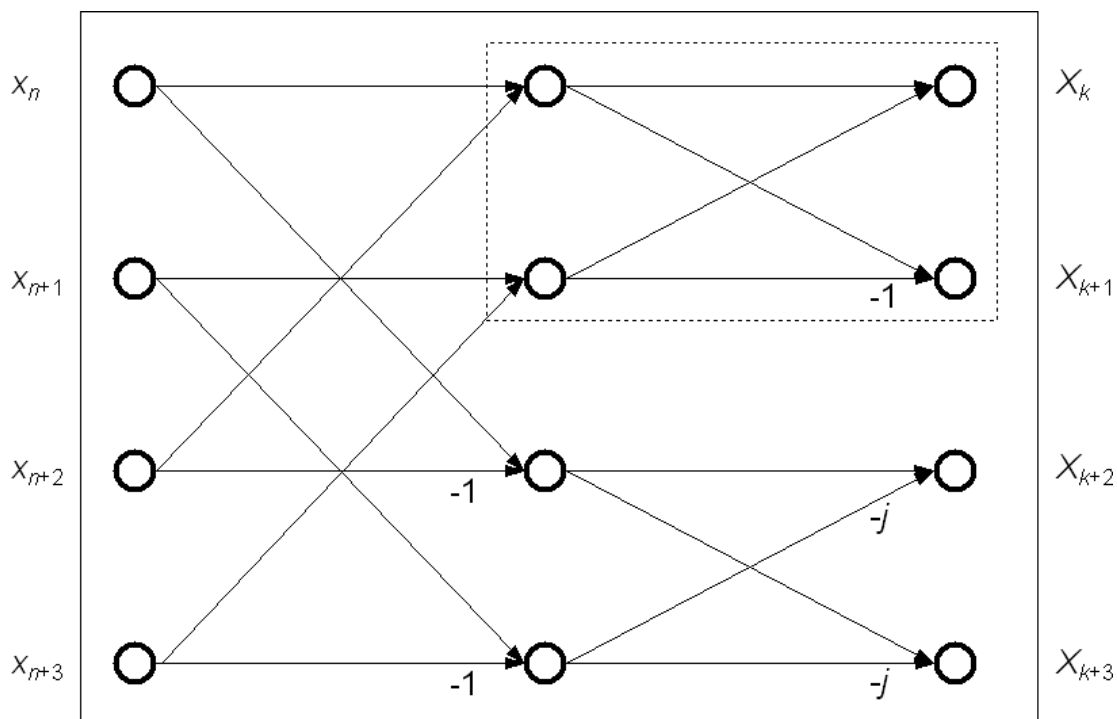


图 10 4 点 Split-Radix FFT 蝶形运算图

它由一个 4 点的 L 字碟形和一个 2 点 FFT 所组成（如图中虚框所示）。需要注意的是，图 10 以及其它所谓的蝶形图（参见第 2.6 节）都是对复数运算而言，这对实际编程和优化的意义不大，因为编程时需要对输入数据的实部和虚部分别运算，因此了解各点实部和虚部的数据流向是重要的，见下图：

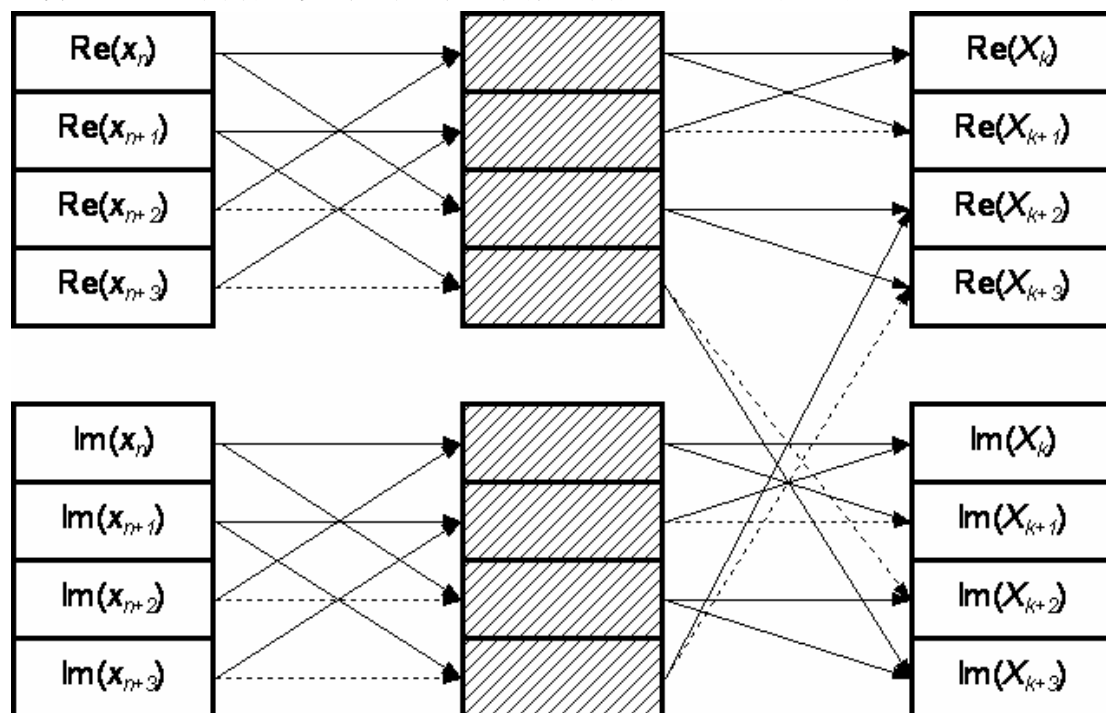


图 11 4 点 Split-Radix FFT 实、虚部数据流向图

在图 11 中，实 / 虚线分别代表加 / 减法运算，而划有斜线的方框代表暂存的中间数据，这并不需要引入临时变量，而完全可以容纳进 Intel Pentium III/4 CPU 的 XMM 寄存器中（参见 3.4 节）。

可以看出，对于输入序列的实部和虚部的运算已经不是完美的蝶形了。为了适应我们的程序结构（实部数组和虚部数组分开），图 11 将数据的实部和虚部分为两部分表示，即 $\text{Re}(x(n)) \dots \text{Re}(x(n+3))$ 和 $\text{Im}(x(n)) \dots \text{Im}(x(n+3))$ 。同时，由于每个 XMM 寄存器可以存储 4 个单精度（32-bit）浮点数，图 11 以 4 个数据为一个单元，使我们可以看出数据在寄存器间应进行的交换和移动。

下面我们着手 FFT 运算的优化，其中重要的一步是发现运算中的并行性，并将这些可并行进行的运算四四合并，以适应 SIMD 指令集的操作。对于图 11 左起第一组碟形，实际的运算为：

$$\begin{aligned}
 R'_0 &= R_0 + R_2 \\
 R'_1 &= R_1 + R_3 \\
 R'_2 &= R_0 - R_2 \\
 R'_3 &= R_1 - R_3
 \end{aligned}
 \tag{57}$$

式中的 R_0 、 R_1 、 R_2 和 R_3 分别代表 $\text{Re}(x(n)) \dots \text{Re}(x(n+3))$ ，由于虚部的运算和实部是一致的，这里从略。

显而易见，式(57)所示的运算是可以合并的，这是因为减法终究可以归并为加法：

$$\begin{aligned}
 R'_0 &= R_0 + R_2 \\
 R'_1 &= R_1 + R_3 \\
 R'_2 &= R_0 + (-R_2) \\
 R'_3 &= R_1 + (-R_3)
 \end{aligned}
 \tag{58}$$

但这步运算进行前需要对寄存器中的输入数据进行重排，通过对式(58)进行等价变换，我们有两种方案，如下图所示：

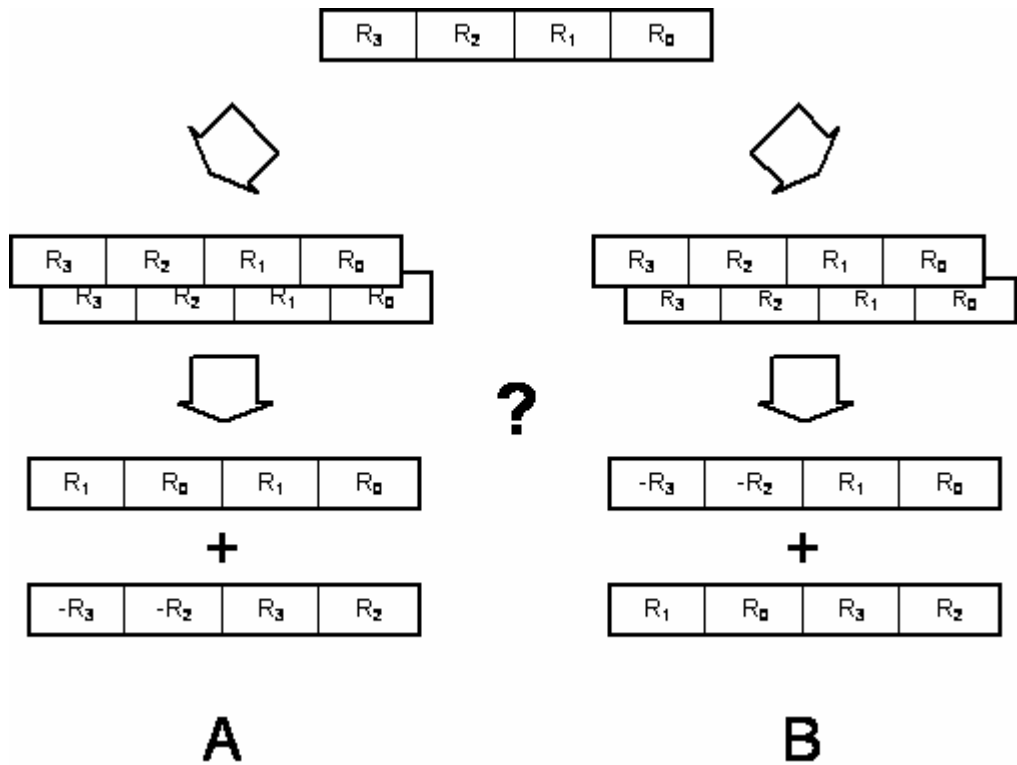


图 12 4 点 Split-Radix FFT：并行加法的两种方案

可见，方案 A 需要对输入数据寄存器进行两次重排，而方案 B 只需一次；另外，两种方案都需要对寄存器中的两个数据求反。结合 3.4 的说明，SSE 寄存器内重排（使用 `shufps` 指令）是一种费时的操作，因此 B 方案更胜一筹。

至于两个数据的求反操作，也有两种方案，如所示。方案 C 需要进行一次浮点乘法操作，而方案 D 只需要一次逻辑异或（XOR）操作，显然方案 D 更快捷。

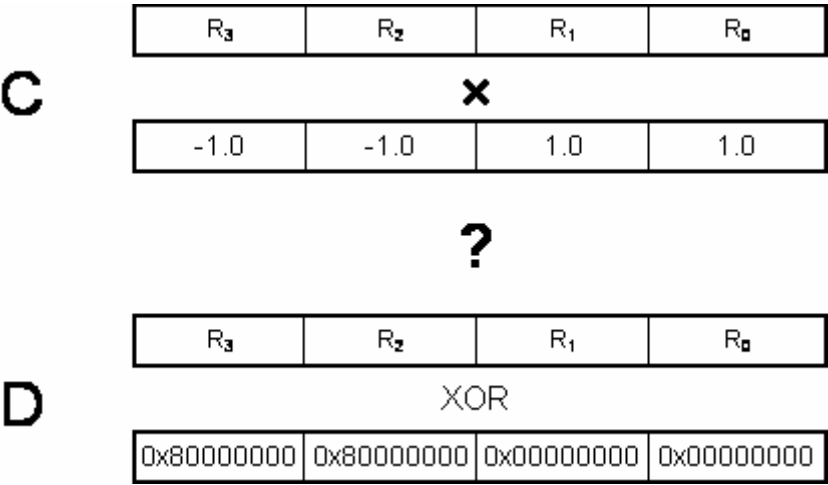


图 13 4 点 Split-Radix FFT：数值求反的两种方案

可以使用异或（XOR）进行求反运算的基础在于 Intel 架构 PC 机中采用的 IEEE 754 标准浮点数存储格式^[18]，它规定浮点数的二进制最高位为符号位，将这一位异或逻辑“1”，就可以改变该浮点数的符号。

图 11 中的第二组蝶形需要进行的操作是：

$$\begin{aligned} R_0'' &= R_0' + R_1' & I_0'' &= I_0' + I_1' \\ R_1'' &= R_0' - R_1' & I_1'' &= I_0' - I_1' \\ R_2'' &= R_2' + I_3' & I_2'' &= I_2' - R_3' \\ R_3'' &= R_2' - I_3' & I_3'' &= I_2' + R_3' \end{aligned} \tag{59}$$

其中， I_0 、 I_1 、 I_2 和 I_3 分别代表 $\text{Im}(x(n))\dots\text{Im}(x(n+3))$ ，字母上的“'”代表第一步的结果。这几个运算可以通过 4 个 SSE 寄存器重排指令、2 个 SSE 异或和 2 个 SSE 加法指令完成，流程如下图所示：

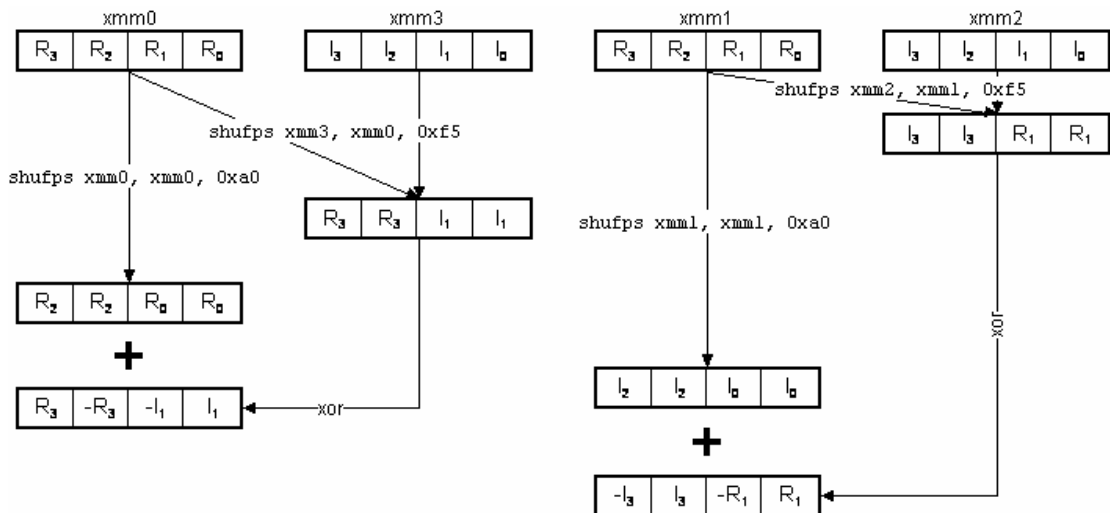


图 14 4 点 Split-Radix FFT：第二组蝶形运算

至此，我们成功完成了 4 点 Split-Radix FFT 内核的优化实现。现将此实现的指令统计数据罗列如下：

SIMD 加法 ¹⁵	SIMD 乘法	寄存器内重排	内存读取 ¹⁶	内存写入 ¹⁷
4	0	6	4	2

表 8 4 点 FFT 内核指令数统计

4.4.2 8 点 FFT 内核的实现

与 4 点内核的实现类似，我们可以尽可能地发现 8 点和 16 点 Split-Radix FFT 内部存在的并行性，从而可将大量的运算合并。由于篇幅的关系，这一节以及下一节中，我们不再详述 FFT 内核的实现细节，而仅给出 8 点和 16 点 Split-Radix FFT 的蝶形运算图，及 8 点 FFT 的实、虚部数据流程图。同时，我们也列出了这两个 FFT 内核的指令统计结果。

¹⁵ SIMD 减法也包含在内，指令执行速度与加法一致。

¹⁶ 这里的内存读取包括载入两个用于数值求反的异或因子的操作。

¹⁷ 内存读取和内存写入都以 16 字节，即 4 个单精度浮点数为单位。

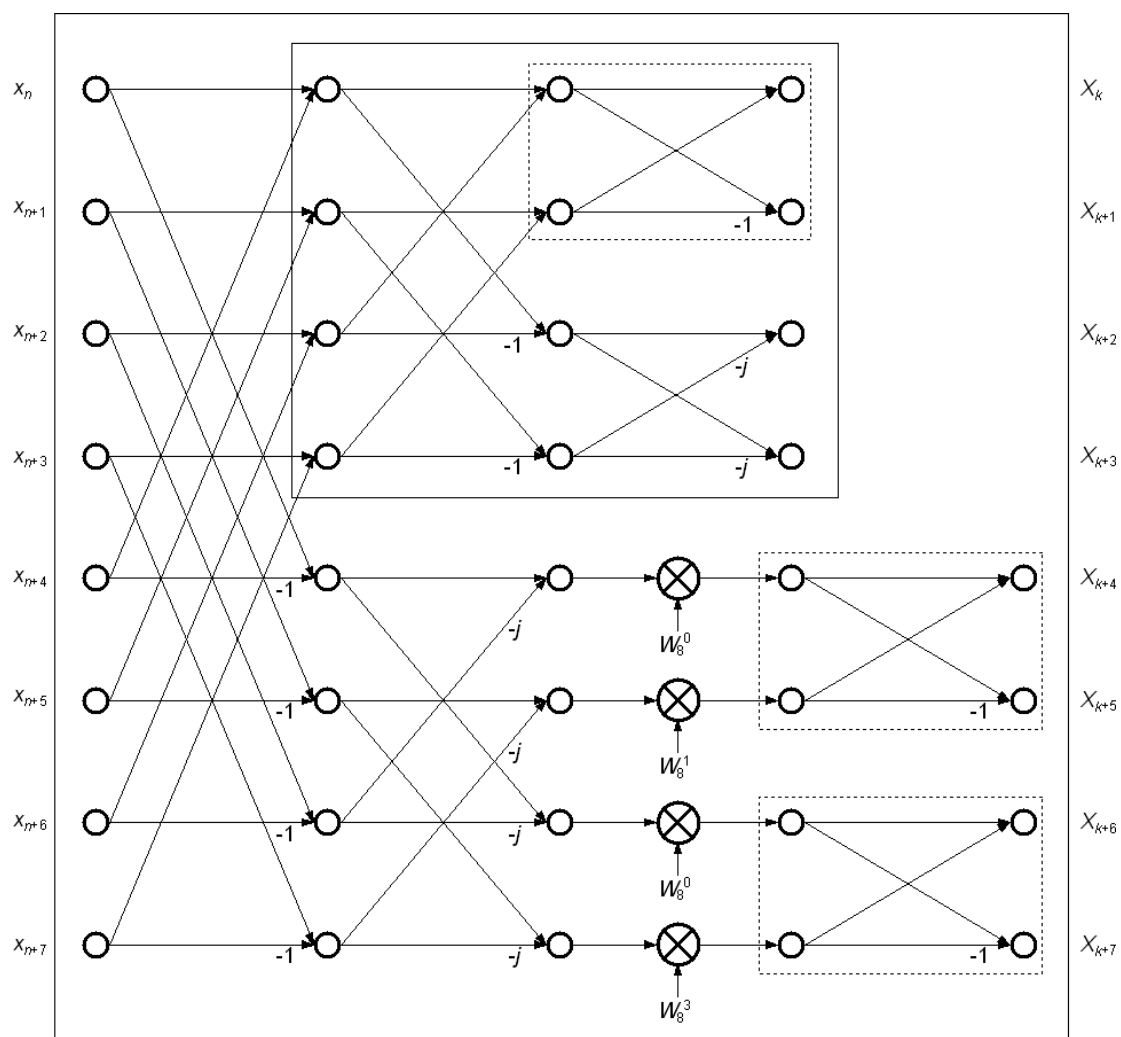


图 15 8 点 Split-Radix FFT 蝶形运算图

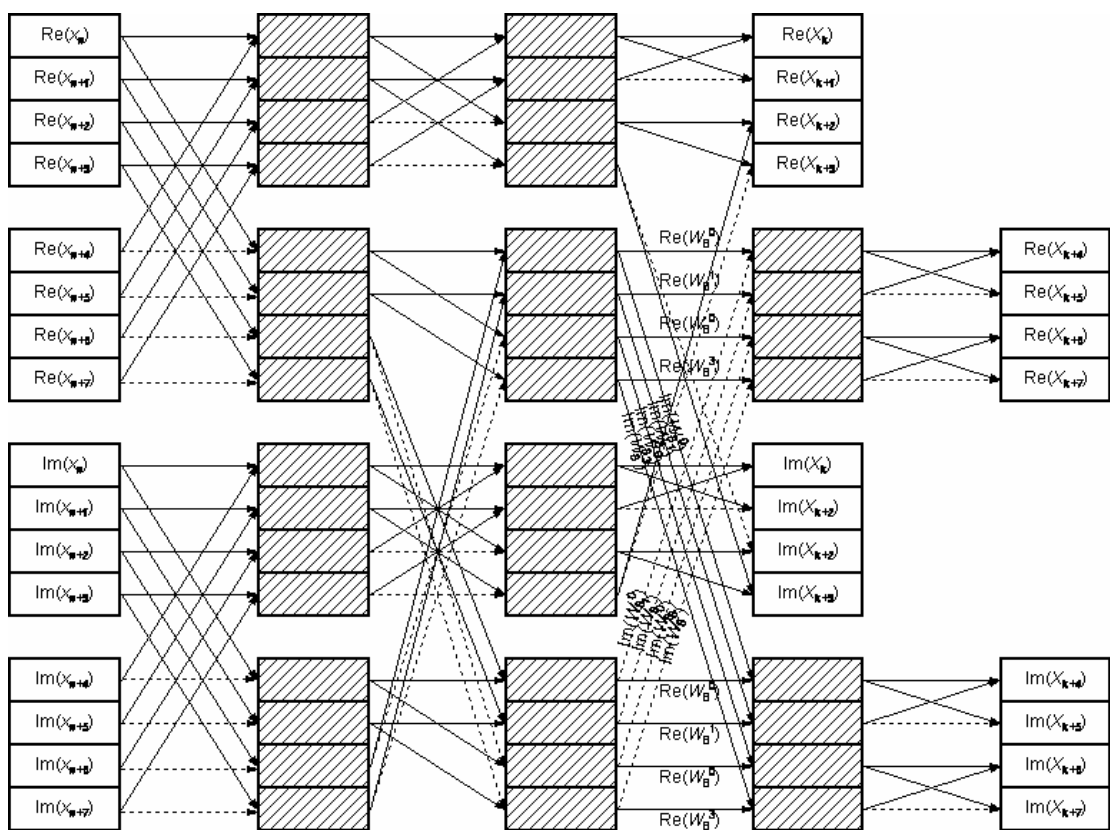


图 16 4 点 Split-Radix FFT 实、虚部数据流向图

SIMD 加法	SIMD 乘法	寄存器内重排	内存读取	内存写入
14	4	12	8	4

表 9 8 点 FFT 内核指令数统计

4.4.3 16 点 FFT 内核的实现

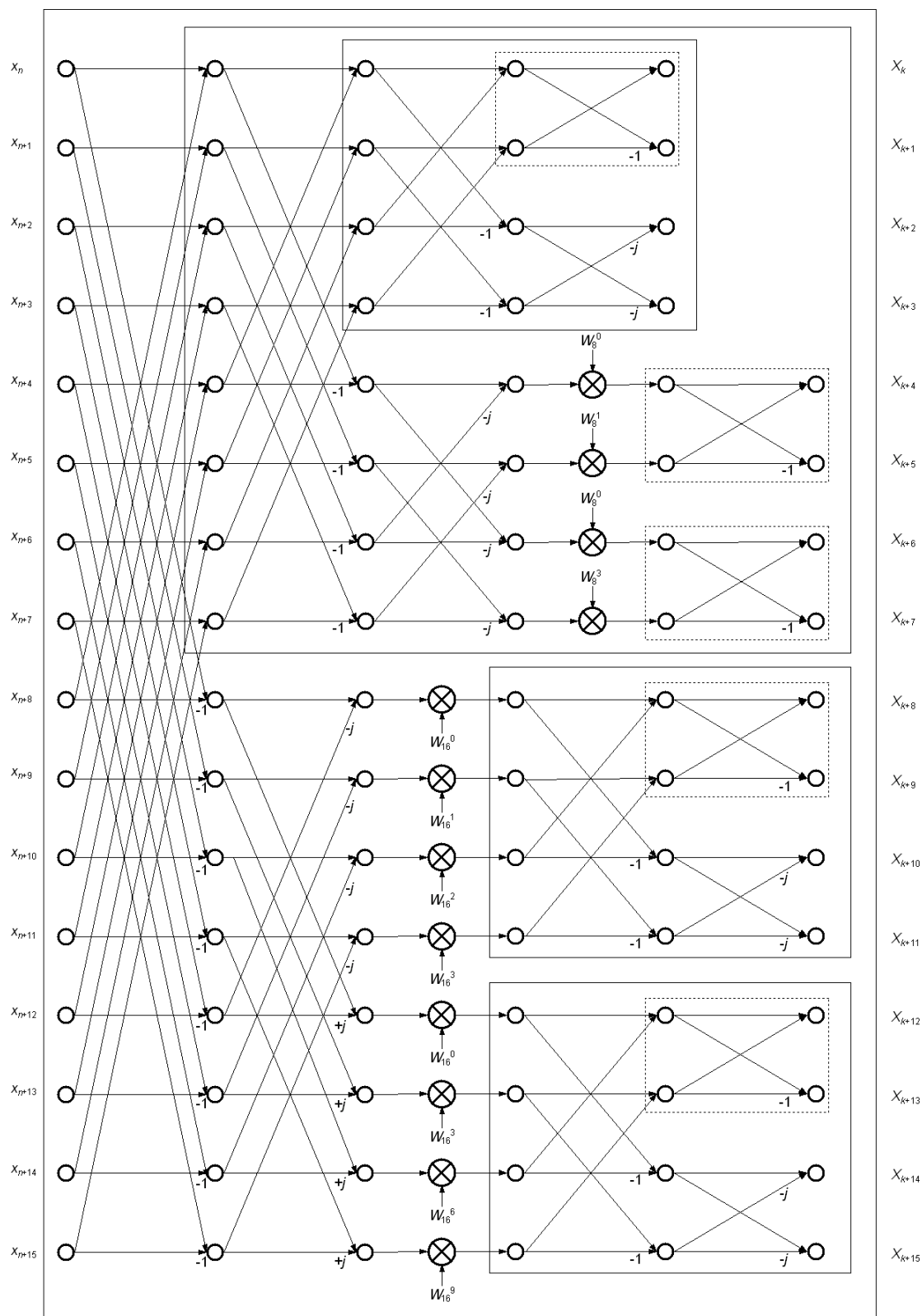


图 17 16 点 Split-Radix FFT 蝶形运算图

SIMD 加法	SIMD 乘法	寄存器内重排	内存读取	内存写入
38	12	24	20	12

表 10 16 点 FFT 内核指令数统计

4.4.4 IFFT 内核的优化实现

对于 IFFT 同样有 4、8 和 16 点的小代码内核函数。从式(25)可以看出, IFFT 实际只需将图 10、图 15 和图 17 蝶形运算图中的 j 改为 $-j$, $-j$ 改为 j , 并把所有的因子 W_N 替换成它的共轭 W_N^* 即可得到。因此我们只需要在前述的 4 点、8 点和 16 点 FFT 内核的基础上作少许修改, 就可以得到 IFFT 内核函数。但要注意输出结果还要乘上 $1/N$ 系数, 见(25)式。

4.5 通用内核函数的实现

当 FFT 点数大于 16 时, 需要调用通用内核函数对序列进行分解并乘以适当的旋转因子。该函数使用一重循环对输入序列进行处理, 每次循环可计算 16 个点, 对 N 点的输入序列共需要 $N/16$ 次循环。

假设通用内核函数输入序列的点数为 N' , 它是点数为 N ($N > N'$) 的序列中从序号 n 起始的一部分, 则函数的首次循环分别从序号 n 、 $n + N'/4$ 、 $n + N'/2$ 和 $n + 3N'/4$ 处输入 4 点, 根据图 3 所示的蝶形进行运算, 并输出相应的结果。此后每次循环 n 递增 4, 继续对其余点进行运算, 直到循环了 $N'/16$ 次。

通用内核函数需要使用较多的通用寄存器, 用于循环计数和数据指针。从上面的描述可以看出, 由于函数将输入序列分为了四个部分, 每部分均要使用一个寄存器存储其起始地址; 由于输入序列分为实部和虚部, 这就需要两个寄存器分别保存实部数组和虚部数组的起始地址; 旋转因子采用查找表保存, 需要一个寄存器保存此查找表的地址; 此外还需要一个寄存器保存循环计数值; 这样就至少需要 8 个通用寄存器。而一般而言, 在嵌入式汇编语言里仅有 EAX、EBX、ECX、EDX、ESI 和 EDI 六个通用寄存器可供使用, 这就给编程带来极大不便。

一种解决办法是引入临时变量, 代替寄存器存储一些变量值, 但这违反了我们

的优化原则。另一种办法是复用某些通用寄存器，在程序需要将寄存器挪作它用时，使用 **PUSH** 指令暂存寄存器的上一个值，完成任务后再用 **POP** 指令将寄存器恢复原值，继续进行其它处理，但这种方法要使用堆栈，同样增加了内存存取次数，违反了我们的优化原则，是不可取的。

由此可见，上述两种常规的做法都不是最佳的。能否将 **ESP** 和 **EBP** 这两个寄存器也加以利用呢？这样就能够刚好凑足 8 个寄存器，满足我们的需要。但如 4.5 节所提，这两个寄存器通常被 C/C++ 程序用于堆栈指针和框架指针，是不能随意更改的。而且，程序也不能通过常规的做法同时保存这两个寄存器：若我们使用临时变量保存它们，**ESP** 可被正常保存并移作它用，**EBP** 则不行，这是因为 C/C++ 中的使用 **EBP** 保存函数变量地址，**EBP** 即使被保存在临时变量中，该临时变量的地址也会因 **EBP** 随后的改变而丢失，导致 **ESP** 和 **EBP** 均无法恢复；相反，若我们采用 **PUSH** 指令将这两个寄存器保存在堆栈中，由于堆栈指针 **ESP** 被修改，同样会导致两个寄存器均无法恢复。

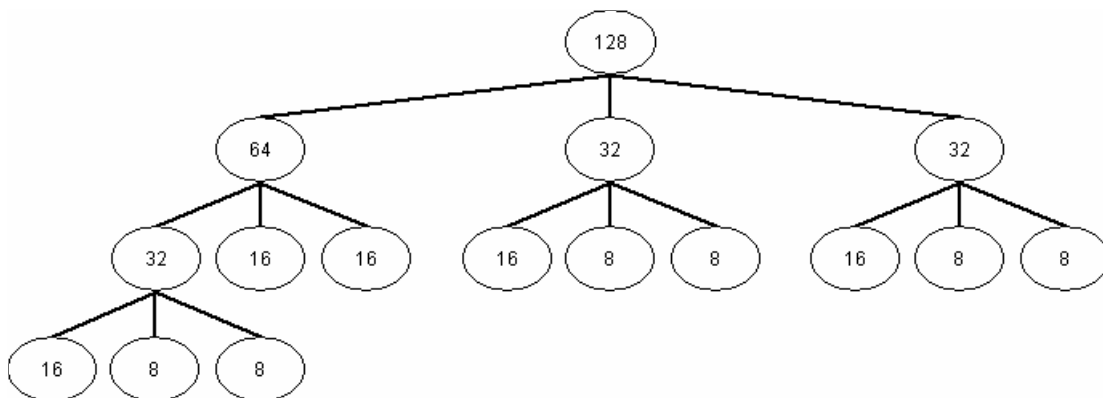
这时就要构思一种非常规的做法。考虑到 Pentium III/4 还有 7 个 MMX 寄存器可供利用，我们可以在函数的起始将 **ESP**、**EBP** 的值暂存入 MMX 寄存器，在函数的末尾再将它们恢复，这可以方便地使用 **MOVD** 指令实现。这样一来，**ESP** 和 **EBP** 寄存器就可以自由地被程序更改和使用了。但有两点需要注意：

1. 在 **EBP** 寄存器被修改后，程序就不再能够引用任何变量，包括函数的入口参数变量和函数中声明的临时变量，因此，对 **EBP** 的修改必须在函数变量使用完毕后。
2. 由于程序使用了 MMX 寄存器，在程序末端必须调用 **EMMS** 指令，恢复 FPU 状态（参见 3.4.3 节）；但由于 **EMMS** 是一个非常耗时的指令，并且在 SFFT 的其余地方均没有使用 FPU 的正常浮点运算指令，我们可以将对 **EMMS** 的调用留待完成了一次完整的 FFT/IFFT 变换后，通常放在 SFFT 的“固定点数接口函数”中（参阅 4.6.2 节）。

4.6 接口函数的优化实现

外部程序通过调用 SFFT 接口函数进行 FFT/IFFT 运算。接口函数分为两类：固定点数 FFT/IFFT 接口函数和通用接口函数。前者的职责是调用内核函数（包括小代码内核函数和通用内核函数）完成指定点数的 FFT 或 IFFT 运算，4、8、16、32、64、128、256、512、1024、2048 和 4096 点 FFT 和 IFFT 分别有各自的接

固定点数接口函数对内核函数的调用可以用“分解树 (Breakdown Tree)”形象地解释。以 128 点的 FFT 运算为例 (如图 18), 128 点 FFT 接口函数首先调用通用内核函数将输入序列分解为 64 点、32 点、32 点三部分, 并乘以适当的旋转因子; 64 点部分可继续分解为 32 点、16 点、16 点三部分, 16 点可以调用小代码内核函数进行运算, 无需再分, 而 32 点部分则可继续细分为 16、8、8 三部分, 再调用小代码内核函数分别计算。



对于其余部分的运算可依此类推：若程序发现分解树的某个节点无法用小代码内核解决，则调用通用内核将其细分，直到树的每个节点都可以由小代码内核计算为止（点数为 16、8 或 4）。这一过程可以方便的由递归算法实现，而由于各种点数的分解树形式是固定的，且分解的结果无非是为了调用各种内核函数解决问题，我们可编程事先执行这一步骤，并生成相应的直列代码，避免在实际运算中重现这一耗时的过程。下面由 SFFT 软件包中摘取了一段 128 点 FFT 接口函数的代码，如下所示¹⁸：

¹⁸ 为方便排版,某些函数名和变量名作了删改,实际的代码请参阅 SFFT 源程序。

```

06 core_o3_fw(p_re, p_im, 24);
07 core_o4_fw(p_re, p_im, 32);
08 core_o4_fw(p_re, p_im, 48);
09 core_loop_fw(table_wn_o3, p_re, p_im, 16, 24, 8, 64);
10 core_o4_fw(p_re, p_im, 64);
11 core_o3_fw(p_re, p_im, 80);
12 core_o3_fw(p_re, p_im, 88);
13 core_loop_fw(table_wn_o3, p_re, p_im, 16, 24, 8, 96);
14 core_o4_fw(p_re, p_im, 96);
15 core_o3_fw(p_re, p_im, 112);
16 core_o3_fw(p_re, p_im, 120);
17
18 swap_o7(p_re);
19 swap_o7(p_im);
20
21 __asm emms;

```

上述代码中，`core_loop_fw` 为 FFT 通用内核函数，“fw”是英文“forward”的缩写，代表正变换。`core_o4_fw` 和 `core_o3_fw` 分别为 16 点和 8 点的小代码内核函数，其中“o4”和“o3”分别代表“order 4”和“order 3”，即 2 的 4 次方和 3 次方。`p_re` 和 `p_im` 分别是输入实部数组和虚部数组。`table_wn_o?` 是旋转因子表的地址。

可以清楚的看到，上述代码的第 1 至 8 行完成了图 18 所示分解树的左分支，第 9 至 12 行完成的是中间的分支，而 13 至 16 行完成了右分支的计算。第 18 和 19 行是完成位逆序交换的函数，由于我们采用的 Split-Radix 算法输出是位逆序的，需要调用这两个函数重排输出元素。第 21 行使用 `EMMS` 指令恢复 FPU 状态，这是因为通用内核函数和位逆序函数中使用了 `MMX` 指令和寄存器。

4.6.2 通用接口函数的实现

通用接口根据输入参数（是 FFT 还是 IFFT，FFT/IFFT 的点数等），通过函数表的方法调用相应的固定点数接口函数。

在解释函数表法之前，我们需要对 C/C++ 语言的函数调用方式有一些了解。C/C++ 语言中可使用关键字设定多种不同的函数调用约定，最常用的有三种：

- `__cdecl`
- `__stdcall`

- `__fastcall`

`__cdecl` 是 C/C++ 语言中默认的函数调用约定，在不使用任何关键字修饰函数声明时，C/C++ 语言采用这种方式调用函数。这种调用方式使用堆栈传递参数，调用方在调用某一函数时，将函数参数按从右向左的顺序依次 `PUSH` 入堆栈，在函数返回后，调用方需要使用 `POP` 指令（或通过调整 `ESP` 寄存器的值）清除 `PUSH` 入的参数。由于清除堆栈的职责在于调用方，这种约定使调用不定参数的函数¹⁹成为可能。

`__stdcall` 是 Windows API 惯用的函数调用约定，它与 `__cdecl` 约定类似，是一种使用堆栈传递参数的调用方式，主要的区别是在使用这种约定时，被调用的函数主动将参数 `POP` 出堆栈。

`__fastcall` 方式通过 `ECX` 和 `EDX` 寄存器分别传递第一个和第二个参数，更多的参数则按从右向左的顺序使用堆栈传送。由于这种调用约定优先使用寄存器传递参数，在恰当使用的前提下是一种较为快速的函数调用方式。

下面举一个简单的例子说明上述三种函数调用约定。在例子中，我们使用 `__cdecl`、`__stdcall` 和 `__fastcall` 三种调用约定分别编写三个函数：`func1`、`func2` 和 `func3`，如下所示：

```
int func1(int a, int b, int c) // __cdecl is omitted because it
{                               // is default to the compiler.
    return (a + b + c);
}

int __stdcall func2(int a, int b, int c)
{
    return (a + b + c);
}

int __fastcall func3(int a, int b, int c)
{
    return (a + b + c);
}
```

¹⁹ 不定参数函数如最常使用的 C 函数 `printf` 等；在声明不定参数时，使用“...”代表不确定的参数列表。详情可参阅 MSDN。

我们使用以下的代码调用上述三个函数，并给出了编译器编译出的相应的汇编程序，可以清楚的看出三种调用方式的不同。

```
// c code that calls the three functions above -----  
func1(1, 2, 3); // __cdecl convention  
func2(1, 2, 3); // __stdcall convention  
func3(1, 2, 3); // __fastcall convention
```

```
// corresponding assembly code -----  
push    3          ;  
push    2          ; push parameters right to left  
push    1          ;  
call    func1      ; call func1 using __cdecl  
add     esp, 12     ; adjust stack  
  
push    3          ;  
push    2          ; push parameters right to left  
push    1          ;  
call    func2      ; call func2 using __stdcall  
                    ; there is no need to adjust stack here  
                    ; because the stack is already popped  
                    ; by the callee  
  
mov     edx, 2      ; pass the second parameter through edx  
mov     ecx, 1      ; pass the first parameter through ecx  
push    3          ; pass the third parameter by stack  
call    func3      ; call func3 using __fastcall
```

除了函数调用约定外，另一个需要了解的事实是计算机实际是根据函数的入口地址（即函数第一条指令的内存地址）来调用函数的，函数表的工作原理就在于此。我们事先将每个固定点数 FFT/IFFT 接口函数的入口地址存入一个数组，就可以将输入的参数直接翻译成函数表的索引，再使用 **CALL** 指令调用索引处的地址值。例如，假设 128 点 FFT 入口地址位于函数表 **FUNC_TABLE** 的索引 14 处，通用接口函数的输入参数为 7（128 是 2 的 7 次方），则可以使用诸如 **CALL FUNC_TABLE[7 * 2]** 的方式调用 128 点 FFT 函数。

4.7 位逆序函数的优化实现

位逆序是多数快速傅里叶变换算法的必需步骤，Split-Radix FFT 也不例外。我

们使用的 Split-Radix 算法具有输入序列正序，输出序列逆序的特征。

以 8 点 FFT/IFFT 为例，输入序列序号依次为 0, 1, 2, 3, 4, 5, 6, 7，输出序列的序号由于位逆序，变为 0, 4, 2, 6, 1, 5, 3, 7。为了将输出序列正序，程序实际需要做的的工作是将以下两组样点的值交换，它们是(1, 4)和(3, 6)。

交换样点的值看起来是一个十分简单的过程，但若处理不当，则会使程序的效率大打折扣。有以下的方法完成样点交换的工作：

1. 使用 MOV 指令，以通用寄存器（EAX、EBX、ECX、EDX 等）为中转媒介，交换样点的值；
2. 使用 XCHG 指令，以通用寄存器（EAX、EBX、ECX、EDX 等）为中转媒介，交换样点的值；
3. 使用 MOVD 指令，以 MMX 寄存器（MM0, MM1, MM2, ..., MM7）为中转媒介，交换样点的值；
4. 使用 MOVD 指令，以 XMM 寄存器（XMM0, XMM1, XMM2, ..., XMM7）为媒介，交换样点的值。

经实验，发现方法 3 是速度最快的，而方法 2 速度最慢。

此外，交换样点时的内存存取顺序也是有讲究的，以下几点规则我们应当注意：

1. 对样点的读写尽量按照内存地址顺序，由低地址向高地址推进。例如要交换 (1, 4)和(3, 6)两组样点，应该按照以下的顺序读写内存：

```
LOAD  r1, x(1)
LOAD  r2, x(3)
LOAD  r3, x(4)
LOAD  r4, x(6)
SAVE  x(1), r3
SAVE  x(3), r4
SAVE  x(4), r1
SAVE  x(6), r2
```

其中，r1、r2、r3、r4 是中转寄存器，LOAD 和 SAVE 分别是内存读写指令，在我们所采用的方法（法 3）中，它们相应于 MMX 寄存器和 MOVD 指令。

2. 对同一内存地址的读和写指令尽量分开，如上面的例子中，在完成一系列读操作之后，再进行写操作。

上述规则的目的在于尽量减少微处理器高速缓存的刷新，从而加快读写速度。

4.8 软件包的封装与调用

在完成 SFFT 各函数的优化实现后，最后需要进行的工作是对 SFFT 软件包的封装，以便外部程序调用。根据常规的做法，我们可将该软件包封装成 Windows 动态链接库（DLL）或 C 语言静态链接库（*.lib）的形式。这样，外部程序只要在程序中包含 SFFT 软件包的头文件（sfft.h），并将 SFFT 动态链接库的 C 语言输入库文件（sfttdll.lib）或 SFFT 静态链接库（sfftlib.lib）加入到程序的链接选项中，就可以使用 SFFT 动态链接版或静态链接版提供的功能。使用动态链接版时，编译出的应用程序需要配合 SFFT 动态链接库（sfttdll.dll）方可使用；使用静态连接版不需要额外的 DLL 支持，但编译出的可执行文件会大些。

4.9 小结

这一章首先讨论了在 Intel Pentium III/4 实现 FFT 的算法选择和程序设计思想，我们陈述了在此平台上进行程序优化的一般原则和技巧。接着，我们利用 Intel Pentium III/4 提供的 SSE（Streaming SIMD Extensions）指令集，开始了 FFT 谱分析器的实际设计和优化实现。优化过程中最重要的问题是发现程序中的并行性，从而将尽可能多的运算向量化（Vectorize），使用 SIMD 指令集实现。我们将最终编写成功的 FFT 谱分析器封装成了一个完整的软件包“SFFT”，以方便外部程序的调用。

第5章 SFFT 软件包性能测试与评价

在这一章中，我们将对第4章完成的SFFT软件包进行综合测试与评价。测评分为三个方面：精度测评、速度测评和应用测评。在应用测评中，我们将有幸目睹由SFFT计算出的模拟血流谱的显示结果。

5.1 精度测评

精度测评的主要目的是测量SFFT计算结果的精确性，从而评价该软件包的实用性。我们采用如下的测量方法：

1. 首先生成一组随机数据。这样做的目的是为了使计算出的频谱类似于白噪声的频谱，各点均可得到非零值；
2. 然后，使用SFFT软件包中的函数对上述随机数据进行FFT和IFFT，得到的结果和原数据比较，可以得到各点的误差值 $\Delta x(n) = |x'(n) - x(n)|$ ；
3. 根据各点的误差值，由下面的公式求出平均误差（M.E.），用于衡量SFFT软件包的计算精度。

$$M.E. = \sum_{n=0}^{n < N'} \frac{\Delta x(n) \times 2}{|x'(n)| + |x(n)| + \varepsilon} / N' \quad (60)$$

式(60)中的 ε 是一个很小的正数，以防止分母为0。 N' 是被测点数据总数，每被测点提供实部和虚部两个数据，故对于 N 点的FFT而言， $N' = 2N$ 。

根据这种方法我们得到了SFFT的精度测评结果。为了增加准确性，我们对各种点数的FFT分别取6组不同的随机数进行计算，得出各自的平均误差，再根据参与测量的所有数据，得出了SFFT结果的总平均误差，如下表所示：

FFT/IFFT 点数	平均误差 (M.E.)
4	8.04×10^{-8}
8	2.76×10^{-7}
16	2.04×10^{-7}

32	2.95×10^{-7}
64	4.86×10^{-7}
128	7.11×10^{-7}
256	4.19×10^{-7}
512	1.01×10^{-6}
1024	5.14×10^{-7}
2048	6.71×10^{-7}
4096	2.18×10^{-8}
总平均误差	3.36×10^{-7}

表 11 SFFT 精度测评结果

5.2 速度测评

通过与几种国际上主流且性能较高的 FFT 运算库进行计算速度的横向对比，我们可以看出 SFFT 的优势。

振奋人心的是，对于短时快速傅里叶变换，包括长度 16、32、64、128 和 256 点的场合，SFFT 的表现是几乎总所有 FFT 运算库中最好的，其速度达到甚至超过了 Intel 公司专为 Pentium III/4 CPU 设计的数值计算库 Math Kernel Library 和信号处理函数库 Signal Processing Library。另一方面，在所有开放源代码的 FFT 运算库中，SFFT 无疑是最快的，其速度大大超过了 FFTW，同时也超过了卡内基·梅隆大学 SPIRAL 计划中所编写的 FFT 软件包^{[6][7]}。

5.2.1 测评方法

测评一段程序的速度通常使用测量该程序运行时间的方法，一种最通俗且惯用的测量流程是：

1. 从特定的时钟源读取当前时间，保存在变量 t_0 中；
2. 运行被测程序；为达到一定精度，常需循环运行 M ($M \gg 1$) 次；
3. 从特定的时钟源读取当前时间 t_1 ，与 t_0 相减就得到了程序运行 M 次的时间 T ($T = t_1 - t_0$)。

5.2.1.1 时钟源的选择

选取恰当的时钟源是测评准确性的关键，在 Windows 操作系统中主要提供了三个 API 函数获取时间^[22]，它们是：

- 调用 `GetTickCount()`
- 调用 `timeGetTime()`
- 调用 `QueryPerformanceCounter()`

`GetTickCount()`和 `timeGetTime()`函数返回自 Windows 系统启动始所经历的时间，以毫秒为单位，但它们的实际精度往往大于 1 毫秒，其中 `GetTickCount()`最为粗略，精度只能达到 5~55ms 左右，`timeGetTime()`函数精度稍高，在 Windows NT/2000 系统中，它的默认精度约为 5ms，可作为多媒体应用程序的定时器来源。

Windows 中精度最高的时间测量函数是 `QueryPerformanceCounter()`，它返回硬件系统高精度时钟计数器（High-resolution Performance Counter）的当前值，这个计数器并不是在所有的硬件平台上都得到支持的。在 Intel Pentium III/4 平台上，此函数的最高精度约为 2.79e-007 秒，合 279 纳秒。

除了可以使用以上的 Windows API 获取时间信息外，我们还可以编制程序直接由系统硬件获取时间值，这一机制是通过读取 Intel Pentium III/4 系列 CPU 的 Time-Stamp 计数器（TSC）的值实现的。TSC 的当前值存在一个 64 位寄存器中，它记录计算机启动时起所经过的时钟周期数，并以 CPU 的时钟频率刷新（即每时钟周期增加 1），因此，TSC 是 Intel 架构计算机中最精细的时钟源，也是我们这次测试所最终选择的时钟源。

时钟源	最高精度
<code>GetTickCount()</code>	5 ~ 55 ms
<code>timeGetTime()</code>	1 ~ 5 ms
<code>QueryPerformanceCounter()</code>	< 1 μ s
Time-Stamp Counter	CPU 时钟周期

表 12 Windows 系统中可用时钟源精度对比

Pentium Time-Stamp 计数器可使用 `rdtsc` 指令读取，它将 TSC 寄存器的低 32

位存入 EAX 寄存器，高 32 位存入 EDX 寄存器。一个使用 C 语言和嵌入式汇编读取 TSC 寄存器的示例函数如下：

```
__declspec(naked) __int64 read_tsc(void)
{
    __asm rdtsc;
    __asm ret; /* return value saved in EDX:EAX */
}
```

5.2.1.2 提高测量精度的策略

为了进一步提高测量精度，除了采用精确的时间源和循环运行被测程序外，还有若干需要注意的问题。我们知道 Windows 是一种多任务的操作系统，在同一时刻可有多个进程在系统中运行，系统调度程序根据进程优先级为这些进程分配时间片²⁰，从而达到多任务同时运行的效果。

因为这个原因，我们的测试程序在运行时常会受到系统中同时运行的其它进程的“干扰”。并且，各进程所拥有的时间片不是均匀的，而是有赖于系统调度程序的调整和控制，这就会为测量数据引入一定的随机噪声。为了降低这个问题的影响，测试程序将自身的进程优先级和线程优先级设为最高²¹，这样在测试过程中就可以达到独占 CPU 的效果，将外来干扰降到最低值。

5.2.2 测评结果

在一台 Pentium 4 1.4G CPU / 256MB RDRAM、装有 Windows 2000 操作系统的计算机上，我们对各 FFT 软件包的速度比较结果如图 19 至图 22 所示。

参与评测的软件包包括：

- **FFTW** : FFTW 2.0 函数库 (<http://www.fftw.org>);
- **FFTD** : Spiral FFTD 函数库 (针对 Pentium II 汇编优化);

²⁰ 实际上操作系统是根据线程优先级为每一线程分配时间片。一个进程至少有一个线程，称为主线程，线程优先级由所属的进程优先级加上线程相对优先级决定。

²¹ Windows 系统中最高的进程优先级为“实时优先级 (REALTIME_PRIORITY_CLASS)”，由于此优先级超越了 Windows 中的所有系统进程，在测试程序运行中，包括鼠标、键盘等输入设备都会短时间停止响应。

- **IMKL** : Intel Math Kernel Library V5.2.1 (针对 Pentium 4 优化);
- **ISPL** : Intel Signal Processing Library 4.5
- **SFFT** : 我们编写的 FFT 软件包。

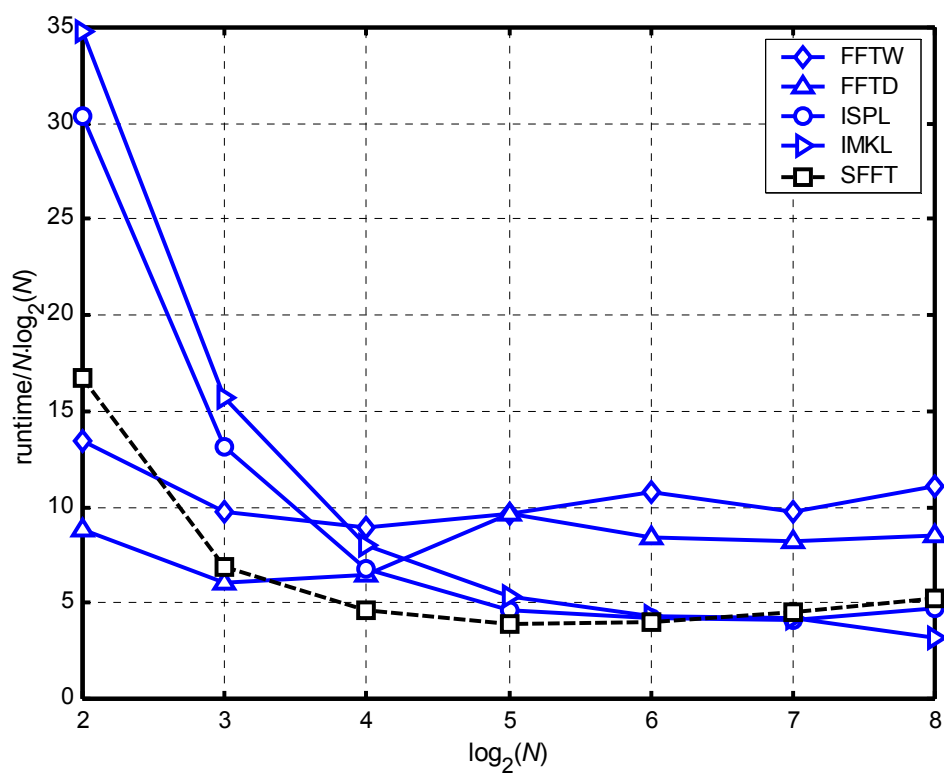


图 19 SFFT 静态链接库版速度测评 (MSC++编译)

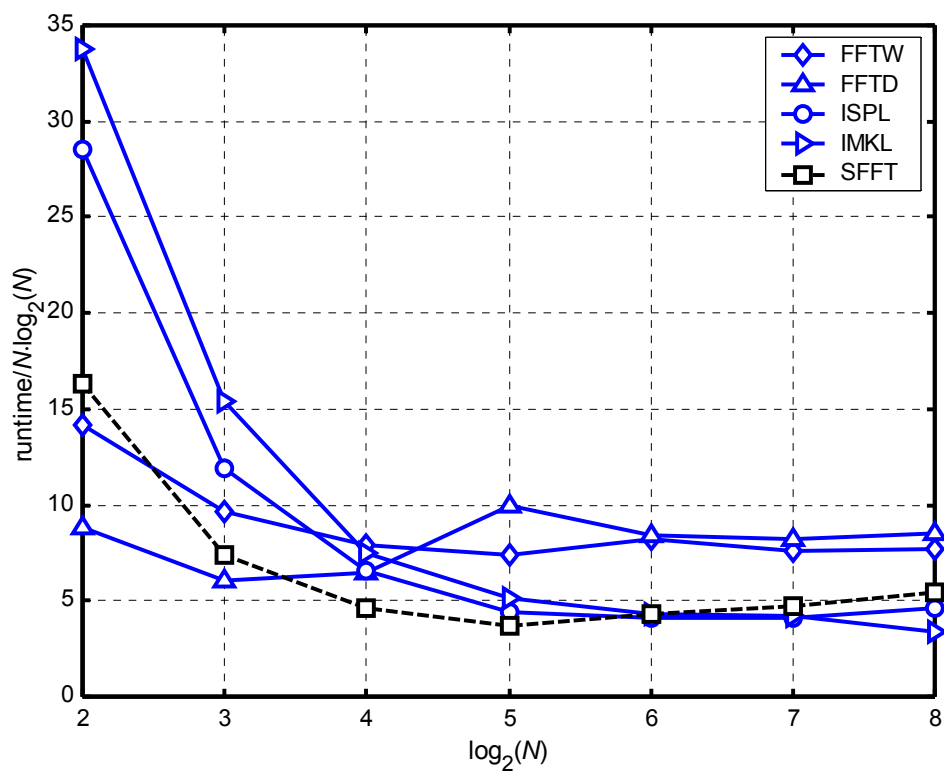


图 20 SFFT 静态链接库版速度测评 (Intel C++编译)

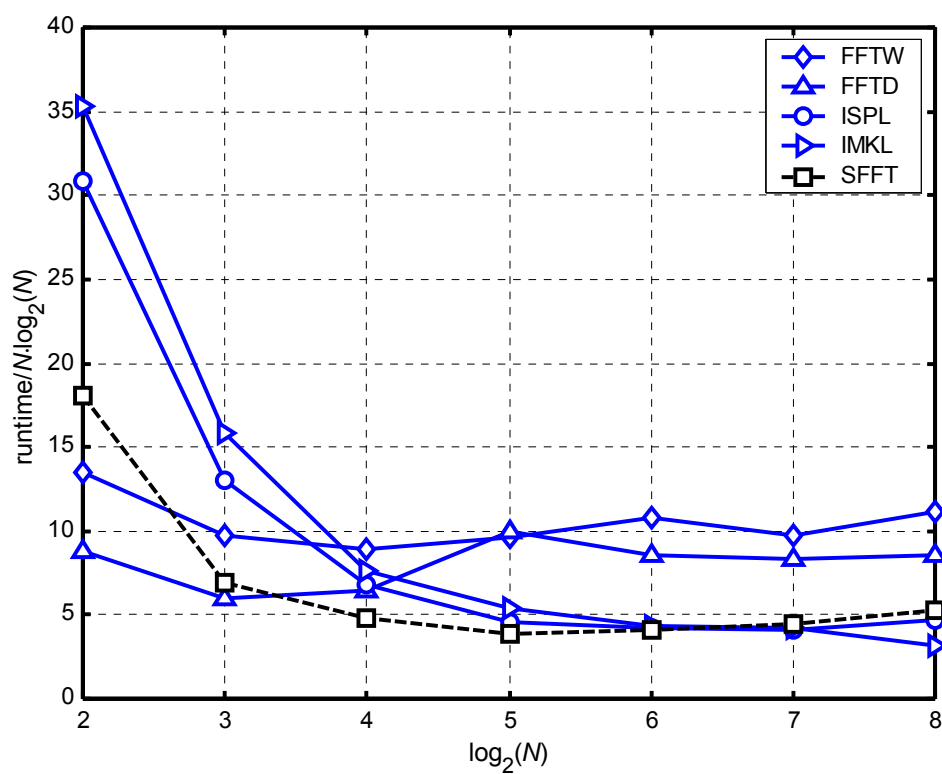


图 21 SFFT 动态链接库版速度测评 (MSC++编译)

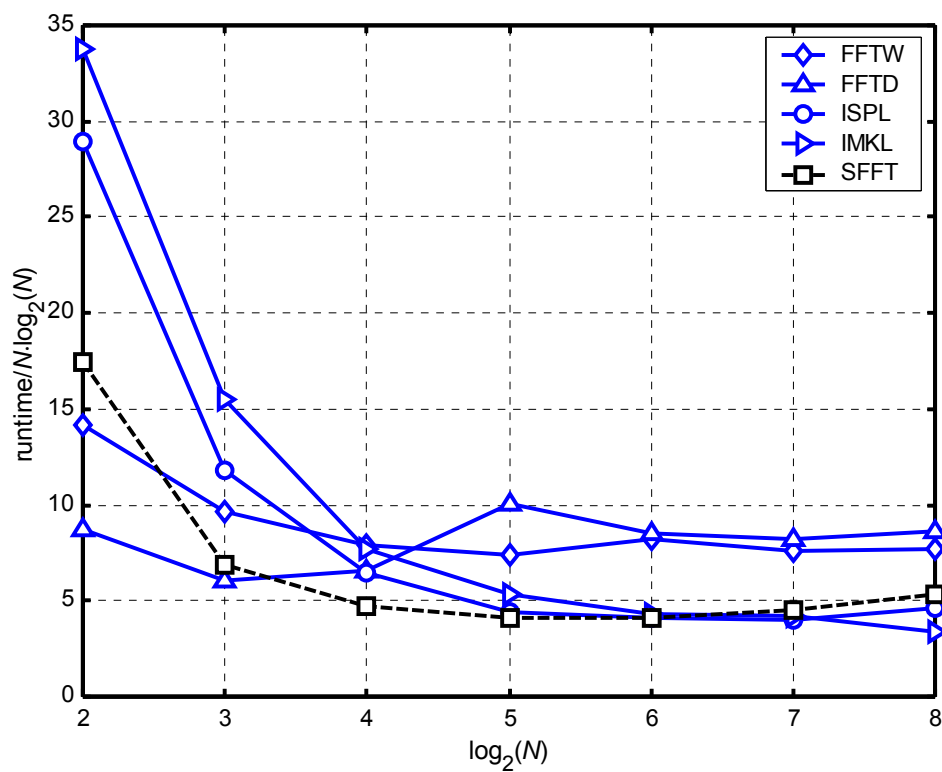


图 22 SFFT 动态链接库版速度测评 (Intel C++编译)

5.2.3 误差分析

由于时钟源的最大精度是确定的，对于执行较快的被测程序，测量精度将会较低，由此带来的误差我们称之为量化误差（Quantization Error）。如前所述，为降低量化误差，一般将被测程序循环执行 M ($M \gg 1$) 遍，将测得的总时间除以 M ，则得到被测程序的平均执行时间，采取此措施后量化误差将会很低。此外，由于循环引入了多余的指令，将耗费一些额外的执行时间，但这对进行速度比较没有影响。

除了量化误差，其余的测量误差将主要来源于两方面。一是操作系统的进程调度的影响；虽然我们已将测试程序的进程优先级和线程优先级都设为最高，但对于多任务的 Windows 操作系统，仍不能保证在测试期间仅有一个进程在系统中运行，这将对测量精度造成一些影响。另一方面是由于微处理器本身在执行各段被测程序时，一些执行环境如处理器高速缓存和流水线中的当前状态、内存地址的不同以及温度、湿度等物理条件的影响都可能对速度产生一定的影响，这是难以避免的。

5.3 应用测评

基于 SFFT，我编制了一个脉冲多普勒血流成像系统的软件部分，名为“Ultrasound Imaging System（超声成像系统）”，简称“UIS”。该软件底层与高速数据采集卡接口，从超声探头采集多普勒回波信号，加以数字化。数字化后的信号送入软件进行正交解调、滤波、谱分析和对数压缩等处理，最后调用 DirectDraw 接口进行频谱图像显示。UIS 软件的主界面如图 23 所示。

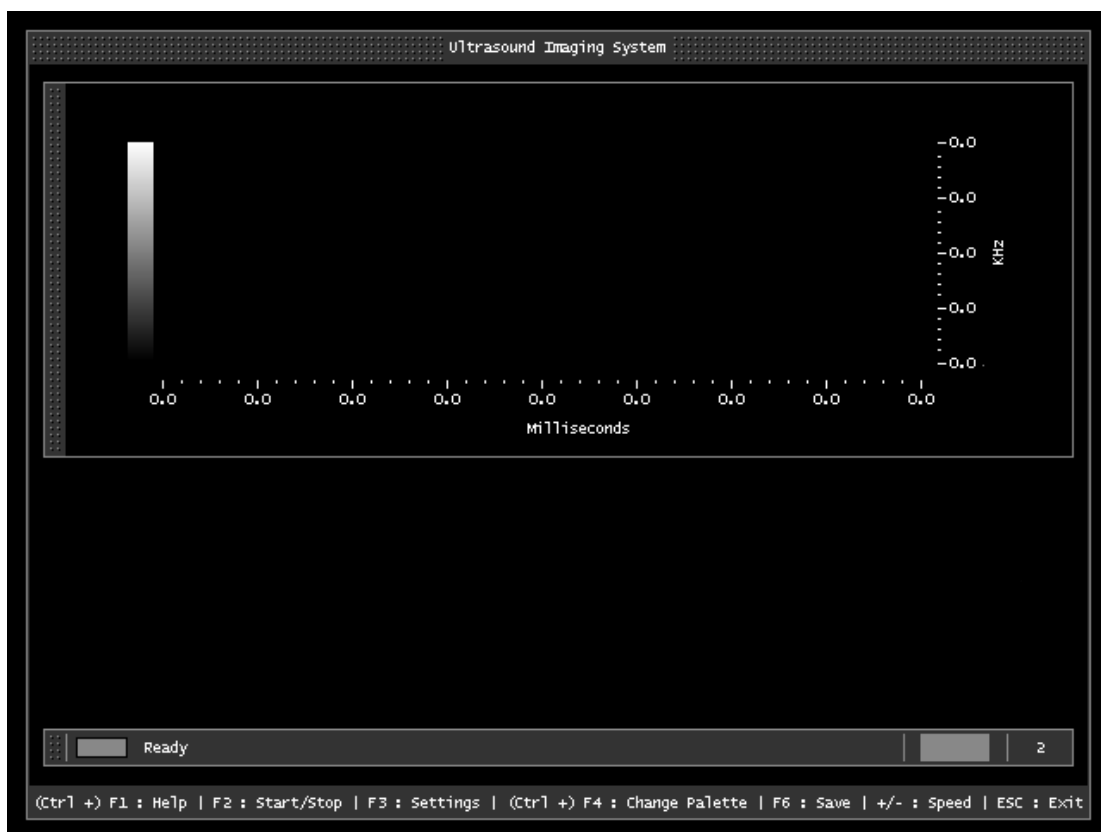


图 23 超声成像系统 (UIS) 主界面

下面将使用该软件实测 SFFT 谱分析的效果。由于该脉冲多普勒血流成像系统的底层硬件尚未完全实现，测试中将使用软件模拟一段多普勒信号。假设超声探头发射频率为 3MHz，多普勒回波信号实际可看作对该 3MHz 载波信号的调频，因此，我们编程预先生成了两个调频信号的数据文件，分别为一个周期的-500~+500Hz 的锯齿波和方波，它们都被调制在 3MHz 的载频上。信号使用 20MHz 的频率进行采样，故生成的数据文件分别包含 20M 个数据点。

UIS对输入数据进行处理的大致步骤如下,关于更详细的处理流程可以参阅[24]:

1. 分别从两个数据文件中循环读取调频波数据，每次读取 5000 个样点；
2. 从读取到的样点序列中某个固定位置起抽取 20 个连续的样点，进行正交解调，滤波，而后求和；
3. 求和结果被连续送入一个可容纳 128 样点的临时缓冲区，待该缓冲区填满后，数据被送往以 SFFT 为核心的谱分析器进行频谱计算；
4. 计算所得频谱经过对数压缩，送往显示模块进行灰度变换和显示。

2. 从读取到的样点序列中某个固定位置起抽取 20 个连续的样点，进行正交解调，滤波，而后求和；

3. 求和结果被连续送入一个可容纳 128 样点的临时缓冲区，待该缓冲区填满后，数据被送往以 SFFT 为核心的谱分析器进行频谱计算；

4. 计算所得频谱经过对数压缩, 送往显示模块进行灰度变换和显示。

在实测中，模拟血流谱显示快而流畅，画面清晰，显示了 SFFT 的威力。图 24 和图 25 是模拟血流谱的显示结果：

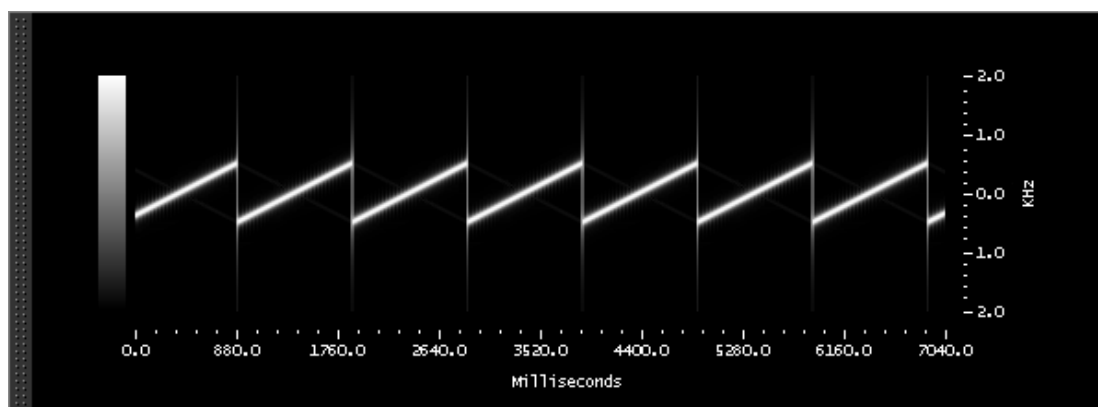


图 24 SFFT 应用评测结果—锯齿波频谱

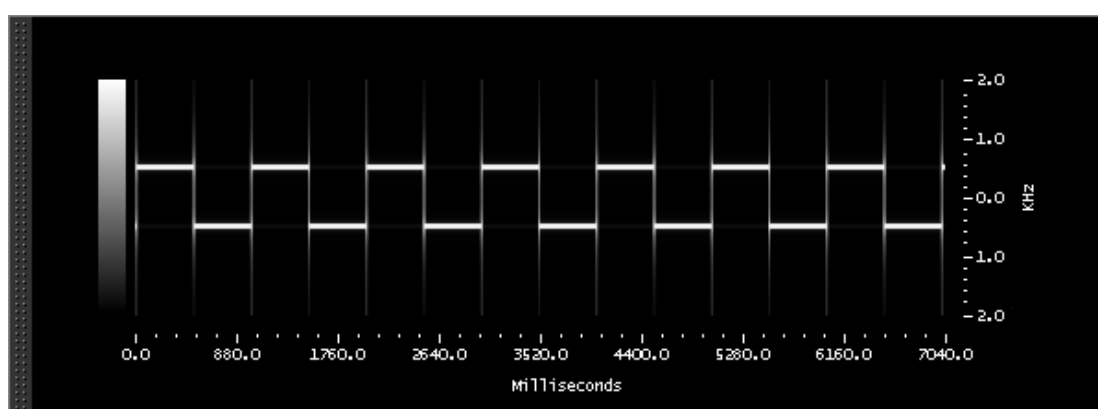


图 25 SFFT 应用评测结果—方波频谱

5.4 结论

从上面的测评结果可以看出，SFFT 是一个完全达到了实用要求的快速傅里叶变换软件包，并且由于其充分发挥了 Intel Pentium III/4 微处理器的内在潜力，在主流 PC 平台上有着其它同类软件无法比拟的速度优势，达到了脉冲多普勒血流成像系统快速谱计算的目的。同时，由于 SFFT 软件包的通用性，它也可以被方便的移植到其它应用场合。而编制此软件包过程所用的思想和优化技巧，也同样适用于今后的其它开发场合，为在主流 PC 平台上实现完整而实用的医学超声成像系统提供了经验和借鉴。



参考文献

- [1] 冯若,《超声手册》,南京,南京大学出版社,1999年;第七章,第732~824页。
- [2] 郑君里、杨为理、应启珩,《信号与系统》,高等教育出版社,1981年,第191~204页。
- [3] 刘立人,《并发程序设计》,上海,上海科学技术文献出版社,1987年;第1~8页。
- [4] 陈思平,“从北美放射学会看医学超声的发展”,深圳安科高技术股份有限公司,2000年。
- [5] Gavin Haentjens, “An Investigation of Cooley-Tukey Decompositions for the FFT”, Electrical and Computer Engineering Department, Carnegie Mellon University, May 18, 2000.
- [6] David Sepiashvili, “Performance Models and Search Methods for Optimal FFT Implementations”, Electrical and Computer Engineering Department, Carnegie Mellon University, May 1, 2000.
- [7] Franz Franchetti and Markus Püschel, “A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms”, Proc. International Parallel and Distributed Processing Symposium (IPDPS), 2002.
- [8] Matteo Frigo. “A fast Fourier transform compiler”. In Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, May 1999.
- [9] Matteo Frigo and Steven G. Johnson. “The fastest Fourier transform in the West. Technical report”, MIT Laboratory for Computer Science, September 1997.
- [10] Matteo Frigo and Steven G. Johnson. “FFTW: An Adaptive Software Architecture for the FFT”, In Proceedings of the IEEE International Conference On Acoustics, Speech and Signal Processing, volume 3, pp. 1381-1384, Seattle, WA, May 1998.
- [11] Bevan M. Bass, “A Approach to Low-Power, High-Performance, Fast Fourier Transform Processor Design”, Department of Electrical Engineering, Stanford University, February, 1999.
- [12] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, "Numerical Recipes in C", Cambridge University Press, 1997,

Chapter 12, pp. 496 - 536.

- [13]Aravind Ganapathiraju, Jonathan Hamaker, Joseph Picone, Anthony Skjellum, "Contemporary View of FFT Algorithms", Mississippi State University.
- [14]Dragan Mirković, Rishad Mahasoom, Lennart Johnsson, "An Adaptive Software Library for Fast Fourier Transforms", Department of Computer Science, University of Houston, 2000.
- [15]Martin Vetterli and Pierre Duhamel, "Split-Radix Algorithms for Length- p^m DFT's", IEEE Trans. On ASSP, Vol. 37, No. 1, January 1989.
- [16]Intel Corporation, AP-808, "Split-Radix Fast Fourier Transform Using Streaming SIMD Extensions", January, 1999.
- [17]Intel Corporation, AP-900, "Identifying Support for Streaming SIMD Extensions in the Processor and Operating System", January, 1999.
- [18]Intel Corporation, "IA-32 Intel® Architecture Software Developer's Manual", <http://developer.intel.com>, 2001.
- [19]Intel Corporation, "Intel® Pentium® 4 and Intel® Xeon™ Processor Optimization Reference Manual", <http://developer.intel.com>, 2001.
- [20]Intel Corporation, "Desktop Performance and Optimization for Intel® Pentium® 4 Processor", <http://developer.intel.com>, February, 2001.
- [21]Intel Corporation, "Three Vectors of Performance", May, 1997.
- [22]Microsoft Corporation, Microsoft Developer Network (MSDN) Library, <http://msdn.microsoft.com>, 2002.
- [23]Advanced Micro Devices, Inc., AMD Athlon™ Processor x86 Code Optimization Guide, 2002.
- [24]李康,《超声成像系统 (UIS) 开发者指南》, 南京大学声学研究所, 医学超声工程实验室, 2002 年。