南開大學
Nankai University

计算机学院

并行程序设计期末报告

基于频域插值的音频变调方法的并行加速算法

姓名：杜岱玮 陈静怡

学号：2011421 2012885

专业：计算机科学与技术

2022 年 3 月 30 日

目录

Abstract

本文作者杜岱玮（学号 2011421）、陈静怡（学号 2012885）打算组队完成并行程序设计课程的期末作业。我们的研究题目是语音变调技术的并行优化。语音变调是数字信号处理的重要应用，在互联网社交平台与大多数人的生活密不可分的当下，语音变调技术找到了越来越广阔的应用场景。语音变调已经得到了广泛的研究。目前已有的变调方法主要分为两大类：一类是时域插值拼接方法，例如 SOLA-FS 法；另一类是频域方法，常被称为相位声码器（phase-vo-coder）。我们计划主要研究基于频域插值的音频变调方法，该方法包含快速傅里叶变换、频域插值、时域卷积滤波等多个流程。我们将尝试运用 SIMD、多线程等多种不同的并行优化算法对其进行并行加速。

关键字：语音处理、并行优化、快速傅里叶变换、卷积滤波、频域插值

1 问题描述

我们的研究题目是基于相位声码器的语音变调。语音变调是数字信号处理的重要应用，特别是在互联网社交平台盛行的当下，基于语音变调的变声器技术被广泛应用在社交、直播软件中，不仅增加了在线语音聊天的趣味性，还起到保护隐私的效果。虚拟语音社交已经成为流行趋势，2020 年 4 月发布的语音社交平台 Clubhouse，截至 2021 年 2 月使用人数已经超过 200 万；与此同时，虚拟主播（vtuber）正在全世界的年轻群体中找到越来越广泛的受众。与文字不同，语音本身带有说话者的个人特征，各类分析技术可以从中轻易获取说话者的大量信息，引起了人们关于语音聊天泄露隐私的担忧。基于变调的变声技术可以在保留语音的交流功能的同时，掩盖其中的敏感信息，有效消除隐私泄露的隐患。由此，对于语音变调的研究非常具有现实意义。

2 研究背景

作为一项有广阔应用前景的技术，语音变调已经得到了广泛的研究。目前已有的变调方法主要分为两大类：一类是时域插值拼接方法，例如 SOLA-FS 法；另一类是频域方法，常被称为相位声码器（phase-vo-coder）。时域处理方法的优点是计算量小，而且变调结果自然度很好，但是由于拼接处理会带来相位不连续，产生噪音；频域方法由于要进行时频转换、估计相位和计算真实频率，需要的运算量较大，而且变调后语音不自然，有金属声。[7]

变调方法一般包括变调和时长规整两步，不同的变调方法的变调步骤一般是相同的，差异主要体现在时长规整步骤。具体来说，首先通过变采样率方法同时改变语调语速，然后以某种变速不变调方法改变音频时长，恢复原本的语速，最终只有语调被改变了。

所谓改变采样率，实际上就是我们自然理解中的改变音频播放速率：快进播放的音频变得尖锐，慢速播放的音频变得低沉，实际上就是改变了声调高低。

时长规整步骤有时域处理和频域处理两类方法。时域处理类方法直接在时域上对音频进行操作；而频域处理类方法先通过傅里叶变换等技术将时域信号转换到频域，然后在频域上进行处理，最后通过逆变换获得时域上的处理结果。

值得注意的是，变调和时长规整的顺序是可以改变的，可以先变调再时长规整，也可以先时长规整再进行变调。

举一个例子。假如我们希望将一段音频提高一个半音（频率乘以因子 1.0594），我们可以先将音频长度拉伸 1.0594 倍，保持音调不变，然后通过改变采样率的方法将其压缩 1.0594 倍，最终得到一段时长不变、频率提高 1.0594 倍的音频。[1]

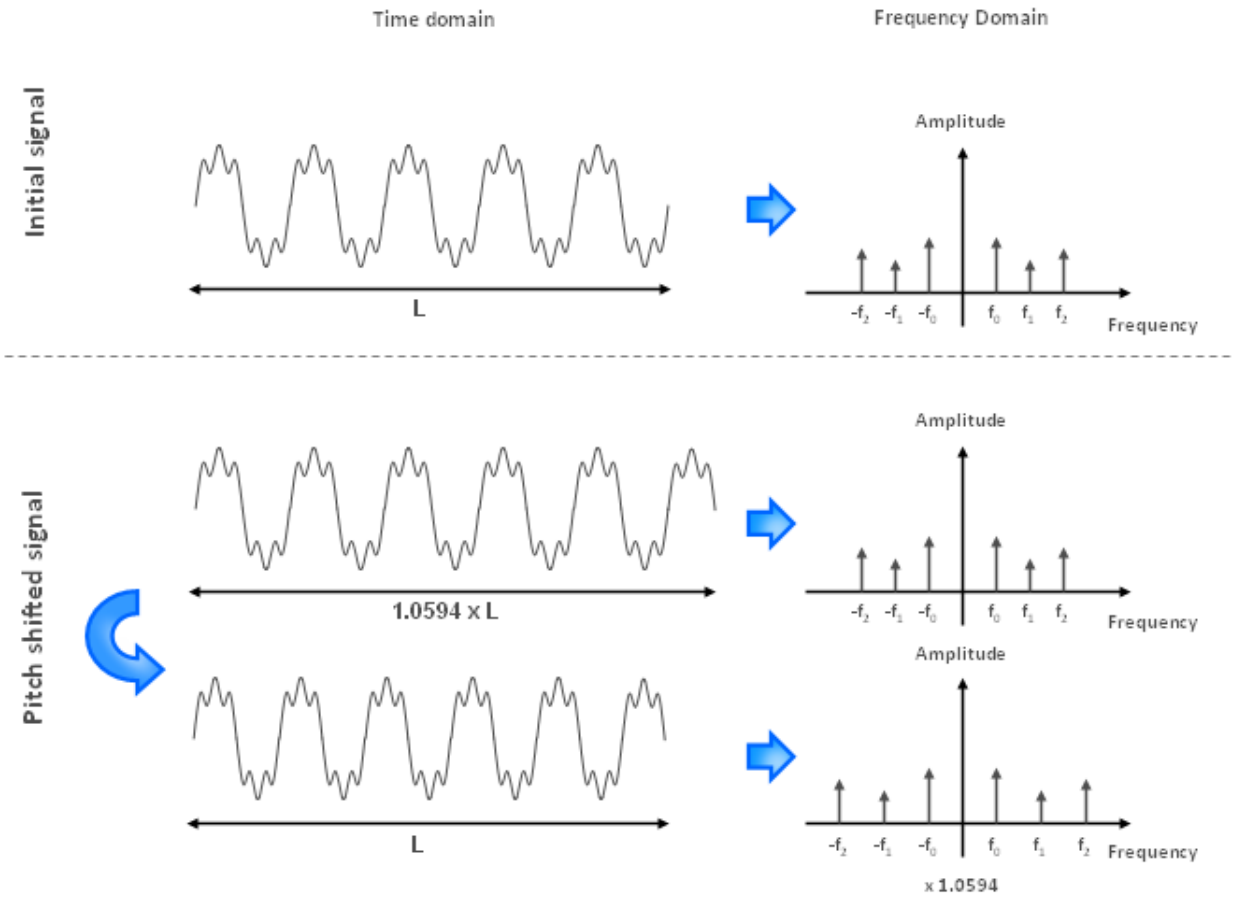

图 2.1: 将一段音频提高一个半音

3 算法流程

我们打算研究基于频域插值的变调方法的并行加速技术。该算法包含五个步骤：改变采样率、傅里叶变换、频域插值、傅里叶逆变换、卷积滤波去噪。我们已经用 python 对该算法进行了初步验证，可以确保该算法是可行的。

3.1 变速变调

上一节提到，改变采样率的效果是改变音频的播放速率从而改变其音调。为了具体地说明这一方法，我先解释一下音频文件的格式。音频文件由一个表示声波变化趋势的数列组成，数列中的数值越大，对应时刻的振动越强。每一秒的音频对应固定数量的数据点。为了加快或减慢语速，我们需要在保持数列的变化趋势不变的同时，用更少或更多的数据点表示它。

音频设备每秒播放固定数量的数据点，如果更少的数据点表示了更长的变化趋势，这一趋势就会在更少的时间里被播放出来，从而达到加速播放的效果；减速播放的原理也是类似的。在实践中，我们先对表示原始音频的数列进行插值，将离散的数据点变成时间上连续的函数，然后以不同的采样率对这个函数进行采样，从而用更少或更多的离散数据点表示它。

例如，为了将播放速度提升到两倍，我们要用一半的数据点表示原始音频。最简单的办法就是每隔一个数据点保留一个数据点，得到一个长度为原来一半的数列。

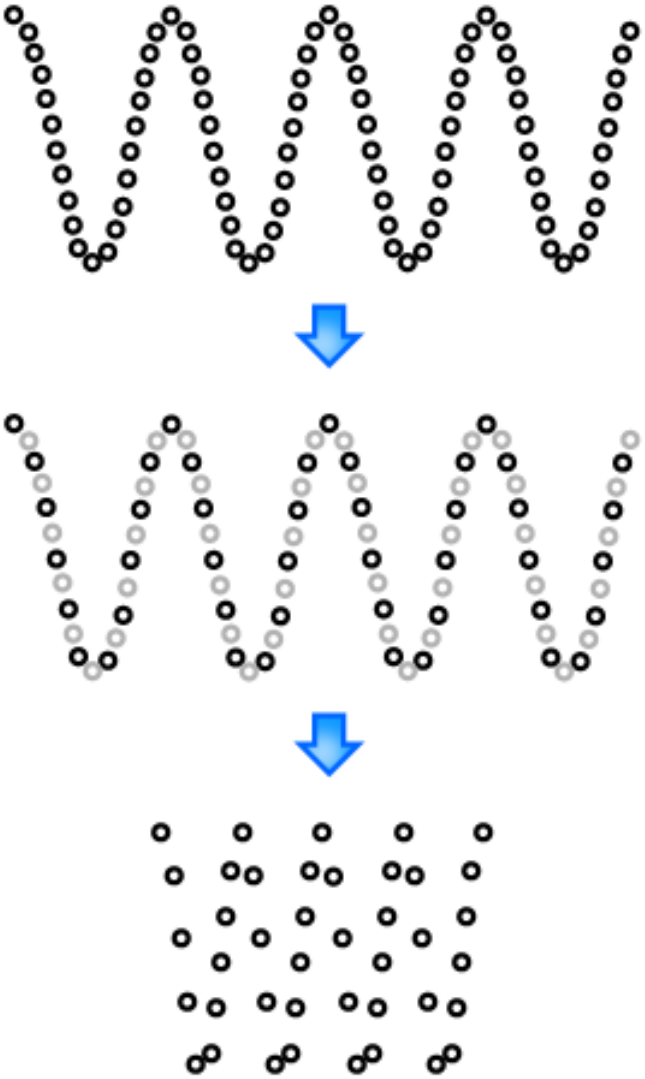

图 3.2: 以二倍采样率重新采样

3.2 时长规整

我们用频域插值的办法实现音频的变速不变调，使音频恢复原本的时长。

声波在较长时段上的变化是不规则的，但在极短的时间内，它可以被视为周期信号，如图3.3所示。我们把音频切分为多个极短的区间，然后均匀地对每个区间进行拉伸，最后再把所有区间拼接起来得到处理结果。小区间长度通常定为 20 毫秒左右。[2]

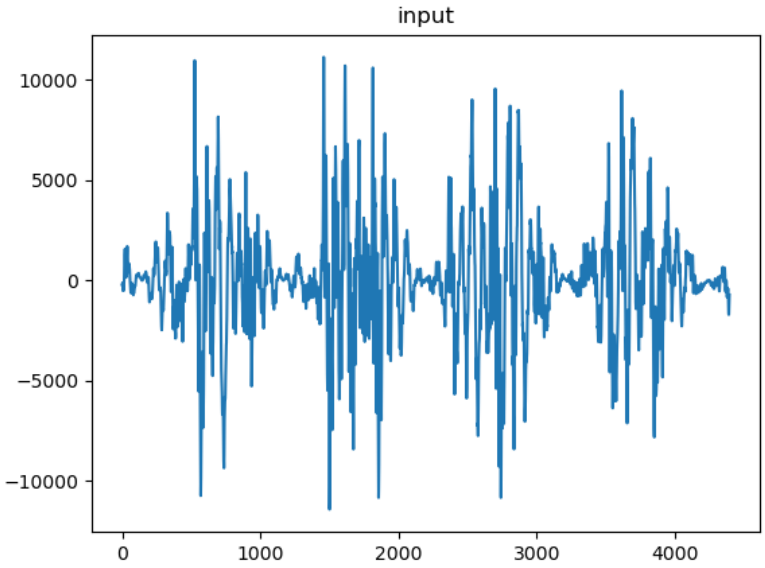

图 3.3: 极短时间内时域信号可以视为周期信号

对于每个区间，处理步骤分为傅里叶变换、频域插值、傅里叶逆变换三步。通过傅里叶变换我们得到信号的频谱。为了达到时域拉伸的目标，我们先把信号转到频域，在频域进行拉伸，然后恢复到时域。我们要做的是，在确保频谱远远看上去没有变化的同时，使其变得稀疏或者稠密。总体趋势不变确保了逆变换得到的时域信号音调不变，频谱密度的改变减少或增加了逆变换所得信号的数据点，从而改变其时长。与变速变调部分类似，我们用插值方法改变频谱密度。

我们采用快速傅里叶变换算法进行高效的时域-频域转换，效果如图3.4所示。该算法技术性较强，将在下一节得到更详细的讨论。

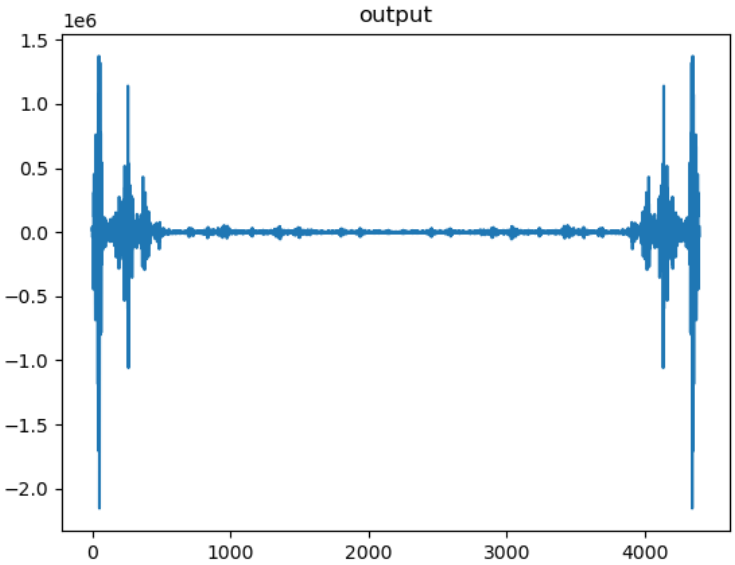

图 3.4: 转换到频域上的信号

3.3 滤波去噪

对音频进行时长规整处理时，我们采用的是分段处理的方法，在各个分段的衔接处会出现一些不连续点，产生一些高频噪音。为此，我们使用 M 点移动平均滤波器对变调处理后的语音进行低通滤波。在基于 python 的可行性验证中，我们将平滑点数取为 5，有效抑制了部分噪声。

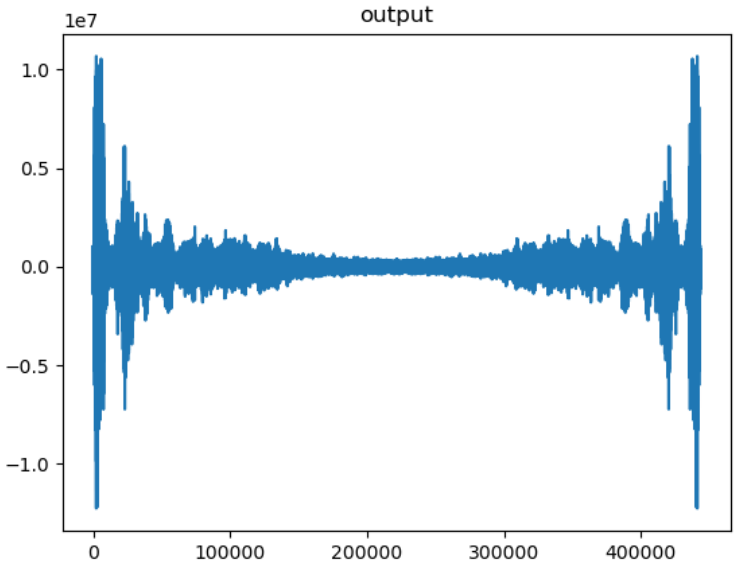

图 3.5: 滤波前的效果

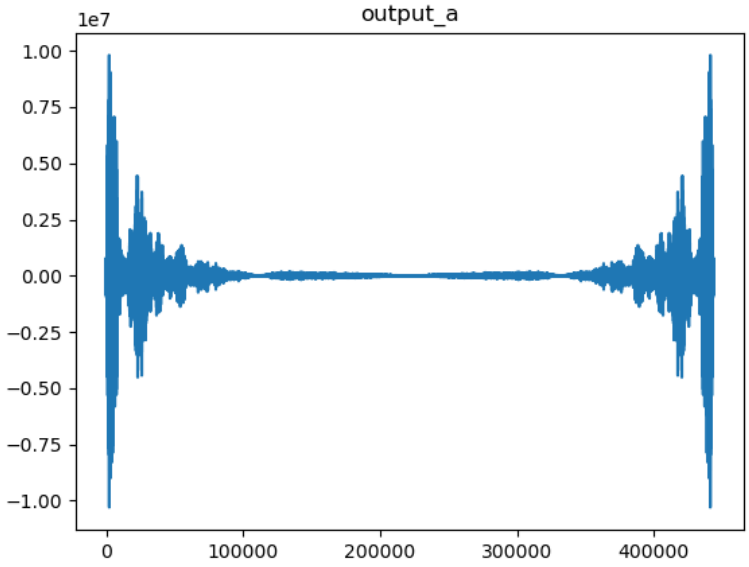

图 3.6: 滤波后的效果

具体来说，M 点平均滤波器使用一个数列 h 对输入信号做卷积，h 的定义为:

$$h[n]=\begin{cases}1/M, & 0\le n\le M-1\\ 0, & \text{otherwise}\end{cases}$$

根据傅里叶变换的性质，我们知道时域上的卷积等价于频域上的对位相乘。也就是说，原始信号经过滤波后，它的每个频率都被乘以一个复常数。通过计算 M 点平均滤波器的频域响应，我们画出它的强度响应图谱:

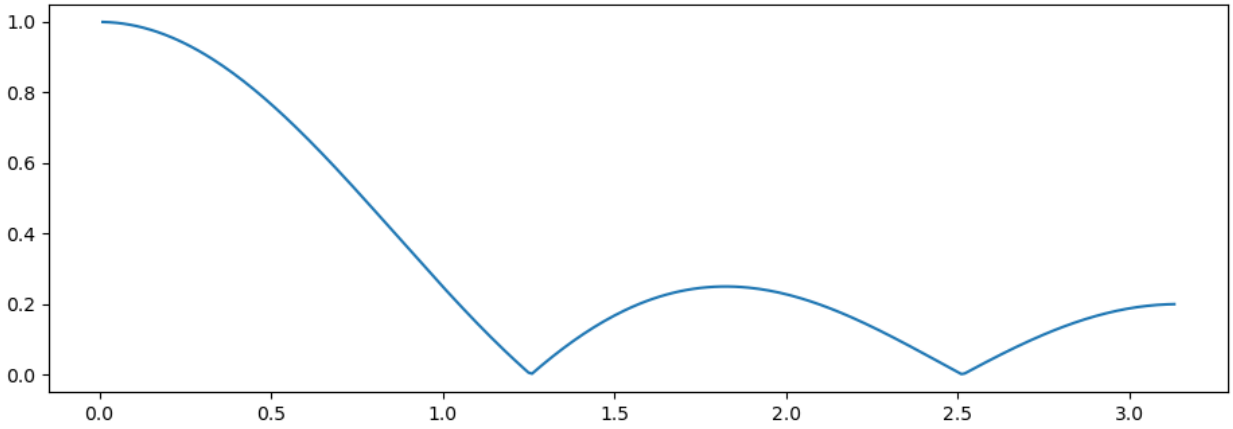

图 3.7: 5 点平均滤波器的强度响应

该图横轴表示频率，纵轴表示该频率的强度响应。也就是说，输出信号相应频率的强度等于输入信号相应频率的强度乘以图中该频率对应的数值。可以看到，经过滤波处理，每个频率的强度都有所损失，而高频信号的损失更为显著；因此 M 点平均滤波器对于高频噪声有明显的抑制作用，很好地满足了我们的需要。

4 快速傅里叶变换

4.1 快速傅里叶变换

离散傅里叶变换（Discrete Fourier Transform，DFT），可以将信号的时域采样变换为其 DTFT 的频域采样。而快速傅立叶变换（Fast Fourier Transform，FFT）是是一种高效实现 DFT 的算法，在频谱分析领域更加常用。

FFT 算法的基本思想是分治。它的分治思想体现在将多项式分为奇次项和偶次项处理。[3][6]
对于一个多项式:

$$f(x)=\sum_{i=0}^{N}a_i*x^i$$

按照奇偶分成两组:

$$f(x)=\sum_{i=0}^{N}(a_{2i}*x^{2i}+a_{2i+1}*x^{2i+1})$$

记:

$$G(x) = \sum_{i=0}^{N} a_{2i} * x^{2i}$$

$$H(x) = \sum_{i=0}^{N} a_{2i+1} * x^{2i}$$

则:

$$f(x) = G(x) + x * H(x)$$

利用单位复根的性质得到:

$$\begin{aligned}
\mathrm{DFT}\left(f\left(\omega_n^k\right)\right) &= \mathrm{DFT}\left(G\left(\left(\omega_n^k\right)^2\right)\right) + \omega_n^k \times \mathrm{DFT}\left(H\left(\left(\omega_n^k\right)^2\right)\right) \\
&= \mathrm{DFT}\left(G\left(\omega_n^{2k}\right)\right) + \omega_n^k \times \mathrm{DFT}\left(H\left(\omega_n^{2k}\right)\right) \\
&= \mathrm{DFT}\left(G\left(\omega_{n/2}^{k}\right)\right) + \omega_n^k \times \mathrm{DFT}\left(H\left(\omega_{n/2}^{k}\right)\right)
\end{aligned}$$

同理:

$$\begin{aligned}
\mathrm{DFT}\left(f\left(\omega_n^{k+n/2}\right)\right) &= \mathrm{DFT}\left(G\left(\omega_n^{2k+n}\right)\right) + \omega_n^{k+n/2} \times \mathrm{DFT}\left(H\left(\omega_n^{2k+n}\right)\right) \\
&= \mathrm{DFT}\left(G\left(\omega_n^{2k}\right)\right) - \omega_n^k \times \mathrm{DFT}\left(H\left(\omega_n^{2k}\right)\right) \\
&= \mathrm{DFT}\left(G\left(\omega_{n/2}^{k}\right)\right) - \omega_n^k \times \mathrm{DFT}\left(H\left(\omega_{n/2}^{k}\right)\right)
\end{aligned}$$

由此，我们可以对 $G(x)$ 和 $H(x)$ 递归求解，伪代码如下:

Algorithm 1 递归版 FFT，时间复杂度 O(nlog(n))

```
0: function FFT(complex * f, intn)
0:     if(n == 1)return
0:     tmp ← f
0:     f[n/2 : n] ← oddterms
0:     f[0 : n/2] ← eventerms
0:     g ← f
0:     h ← f + n/2
0:     function FFT(f, n/2)
0:     end function
0:     function FFT(f + n/2, n/2)
0:     end function
0:     complexcur = 1, step = exp(I * (2 * M_P I/n * rev)
1: for each i ∈ [0, n/2] do
1:         tmp[i] ← g[i] + cur * h[i] ;
1:         tmp[i = n/2] ← g[i] − cur * h[i] ;
1:         cur* = step;
2: end for
2:     f ← tmp
2: end function=0
```

在递归版 FFT 的基础上，可通过位逆序置换（bit-reversal permutation）进一步优化。在递归过程中，下标为偶数的放在左边，下标为奇数的放在右边，即二进制末尾为 0 的在左边，为 1 的在右边。归纳推理可知，最终各个叶子节点会出现一个位逆序置换（假设数据规模 n 是 2 的整数幂），即一个数的最终位置是它原本下标二进制对称反转后的数。
例如，假设共有 8 个元素：

$$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$$

对应下标的二进制表示：

$$[000, 001, 010, 011, 100, 101, 110, 111]$$

变换后的位置：

$$[a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$$

对应下标的二进制表示：

$$[000, 100, 010, 110, 001, 101, 011, 111]$$

由此，我们可以预见每个数最终的位置，将递归交换位置替换为直接将数移动到目标位置。同时，我们发现，如果要求一个数 x 二进制反转后的数（记为 R(x)），可以先把它先右移一位，然后取对称，再右移一位，如果原数是奇数首位补 1，就得到了反转后的 x，即：

$$R(x) = \left\lceil \frac{R\left(\left\lfloor \frac{x}{2} \right\rfloor\right)}{2} \right\rceil + (x \bmod 2) \times \frac{\text{len}}{2}$$

这也是目前应用最广的 FFT 算法。

4.2 快速傅里叶逆变换

快速傅里叶逆变换（IDFT）是一个已知目标多项式的点值，将其转回多项式系数形式的算法。考虑到 FFT 本身是个线性变换，我们可以将 FFT 过程写成矩阵乘法的形式：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

所以我们只需要乘中间矩阵的逆矩阵就能求得系数 a 向量，用线性代数的知识推理可知，它的逆矩阵是每一项取倒数再除以 n。为了求倒数，使用欧拉公式：

$$\frac{1}{\omega_k} = \omega_k^{-1} = e^{-\frac{2\pi i}{k}} = \cos\left(\frac{2\pi}{k}\right) + i \cdot \sin\left(-\frac{2\pi}{k}\right)$$

因此我们可以以把单位根 ω_k 取成 $e^{-\frac{2\pi i}{k}}$，其他操作与 FFT 完全一致。这就是目前最常见的串行 FFT 算法。

5 并行算法 [5]

5.1 SIMD 优化

根据 Flynn's taxonomy（弗林分类法），计算机体系结构可以分为 Single instruction stream single data stream (SISD), Single instruction stream multiple data streams (SIMD), Multiple instruction streams single data stream (MISD), Multiple instruction streams multiple data streams (MIMD) 四种，如图5.8所示：[4]

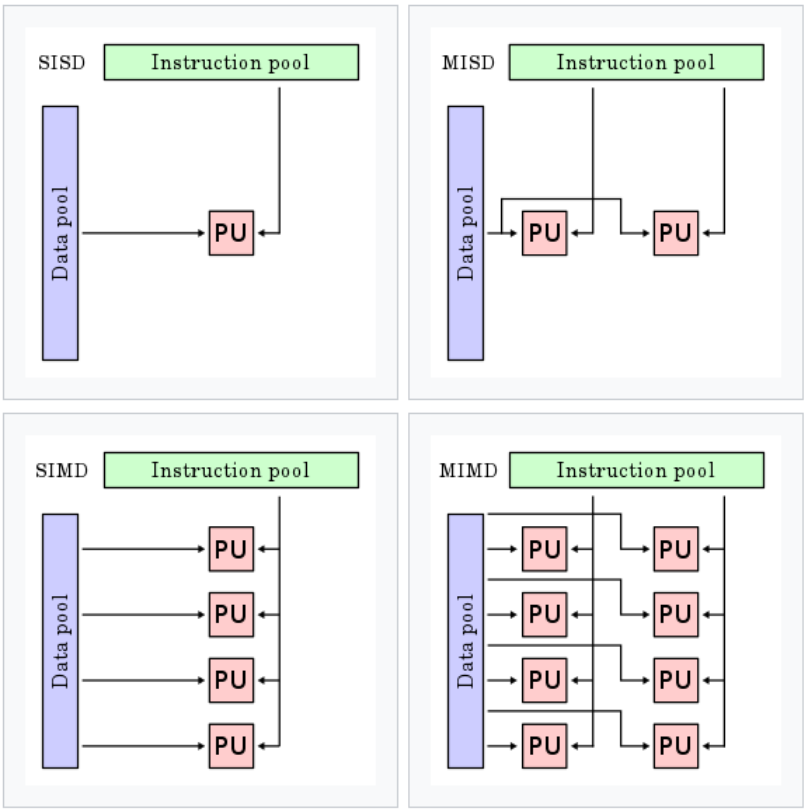

图 5.8: 四种体系结构

FFT 算法中存在大量相同操作的数据，适合用 SSE、AVX、Neon 等架构进行 SIMD 优化。待处理序列以数组的形式存储，对前后两部分有计算，可使用向量寄存器将两部分分别存储为向量，进行向量批量计算。

另一方面，卷积操作由一系列对位相乘求和组成，也适合进行 SIMD 优化。我们用于去噪的低通滤波器包含一个很小的卷积核，这个卷积核完全可以放进一个向量寄存器中。卷积操作相当于用卷积核与待处理序列对位相乘然后求和，将卷积核平移，继续对位相乘求和，不断进行这一操作直到序列被处理完毕。为了使用 SIMD 优化，我们可以在上述的每一步将序列的一部分加载进向量寄存器，然后将它与预先准备好的储存了卷积核的寄存器对位相乘并求和。

5.2 多线程优化

多线程是 CPU 在操作系统的支持下同时支持多个线程的执行的能力。在多线程应用程序中，线程共享单个或多个内核的资源，包括计算单元、CPU 缓存等。多线程旨在通过使用线程级并行以及指

令级并行来提高单个内核的利用率，这两种技术是互补的，几乎在所有具有多个多线程 CPU 和具有多个多线程内核的 CPU 的现代系统架构中，这两种技术都结合在一起使用。

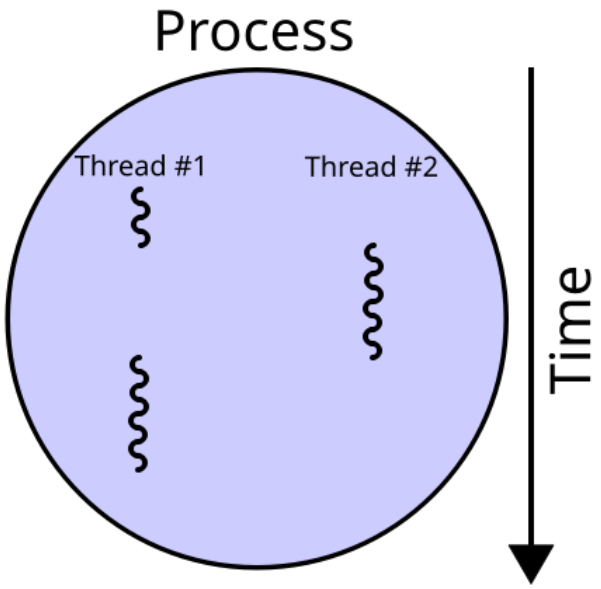

图 5.9: 两个线程可以运行在一个处理器单元上

由于语音数字信号只能在很短的区间内被视为周期性的波，因此必须将频域信号分段处理，各段信号之间互相不依赖，这部分处理非常适合使用多线程的并行优化算法。我们将会使用 MPI、pthreads 以及 openMP 等函数库，分别在 arm 和 x86 两个平台上尝试进行分布式内存和共享式内存的编程，以及测试、分析数据。

6 分工

- 杜岱玮负责设计整体算法流程和变速不变调、滤波去噪的代码实现，主要是频域信号上的工作，再使用陈静怡负责的快速傅里叶逆变换转回原来的时域音频信号。
- 陈静怡负责调查快速傅里叶变换的原理，以及快速傅里叶变换和快速傅里叶逆变换的代码实现。为杜岱玮的后续工作提供处理好的频域信号。

参考文献

[1] http://www.guitarpitchshifter.com/algorithm.html#33.

[2] http://www.panix.com/~jens/pvoc-dolson.par.

[3] https://oi-wiki.org/.

[4] https://en.wikipedia.org/wiki/Flynn%27s_taxonomy.

[5] Peter S. Pacheco. *An Introduction to Parallel Programming*. China Machine Press, 2012.

[6] Charles E. Leiserson Thomas H. Cormen. *Introduction to Algorithms, Third Edition*. China Machine Press, 2013.

[7] 田岚张晓蕊. 语音变调方法分析及音效评估. 山东大学学报 *(工学版)*, 2011.