南開大學
Nankai University

计算机学院

并行程序设计作业

SIMD 编程实验

姓名：杜岱玮

学号：2011421

专业：计算机科学与技术

2022 年 4 月 10 日

目录

1 我做了什么 **2**
1.1 github 地址 2

2 对序列拉伸的 SIMD 优化 **2**
2.1 问题描述 2
2.2 第一种 SIMD 优化算法 3
2.3 测试结果与分析 4
2.4 第二种 SIMD 优化算法 5
2.5 测试结果与分析 6

3 卷积低通滤波的 SIMD 优化 **7**
3.1 问题描述 7
3.2 SIMD 优化 7
3.3 测试结果和分析 8

4 AVX SIMD 优化实验 **9**
4.1 从 neon 转换到 AVX 9
4.2 测试结果 9
4.2.1 序列拉伸 9
4.2.2 卷积滤波 9

5 附录：测试代码 **10**
5.1 arm-neon 测试代码 10
5.2 x86-AVX 测试代码 16

1 我做了什么

这学期我（杜岱玮，2011421）计划与陈静怡（2012885）共同完成期末作业，我们的选题是基于频域插值的音频变调算法。整个算法中最为复杂的环节是快速傅里叶变换（FFT），我们将它单独分割出来由陈静怡负责，而我负责算法其余部分的设计和优化，即主要负责时域和频域的音频处理。关于算法的细节，请参阅我们的开题报告，我就不在这里多说了。

在这篇实验报告中，我将展示我在 SIMD 并行优化方面进行的各种探索。

我发现由我负责的时频域处理方面有两个环节适合进行 SIMD 优化：一个是通过线性插值进行序列长度的拉伸，另一个是通过卷积低通滤波去除音频杂音。对于第一个环节，我进行了两次 SIMD 优化尝试：在第一次尝试中，我惊讶地发现算法性能不升反降；对算法进行了进一步优化后，我终于在第二次尝试中取得了相对于基础算法的性能提升。我将尝试对算法的表现进行解释。对于第二个环节卷积滤波，SIMD 优化不出所料地使算法性能有了巨大提升。

这些实验是在 arm 服务器上进行的，我又在 x86 架构的笔记本电脑上重复了上述实验。

1.1 github 地址

https://github.com/ephratica/parallel-homework

2 对序列拉伸的 SIMD 优化

2.1 问题描述

在时域及频域的数字信号处理中经常需要面对这样的问题：给定一个长度为 len 的序列 A，我需要将它拉伸以获得一个长度为 $newlen$ 的序列 B。由于 B 是由 A 经拉伸获得的，我要确保它们的变化趋势相同。

实际做法中通常用线性插值解决这个问题。令 $rate = \frac{newlen}{len}$ 表示拉伸比例,$p = \frac{i}{rate}$ 表示序列 B 中的第 i 位对应的拉伸前的位置，则序列 B 可以用下面的方法计算：

$$B[i] = (\lfloor p \rfloor + 1 - p)A[\lfloor p \rfloor] + (p - \lfloor p \rfloor)A[\lceil p \rceil]$$

.

其中 $\lfloor\rfloor$ 表示向下取整。由于 i 是序列 B 的一个下标，它是整数，但它在序列 A 中对应的位置 p 不一定是整数；所以上面的式子用向下取整找到距离 p 最近的两个整数，并用对应位置的序列 A 的值进行线性插值计算 $B[i]$。

直接根据公式很容易写出序列拉伸的朴素算法：

序列拉伸的朴素算法

```
1  std::pair<float *, int> stretch(float *data, int len, int newlen)
2  {
3    float rate = (float)newlen / len;
4    float *newdata = new float[newlen];
5    for (int i = 0; i < newlen; i++)
6    {
7      float pos = i / rate;
8      int left = (int)pos, right = left + 1;
```

```
        newdata[i] = data[left] * (right - pos) + data[right] * (pos - left);
      }
      return { newdata, newlen };
    }
```

2.2 第一种 SIMD 优化算法

首先我考虑直接用 Neon 对朴素算法进行 4 路向量化。

为了计算朴素算法中的 pos, left, right，我首先需要把连续的 4 个下标加载到一个向量中。为了方便的实现这一点，我预先将所有可能用到的自然数连续存放在一个数组中，这样需要取连续下标到向量中时只要用 vld1q_s32 指令从数组的对应位置加载就好了。

我遇到的一个问题是如何在浮点型向量和整型向量之间转换。由于算法中用到了向下取整，我需要通过将浮点型向量转换为整型向量来实现这一点。通过查阅资料，我了解到 vcvtq_f32_s32 可以将 32 位浮点数转换为有符号整数，vcvtq_s32_f32 可以将 32 位有符号整数转换为 32 位浮点数。

另外一个问题是实现朴素算法中的 “right = left + 1” 语句。它对应向量与标量的加法，而问题在于 neon 中没有向量加标量的指令。作为解决方案，我在循环开始前定义了一个向量 vones = (1,1,1,1)，这样就可以用向量 vones 代替标量 1，把向量加标量转换为向量加向量。

下面的代码用 4 路向量计算朴素算法中的 pos,left,right,(right - pos),(pos - left):

一部分向量计算

```
    for (int i = 0; i < newlen4; i += 4)
    {
      int32x4_t vint = vld1q_s32(constants.numbers + i);  //numbers[i] = i
      float32x4_t vpos = vcvtq_f32_s32(vint); //pos
      vpos = vmulq_n_f32(vpos, _rate);  //_rate = 1 / rate
      int32x4_t vleft = vcvtq_s32_f32(vpos);  //left = floor(i / rate)
      int32x4_t vright = vaddq_s32(vleft, vone);//right = left + 1
      float32x4_t vfl = vcvtq_f32_s32(vleft);
      float32x4_t vfr = vcvtq_f32_s32(vright);
      vfl = vsubq_f32(vpos, vfl);     //pos - left
      vfr = vsubq_f32(vfr, vpos);     //right - pos

        ......
    }
```

接下来我需要读取数组 data 中下标对应 vleft 和 vright 的元素，并用它们组成两个向量。对于这个操作我没有找到好的优化方法，只能逐一读取。经过我的测试，访问向量单独元素的 vsetq_lane_f32 和 vgetq_lane_f32 效率较低；为了更快地单独访问 vleft,vright 的每个元素，我先把它们存为数组。

从数组里读取数据

```
    for (int i = 0; i < newlen4; i += 4)
    {
        ......

      vst1q_s32(tmpi, vleft);
      tmpf[0] = data[tmpi[0]];
```

```
    tmpf[1] = data[tmpi[1]];
    tmpf[2] = data[tmpi[2]];
    tmpf[3] = data[tmpi[3]];
    float32x4_t vdataleft = vld1q_f32(tmpf);
    vst1q_s32(tmpi, vright);
    tmpf[0] = data[tmpi[0]];
    tmpf[1] = data[tmpi[1]];
    tmpf[2] = data[tmpi[2]];
    tmpf[3] = data[tmpi[3]];
    float32x4_t vdataright = vld1q_f32(tmpf);

    ......
}
```

最后一步是计算 data[left] * (right - pos) + data[right] * (pos - left)。现在 data[left] 被存在 vdataleft 中，right-pos 被存在 vfr 中，dataright 被存在 vdataright 中，pos-left 被存在 vlr 中。上述计算就变成了向量运算 vdataleft*vfr+vdataright*vfl：

完成计算

```
for (int i = 0; i < newlen4; i += 4)
{
    ......

    float32x4_t vterm1 = vmulq_f32(vdataleft, vfr);
    float32x4_t vterm2 = vmulq_f32(vdataright, vfl);
    vterm1 = vaddq_f32(vterm1, vterm2);
    vst1q_f32(newdata + i, vterm1);      //运算结果存入newdata

}
```

2.3 测试结果与分析

我对朴素算法和 SIMD 优化算法在不同输入输出规模上进行了测试和比较，测试代码请参照附录。得到的结果令人吃惊，如下表所示：

输入序列长度	输出序列长度	运行次数	朴素算法运行时间/s	SIMD 运行时间/s
1000	1800	50000	1.05	3.14
10000	18000	5000	1.05	3.14
100000	180000	500	1.05	3.15
1000000	1800000	50	1.08	3.14

表 1: 性能测试结果

为了使测试用时相差不多，我让不同测试的输入输出长度 x 运行次数相等。可见，同一算法对于不同规模的输入输出的效率基本一致。出人意料的是，我的 SIMD” 优化” 令算法变慢 3 倍!

我认为算法变慢有两个主要原因：

- 向量操作数量过多，额外开销巨大。为了完成较为复杂的计算，我在每次循环调用了 4 次类型转

换、2 次向量加法、2 次向量减法、1 次标量乘法、2 次向量乘法、3 次向量载入、3 次向量存储。虽然理论上朴素算法的每次循环也会执行对应的操作，但过多的向量操作的额外开销显然大大超过了它们带来的优势。

- 内存读取写入数量太多。朴素算法在计算每个元素时，需要从内存读取 2 次，向内存写入 1 次。而对于 SIMD 算法，它需要将向量化计算的数组下标先写入再读取以访问向量的每一位，再将对应位置的数据通过读取-存储-读取整理成向量，最后把计算结果存入内存。它对于每个数组元素总计读取 5 次、写入 4 次，总数是朴素算法的 3 倍。内存访问操作很可能拖累了算法的效率。

针对这些问题，我对 SIMD 算法进行了进一步的优化。

2.4 第二种 SIMD 优化算法

首先我考虑如何减少向量操作的次数。我注意到，在实际的算法流程中，音频会被分割成小段进行处理，每一小段的长度是相同的，而这样就会产生很多重复计算。尤为值得注意的是两个系数 (right - pos) 和 (pos - left)，它们在算法每次运行的过程中都是相同的，却需要好几个向量指令来计算。我认为我可以在开始算法的多轮执行之前将这两个系数预处理出来，这样每次循环开始时只需调用 vld1q_f32 就可以将它们加载进向量寄存器。

另一个可以优化之处是减少内存操作。我发现大量内存操作是在计算数组下标并取回那个下标位置的数的过程中产生的：由于计算得到的下标在向量寄存器里，我要先把它们移出寄存器，再取来对应数据，最后把这些数据打包成向量。一种减少内存操作的办法是用 vsetq_lane_f32 和 vgetq_lane_f32 代替两次内存操作，但这两个指令效率同样不高。最终，我选择放弃 SIMD，直接计算数组下标并储存在临时变量中，这样不仅节省了 1 次读取和 1 次写入，还减少了好几个向量指令。

还有一个可优化之处在于将一个乘法指令和一个加法指令合并为一个乘加指令。循环的最后执行了两个向量乘法和一个向量加法，我将其中一个乘法与加法合并，用乘加指令 vmlaq_f32 代替。

优化的 SIMD 算法

```
float *tmpl, *tmpr;
void stretch2_prepare(int len, int newlen)
{
  float rate = (float)newlen / len;
  tmpl = new float[newlen];
  tmpr = new float[newlen];
  for (int i = 0; i < newlen; i++)
  {
    float pos = i / rate;
    tmpl[i] = pos - (int)pos;
    tmpr[i] = 1 - tmpl[i];
  }
}
std::pair<float *, int> stretch2(float *data, int len, int newlen)
{
  float rate = (float)newlen / len, _rate = 1 / rate;
  float *newdata = new float[newlen];
  int newlen4 = newlen / 4 * 4;
  float tmp[4];
  for (int i = 0; i < newlen4; i += 4)
```

```
21      {
22        float32x4_t vl = vld1q_f32(tmpl + i);
23        float32x4_t vr = vld1q_f32(tmpr + i);
24
25        int l0 = i / rate, l1 = (i + 1) / rate, l2 = (i + 2) / rate, l3 = (i + 3) /
              rate;
26        tmp[0] = data[l0]; tmp[1] = data[l1];
27        tmp[2] = data[l2]; tmp[3] = data[l3];
28        float32x4_t vdataleft = vld1q_f32(tmp);
29        tmp[0] = data[l0 + 1]; tmp[1] = data[l1 + 1];
30        tmp[2] = data[l2 + 1]; tmp[3] = data[l3 + 1];
31        float32x4_t vdataright = vld1q_f32(tmp);
32
33        float32x4_t vterm1 = vmulq_f32(vdataleft, vr);
34        float32x4_t vterm2 = vmlaq_f32(vterm1, vdataright, vl);
35        vst1q_f32(newdata + i, vterm2);
36      }
37      for (int i = newlen4; i < newlen; i++)  //下面处理边界情形
38      {
39        float pos = i / rate;
40        int left = (int)pos, right = left + 1;
41        newdata[i] = data[left] * (right - pos) + data[right] * (pos - left);
42      }
43      newdata[newlen - 1] = data[len - 1];
44      return { newdata, newlen };
45    }
```

2.5 测试结果与分析

我用相同的数据对改进过的 SIMD 算法进行了测试。优化效果十分明显，改进过的 SIMD 算法大大优于原版 SIMD 算法，而且相较朴素算法也有约 20% 的改进。

输入序列长度	输出序列长度	运行次数	朴素算法运行时间/s	SIMD 运行时间/s	优化 SIMD 运行时间/s
1000	1800	50000	1.05	3.14	0.86
10000	18000	5000	1.05	3.14	0.86
100000	180000	500	1.05	3.15	0.87
1000000	1800000	50	1.08	3.14	0.87

表 2: 性能测试结果

然而，我原本期望 SIMD 能够带来成倍的性能提升，实际的 SIMD 优化显然没有达到我的预期。我认为原因在于，序列拉伸算法的很大一部分不适合 SIMD 优化。SIMD 最适合对依次排列的数据进行统一的操作然后整体放回，而序列拉伸的很大一部分工作量在于数据的重新排列；只有计算线性插值公式的部分是最适合 SIMD 的。所以，哪怕 SIMD 能将这一部分大大提速，它的整体优化效果也不会那么明显。

3 卷积低通滤波的 SIMD 优化

3.1 问题描述

卷积低通滤波器常常被用于去除音频中的噪声。一个平滑系数为 k 的卷积滤波器将输入序列的每一个点的值替换为它周围的 k 个点的算术平均值。在这里，我研究的是平滑系数为 5 的卷积滤波器。

卷积低通滤波器的朴素算法很简单，它首先计算每个点周围的五个点的和，然后将它们全部除以 5。

卷积低通滤波器的朴素算法

```
1  std::pair<float *, int> filter(float *data, int len)
2  {
3    float *ret = new float[len];
4    memset(ret, 0, sizeof(float) * len);
5    for (int i = -2; i <= 2; i++)
6      for (int j = 0; j < len; j++)
7        if (j + i >= 0 && j + i < len)
8          ret[j + i] += data[j];
9    for (int i = 2; i < len - 2; i++)
10     ret[i] /= 5;
11   ret[1] /= 4, ret[len - 2] /= 4;
12   ret[0] /= 3, ret[len - 1] /= 3;
13   return { ret,len };
14 }
```

3.2 SIMD 优化

卷积滤波器的朴素算法有两点明显的可优化之处。

在朴素算法的第 6-8 行，data 数组被一一对应地加到 (ret + i) 数组上。这正是 SIMD 应用的典型场景，可以通过向量加法得到很好的优化。

唯一一个问题在于，SIMD 难以实现第 7 行的条件判断，容易造成内存访问越界。为了解决这个问题，我在分配 ret 数组的空间时在数组两端额外分配一些空闲空间，使得程序的轻微访问越界不会引发异常。

求和部分的向量化

```
1      float *ret = new float[len + 4];
2  memset(ret, 0, sizeof(float) * (len + 4));
3  ret += 2;
4  int len4 = len / 4 * 4;
5  for (int i = -2; i <= 2; i++)
6  {
7    for (int j = 0; j < len4; j += 4)
8    {
9      float32x4_t vdata = vld1q_f32(data + j);
10     float32x4_t vret = vld1q_f32(ret + j + i);
11     vret = vaddq_f32(vret, vdata);
```

```
12          vst1q_f32(ret + j + i, vret);
13        }
14        for (int j = len4; j < len; j++)
15          ret[j + i] += data[j];
16      }
```

另一处可优化点在朴素算法的第 9-10 行，ret 数组的每个元素都被除以 5。这同样可以简单地使用向量化进行加速。

一个小问题是 neon 不支持标量除法，于是我把除以 5 转化为乘以 0.2，然后使用标量乘法指令 vmulq_n_f32。

由于 ret 数组的起始位置被我改变了，如果有程序在函数外使用 “delete[] ret” 会引发异常；因此我在将 ret 的每个元素除以 5 的同时将它拷贝到另外一个数组 tmp 中，并以 tmp 作为返回值。

向量化的标量乘法

```
1     float *tmp = new float[len];
2     int lim = 2 + (len - 4) / 4 * 4;
3     for (int i = 2; i < lim; i += 4)
4     {
5       float32x4_t vret = vld1q_f32(ret + i);
6       vret = vmulq_n_f32(vret, 0.2);
7       vst1q_f32(tmp + i, vret);
8     }
9     for (int i = lim; i < len - 2; i++)
10      tmp[i] = ret[i] / 5;
11    tmp[1] = ret[1] / 4, tmp[len - 2] = ret[len - 2] / 4;
12    tmp[0] = ret[0] / 3, tmp[len - 1] = ret[len - 1] / 3;
13    delete[](ret - 2);
14    return { tmp,len };
```

3.3 测试结果和分析

我对朴素算法和 SIMD 优化算法在不同输入输出规模上进行了测试和比较，测试代码请参照附录。测试结果如下：

输入序列长度	输出序列长度	运行次数	朴素算法运行时间/s	SIMD 运行时间/s
1000	1800	50000	2.37	0.94
10000	18000	5000	1.70	0.96
100000	180000	500	1.72	0.97
1000000	1800000	50	1.73	1.02

表 3: 性能测试结果

大致来说，SIMD 优化使得卷积滤波提速 75% 左右，符合我的预期。与前文所述的序列拉伸相比，对卷积滤波的 SIMD 优化更加简单直接，额外开销更少，能够取得更好的效果完全在情理之中。

4 AVX SIMD 优化实验

4.1 从 neon 转换到 AVX

为了在 x86 平台测试 SIMD 优化的效果，我将代码中的所有 neon 指令转换为 AVX 指令。

neon 和 AVX 的主要区别在于寄存器位数，neon 寄存器为 128 位而 AVX 寄存器为 256 位。于是代码的结构修改主要在于从 4 路向量化改为 8 路向量化。

neon 的类型或指令与 AVX 的类型或指令基本是一一对应的，例如：

- float32x4_t 对应 ___m256
- vld1q_f32 对应 _mm256_load_ps
- vaddq_f32 对应 _mm256_add_ps
- vmlaq_f32 对应 _mm256_fmadd_ps
- .....

其中有一些例外需要注意。比如，neon 中有 vmulq_n_f32 这样的向量-标量运算指令，而 AVX 只有向量-向量运算指令，只能先用 _mm256_set1_ps 这样的指令获得一个每一位都相同的向量，然后再进行向量-向量运算。

另一个更加危险的例外是乘加指令：neon 的 vmlaq_f32(a,b,c) 返回 a+b*c，而 _mm256_fmadd_ps 返回 a*b+c。我险些在转换时弄错参数顺序。

然而大体上，从 neon 到 AVX 的转换还是相当机械化、相当容易的。

4.2 测试结果

我用相同的测试数据在自己的笔记本电脑上对 AVX 版本代码进行了测试。为免本节变得过于冗长，我将所有 AVX 版本代码都放在了附录中。

4.2.1 序列拉伸

限于精力，我仅仅将 2.4 节中的第二种 SIMD 算法改写为 AVX 版本并进行测试，结果如下：

输入序列长度	输出序列长度	运行次数	朴素算法运行时间/s	SIMD 运行时间/s
1000	1800	50000	1.896	0.432
10000	18000	5000	1.749	0.437
100000	180000	500	1.871	0.543
1000000	1800000	50	1.849	0.509

表 4: 性能测试结果

总体而言，基于 AVX 的 SIMD 优化算法的速度是朴素算法的 3-4 倍，加速比例远远超过 neon 的约 1.2 倍。AVX 的 8 路向量化相较于 neon 的 4 路向量化的优势可以部分解释加速比的差距，但我认为这难以解释为何差距如此之大；我只能认为一些不为我所知的因素，如架构差异或硬件差异，导致了这个结果。

4.2.2 卷积滤波

我将 3.2 节描述的 SIMD 卷积滤波改写为 AVX 版本并进行了测试，结果如下表所示。

输入序列长度	输出序列长度	运行次数	朴素算法运行时间/s	SIMD 运行时间/s	加速比
1000	1800	50000	0.929	0.265	3.51
10000	18000	5000	1.224	0.236	5.19
100000	180000	500	1.115	0.205	5.44
1000000	1800000	50	1.053	0.35	3.01

表 5: 性能测试结果

这次 SIMD 算法的加速比在不同规模的测试数据上变化较大，因此我将它单独列出。总体来看，SIMD 算法的效率应当是朴素算法的 3-6 倍；作为对比，基于 neon 的 SIMD 算法加速比约为 2 倍。我认为这个结果在情理之中，毕竟 AVX 的 256 位寄存器的长度是 neon 的 128 位寄存器的 2 倍，它的加速比达到 neon 的 2 倍左右是完全合理的。

5 附录：测试代码

简要介绍一下代码结构：

- Constants 类包含代码中用到的一些辅助性的常量数组
- 同一个函数的非 SIMD 版本和 SIMD 版本名称相同，只是 SIMD 版本位于命名空间 simd 中
- Test 类负责重复运行函数并计时，测试数据在构造函数中生成，重载的括号被用于接收被测试函数

5.1 arm-neon 测试代码

neon 测试代码

```
#include<cstdio>
#include<arm_neon.h>
#include<utility>
#include<time.h>
#include<math.h>
#include<iostream>
#include<cstring>
#include<vector>

class Constants {
public:
  static const int MAXN = 3000000;
  int *numbers;
  int *ones;

  Constants()
  {
    numbers = new int[MAXN];
    ones = new int[MAXN];
    for (int i = 0; i < MAXN; i++)
```

```
21        numbers[i] = i, ones[i] = 1;
22    }
23    ~Constants() { delete[] numbers, delete[] ones; }
24  } constants;
25
26  namespace simd {
27
28    std::pair<float *, int> filter(float *data, int len)
29    {
30      float *ret = new float[len + 4];
31      memset(ret, 0, sizeof(float) * (len + 4));
32      ret += 2;
33      int len4 = len / 4 * 4;
34      for (int i = -2; i <= 2; i++)
35      {
36        for (int j = 0; j < len4; j += 4)
37        {
38          float32x4_t vdata = vld1q_f32(data + j);
39          float32x4_t vret = vld1q_f32(ret + j + i);
40          vret = vaddq_f32(vret, vdata);
41          vst1q_f32(ret + j + i, vret);
42        }
43        for (int j = len4; j < len; j++)
44          ret[j + i] += data[j];
45      }
46      float *tmp = new float[len];
47      int lim = 2 + (len - 4) / 4 * 4;
48      for (int i = 2; i < lim; i += 4)
49      {
50        float32x4_t vret = vld1q_f32(ret + i);
51        vret = vmulq_n_f32(vret, 0.2);
52        vst1q_f32(tmp + i, vret);
53      }
54      for (int i = lim; i < len - 2; i++)
55        tmp[i] = ret[i] / 5;
56      tmp[1] = ret[1] / 4, tmp[len - 2] = ret[len - 2] / 4;
57      tmp[0] = ret[0] / 3, tmp[len - 1] = ret[len - 1] / 3;
58      delete[](ret - 2);
59      return { tmp,len };
60    }
61
62    float *tmpl, *tmpr;
63    void stretch2_prepare(int len, int newlen)
64    {
65      float rate = (float)newlen / len;
66      tmpl = new float[newlen];
67      tmpr = new float[newlen];
68      for (int i = 0; i < newlen; i++)
69      {
```

```
70        float pos = i / rate;
71        tmpl[i] = pos - (int)pos;
72        tmpr[i] = 1 - tmpl[i];
73      }
74    }
75    std::pair<float *, int> stretch2(float *data, int len, int newlen)
76    {
77      float rate = (float)newlen / len, _rate = 1 / rate;
78      float *newdata = new float[newlen];
79      int newlen4 = newlen / 4 * 4;
80      float tmp[4];
81      for (int i = 0; i < newlen4; i += 4)
82      {
83        float32x4_t vl = vld1q_f32(tmpl + i);
84        float32x4_t vr = vld1q_f32(tmpr + i);
85
86        int l0 = i / rate, l1 = (i + 1) / rate, l2 = (i + 2) / rate, l3 = (i + 3) /
              rate;
87        tmp[0] = data[l0]; tmp[1] = data[l1];
88        tmp[2] = data[l2]; tmp[3] = data[l3];
89        float32x4_t vdataleft = vld1q_f32(tmp);
90        tmp[0] = data[l0 + 1]; tmp[1] = data[l1 + 1];
91        tmp[2] = data[l2 + 1]; tmp[3] = data[l3 + 1];
92        float32x4_t vdataright = vld1q_f32(tmp);
93
94        float32x4_t vterm1 = vmulq_f32(vdataleft, vr);
95        float32x4_t vterm2 = vmlaq_f32(vterm1, vdataright, vl);
96        vst1q_f32(newdata + i, vterm2);
97      }
98      for (int i = newlen4; i < newlen; i++)  //下面处理边界情形
99      {
100       float pos = i / rate;
101       int left = (int)pos, right = left + 1;
102       newdata[i] = data[left] * (right - pos) + data[right] * (pos - left);
103     }
104     newdata[newlen - 1] = data[len - 1];
105     return { newdata, newlen };
106   }
107   std::pair<float *, int> stretch(float *data, int len, int newlen)
108   {
109     float rate = (float)newlen / len, _rate = 1 / rate;
110     float *newdata = new float[newlen];
111     int newlen4 = newlen / 4 * 4;
112     int32x4_t vone = vld1q_s32(constants.ones);
113     int tmpi[4];
114     float tmpf[4];
115     for (int i = 0; i < newlen4; i += 4)
116     {
117       int32x4_t vint = vld1q_s32(constants.numbers + i);  //numbers[i] = i
```

```
118         float32x4_t vpos = vcvtq_f32_s32(vint);
119         vpos = vmulq_n_f32(vpos, _rate);  //_rate = 1 / rate
120         int32x4_t vleft = vcvtq_s32_f32(vpos);  //floor(i / rate)
121         int32x4_t vright = vaddq_s32(vleft, vone);
122         float32x4_t vfl = vcvtq_f32_s32(vleft);
123         float32x4_t vfr = vcvtq_f32_s32(vright);
124         vfl = vsubq_f32(vpos, vfl);
125         vfr = vsubq_f32(vfr, vpos);
126 
127       /*  float32x4_t vdataleft;
128         vdataleft = vsetq_lane_f32(data[vgetq_lane_s32(vleft, 0)], vdataleft, 0);
129         vdataleft = vsetq_lane_f32(data[vgetq_lane_s32(vleft, 1)], vdataleft, 1);
130         vdataleft = vsetq_lane_f32(data[vgetq_lane_s32(vleft, 2)], vdataleft, 2);
131         vdataleft = vsetq_lane_f32(data[vgetq_lane_s32(vleft, 3)], vdataleft, 3);
132         float32x4_t vdataright;
133         vdataright = vsetq_lane_f32(data[vgetq_lane_s32(vright, 0)], vdataright, 0);
134         vdataright = vsetq_lane_f32(data[vgetq_lane_s32(vright, 1)], vdataright, 1);
135         vdataright = vsetq_lane_f32(data[vgetq_lane_s32(vright, 2)], vdataright, 2);
136         vdataright = vsetq_lane_f32(data[vgetq_lane_s32(vright, 3)], vdataright, 3);*/
137         vst1q_s32(tmpi, vleft);
138         tmpf[0] = data[tmpi[0]];
139         tmpf[1] = data[tmpi[1]];
140         tmpf[2] = data[tmpi[2]];
141         tmpf[3] = data[tmpi[3]];
142         float32x4_t vdataleft = vld1q_f32(tmpf);
143         vst1q_s32(tmpi, vright);
144         tmpf[0] = data[tmpi[0]];
145         tmpf[1] = data[tmpi[1]];
146         tmpf[2] = data[tmpi[2]];
147         tmpf[3] = data[tmpi[3]];
148         float32x4_t vdataright = vld1q_f32(tmpf);
149 
150         float32x4_t vterm1 = vmulq_f32(vdataleft, vfr);
151         float32x4_t vterm2 = vmulq_f32(vdataright, vfl);
152         vterm1 = vaddq_f32(vterm1, vterm2);
153         vst1q_f32(newdata + i, vterm1);
154 
155       }
156       for (int i = newlen4; i < newlen; i++)
157       {
158         float pos = i / rate;
159         int left = (int)pos, right = left + 1;
160         newdata[i] = data[left] * (right - pos) + data[right] * (pos - left);
161       }
162       newdata[newlen - 1] = data[len - 1];
163       return { newdata, newlen };
164     }
165 
166 }
```

```
167
168 class Test {
169 public:
170   float *data;
171   int len;
172   int newlen;
173   int times;
174   Test(int n, int m, int times)
175   {
176     len = n;
177     newlen = m;
178     this->times = times;
179     data = new float[n];
180     for (int i = 0; i < n; i++)
181       data[i] = sin(i);
182   }
183   ~Test() { delete[] data; }
184
185   double operator () (std::pair<float *, int>(*f)(float *, int, int))
186   {
187     long long st = clock();
188     for (int i = 0; i < times; i++)
189       delete[] f(data, len, newlen).first;
190     return (1.0 * clock() - st) / CLOCKS_PER_SEC;
191   }
192   double operator () (std::pair<float *, int>(*f)(float *, int))
193   {
194     long long st = clock();
195     for (int i = 0; i < times; i++)
196       delete[] f(data, len).first;
197     return (1.0 * clock() - st) / CLOCKS_PER_SEC;
198   }
199 };
200
201 std::pair<float *, int> stretch(float *data, int len, int newlen)
202 {
203   float rate = (float)newlen / len;
204   float *newdata = new float[newlen];
205   for (int i = 0; i < newlen; i++)
206   {
207     float pos = i / rate;
208     int left = (int)pos, right = left + 1;
209     newdata[i] = data[left] * (right - pos) + data[right] * (pos - left);
210   }
211   return { newdata, newlen };
212 }
213
214 std::pair<float *, int> filter(float *data, int len)
215 {
```

```
216   float *ret = new float[len];
217   memset(ret, 0, sizeof(float) * len);
218   for (int i = -2; i <= 2; i++)
219     for (int j = 0; j < len; j++)
220       if (j + i >= 0 && j + i < len)
221         ret[j + i] += data[j];
222   for (int i = 2; i < len - 2; i++)
223     ret[i] /= 5;
224   ret[1] /= 4, ret[len - 2] /= 4;
225   ret[0] /= 3, ret[len - 1] /= 3;
226   return { ret,len };
227 }
228
229 void test_time()
230 {
231   Test tests[] = { Test(1000, 1800, 50000), Test(10000, 18000, 5000),
232     Test(100000, 180000, 500),Test(1000000, 1800000, 50) };
233   std::vector<float> t1, t2;
234
235   for (Test &test : tests) {
236     t1.push_back(test(filter));
237     t2.push_back(test(simd::filter));
238   }
239   printf("  filter_no_simd: ");
240   for (float t : t1)
241     printf("%f ", t);
242   printf("\nfilter_with_simd: ");
243   for (float t : t2)
244     printf("%f ", t);
245   printf("\n");
246
247   t1.clear();
248   t2.clear();
249   std::vector<float> t3;
250   for (Test &test : tests) {
251     t1.push_back(test(stretch));
252     t2.push_back(test(simd::stretch));
253     simd::stretch2_prepare(test.len, test.newlen);
254     t3.push_back(test(simd::stretch2));
255   }
256   printf("   stretch_no_simd: ");
257   for (float t : t1)
258     printf("%f ", t);
259   printf("\nstretch_with_simd1: ");
260   for (float t : t2)
261     printf("%f ", t);
262   printf("\nstretch_with_simd2: ");
263   for (float t : t3)
264     printf("%f ", t);
```

```
265   printf("\n");
266 /*  auto r = simd::filter(test.data, test.len);
267   auto r2 = filter(test.data, test.len);
268   float sum = 0;
269   for (int i = 0; i < test.len; i++)
270     sum += fabs(r.first[i] - r2.first[i]);
271   printf("avgdiff: %f\n", sum / test.len);*/
272 }
273
274 void test_correctness()
275 {
276   float data[] = { 1,2,3,4 };
277   int len = 4, newlen = 7;
278   auto r = simd::stretch(data, len, newlen);
279   auto r2 = stretch(data, len, newlen);
280   for (int i = 0; i < len; i++)
281     printf("%.1f%c", data[i], i == len - 1 ? '\n' : ' ');
282   for (int i = 0; i < newlen; i++)
283     printf("%.1f%c", r.first[i], i == newlen - 1 ? '\n' : ' ');
284   for (int i = 0; i < newlen; i++)
285     printf("%.1f%c", r2.first[i], i == newlen - 1 ? '\n' : ' ');
286 }
287
288 int main()
289 {
290   test_time();
291   /*test_correctness();*/
292
293   return 0;
294 }
```

5.2 x86-AVX 测试代码

AVX 测试代码

```
1  #include <immintrin.h>
2  #include <cstdio>
3  #include <utility>
4  #include <cstring>
5  #include <vector>
6  #include <time.h>
7
8  class Constants {
9  public:
10   static const int MAXN = 3000000;
11   int *numbers;
12   int *ones;
13
```

```
14    Constants()
15    {
16      numbers = new int[MAXN];
17      ones = new int[MAXN];
18      for (int i = 0; i < MAXN; i++)
19        numbers[i] = i, ones[i] = 1;
20    }
21    ~Constants() { delete[] numbers, delete[] ones; }
22  } constants;
23
24  namespace simd {
25
26    std::pair<float *, int> filter(float *data, int len)
27    {
28      float *ret = new float[len + 4];
29      memset(ret, 0, sizeof(float) * (len + 4));
30      ret += 2;
31      int len8 = len / 8 * 8;
32      for (int i = -2; i <= 2; i++)
33      {
34        for (int j = 0; j < len8; j += 8)
35        {
36          __m256 vdata = _mm256_load_ps(data + j);
37          __m256 vret = _mm256_load_ps(ret + j + i);
38          vret = _mm256_add_ps(vret, vdata);
39          _mm256_store_ps(ret + j + i, vret);
40        }
41        for (int j = len8; j < len; j++)
42          ret[j + i] += data[j];
43      }
44      float *tmp = new float[len];
45      int lim = 2 + (len - 4) / 8 * 8;
46      for (int i = 2; i < lim; i += 8)
47      {
48        __m256 vret = _mm256_load_ps(ret + i);
49        vret = _mm256_mul_ps(vret, _mm256_set1_ps(0.2));
50        _mm256_store_ps(tmp + i, vret);
51      }
52      for (int i = lim; i < len - 2; i++)
53        tmp[i] = ret[i] / 5;
54      tmp[1] = ret[1] / 4, tmp[len - 2] = ret[len - 2] / 4;
55      tmp[0] = ret[0] / 3, tmp[len - 1] = ret[len - 1] / 3;
56      delete[](ret - 2);
57      return { tmp,len };
58    }
59
60    float *tmpl, *tmpr;
61    void stretch2_prepare(int len, int newlen)
62    {
```

```
63      float rate = (float)newlen / len;
64      tmpl = new float[newlen];
65      tmpr = new float[newlen];
66      for (int i = 0; i < newlen; i++)
67      {
68        float pos = i / rate;
69        tmpl[i] = pos - (int)pos;
70        tmpr[i] = 1 - tmpl[i];
71      }
72    }
73    std::pair<float *, int> stretch2(float *data, int len, int newlen)
74    {
75      float rate = (float)newlen / len, _rate = 1 / rate;
76      float *newdata = new float[newlen];
77      int newlen8 = newlen / 8 * 8;
78      float tmp[8];
79      for (int i = 0; i < newlen8; i += 8)
80      {
81        __m256 vl = _mm256_load_ps(tmpl + i);
82        __m256 vr = _mm256_load_ps(tmpr + i);
83
84        int l0 = i / rate, l1 = (i + 1) / rate, l2 = (i + 2) / rate, l3 = (i + 3) /
              rate,
85          l4 = (i + 4) / rate, l5 = (i + 5) / rate, l6 = (i + 6) / rate, l7 = (i + 7) /
                rate;
86        tmp[0] = data[l0]; tmp[1] = data[l1];
87        tmp[2] = data[l2]; tmp[3] = data[l3];
88        tmp[4] = data[l4]; tmp[5] = data[l5];
89        tmp[6] = data[l6]; tmp[7] = data[l7];
90        __m256 vdataleft = _mm256_load_ps(tmp);
91        tmp[0] = data[l0 + 1]; tmp[1] = data[l1 + 1];
92        tmp[2] = data[l2 + 1]; tmp[3] = data[l3 + 1];
93        tmp[4] = data[l4 + 1]; tmp[5] = data[l5 + 1];
94        tmp[6] = data[l6 + 1]; tmp[7] = data[l7 + 1];
95        __m256 vdataright = _mm256_load_ps(tmp);
96
97        __m256 vterm1 = _mm256_mul_ps(vdataleft, vr);
98        __m256 vterm2 = _mm256_fmadd_ps(vdataright, vl, vterm1);
99        _mm256_store_ps(newdata + i, vterm2);
100     }
101     for (int i = newlen8; i < newlen; i++)  //下面处理边界情形
102     {
103       float pos = i / rate;
104       int left = (int)pos, right = left + 1;
105       newdata[i] = data[left] * (right - pos) + data[right] * (pos - left);
106     }
107     newdata[newlen - 1] = data[len - 1];
108     return { newdata, newlen };
109   }
```

```
110
111 }
112
113 class Test {
114 public:
115   float *data;
116   int len;
117   int newlen;
118   int times;
119   Test(int n, int m, int times)
120   {
121     len = n;
122     newlen = m;
123     this->times = times;
124     data = new float[n];
125     for (int i = 0; i < n; i++)
126       data[i] = sin(i);
127   }
128   ~Test() { delete[] data; }
129
130   double operator () (std::pair<float *, int>(*f)(float *, int, int))
131   {
132     long long st = clock();
133     for (int i = 0; i < times; i++)
134       delete[] f(data, len, newlen).first;
135     return (1.0 * clock() - st) / CLOCKS_PER_SEC;
136   }
137   double operator () (std::pair<float *, int>(*f)(float *, int))
138   {
139     long long st = clock();
140     for (int i = 0; i < times; i++)
141       delete[] f(data, len).first;
142     return (1.0 * clock() - st) / CLOCKS_PER_SEC;
143   }
144 };
145
146 std::pair<float *, int> stretch(float *data, int len, int newlen)
147 {
148   float rate = (float)newlen / len;
149   float *newdata = new float[newlen];
150   for (int i = 0; i < newlen; i++)
151   {
152     float pos = i / rate;
153     int left = (int)pos, right = left + 1;
154     newdata[i] = data[left] * (right - pos) + data[right] * (pos - left);
155   }
156   return { newdata, newlen };
157 }
158
```

```
159 std::pair<float *, int> filter(float *data, int len)
160 {
161   float *ret = new float[len];
162   memset(ret, 0, sizeof(float) * len);
163   for (int i = -2; i <= 2; i++)
164     for (int j = 0; j < len; j++)
165       if (j + i >= 0 && j + i < len)
166         ret[j + i] += data[j];
167   for (int i = 2; i < len - 2; i++)
168     ret[i] /= 5;
169   ret[1] /= 4, ret[len - 2] /= 4;
170   ret[0] /= 3, ret[len - 1] /= 3;
171   return { ret,len };
172 }
173
174 void test_time()
175 {
176   Test tests[] = { Test(1000, 1800, 50000), Test(10000, 18000, 5000),
177     Test(100000, 180000, 500),Test(1000000, 1800000, 50) };
178   std::vector<float> t1, t2;
179
180   for (Test &test : tests) {
181     t1.push_back(test(filter));
182     t2.push_back(test(simd::filter));
183   }
184   printf("  filter_no_simd: ");
185   for (float t : t1)
186     printf("%f ", t);
187   printf("\nfilter_with_simd: ");
188   for (float t : t2)
189     printf("%f ", t);
190   printf("\n");
191
192   t1.clear();
193   t2.clear();
194   std::vector<float> t3;
195   for (Test &test : tests) {
196     t1.push_back(test(stretch));
197     simd::stretch2_prepare(test.len, test.newlen);
198     t3.push_back(test(simd::stretch2));
199   }
200   printf("   stretch_no_simd: ");
201   for (float t : t1)
202     printf("%f ", t);
203   printf("\nstretch_with_simd2: ");
204   for (float t : t3)
205     printf("%f ", t);
206   printf("\n");
207 }
```

```
208
209 void test_correctness()
210 {
211   float data[] = { 1,2,3,4 };
212   int len = 4, newlen = 7;
213   simd::stretch2_prepare(len, newlen);
214   auto r = simd::stretch2(data, len, newlen);
215   auto r2 = stretch(data, len, newlen);
216   for (int i = 0; i < len; i++)
217     printf("%.1f%c", data[i], i == len - 1 ? '\n' : ' ');
218   for (int i = 0; i < newlen; i++)
219     printf("%.1f%c", r.first[i], i == newlen - 1 ? '\n' : ' ');
220   for (int i = 0; i < newlen; i++)
221     printf("%.1f%c", r2.first[i], i == newlen - 1 ? '\n' : ' ');
222 }
223
224 int main()
225 {
226   test_time();
227   //test_correctness();
228
229   return 0;
230 }
```