南開大學
Nankai University

南　开　大　学

计　算　机　学　院

三角形光栅化反走样算法的并行化

姓名：文静静

学号：1811507

年级：2018级

专业：计算机科学与技术

指导教师：王刚

时间：2021年7月4日

摘 要

对图片进行三角形光栅化时，容易出现锯齿、摩尔纹等信号走样的问题，针对三角形光栅化走样的问题，可以通过增加采样率以及抗锯齿反走样的方法来改善。反走样通常可以通过低通滤波和超采样反采样 (SSAA) 或多重采样反走样 (MSAA) 实现，而随着规模增大，使用快速傅里叶变换、频域乘积和逆傅里叶变换以及多重采样反走样 (MSAA) 等算法都有巨大的矩阵和向量运算量，所花费的运行时间也会很长。因此借助反走样算法天然的并行性，本文采用不同的并行策略以加速算法进行时间性能和加速比的对比，探索并行化对三角形光栅化反采样问题的改善。

关键词： 三角形光栅化，反走样，快速傅里叶变换，高斯滤波，逆傅里叶变换，MSAA，并行化

目录

1 绪论

1.1 问题背景

我们在光栅图形显示器上绘制非水平、非垂直的直线或多边形边界时，会呈现锯齿状外观。这是因为直线和多边形的边界是连续的，而光栅则是由离散的点组成。用离散量表示连续的量而引起的失真，称为走样或称为混淆，比如出现锯齿状边界、摩尔纹、车轮效应等等[1]。

走样瑕疵的本质是信号改变的太快而采样太慢。因此针对走样可以有两种改善方法。第一种是通过增加采样率。但是这并不是反走样的思路，为了达到提升采样的频率，可以通过使用高分辨率的仪器，但是这种方法可能不太现实，比如显示器不可能为了采样率做到很高的分辨率，而且为了达到一定的效果，可能需要非常高的分辨率，因此可能需要非常高昂的花费。第二种方法就是反走样算法。典型的反走样算法就是先通过低通滤波器做模糊，之后再进行采样。因为一张图可以看作是由低频、高频、和其他频段的信号组成的，高频信息指的就是那些边界，在频域上使用低通滤波将器可以得到模糊的图像从而改善锯齿问题。

对本次期末研究，我们小组打算研究如何对反走样算法进行并行化加速。我们选择的实现两种反走样算法：1.低通滤波：快速傅里叶变换、频域乘积、逆快速傅里叶变换；2.多重采样超采样。本人主要负责逆快速傅里叶变换和多重采样反走样 (MSAA) 的并行化。

1.2 研究现状

针对 FFT 的并行化处理，可以通过将图像进行分块并行多处理机处理，随着节点数的增加，加速比会不断增大，当处理机达到4个时，并行效率达到最大，之后处理节点增加并行效果并不明显[2]；也可以在多处理机中实现流水线算法、FFT算法的并行化(二元交换算法)、快速傅里叶变换、基本的主从实现等算法，以此解决傅里叶变换和快速傅里叶变换中 N 取较大值时所产生的顺序复杂性，进而使多处理机系统中多个处理机间更加协调地工作，更加有效地利用 CPU[3]。作为频域图像处理中最重要的核心算法之一，

对 FFT 和 IFFT 进行并行化是影响数字图像处理软件系统整体效率的关键。有一种适于 SIMD 计算模式的自然顺序二维 FFT 算法，利用 Intel 处理器提供的新指令对算法进行了改进。同时应用 OpenMP 对算法进行了多核环境下的优化，并设计了与之配套的滚动型缓冲区。这种 FFT 算法在多核下的运行效率最高可达到目前广泛使用的 FFT 算法的4.5倍，对海量图像数据的处理优势尤为显著[4]。

三角形光栅化的性能是决定图形处理器性能的一个重要因素，传统的三角形光栅化算法需要处理大量的无关像素，会降低处理速度。针对采用瓦片渲染的嵌入式 GPU，采用超级采样中的反走样算法，只用加法和移位实现了 RGSS 反走样算法。用 C++ 语言实现了所提出的三角形光栅化及反走样算法，该算法可完成对任意三角形的渲染，抗锯齿效果明显[5]。但是针对普遍情况的三角形光栅化反走样，此种方法并不适用。从三角形光栅化单元结构出发，构建一种非阻塞并行三角形光栅化单元结构，将三角形覆盖的像素 Tile 分为完全在内 Tile 和部分在内 Tile 两种类型，并分别设置两条并行的处理通道，能够在保持三角形 Tile 输出顺序的前提下，实现两类 Tile 的非阻塞并行扫描，提升资源利用率，三角形处理能力和像素生成能力,尤其对于较大三角形图元来说更为有效[6]。但是这种方法的三角形光栅化单元结构需要7个功能流水级，略微复杂，并不实用。三角形光栅化作为图形处理的关键环节，其并行化处理并不普遍成熟实用。当前光栅化插值扫描主要依靠单扫描线方法，然而随着硬件技术的发展，当前单扫描线方法已无法充分利用高速并行的硬件资源，极大限制图形处理速度。因此本次实验将具体研究多线程扫描方法，提升三角形光栅化反走样的性能。

2 串行算法

2.1 低通滤波

低通滤波可以简单的认为：设定一个频率点，当信号频率高于这个频率时不能通过，在数字信号中，这个频率点也就是截止频率，当频域高于这个截止频率时，则全部赋值为0。因为在这一处理过程中，让低频信号全部通过，所以称为低通滤波。

低通过滤的概念存在于各种不同的领域，诸如电子电路，数据平滑，声学阻挡，图像模糊等领域经常会用到。

在数字图像处理领域，从频域看，低通滤波可以对图像进行平滑去噪处理。

那么对三角形模糊处理具体是在频域上使用低通滤波将器乘以这个三角形的频域信号即可，也就是模糊操作，那对应到时域上我们可以使用一个像素大小的卷积核对单个像素做卷积操作。时域（spatial domain）上的卷积等价于频域（frequency domain）上的乘积；时域上的乘积等价于频域上的卷积；我们先把图片和卷积核通过傅里叶变换转化到频域上，然后在频域上与卷积核进行傅里叶变换的频域做乘积，最后将结果做逆傅里叶变换即可，即完成图片的模糊操作[7]。

2.1.1 快速傅里叶变换

FFT 算法是频域图像处理中最重要的核心算法之一，是影响数字图像处理软件系统整体效率的关键。

```
void fft(int width, int height, Vector3f** fxRealTwo, Vector3f**
   fxImagTwo, Vector3f** RealTwo)
{
    cout << width << " " << height << endl;
    for (int v = 0; v < width; v++)
    {
        for (int u = 0; u < height; u++)
        {
            fxRealTwo[v][u] = {0, 0, 0};
            fxImagTwo[v][u] = {0, 0, 0};
        }
    }
    for (int v = 0; v < width; v++)
    {
        for (int u = 0; u < height; u++)
        {
            for (int j = 0; j < width; j++)
            {
                for (int i = 0; i < height; i++)
                {
                    float w = 2 * MY_PI * u * i / height + 2 * MY_PI * v
                        * j / width;
                    fxRealTwo[v][u] += RealTwo[j][i] * cos(w);
                    fxImagTwo[v][u] -= RealTwo[j][i] * sin(w);
                }
            }
        }
    }
}
```

我们将应用 OpenMP 对算法进行了多线程的优化，从而得到的算法对海量图像数据的处理优势尤为显著，能将图像从时域转换到频域从而进行乘积操作，同时尝试 MPI 进行多核运算。[4]。

2.1.2 频域上的乘积

当使用傅里叶变换从时域转换到频域后，可以进行模糊操作了。模糊也就是将信号的高频信息过滤掉，去掉高频信号以后，频谱不会再发生堆叠。我们可以将变换后的频域与卷积核进行傅里叶变换的频域相乘，我们在频域的操作实际上就是对原图的频域进行了一个低通滤波。

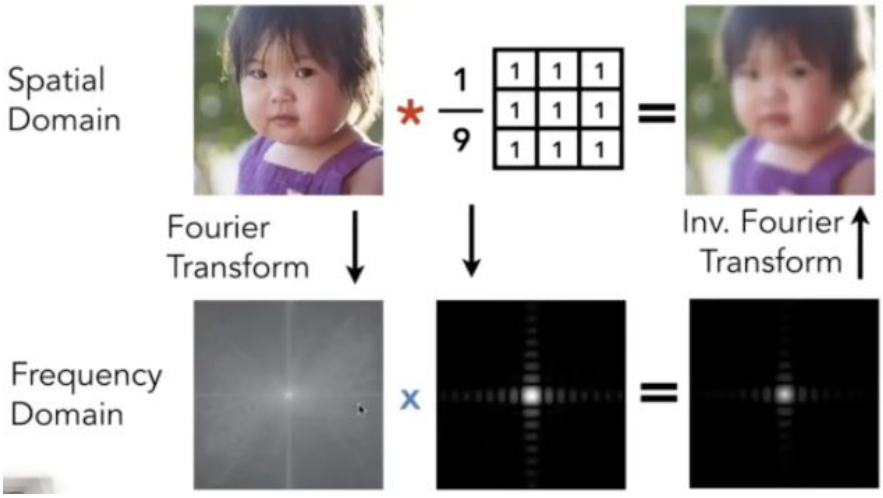
2.1.3 逆快速傅里叶变换

在频域上做完乘积之后，我们需要将结果通过逆傅里叶变换转换回时域进行采样操作。逆傅里叶变换的串行代码如下所示：

```
1  void ifft(int width, int height, Vector3f** ResultReal, Vector3f**
       ResultImage, Vector3f** ifxRealTwo, Vector3f** ifxImageTwo)
2  {
3      cout << width << " " << height << endl;
4      for (int v = 0; v < width; v++)
5      {
6          for (int u = 0; u < height; u++)
7          {
8              ifxRealTwo[v][u] = {0, 0, 0};
9              ifxImageTwo[v][u] = {0, 0, 0};
10         }
11     }
12     for (int v = 0; v < width; v++)
13     {
14         for (int u = 0; u < height; u++)
15         {
16
17             for (int j =0; j < width; j++)
18             {
19                 for (int i = 0; i < height; i++)
20                 {
21                     float w = 2 * MY_PI * u * i / height + 2 * MY_PI * v
                           * j / width;
22                     ifxRealTwo[v][u] += ResultReal[j][i] * cos(w) -
                           ResultImage[j][i] * sin(w);
```

```
                    ifxImageTwo[v][u] += ResultImage[j][i] * cos(w) +
                        ResultReal[j][i] * sin(w);
                }
            }
            ifxRealTwo[v][u] /= (width * height);
            ifxImageTwo[v][u] /= (width * height);
        }
    }
}
```

通过逆傅里叶变换的操作，我们简化了时域的卷积，实现了模糊操作，如下图所示：



2.2 MSAA

MSAA 把一个像素点划分成4个 child-points。然后通过 child-point 覆盖的数量来计算对应的颜色 (c, d)[8]。串行算法如下：

```
// 格子里的细分四个小点坐标
std::vector<Eigen::Vector2f> pos
{
    {0.25,0.25},
    {0.75,0.25},
    {0.25,0.75},
    {0.75,0.75},
};
```

```
for (int x = min_x; x <= max_x; x++) {
    for (int y = min_y; y <= max_y; y++) {
        // 记录最小深度
        float minDepth = FLT_MAX;
        // 四个小点中落入三角形中的点的个数
        int count = 0;
        // 对四个小点坐标进行判断
        for (int i = 0; i < 4; i++) {
            // 小点是否在三角形内
            if (insideTriangle((float)x + pos[i][0], (float)y + pos[i
                ][1], t.v)) {
                // 如果在，对深度进行插值z
                auto tup = computeBarycentric2D((float)x + pos[i][0], (
                    float)y + pos[i][1], t.v);
                float alpha;
                float beta;
                float gamma;
                std::tie(alpha, beta, gamma) = tup;
                float w_reciprocal = 1.0 / (alpha / v[0].w() + beta / v
                    [1].w() + gamma / v[2].w());
                float z_interpolated = alpha * v[0].z() / v[0].w() +
                    beta * v[1].z() / v[1].w() + gamma * v[2].z() / v[2].
                    w();
                z_interpolated *= w_reciprocal;
                minDepth = std::min(minDepth, z_interpolated);
                count++;
            }
        }
        if (count != 0) {
            if (depth_buf[get_index(x, y)] > minDepth) {
                Vector3f color = t.getColor() * count / 4.0;
                Vector3f point(3);
                point << (float)x, (float)y, minDepth;
                // 替换深度
                depth_buf[get_index(x, y)] = minDepth;
                // 修改颜色
                set_pixel(point, color);
            }
```

```
42              }
43          }
44      }
```

因此在串行算法中 MSAA 的改进使得计算量增加了4倍。

3 并行策略

关于快速傅里叶变换和卷积核进行傅里叶变换在频域上做乘积相关的并行化由队友完成。

本人将实现逆快速傅里叶变换和 MSAA 的并行化处理。

3.1 IFFT

3.1.1 结合 SIMD

关于 FFT 变换和 IFFT 变换，有很多参考的代码，特别是对于长度为2的整数次幂的序列，实现起来也是非常简易的，而对于非2次幂的序列，就稍微有点麻烦了，FFTW 中是可以实现任意长度FFT的，而Opencv则有选择性的实现了某些长度序列的变换，查看 Opencv 的代码，发现其只有对是 4 的整数次幂的数据部分采用了 SSE 优化，比如 4、16、64、256、1024 这样的序列部分，因此基 4 的 FFT 是最快的，而剩余的部分则依旧是普通的 C 语言实现。

对于 2 维的 FFT 变换，我直接先每行进行一维的 FFT，然后对结果在进行列方向的一维 FFT，由于一维的 FFT 算法需要处理的序列必须是连续的内存，因此，需要对中间的结果进行转置，处理完后在转置回来，为了节省时间，这个转置也应该用 SSE 优化。当 2 维的宽度和高度相同时，这个转置是不需要分配另外一份额外的内存的，这个叫 In-Place 转置，另外一个重要优点就是一维的 FFT 算法也支持 In-Place 操作。由于是对 Complex 数据进行转置，一个 Complex 正好和 double 数据占用同样的内存，这样直接利用和 double 相关的 SSE 指令就可以方便的实现 Complex 相关数据的转置。我们本次实验默认高度和宽度一致，因此可以直接如下优化：

```
inline void Inplace_TransposeSSE2X2D(double *Src, double *Dest, int
    Length)
{
    __m128d S0 = _mm_loadu_pd(Src);
    __m128d S1 = _mm_loadu_pd((Src + Length));
    __m128d D0 = _mm_loadu_pd(Dest);
    __m128d D1 = _mm_loadu_pd((Dest + Length));
    _mm_storeu_pd(Dest, _mm_unpacklo_pd(S0, S1));
    _mm_storeu_pd(Dest + Length, _mm_unpackhi_pd(S0, S1));
    _mm_storeu_pd(Src, _mm_unpacklo_pd(D0, D1));
    _mm_storeu_pd(Src + Length, _mm_unpackhi_pd(D0, D1));
}
```

最后得到的代码如下：

```
int IM_FFTConv2(unsigned char *Src, unsigned char *Dest, int Width, int
    Height, int Stride, float *Kernel, int KerWidth, int KerHeight)
{
    int Channel = Stride / Width;
    if ((Src == NULL) || (Dest == NULL)) return IM_STATUS_NULLREFRENCE;
    if ((Width <= 0) || (Height <= 0)) return IM_STATUS_INVALIDPARAMETER
        ;
    // 卷积核越大，每次的有效计算量就越小了
    if ((KerWidth >50) || (KerHeight > 50)) return
        IM_STATUS_INVALIDPARAMETER;
    if ((Channel != 1) && (Channel != 3) && (Channel != 4)) return
        IM_STATUS_INVALIDPARAMETER;

    int Status = IM_STATUS_OK;

    const int TileWidth = 256, TileHeight = 256;

    int HalfW = KerWidth / 2, HalfH = KerHeight / 2;
    // 第一个加是因为卷积扩展的矩阵大小为1N+M第二个是因为为了取得有效数据，还要对边缘
        进行扩展，扩展的大小为−1,−1KerHeight − 比如1,KerHeight，则每边需要扩展个
        像素，一共扩展个像素=524
    int ValidW = TileWidth + 1 - KerWidth - (KerWidth - 1);
    // 默认的卷积的效果再卷积周边是用来填充的，如果分块处理时，这明显是不能满足要求的，
        会带来明显的块于块之间的分界线0
    int ValidH = TileHeight + 1 - KerHeight - (KerHeight - 1);
```

```
19      // 图像需要分成的块数
20      int TileAmountX = (Width / ValidW) + (Width % ValidW ? 1 : 0);
21      int TileAmountY = (Height / ValidH) + (Height % ValidH ? 1 : 0);
22      // 需要将卷积核扩展到和大小TileWidthTileHeight
23      Complex *Conv = (Complex *)malloc(TileWidth * TileHeight * sizeof(
            Complex));
24      // 每个小块对应的数据，当为位模式时需要份内存，灰度只需一份243
25      Complex *Tile = (Complex *)malloc(TileWidth * TileHeight * sizeof(
            Complex) * 3);
26      // 每个小块取样时的坐标偏移，这样在中间的块也可以取到周边合理的只，在边缘处则位镜
            像值
27      int *RowOffset = (int *)malloc((TileAmountX * ValidW + KerWidth) *
            sizeof(int));
28      int *ColOffset = (int *)malloc((TileAmountY * ValidH + KerHeight) *
            sizeof(int));
29
30      if ((Conv == NULL) || (Tile == NULL) || (RowOffset == NULL) || (
            ColOffset == NULL))
31      {
32          Status = IM_STATUS_OUTOFMEMORY;
33          goto FreeMemory;
34      }
35      // 左右对称
36      Status = IM_GetOffsetPos(RowOffset, Width, HalfW, TileAmountX *
            ValidW + (KerWidth - HalfW) - Width, IM_EDGE_MIRROR);
37      if (Status != IM_STATUS_OK) goto FreeMemory;
38      Status = IM_GetOffsetPos(ColOffset, Height, HalfH, TileAmountY *
            ValidH + (KerHeight - HalfH) - Height, IM_EDGE_MIRROR);
39      if (Status != IM_STATUS_OK) goto FreeMemory;
40      // 卷积核的其他元素都为，这里先整体赋值为00
41      memset(Conv, 0, TileWidth * TileHeight * sizeof(Complex));
42      for (int Y = 0; Y < KerHeight; Y++)
43      {
44          int Index = Y * KerWidth;
45          for (int X = 0; X < KerWidth; X++)
46          {
47              // 卷积核需要放置在左上角
48              Conv[Y * TileWidth + X].Real = Kernel[Index + X];
```

```
49          }
50      }
51      // 对卷积核进行变换，注意行方向上下部都为，这样可以节省部分计算时间FFT0
52      Status = FFT2D(Conv, Conv, TileWidth, TileHeight, false, 0,
            TileHeight - KerHeight);
53      if (Status != IM_STATUS_OK)     goto FreeMemory;
54      // 单通道时可以一次性处理个块2
55      if (Channel == 1)
56      {
57
58      }
59      else if (Channel == 4)
60      {
61          Complex *TileBG = Tile, *TileRA = Tile + TileWidth * TileHeight;
62          for (int TileY = 0; TileY < TileAmountY; TileY++)
63          {
64              for (int TileX = 0; TileX < TileAmountX; TileX++)
65              {
66                  IM_Rectangle SrcR, ValidR;
67                  IM_SetRect(&SrcR, TileX * ValidW, TileY * ValidH, TileX
                        * ValidW + ValidW, TileY * ValidH + ValidH);
68                  IM_SetRect(&ValidR, 0, 0, Width, Height);
69                  IM_IntersectRect(&ValidR, ValidR, SrcR);
70                  for (int Y = ValidR.Top; Y < ValidR.Bottom + KerHeight -
                        1; Y++)
71                  {
72                      byte *LinePS = Src + ColOffset[Y] * Stride;
73                      int Index = (Y - ValidR.Top) * TileWidth;
74                      for (int X = ValidR.Left; X < ValidR.Right +
                            KerWidth - 1; X++)
75                      {
76                          byte *Sample = LinePS + RowOffset[X] * 4;
77                          TileBG[Index].Real = Sample[0];
78                          TileBG[Index].Imag = Sample[1];
79                          TileRA[Index].Real = Sample[2];
80                          TileRA[Index].Imag = Sample[3];
81                          Index++;
82                      }
```

```
83                  }
84                  Status = FFT2D(TileBG, TileBG, TileWidth, TileHeight,
                        false, 0, TileHeight - (ValidR.Bottom + KerHeight - 1
                        - ValidR.Top));
85                  if (Status != IM_STATUS_OK)     goto FreeMemory;
86
87                  Status = FFT2D(TileRA, TileRA, TileWidth, TileHeight,
                        false, 0, TileHeight - (ValidR.Bottom + KerHeight - 1
                        - ValidR.Top));
88                  if (Status != IM_STATUS_OK)     goto FreeMemory;
89
90                  for (int Y = 0; Y < TileWidth * TileHeight; Y++)
91                  {
92                      float Temp = TileBG[Y].Real;
93                      TileBG[Y].Real = TileBG[Y].Real * Conv[Y].Real -
                            TileBG[Y].Imag * Conv[Y].Imag;
94                      TileBG[Y].Imag = Temp * Conv[Y].Imag + TileBG[Y].
                            Imag * Conv[Y].Real;
95                      Temp = TileRA[Y].Real;
96                      TileRA[Y].Real = TileRA[Y].Real * Conv[Y].Real -
                            TileRA[Y].Imag * Conv[Y].Imag;
97                      TileRA[Y].Imag = Temp * Conv[Y].Imag + TileRA[Y].
                            Imag * Conv[Y].Real;
98                  }
99                  FFT2D(TileBG, TileBG, TileWidth, TileHeight, true);
100                 if (Status != IM_STATUS_OK)     goto FreeMemory;
101
102                 FFT2D(TileRA, TileRA, TileWidth, TileHeight, true);
103                 if (Status != IM_STATUS_OK)     goto FreeMemory;
104
105                 for (int Y = ValidR.Top; Y < ValidR.Bottom; Y++)
106                 {
107                     byte *LinePD = Dest + Y * Stride + ValidR.Left * 4;
108                     int Index = (Y - ValidR.Top + KerHeight - 1) *
                            TileWidth + KerWidth - 1;
109                     for (int X = ValidR.Left; X < ValidR.Right; X++)
110                     {
111                         LinePD[0] = IM_ClampToByte(TileBG[Index].Real);
```

```
112                             LinePD[1] = IM_ClampToByte(TileBG[Index].Imag);
113                             LinePD[2] = IM_ClampToByte(TileRA[Index].Real);
114                             LinePD[3] = IM_ClampToByte(TileRA[Index].Imag);
115                             LinePD += 4;
116                             Index++;
117                         }
118                     }
119                 }
120             }
121         }
122         FreeMemory:
123         if (RowOffset != NULL) free(RowOffset);
124         if (ColOffset != NULL) free(ColOffset);
125         if (Conv != NULL) free(Conv);
126         if (Tile != NULL) free(Tile);
127         return Status;
128     }
```

数据源采用了 Complex 格式，这样数据的实部和虚部在内存中是连续的，用 SSE 加载数据是也就很方便了。但是由于本次我们采用的 Eigen 中的 Vector3f 存储每个像素点的 RGB 颜色变换，相邻像素点的相同颜色通道并不是相邻存储，因此这种存储方式并不适用于 SIMD 的并行化。而且根据对 Opencv 的探究，我们发现其对 SSE 的优化并不重视。从 1998 年 CPU 加入 3D 图形相关的指令，12年来 CPU 的 SIMD 性能也没有达到同时代的显卡，因此在我们的实验基础上， SIMD 的优化是不适用的。

3.1.2 结合 OpenMp

在 IFFT 计算过程中将循环对图像每行和每列的数据依次进行计算，而在不同行列之间的计算过程是彼此独立的。各循环中的任意一次迭代不依赖于其他迭代的结果，执行循环时不存在循环依赖，所以可以将整个计算过程并行处理。如果用多线程处理方式，可以产生两个以上的线程“同时”计算，能大大提升计算的性能和效率。OpenMP 是用以编写可移植的多线程应用程序的 API 库，内部使用 Windows 线程池，可以很好的支持多线程的并行程

序设计。

```
void ifft()
{
    # pragma omp parallel for num_threads(THREAD_NUM)
    for (int v = 0; v < width; v++)
    {
        for (int u = 0; u < height; u++)
        {
            for (int j = 0; j < width; j++)
            {
                for (int i = 0; i < height; i++)
                {
                    float w = 2 * MY_PI * (float)u * (float)i / (float)
                        height + 2 * MY_PI * (float)v * (float)j / (float
                        )width;
                    ifxRealTwo[v][u] += fxRealTwo[j][i] * cos(w) -
                        fxImageTwo[j][i] * sin(w);
                    ifxImageTwo[v][u] += fxImageTwo[j][i] * cos(w) +
                        fxRealTwo[j][i] * sin(w);
                }
            }
            ifxRealTwo[v][u] /= (width * height);
            ifxImageTwo[v][u] /= (width * height);
        }
    }
}
```

3.1.3 结合 MPI

如果用单核多线程处理方式，这样得到的效率提升比较有限，因为这本质上是对算法的并行化模拟，实际上还是单个处理器串行化完成计算任务，而由于处理器还需要对多个线程进行调度和维护，反而影响了计算任务本身的效率。如果针对多核处理器对程序进行改进，则系统将把整个计算任务分配给多个物理核心共同分担，而又因为不存在循环依赖，核心之间也不需要大规模的通信，所以各核心可以独立的完成计算任务，性能提升将会更好，

因此我还尝试结合 MPI 进行多核计算。

由于结合 MPI 需要在服务器上运行，而整体光栅化模型部署到服务器上比较麻烦，需要下载多个图形学相关的库，因此，我将逆傅里叶变换单独进行并行化，探究性能变换并分析，然后选择并行化最优方法处理原模型中的逆傅里叶变换函数，得到光栅化三角形反走样算法的并行化结果，与串行化进行对比。

使用了4个节点，每个节点块划分，节点之间不需要通信，相互独立，只需要最后将处理之后的结果发送给 0 节点。

```
#define MY_PI 3.1415926
#define THREAD_NUM 4
double t;
const int width = 32;
const int height = 32;
float **RealTwo, **fxRealTwo, **fxImageTwo, **ifxRealTwo, **ifxImageTwo;

void init()
{
    RealTwo = new float *[width];
    fxRealTwo = new float *[width];
    fxImageTwo = new float *[width];
    ifxRealTwo = new float *[width];
    ifxImageTwo = new float *[width];
    for (int v = 0; v < width; v++)
    {
        RealTwo[v] = new float[height];
        fxRealTwo[v] = new float[height];
        fxImageTwo[v] = new float[height];
        ifxRealTwo[v] = new float[height];
        ifxImageTwo[v] = new float[height];
        for (int u = 0; u < height; u++)
        {
            RealTwo[v][u] = (rand() % 100 + 1) / 10;
            fxRealTwo[v][u] = 0.0;
            fxImageTwo[v][u] = 0.0;
            ifxRealTwo[v][u] = 0.0;
            ifxImageTwo[v][u] = 0.0;
```

```
29          }
30      }
31  }
32  void fft()
33  {
34      for (int v = 0; v < width; v++)
35      {
36          for (int u = 0; u < height; u++)
37          {
38              for (int j = 0; j < width; j++)
39              {
40                  for (int i = 0; i < height; i++)
41                  {
42                      float w = 2 * MY_PI * (float)u * (float)i / (float)
                            height + 2 * MY_PI * (float)v * (float)j / (float
                            )width;
43                      fxRealTwo[v][u] += RealTwo[j][i] * cos(w);
44                      fxImageTwo[v][u] -= RealTwo[j][i] * sin(w);
45                  }
46              }
47          }
48      }
49  }
50  void compare()
51  {
52      for (int v = 0; v < width; v++)
53      {
54          for (int u = 0; u < height; u++)
55          {
56              if (fabs(ifxRealTwo[v][u] - RealTwo[v][u]) > 1e-2)
57              {
58                  cout << "ifx: " << ifxRealTwo[v][u] << " real: " <<
                        RealTwo[v][u] << endl;
59                  cout << "Wrong!" << endl;
60                  return;
61              }
62          }
63      }
```

```
64      cout << "Correct!" << endl;
65  }
66
67  int main(int argc, char **argv)
68  {
69      // 初始化
70      MPI_Init(&argc, &argv);
71      // rank 当前进程，size 进程数量
72      int rank, size;
73      MPI_Status status;
74      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
75      MPI_Comm_size(MPI_COMM_WORLD, &size);
76      double start_time = 0;
77      double end_time = 0;
78      int n = width / size;
79      float **subReal = new float *[n];
80      float **subImage = new float *[n];
81      for (int i = 0; i < n; i++)
82      {
83          subReal[i] = new float[height];
84          subImage[i] = new float[height];
85      }
86      init();
87      fft();
88      start_time = MPI_Wtime();
89      // 开始计算
90      for (int v = 0; v < width; v++)
91      {
92          if (rank == v / n)
93          {
94              int row = v % n;
95              for (int u = 0; u < height; u++)
96              {
97                  for (int j = 0; j < width; j++)
98                  {
99                      for (int i = 0; i < height; i++)
100                     {
101                         float w = 2 * MY_PI * (float)u * (float)i / (
```

```
                        float)height + 2 * MY_PI * (float)v * (float)
                            j / (float)width;
102                     subReal[row][u] += fxRealTwo[j][i] * cos(w) -
                            fxImageTwo[j][i] * sin(w);
103                     subImage[row][u] += fxImageTwo[j][i] * cos(w) +
                            fxRealTwo[j][i] * sin(w);
104                 }
105             }
106             subReal[row][u] /= (width * height);
107             subImage[row][u] /= (width * height);
108         }
109     }
110 }
111 if (rank == 0)
112 {
113     for (int i = 1; i < size; i++)
114     {
115         for (int j = 0; j < n; j++)
116         {
117             MPI_Recv(&ifxRealTwo[i * n + j][0], height, MPI_FLOAT, i
                    , j, MPI_COMM_WORLD, &status);
118         }
119     }
120     for (int i = 0; i < n; i++)
121     {
122         for (int j = 0; j < height; j++)
123         {
124             ifxRealTwo[i][j] = subReal[i][j];
125         }
126     }
127     end_time = MPI_Wtime();
128     cout << "Running time = " << (end_time - start_time) * 1000 << "
            ms" << endl;
129     compare();
130 }
131 else
132 {
133     for (int i = 0; i < n; i++)
```

```
134             {
135                 MPI_Send(&subReal[i][0], height, MPI_FLOAT, 0, i,
                        MPI_COMM_WORLD);
136             }
137         }
138         MPI_Finalize();
139         return 0;
140     }
```

3.1.4 结合 OpenMp 和 MPI

结合 OpenMp 和 MPI，使用多核多线程，每个节点两个线程：

```
1   // 初始化
2   int provided;
3   MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
4   // rank 当前进程，size 进程数量
5   int rank, size;
6   MPI_Status status;
7   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8   MPI_Comm_size(MPI_COMM_WORLD, &size);
9   double start_time = 0;
10  double end_time = 0;
11  int n = width / size;
12  float **subReal = new float *[n];
13  float **subImage = new float *[n];
14  for (int i = 0; i < n; i++)
15  {
16      subReal[i] = new float[height];
17      subImage[i] = new float[height];
18  }
19  init();
20  fft();
21  start_time = MPI_Wtime();
22  // 开始计算
23  for (int v = 0; v < width; v++)
24  {
25      if (rank == v / n)
```

```
26      {
27          int row = v % n;
28          #pragma omp parallel for num_threads(provided)
29          for (int u = 0; u < height; u++)
30          {
31              for (int j = 0; j < width; j++)
32              {
33                  for (int i = 0; i < height; i++)
34                  {
35                      float w = 2 * MY_PI * (float)u * (float)i / (float)
                            height + 2 * MY_PI * (float)v * (float)j / (float
                            )width;
36                      subReal[row][u] += fxRealTwo[j][i] * cos(w) -
                            fxImageTwo[j][i] * sin(w);
37                      subImage[row][u] += fxImageTwo[j][i] * cos(w) +
                            fxRealTwo[j][i] * sin(w);
38                  }
39              }
40              subReal[row][u] /= (width * height);
41              subImage[row][u] /= (width * height);
42          }
43      }
44  }
45  if (rank == 0)
46  {
47      for (int i = 1; i < size; i++)
48      {
49          for (int j = 0; j < n; j++)
50          {
51              MPI_Recv(&ifxRealTwo[i * n + j][0], height, MPI_FLOAT, i, j,
                    MPI_COMM_WORLD, &status);
52          }
53      }
54      for (int i = 0; i < n; i++)
55      {
56          for (int j = 0; j < height; j++)
57          {
58              ifxRealTwo[i][j] = subReal[i][j];
```

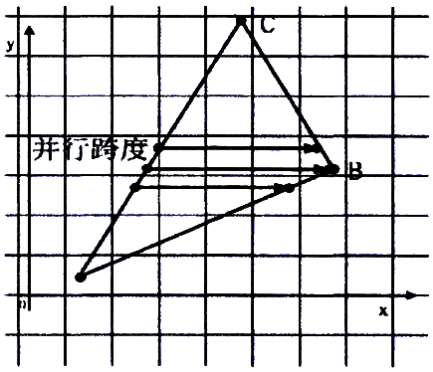
```
59            }
60        }
61        end_time = MPI_Wtime();
62        cout << "Running time = " << (end_time - start_time) * 1000 << " ms"
              << endl;
63        compare();
64    }
65    else
66    {
67        for (int i = 0; i < n; i++)
68        {
69            MPI_Send(&subReal[i][0], height, MPI_FLOAT, 0, i, MPI_COMM_WORLD
                  );
70        }
71    }
72    MPI_Finalize();
```

3.2 MSAA

由于 MSAA 每一个像素需要同时实现4个 child-points 的计算，当像素点比较多的时候，计算繁重，我们使用 OpenMp 通过多线程对像素计算进行并行化优化：

有两种并行策略：

(1)建立多条扫描线程，每个线程扫描一行，如下图：



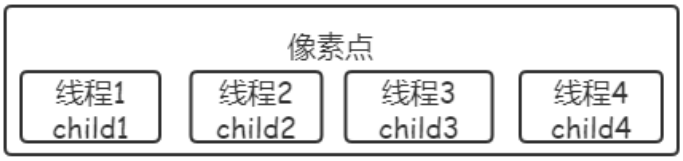

```
1    gettimeofday(&tv_begin, NULL);
2    // 格子里的细分四个小点坐标
```

```
std::vector<Eigen::Vector2f> pos
{
    {0.25,0.25},
    {0.75,0.25},
    {0.25,0.75},
    {0.75,0.75},
};
#pragma omp parallel for num_threads(4)
for (int x = min_x; x <= max_x; x++) {
    for (int y = min_y; y <= max_y; y++) {
        // 记录最小深度
        float minDepth = FLT_MAX;
        // 四个小点中落入三角形中的点的个数
        int count = 0;
        // 对四个小点坐标进行判断
        for (int i = 0; i < 4; i++) {
            // 小点是否在三角形内
            if (insideTriangle((float)x + pos[i][0], (float)y + pos[i
                ][1], t.v)) {
                // 如果在，对深度进行插值z
                auto tup = computeBarycentric2D((float)x + pos[i][0], (
                    float)y + pos[i][1], t.v);
                float alpha;
                float beta;
                float gamma;
                std::tie(alpha, beta, gamma) = tup;
                float w_reciprocal = 1.0 / (alpha / v[0].w() + beta / v
                    [1].w() + gamma / v[2].w());
                float z_interpolated = alpha * v[0].z() / v[0].w() +
                    beta * v[1].z() / v[1].w() + gamma * v[2].z() / v[2].
                    w();
                z_interpolated *= w_reciprocal;
                minDepth = std::min(minDepth, z_interpolated);
                count++;
            }
        }
        if (count != 0) {
            if (depth_buf[get_index(x, y)] > minDepth) {
```

```
                Vector3f color = t.getColor() * count / 4.0;
                Vector3f point(3);
                point << (float)x, (float)y, minDepth;
                // 替换深度
                depth_buf[get_index(x, y)] = minDepth;
                // 修改颜色
                set_pixel(point, color);
            }
        }
    }
}
gettimeofday(&tv_end, NULL);
cout << "MSAA runing time = " << (float)(tv_begin.tv_sec - tv_end.tv_sec
    ) * 1000.0 + (float)(tv_begin.tv_usec - tv_end.tv_usec) / 1000.0 << "
    ms" << endl;
```

(2)建立4个线程，每个线程负责一个像素点中的一个 child-point 的计算：

如下图所示：




```
gettimeofday(&tv_begin, NULL);
// 格子里的细分四个小点坐标
std::vector<Eigen::Vector2f> pos
{
    {0.25,0.25},
    {0.75,0.25},
    {0.25,0.75},
    {0.75,0.75},
};
for (int x = min_x; x <= max_x; x++) {
    for (int y = min_y; y <= max_y; y++) {
        // 记录最小深度
        float minDepth = FLT_MAX;
```

```
14          // 四个小点中落入三角形中的点的个数
15          int count = 0;
16          // 对四个小点坐标进行判断
17          #pragma omp parallel for num_threads(4)
18          for (int i = 0; i < 4; i++) {
19              // 小点是否在三角形内
20              if (insideTriangle((float)x + pos[i][0], (float)y + pos[i
                    ][1], t.v)) {
21                  // 如果在，对深度进行插值z
22                  auto tup = computeBarycentric2D((float)x + pos[i][0], (
                        float)y + pos[i][1], t.v);
23                  float alpha;
24                  float beta;
25                  float gamma;
26                  std::tie(alpha, beta, gamma) = tup;
27                  float w_reciprocal = 1.0 / (alpha / v[0].w() + beta / v
                        [1].w() + gamma / v[2].w());
28                  float z_interpolated = alpha * v[0].z() / v[0].w() +
                        beta * v[1].z() / v[1].w() + gamma * v[2].z() / v[2].
                        w();
29                  z_interpolated *= w_reciprocal;
30                  minDepth = std::min(minDepth, z_interpolated);
31                  count++;
32              }
33          }
34          if (count != 0) {
35              if (depth_buf[get_index(x, y)] > minDepth) {
36                  Vector3f color = t.getColor() * count / 4.0;
37                  Vector3f point(3);
38                  point << (float)x, (float)y, minDepth;
39                  // 替换深度
40                  depth_buf[get_index(x, y)] = minDepth;
41                  // 修改颜色
42                  set_pixel(point, color);
43              }
44          }
45      }
46  }
```

```
47 gettimeofday(&tv_end, NULL);
48 cout << "MSAA runing time = " << (float)(tv_begin.tv_sec - tv_end.tv_sec
   ) * 1000.0 + (float)(tv_begin.tv_usec - tv_end.tv_usec) / 1000.0 << "
    ms" << endl;
```

这两种方法主要区别在于第一种方法中的线程需要完成的计算比第二种方法中的线程要多。第二种方法中的子线程只需要判断像素点是否在三角形内，进行深度插值即可，而第一种方法中的子线程需要处理一行像素的数据，每个像素要计算4个 child point，还要进行深度替换，设置像素颜色等等操作。

本次实验，将结合多线程设计模型，利用并行处理扫描加快像素计算，缩短了 MSAA 处理时间并提高了图形处理效率，同时做到了反走样。

4 运行环境

4.1 三角形光栅化运行环境

4.1.1 环境安装

本次实验是一个完整的三角形光栅化模型，需要运行在 Oracle VM VirtualBox 虚拟机上。

(1)安装虚拟机

如果你使用 Windows 系统，可以直接下载 `https://download.virtualbox.org/virtualbox/6.1.4/VirtualBox-6.1.4-136177-Win.exe`，下载完成后按照指示完成安装。

如果你使用 Mac OS 系统，可以直接下载 `https://download.virtualbox.org/virtualbox/6.1.4/VirtualBox-6.1.4-136177-OSX.dmg`，下载完成后按照指示完成安装。

如果你使用 Linux 内核的系统，你可以查看 `https://www.virtualbox.org/wiki/Linux\_Downloads`，找到你使用的系统，按照对应的指示完成安装。

(2)下载虚拟硬盘

虚拟硬盘文件的下载地址为 https://cloud.tsinghua.edu.cn/f/103133da1bf8451b8ba6，密码为 games101。下载完成后得到 GAMES101_Ubuntu 18.04.2(64bit).rar，将其解压后得到虚拟硬盘文件 GAMES101_Ubuntu 18.04.2 (64bit).vdi。

(3)配置虚拟机

打开 Virtual Box，点击新建，设置类型为 Linux，版本为 Ubuntu-64 bit，建议设置虚拟机的内存大小为 2GB，然后选择使用已有的虚拟硬盘文件，设置为之前解压得到的 GAMES101_Ubuntu 18.04.2 (64bit).vdi，最后点击创建完成虚拟机的配置工作，Ubuntu 系统的密码为 Ilovegraphics。

4.1.2 运行指令

在 **main.cpp** 所在目录下，打开终端 (命令行)，依次输入：

- **mkdir build**: 创建名为 build 的文件夹。
- **cd build**: 移动到 build 文件夹下。
- **cmake** ..: 注意其中’..’ 表示上一级目录，若为’.’ 则表示当前目录。
- **make**: 编译程序，错误提示会显示在终端中。
- **./Rasterizer**：若上一步无错误，则可运行程序。

4.1.3 处理多线程

由于以上虚拟硬盘得到的虚拟机是单核机器，编译器没有将 openMP 设为 Yes，编译也不会报错，将自动忽略 #pragma 这行代码，然后按照传统单核串行的方式编译运行。只需要将 Visual Studio 对应目录下的 vcomp140d.dll 和 vcomp140.dll 拷贝到工程文件目录下即可。

4.2 OpenMp 运行环境

操作系统环境： Windows 10 专业版 (64 位)

处理器： Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz (4 CPUs), 2.5GHz

RAM: 8.0GB

多线程的编译选项：-fopenmp

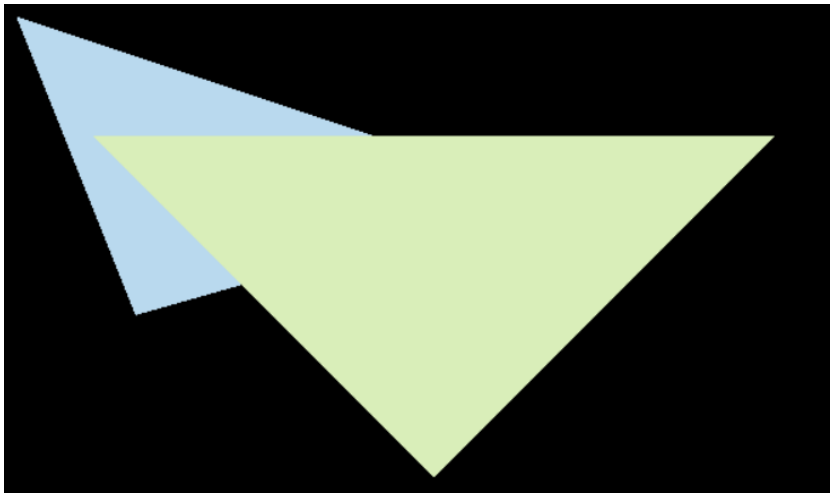
4.3 MPI 运行环境

MPI 编程实验在基于金山云的虚拟机集群上运行，虚拟机集群包括 1 个 master 节点和 32 个计算节点，每个计算节点 2 核，共 64 核可供计算。

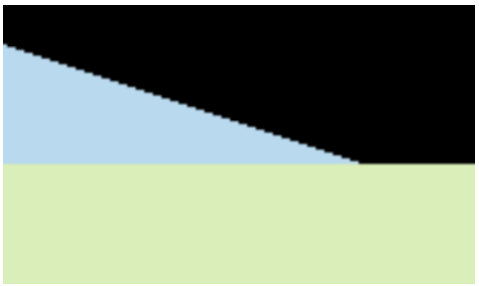
5 结果分析

5.1 反走样效果分析

在反走样之前，两个三角形光栅化的效果如下：



可以看到小三角形的锯齿感是比较明显的：



使用 MSAA 之后，锯齿感已经没有那么明显了：

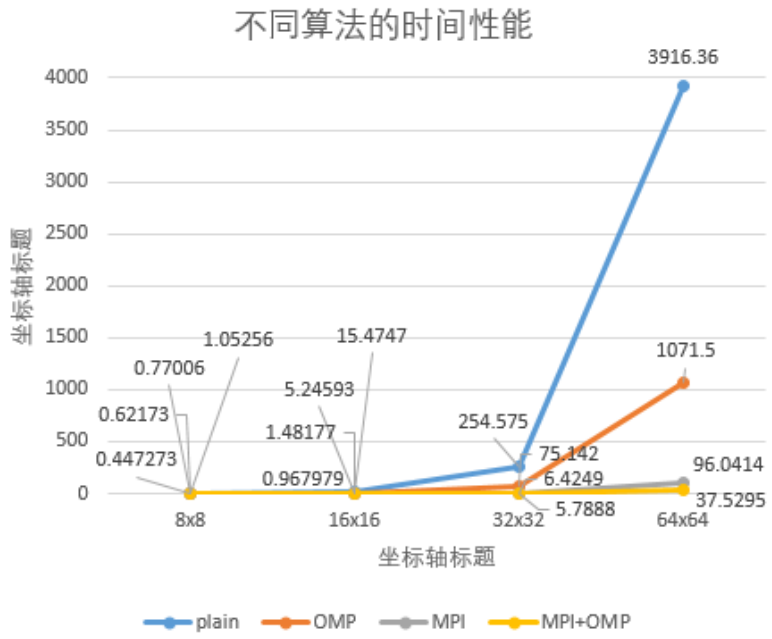
使用低通滤波之后的效果：

可以发现，三角形边缘明显变模糊了，达到了反走样的效果。

5.2 逆傅里叶变换

改变规模的大小，这个规模代表的是光栅化区域的长和宽。测试逆傅里叶变换的时间如下：

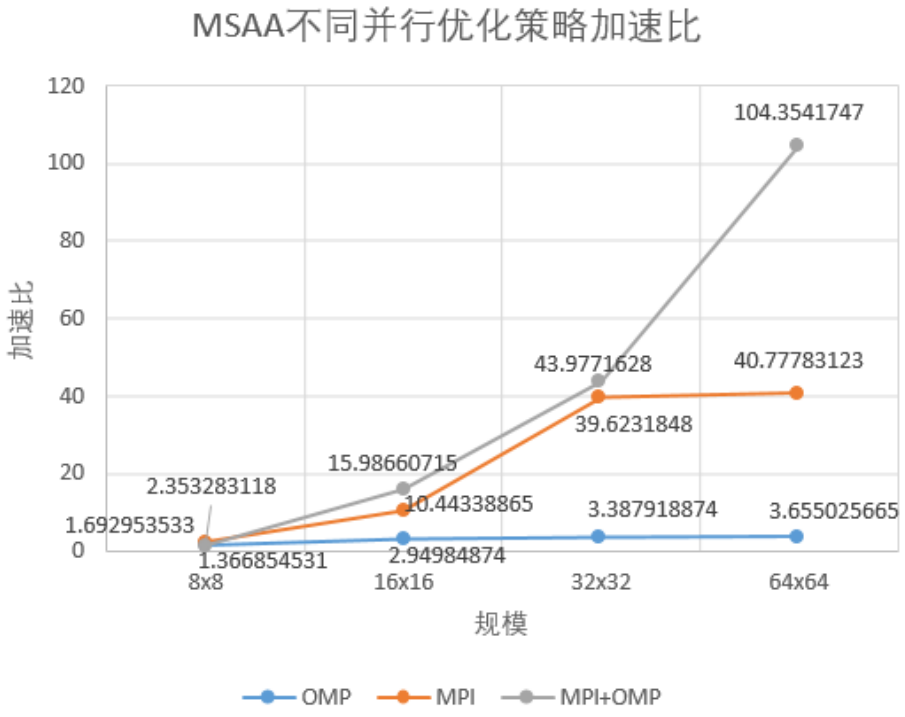
规模	plain	OMP	MPI	MPI+OMP
8x8	1. 05256	0. 62173	0. 447273	0. 77006
16x16	15. 4747	5. 24593	1. 48177	0. 967979
32x32	254. 575	75. 142	6. 4249	5. 7888
64x64	3916. 36	1071. 5	96. 0414	37. 5295



得到 OpenMp、MPI 和 OpenMp 结合 MPI 的加速比如下：

规模	OMP	MPI	MPI+OMP
8x8	1. 6929535	2. 353283	1. 366855
16x16	2. 9498487	10. 44339	15. 98661
32x32	3. 3879189	39. 62318	43. 97716
64x64	3. 6550257	40. 77783	104. 3542

由此得到折线图如下：



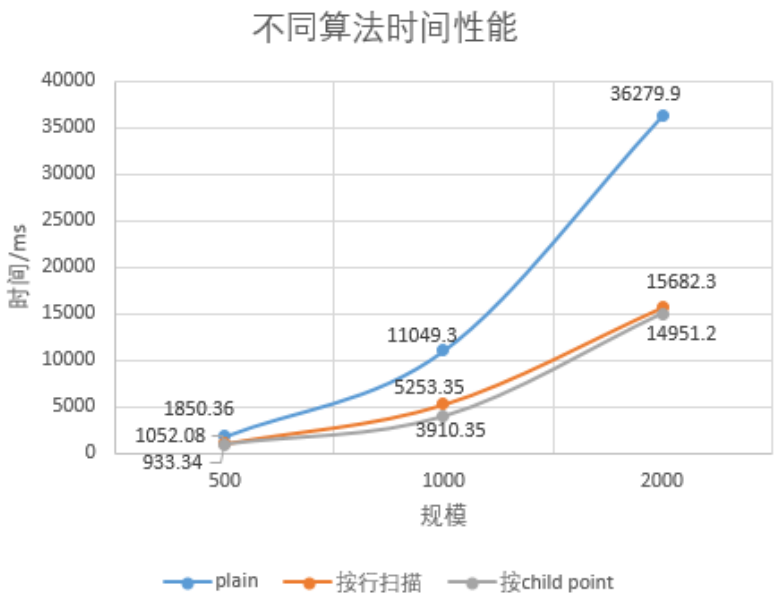
通过折线图可以发现，OpenMp 的并行处理对于性能的改善比较有限，而且每个线程对于每个规模的时间性能加速比基本一致。而 MPI 的并行处理使用了4个节点，每个节点块划分，节点之间不需要通信，相互独立，只需要最后将处理之后的结果发送给 0 节点，使用 MPI 的效果很好，但是当数据规模从 32x32 变成 64x64 之后，加速比没有明显的增长，应该是数据量过大，致使节点计算量过多，加速比没有增长。结合了 OpenMp 之后，尽管数据量增长，一个节点有两个线程，还能迅速处理数据，因此当数据量增长到 64x64 之后，加速比也继续增长。

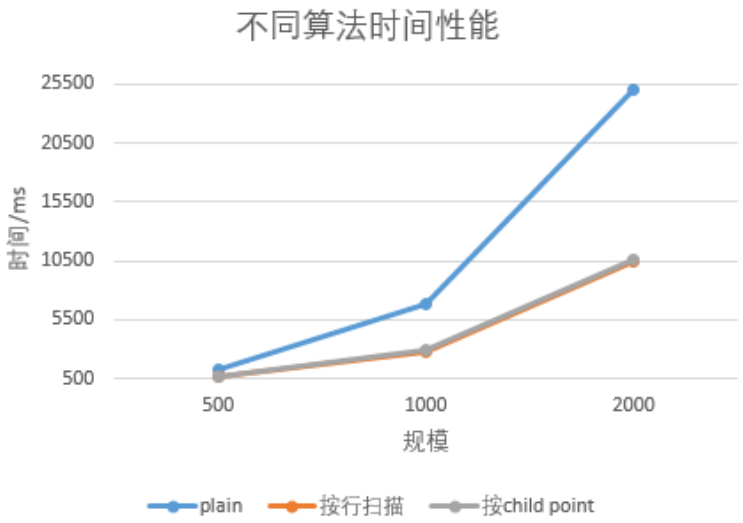
5.3 MSAA

改变规模的大小，测试 MSAA 的时间，十个 frame 取一次平均，得到两个三角形光栅化的时间分别如下：

规模	plain	按行扫描	按child point
500	1850. 36	1052. 08	933. 34
1000	11049. 3	5253. 35	3910. 35
2000	36279. 9	15682. 3	14951. 2

规模	plain	按行扫描	按child point
500	1196. 34	662. 555	654. 13
1000	6750. 72	2830. 47	2912. 35
2000	25065	10409. 65	10509. 55





计算得到加速比和折线图如下：

规模	按行扫描	按child point
500	1.7588	1.9825
1000	2.1033	2.8257
2000	2.3134	2.4266

规模	按行扫描	按child point
500	1.8056	1.8289
1000	2.3850	2.3180
2000	2.4079	2.3850

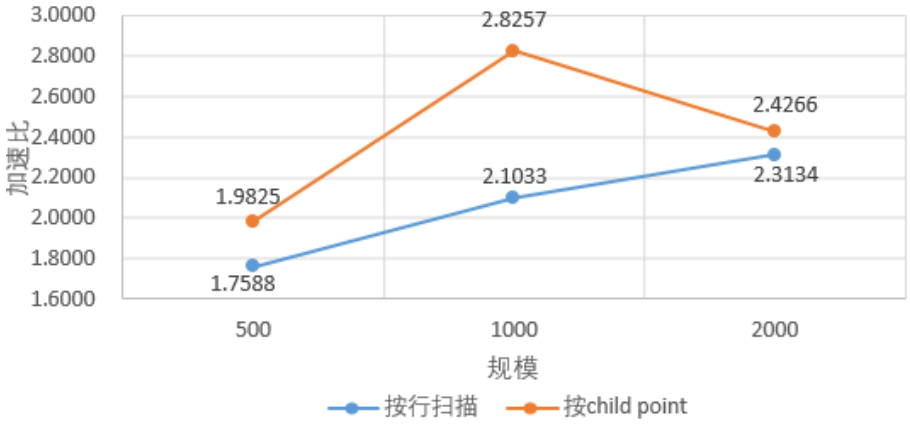


图 1: 大三角形加速比

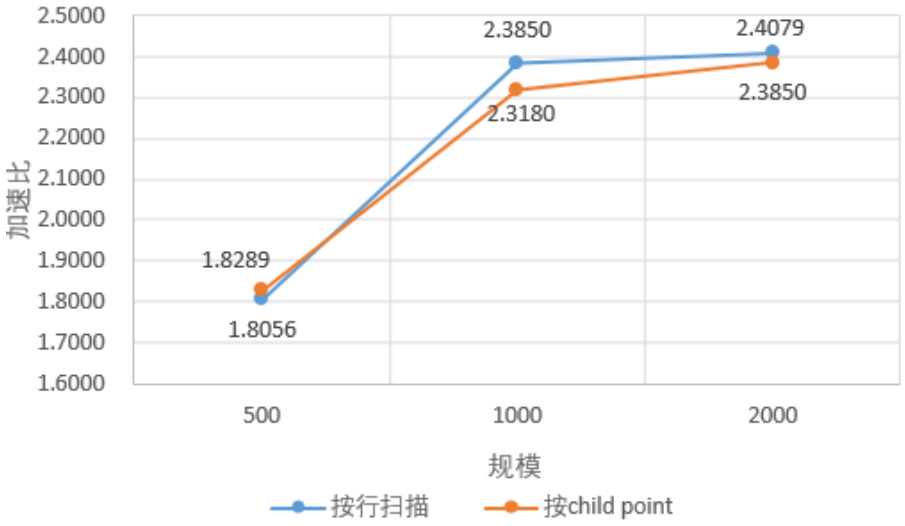


图 2: 小三角形加速比

可以发现，在光栅化小三角形的时候，按行扫描和按 child point 扫描的时间性能相近，而且按 child point 的时间性能相对于按行扫描略差一些，是因为一个 child point 的计算内容并不多，因此四个线程相比于总体，可做的并不多，在按行扫描时，每个线程的计算量比较大，时间性能改善更多，但是当规模增大的时候，按行扫描相对于按 child point 扫描的性能优势显然下降，这是因为当规模变大之后，每一行的位于三角形内的像素个数差别变大，那么负责每一行的线程之间的工作量会差别也会变大，由于使用的是静态调度方式，因此可能会导致出现线程空闲等待的情况，反而影响性能。

在光栅化大三角形的时候，形况类似小三角形规模比较大的情况，因此此时按 child point 扫描的性能优于按行扫描，但是当规模进一步增大的时候，像素变多，加上线程开销，child point 多线程的性能优化对于总体的计算量变小，因此加速比也开始相应下降。

5.4 不同平台影响

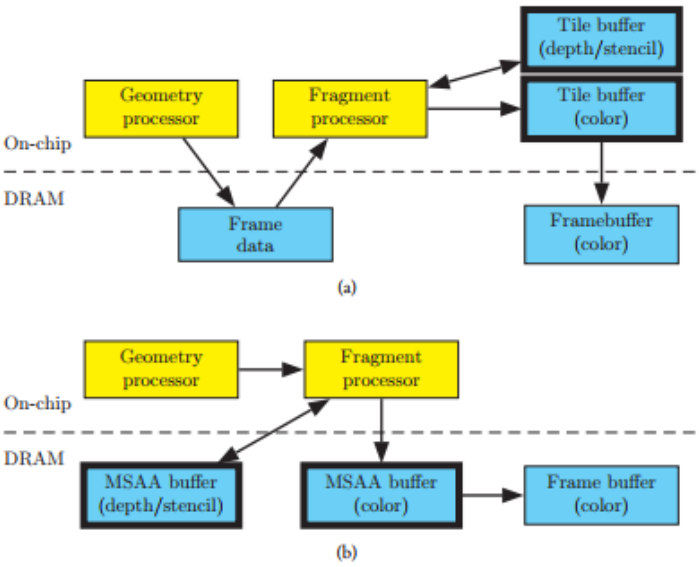
根据探究，在不同 API 版本下，MSAA 对于缓冲区的要求：

架构	是否需要更大的颜色、深度、缓冲区
Direct3D 9/11/12	需要
OpenGL ES 2.0	看GPU架构：TBR(Mali Qualcomm Adreno(300系列之前))、TBDR（PowerVR） 不需要；IMR（nVidia Tera Qualcomm Adreno 300系列以及之后可以在IMR、TBR之间切换）需要。 如果使用 GL_EXT_multisampled_render_to_texture 也需要（跟硬件实现有关（enabling MSAA the right way in OpenGL ES））。
OpenGL ES 3.0/3.1	如果是系统提供的framebuffer,那么同OpenGL ES 2.0的版本。如果是用户创建的framebuffer，那么是需要额外的显存的。

显然，除了增加的计算量会对 MSAA 的性能有影响，需要产生的额外显存也会对 MSAA 的性能产生影响，从之前我们测试的 MSAA 的并行加速比可以看出，使用多线程进行并行处理对于性能的改善有限，最大的加速比也没有超过3。

在 PC 平台上，渲染的方法有 IMR、TBR 和 TBDR，其中 IMR 是立即渲染模式。目前 PC 平台上基本上都是立即渲染模式，CPU 提交渲染数据和渲染命令，GPU 开始执行。TBR 是分块渲染，TBR 把屏幕分成一系列的小块，每个单独来处理，所以可以做到并行。由于在任何时候显卡只需要场景中的一部分数据就可完成工作，这些数据（如颜色、深度等）足够小到可以放在显卡芯片上（on-chip），有效得减少了存取系统内存的次数。它带来的好处就是更少的电量消耗以及更少的带宽消耗，从而会获得更高的性能。TBDR 是分块延迟渲染，TBDR 跟 TBR 有些相似，也是分块，并使用在芯片上的缓存来存储数据（颜色以及深度等），它还使用了延迟技术，叫隐藏面剔除（Hidden Surface Removal），它把纹理以及着色操作延迟到每个像素已经在块中已经确定可见性之后，只有那些最终被看到的像素才消耗处理资源。这意味着隐藏像素的不必要处理被去掉了，这确保了每帧使用最低可能的带宽使用和处理周期数，这样就可以获取更高的性能以及更少的电量消耗。

在移动平台上的 MSAA 的实现如下：



数据流在（a）平铺为基础的和（b）即时模式GPU的多采样渲染。黄色框是计算单位，蓝色框是内存，粗边框表示多采样缓冲区。即时模式 GPU 在芯片上移动多采样像素数据，消耗大量带宽。因此如果相对于IMR模式的显卡来说，TBR或者TBDR的实现MSAA会省很多，因为好多工作直接在on-chip上就完成了。

MSAA 是影响了 GPU 管理的光栅化、片断程序、光栅操作阶段（每个子采样点都要做深度测试）的。每个子采样点都是有自己的颜色和深度存储的，并且每个子采样点都会做深度测试。在移动平台上，是否需要额外的空间来存储颜色和深度需要根据 OpenGL ES 的版本以及具体硬件的实现有关。

MSAA 在一般的情况下（不需要额外空间来存储颜色和深度，直接在 on-chip 上完成子采样点计算，然后直接 resolve 到 framebuffer）是要比 PC 平台上效率高的，因为没有了那么大的带宽消耗。因此我们本次的实现就是直接判断像素点是否处于三角形中，进行深度插值，然后通过 set_pixel 将颜色直接部署到 framebuffer 中的，因此效率还是比较高的。

6 小组分工

1811507 文静静：

1. 逆傅里叶变换并行化程序编写；
2. MSAA 并行化程序编写；
3. 撰写报告。

1811516 余樱童：

1. 傅里叶变换并行化程序编写；
2. 卷积并行化程序编写；
3. 撰写报告。

7 结论

本次实验采用 C++ 语言编写，探究了 FFT 与 IFFT 的 SIMD 优化，但是针对优化的数据特征得到 SIMD 优化的瓶颈，因此对于 FFT 和 IFFT 算法的并行化将通过 OpenMP 和 MPI 来实现，符合算法中不同行列之间的计算过程是彼此独立的特点，能有效完成并行化处理。

对于 MSAA 的并行化，虽然没有特定文献对此并行化进行探究，但是通过多线程对三角形光栅化进行并行化处理是完全可行的，通过 OpenMp 来进行并行优化，同时还探究了不同平台上不同 API 对于 MSAA 并行的处理以及性能的影响，分析影响 MSAA 的性能瓶颈，结合本次使用的方法，进一步提升并行的效率。

参考文献

[1] 杜慧敏, 郝哲, 王鹏超,等. 三角形的光栅化与反走样算法[J]. 西安邮电大学学报, 2015, 20(003):33-38.

[2] 赵晓雷, 王敏, ZHAO,等. 快速傅里叶变换的并行化研究[J]. 渭南师范学院学报, 2011, 12(v.26;No.128):29-32.

[3] 唐俊奇. 多处理机中傅里叶变换的并行算法及实现[J]. 淮海工学院学报(自然科学版), 2006.

[4] 袁泉, 郭子祺, 姚谦,等. 基于并行处理的FFT快速算法[J]. 科学技术与工程, 2008, 8(016):4709-4714.

[5] 刘青楠, 曾泽仓, 杜慧敏,等. 一种易于硬件实现的嵌入式GPU三角形光栅化算法[J]. 微电子学与计算机, 2018, 35(09):32-37.

[6] 张骏, 任向隆, 韩立敏,等. 一种非阻塞并行三角形光栅化单元结构:, CN108009978A[P]. 2018.

[7] 马婷婷, 王静, 朱剑,等. 高速并行信号处理系统及其处理方法:, CN106291501A[P].

[8] 卡斯·W.·埃弗里特, 史蒂文·E.·莫尔纳. 混合多重采样/超采样抗锯齿:, CN101620725B[P]. 2012.

[9] 宋克庆, 黄春. 基于OpenMP快速傅里叶变换并行实现[C]// 2010通信理论与技术新发展——第十五届全国青年通信学术会议. 0.

[10] 聂嬰, 田泽, 马城城. 一种并行扫描的三角形光栅化算法设计与实现[J]. 信息通信, 2016, 000(003):67-68.