Mandelbrot Set Visualization

intro

A complex number c is a member of the Mandelbrot set if, when applying the iteration

> $z_0 = 0$
>
> $z_{n+1} = {z_n}^2 + c, n > 1$

repeatedly, $||z||$ remains bounded.

Neither infinite calculation nor checking boundedness is possible for computer. But we can approximate using a finite number of iterations. The key observation is as follows:

> If $||z_i|| > 2$ for some $i > 0$, then c is NOT a member of the Mandelbrot set.

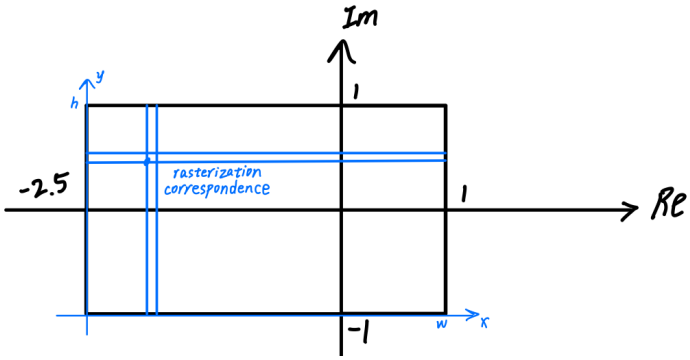
This follows from $||z_{n+1}|| \geq ||z_n||^2 - ||c||$ and that $x^2 > 2x$ for any real number $x > 2$

动力学的结果显示，复数c的微小改变即可引起判定结果的骤然变化，因此 *Mandelbrot set* 的边界非常精致。

介绍

visualization

复平面可以如下图（naive）对应屏幕坐标系，因此这个集合很容易做可视化。



在算法层面，固定使用最直观的 *escape time algorithm*，即 对每个点 (cr, ci)，不断按开头给出的递推式迭代，记录至范数大于2时的迭代次数（或者直到给定的某最大迭代次数时，范数仍小于等于2）。

为了更好地利用所求的迭代次数，可以根据求出的迭代次数对该点的对应像素上色。用了一个简单的库 tinycolormap。上色的原理是，库内部存有 colormap 采集的有限个 index:color 对，因此存在分度值。传入`[0,1]`间参数，找到位于floor和ceil的两种颜色，做线性插值。

我们的目标是，在算法层之下，探索多种手段，提高这个可视化系统的响应速度。

Checkpoint 0: naive implementation

- Windows 10; Visual Studio 2017
- CPU: 1 * Intel Core i7 8750H, 6 cores with Hyper-Threading; ISA Support up to AVX, AVX2, FMA3
- 图形框架是 经典 openGL (immediate mode) with GLUT；本质上只使用其提供的帧缓冲功能

先写出可正确运行的程序，并用传统手段（学习这门课之前就已掌握的）做点能做的。

```
int iterations(const double cr, const double ci)
```

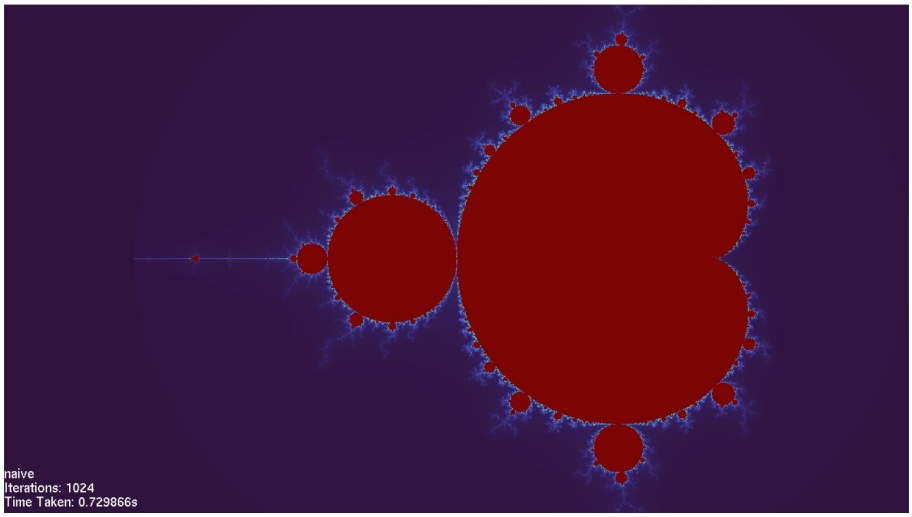
```
{
    double zr = 0, zi = 0;
    int i = 0;
    while (zr * zr + zi * zi < 4.0 && i < max_iterations)
    {
        double re, im;
        re = zr * zr - zi * zi + cr;
        im = 2 * zr * zi + ci;
        zr = re;
        zi = im;
        i++;
    }
    return i;
}
```

避免了 iterations() 循环条件中范数计算所需的平方根；

```
void naive(const ComplexPlane p)
{
    auto ci = p.init_ci;
    for (int i = 0; i < wHeight; i++)
    {
        auto cr = p.init_cr;
        for (int j = 0; j < wWidth; j++)
        {
            pFractal[i][j] = iterations(cr, ci);
            cr += p.cr_step;
        }
        ci += p.ci_step;
    }
}
```

避免了 naive() 二重循环中用乘法计算 ci, cr；

构建配置改到 x86-Release，然后剩下的就交给 MSVC 强大的 /O2 优化了。（至少上这门课之前我是这么想的）



左下角的时间通过如下的计时代码测定

```
```

```
    auto tp1 = steady_clock::now();

    // do something...

    auto tp2 = steady_clock::now();
    duration<double> elapsedTime = tp2 - tp1;
```

虽然效果还行，但是速度差强人意。尤其是后期迭代加入缩放和拖移功能后发现，程序响应速度极慢，宛如崩溃前的系统UI界面一样。

通过二分法，确定代码热点集中在逐像素的遍历过程。这在意料之中。按 1280 x 720 的分辨率计算，这个版本里 CPU 平均每秒处理 1.25M 个像素点，其实挺快了。但按照 0202年 的用户体验，怎么着也要有 30 FPS 吧，这意味着每帧的渲染要控制在 33 ms ； CPU 平均每秒应能够处理 27.65M 个像素点。从这个简单的计算，对 GPU 设计的敬仰之情油然而生。

但是！这么一个简单的 2D 渲染问题，可并行性很高，参考

- A tutorial on this rendering problem, using SIMD and multi-threading tutorial

CPU 其实是有解决这个问题的能力的。老师的 Introduction PPT 给了若干方向。

CKP1: multi-threading

OpenMP

首先探索这个，是因为它真的很简单：在 Introduction PPT fast_blur 的例子里，对应的就是一句

```
#pragma omp parallel for
```

- VS2017 支持 OpenMP 2.0
- Introduction to OpenMP - Tim Mattson (Intel)

这个任务的外层（像素行）循环可并行：

```
#pragma omp parallel for
    for (int i = 0; i < wHeight; i++)
```

感受： 优点。跟后面的手段比，OpenMP 对代码的侵入性较小，主要通过增加编译器指示的方式。 但与此同时，对代码块本身的可并行性要求较高，适合比较简单的并行计算。 比如我们这个例子，发现 for 循环内除了循环变量 i，循环的若干次迭代之间不能有别的数据依赖，也就是说，先前用步进逃掉的乘法现在还得找回来：

```
auto ci = p.init_ci + i * p.ci_step;
...
    auto cr = p.init_cr + j * p.cr_step;
```

编译器指示这种风格也比较古旧了，C 的味道很重。 比如，那个教程PPT后期的高级话题，一旦并行任务之间存在比较多同步要求，编译器指示的行数分分钟超过原有代码，使得代码结构杂乱，不好维护； 同时编译器指示难以调试，感觉不够可靠。

C++17 Concurrency Support

现代 C++ 在语言层面提供了非常优秀的并发支持库，个人 prefer.

- Effective Modern C++
- 线程支持库

std::async 是比 thread 更高级的抽象。 我们只指定需要执行的任务，而将线程管理交由 C++ 实现去做。这允许我们使用 C++ 实现提供的线程池、工作窃取和其他调度技术。

```
    auto NTASKS = std::thread::hardware_concurrency(); //实验机上的结果为 12
    std::array<std::future<void>, NTASKS> tasks{};
    for (size_t k = 0; k < NTASKS; k++)
    {
        auto this_i = k * wHeight / NTASKS, next_i = (k + 1) * wHeight /
NTASKS;
        auto this_ci = p.init_ci + this_i * p.ci_step;
        auto fut = std::async([=]() {
            for (size_t i = this_i; i < next_i; i++)
            {
                ...
            }
            ...
        });
        tasks[k] = std::move(fut);
    }
    for (size_t k = 0; k < NTASKS; k++)
    {
        tasks[k].wait();
    }
```

相比 OpenMP，

- 要封装异步任务，为此得改变代码结构，让外层循环能够只渲染中间几像素行
- 要维护异步任务的容器，手写 任务生成 和 结果等待 的代码

虽然代码量变多了，但可读性和可维护性都更强；同时也更灵活，比如可以保留循环间的步进。

在使用默认 static schedule 时，C++ 原生并发支持 比 OpenMP 快 ~5%，可能是 OpenMP 版本较低，或者有别的 overhead 的缘故。

动态调度

这个问题出于偶然发现：OpenMP 的 parallel for 支持多种调度，我好奇尝试 schedule(dynamic) ，居然发现比我原先的 C++实现 能快上1倍。

- schedule(static) 循环的许多次迭代被均分成 chunk ； 每个 thread 分配一个 chunk
- schedule(dynamic) 有一个 chunk 队列，默认每个 chunk 对应循环的一次迭代；线程们不停从队列中取 chunk ， 直到队列空

经过分析，我认为，我原先的 C++实现 把像素行分成 12 块，这种思路存在缺陷，每一块的计算量不均衡。 观察渲染出来的图片，呈深红色的部分达到了 max_iterations，计算量很大；而深蓝色的部分则几乎没有迭代。 可以看到行与行之间的计算量存在显著区别。这也是为什么 OpenMP 的动态调度能取得更好的速度。

这个问题倒也好弥补。由于 std::async 是异步任务，允许 C++实现 延迟调度，我可以直接增加 NTASKS，比如使得一个任务只写4行像素行。 一次性提交许多细粒度的任务，结合线程池执行，就是动态调度。

这样一来，C++原生并发 和 OpenMP 实现又不相上下了。

最后，多线程带来的 SpeedUp 在 6x 到 10x 不等!

CKP2: SIMD

对 MSVC 编译器生成代码的探究

知道 SIMD 很强大之后，我在第一时间就 `/arch:AVX2` 启用了 MSVC 的矢量化代码生成。 对编译器生成的代码做了逆向，分析如下：

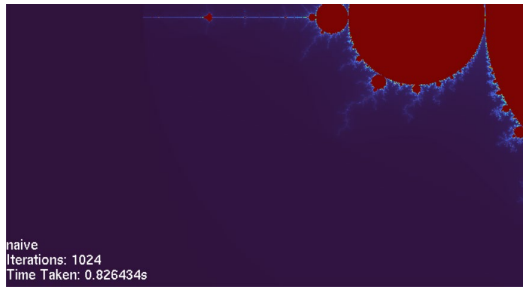
```
void naive(const ComplexPlane p)
{
00473DC0  push        ebp
00473DC1  mov         ebp,esp
00473DC3  and         esp,0FFFFFFF8h
00473DC6  push        ecx
    auto ci = p.init_ci;                CAUTION: only low-half of xmms used
00473DC7  vmovsd      xmm6,qword ptr [ebp+10h]      xmm6 = ci
    auto ci = p.init_ci;
00473DCC  mov         ecx,dword ptr [max_iterations (047F030h)]
00473DD2  mov         edx,offset pFractal (047FE78h)
00473DD7  vmovsd      xmm7,qword ptr [__real@4010000000000000 (047A6D0h)]
00473DDF  push        esi                           xmm7 = 4.0
    for (int i = 0; i < wHeight; i++)
    {
        auto cr = p.init_cr;
00473DE0  vmovsd      xmm5,qword ptr [p]    xmm5 = cr
00473DE5  mov         esi,500h    esi = wWidth
00473DEA  nop         word ptr [eax+eax]  后方指令有对齐要求
00473DF0  vxorpd      xmm2,xmm2,xmm2    xmm2 = zr = 0;
00473DF4  vxorpd      xmm4,xmm4,xmm4    xmm4 = zi = 0
        for (int j = 0; j < wWidth; j++)
        {                                           已经自动内联
            pFractal[i][j] = iterations(cr, ci);
00473DF8  xor         eax,eax
00473DFA  nop         word ptr [eax+eax]
00473E00  vmovaps     xmm3,xmm2
00473E04  cmp         eax,ecx
00473E06  jge         naive+77h (0473E37h)
```

```
        for (int j = 0; j < wWidth; j++)
        {
            pFractal[i][j] = iterations(cr, ci);
00473DF8  xor         eax,eax    j = 0
00473DFA  nop         word ptr [eax+eax]
00473E00  vmovaps     xmm3,xmm2    xmm3 = zr
00473E04  cmp         eax,ecx                    partly optimized to do-while
00473E06  jge         naive+77h (0473E37h)  break if i >= max_iterations
00473E08  vmulsd      xmm1,xmm2,xmm2    xmm1 = zr * zr
00473E0C  vmulsd      xmm0,xmm4,xmm4    xmm0 = zi * zi
00473E10  vsubsd      xmm0,xmm1,xmm0    xmm0 =  zr * zr - zi * zi
00473E14  vaddsd      xmm2,xmm0,xmm5    xmm2 = re; zr = re
00473E18  vaddsd      xmm0,xmm3,xmm3
00473E1C  vmulsd      xmm0,xmm0,xmm4    xmm0 = 2 * zr * zi
00473E20  vaddsd      xmm4,xmm0,xmm6    xmm4 = im; zi = im
00473E24  vmulsd      xmm1,xmm4,xmm4
00473E28  vmulsd      xmm0,xmm2,xmm2
00473E2C  vaddsd      xmm0,xmm1,xmm0    xmm0 = zr * zr + zi * zi
00473E30  inc         eax    j++
00473E31  vcomisd     xmm7,xmm0
00473E35  ja          naive+40h (0473E00h)  if (xmm0 < 4.0)
            cr += p.cr_step;
00473E37  vaddsd      xmm5,xmm5,mmword ptr [ebp+18h]
00473E3C  mov         dword ptr [edx],eax
            cr += p.cr_step;          possible register spilling
00473E3E  add         edx,4
00473E41  sub         esi,1
00473E44  jne         naive+30h (0473DF0h)
        }
        ci += p.ci_step;
00473E46  vaddsd      xmm6,xmm6,mmword ptr [ebp+20h]
00473E4B  cmp         edx,offset frame (0803E78h)
00473E51  jl          naive+20h (0473DE0h)
    }
}
```

- 尽管这循环的结构算简单了，可 MSVC 自动矢量化的表现还是不尽人意。一句话概括， MSVC 只是使用 SSE 的 XMMs 寄存器来做 double 的浮点运算
- 而在串行代码的编译上，老师傅 MSVC 做得挑不出毛病：
    - 汇编代码干净利落，没有一句废话；把 while 变换成 do-while ， 寄存器 xor 清零 和 XMM 寄存器初始化 都是最佳实践
    - 利用了所有 8个 XMMs 寄存器，减少数据依赖
    - 考虑到了对齐；还有合理的自动内联
    - while 循环没有完全优化成 do-while ， 是因为 虽然 i 初始化为 0，但 MSVC 仍无法判断是否会因 i >= max_iterations 而不进入循环；变量 max_iterations 的类型，我图方便用了 int ，max_iterations > 0 的约束是通过程序逻辑实现的
    - 让我在编译器的情境下 handcraft 一段等效的汇编，我没有信心比它做得更好
- 注意到可能存在寄存器溢出。基于x64下有双倍寄存器可用的想法，我试图改为x64构建来“优化”。但没有什么用。我猜可能是因为，我只是倍增了可用的体系结构寄存器，并没有改变物理上的寄存器数量

MSVC 没有生成 x87 浮点指令的代码，我想是因为这套指令已经 deprecated 了， SSE 扩展指令集已经完全覆盖了它的功能，并且没有栈的限制， SSE 指令的速度还可能更快。 为了验证，我调整 MSVC 的代码生成选项，禁用增强指令集，结果如下：

```
            pFractal[i][j] = iterations(cr, ci);
002A5567  sub         esp,10h
002A556A  fxch        st(1)
002A556C  fstp        qword ptr [esp+8]
002A5570  fstp        qword ptr [esp]
002A5573  call        iterations (02A54D0h)
```



IA-32 使用古旧的 x87 浮点栈指令来处理，msvc 做的优化也有限，每次计算时间从 0.7x s 升到 0.8x s

handcraft SIMD code using Compiler Intrinsics

对 inner loop 进行向量化，参考代码

```
    for (int j = 0; j < wWidth; j++)
    {
        pFractal[i][j] = iterations(cr, ci);
        cr += p.cr_step;
    }
```

SIMD 提供的数据级并行是在汇编指令层， 因此不但要内联 *iterations* 函数， 还得思考如何组织 SIMD 原语，使得一种操作同时运用在多个元素上，最后输出还和串行分别处理多个元素一致。 对代码的侵入性很大，并且需要定制。

- Intel Intrinsics Guide

翻译了一会就发现，最难的并不是 *iterations* 函数中核心的浮点运算语句（相反，基本块内代码的翻译是最简单的）， 而是控制逻辑，比如条件与分支。 想象向量部件正在同时让 4 个点 执行 *iterations* 函数。此时某个点不满足 while 循环条件，这个元素的循环应该退出。 然而这不符合 SIMD paradigm: Multiple Data 分别获得了不同的 Instruction.

SIMD paradigm 下，应该通过某种组织，使得：

- 当 4 个点中的某个不满足循环条件时，对这个点对应输出的修改被屏蔽。
- 当 4 个点全部不满足循环条件时，循环退出。

对我们的 *iterations* 函数，每个点的输出是循环次数 i. 即我们要使得：当 4 个点中的某个不满足 *zr * zr + zi * zi < 4.0* 时，对这个点所对应的 i 的自增被屏蔽。

可以通过 mask 来达成这个目的，

```
i++;
```

翻译为：对 4 个点对应的 (zr, zi, i, 4.0) 都做如下操作

```
i += 1 AND (zr * zr + zi * zi < 4.0)
```

debug 体会

这期间深受 VS2017 调试器的 bug 所害：

- 一是 整个调试过程，寄存器窗口显示的 YMMs 的高 128 位不可信，总是显示为 0；
- 二是 如果单步逐条执行指令，不管什么AVX操作，YMMs 的高 128 位都会被清零（而不仅仅是显示错误）！ 必须改用断点才能规避。

特地录了个视频留作纪念，随附。 可以想象，当在苦苦 debug 过程中，发现连 IDE 居然都不可信，那一刻真是血压都上来了。

最后找出 bug: 为了完成将 YMM 数据存回内存， 看到有 intrinsics _mm256_cvtsi256_si32 可以取到 YMM 低 32 位， 就试图结合 3 次右移来做 4 次存回， 错用了 intrinsics _mm256_bsrli_epi128. 这事实上是 SIMD 编程模型用于操作 2 个 128 位数据的移位指令。

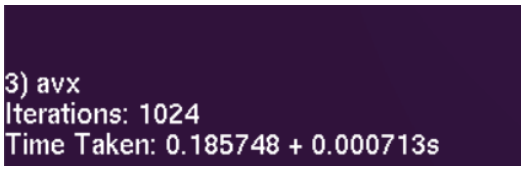
循环次数 i 本是 int 类型，但 AVX 同时处理 4 个元素，每个元素在寄存器中都占 64 位，因此方便起见 i 也提升为 int_64. 但最后需要把 int_64 砍成 int_32 存回数组。想法是把每个 int_64 的低 32 位排到一起，然后用 XMM 的 store 指令。 这就用到 Swizzle 类指令，比如 shuffle permute 等。 AVX 跟 SSE 比有一个 tricky 的地方：

```
When shuffling, YMM is designed as 2 lanes of 128-bit, which means that
you cannot move element from one lane to another.
```

因此这个想法得这么实现：

```
// DW7 | DW6 | DW5 | DW4 | DW3 | DW2 | DW1 | DW0
i_4pi = _mm256_shuffle_epi32(i_4pi, _MM_SHUFFLE(3, 1, 2, 0));
// DW7 | DW5 | DW6 | DW4 | DW3 | DW1 | DW2 | DW0
i_4pi = _mm256_permute4x64_epi64(i_4pi, _MM_SHUFFLE(3, 1, 2, 0));
// DW7 | DW5 | DW3 | DW1 | DW6 | DW4 | DW2 | DW0
```

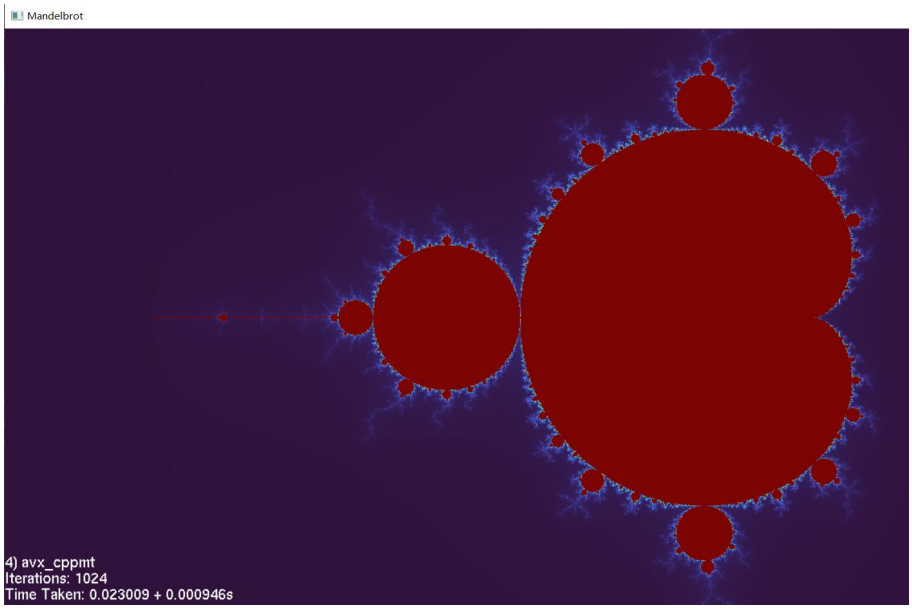
Speedup



可以看到加速比达到了 4x, 并行化的效果非常好。

CKP3: 结合使用 SIMD 和 multi-threading

把异步任务的内层循环改为 AVX 实现，最大限度地利用了摩尔定律带来的福利：



渲染时间从最初的 730 ms 缩短到最终 23 ms ， 加速比达到了 32 倍。 结合使用 SIMD 和 multi-threading

，应用延时基本满足了与用户实时交互的需求， 而只使用了一颗中等层次的笔记本 CPU. 发掘和运用不同层次的并行 带来了流畅的显示。

总结

- 这个问题的渲染并不涉及 *data movement in the slower or shared parts of the memory subsystem*，因此 *loop tiling* 技术不适用。
- 最后的工程里有多种版本的函数，它们所做的事情是一样的。
    - naive 实现是最通用的解决方案，可用性最强
    - 多线程方案的加速比 与可用的硬件线程数成正比，同时以功耗为代价（风扇呼呼转），还可能依赖编译器支持
    - AVX 方案的加速比 与 DLP 的元素个数成正比，额外功耗开销小
    - AVX + 多线程 方案达到了最优秀的性能表现
    - 实际运用，可以去掉 OpenMP 版本的实现
- 可以稍微重构一下减少冗余代码
- 更大规模的并行，可以利用 GPU ， 比如 CUDA