石邢越 16337208

杜云飞教授

超级计算机原理与操作

2018 年 5 月 25 日

关于 Canny 的 SIMD 优化练习

1. 概述

这篇报告描述了我完成关于 Canny 边缘检测算法的 SIMD 优化的过程。首先，我查阅了和练习相关的背景知识，即边缘检测的概念、Canny 算法的大体步骤等（见“2.背景知识”）。之后我尝试运行并根据运行结果和我的目标修改了已提供的串行 Canny 算法程序（见“3.修改串行程序”）。由于刚刚学了 OpenMP 相关的编程知识，我决定先尝试用 OpenMP 进行优化（见“4. 用 OpenMP 进行 SIMD 优化”），但没有等到理想的结果（不仅没有优化，甚至用时更长!）。之后我该用作业建议使用的 IPP 库进行优化（见“5. 用 IPP 库进行 SIMD 优化”），得到了显著的优化。最后，在第 6 部分，我反思了 OpenMP 没有起到优化作用的原因，总结了安装配置 IPP 库的过程中的经验。

2. 背景知识

这部分摘取了课程主页和网络上关于边缘检测和 Canny 算法的介绍，通过了解这些知识，我明确了我要进行的工作——即我的工作重点不在于熟悉并实现 Canny 算法，而在于利用 SIMD 并行对串行的 Canny 算法实现进行优化。

2.1 边缘检测

边缘检测是图像处理和计算机视觉中的基本问题, 边缘检测的目的是标识数字图像中亮度变化明显的点。

图像属性中的显著变化通常反映了属性的重要事件和变化。 这些包括

（i）深度上的不连续、

（ii）表面方向不连续、

（iii）物质属性变化和

（iv）场景照明变化。

边缘检测即将图像中周围属性变化明显的点标识出来，使这些点连成图像中的“边”。

2.2 Canny 算法

Canny 是最早由 John F. Canny 在 1986 年提出的边缘检测算法，并沿用至今。

John F. Canny 给出了评价边缘检测性能优劣的三个指标：

(i) Good detection。即要使得标记真正边缘点的失误率和标记非边缘点的错误率尽量低。

(ii) Good localization。即检测出的边缘点要尽可能在实际边缘的中心；

(iii) Only one response to a single edge。当同一条边有多个响应时，仅能取其一作为标记。即数学上单个边缘产生多个响应的概率越低越好，并且尽量抑制图像噪声产生虚假边缘。

Canny 算法是以上述三个指标为优化目标的求解问题的最优解，即在对图像中物体边缘敏感性的同时，也抑制或消除噪声的影响。其主要步骤如下：

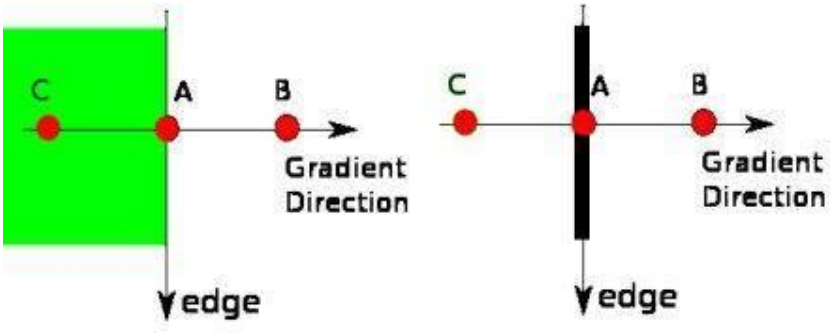
(i) Noise Reduction（可使用高斯滤波器去噪）

(ii) Finding Intensity Gradient of the Image（可在横纵轴分别用 Sobal 算子初步计算出两张梯度图，再最终计算出原图梯度的幅值和方向，其中方向最终近似到四个角度 0, 45, 90, 135）

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

(iii) Non-maximum Suppression（边缘细化，使其更清晰）



(iv) Hysteresis Thresholding（最终使用双阈值来选择边缘像素，生成边缘检测结果）

3. 修改串行程序

我修改的串行代码是已提供的 2 份串行代码中的 Canny-Implementations\fast-canny-edge。原来的程序有一些不符合我要求的地方，我对他们进行了修改，最后得到了几组输入、输出图像对比。

3.1 原程序使用了已经弃用的函数

原程序在读、写文件时使用了 fopen(…)函数和 fscanf(…)函数，这两个函数由于安全性问题已经被弃用。新的替代函数 fopen_s(…) fscanf_s(…)使用的参数略有不同，所以若将这两个已经弃用的函数用新函数替代，代码中需要修改的部分很多，故替换成新函数，而在项目中设置“禁用特定警告 4996”来避免报错。

3.2 原程序的计时方法精确度不够高

```
#include <time.h>
clock_t start = clock();
/*需要计时的部分*/
printf("time elapsed: %f\n", ((double)clock() - start) / CLOCKS_PER_SEC);
```

原程序使用如下方法进行计时：

这种计时方式精确度不够高，当用时比较小的时候，结果会直接被近似成 0，无法观察，

```
#include <Windows.h>
LARGE_INTEGER t1,t2,tc;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
/*需要计时的部分*/
QueryPerformanceCounter(&t2);
printf("time elapsed: %f\n",(t2.QuadPart - t1.QuadPart)*1.0/tc.QuadPart);
```

所以我把各个计时部分改成了下面这种方法：

这种方法计时的精确度更高，可以解决用时被近似成 0 的问题。见下面两次运行结果的对比，第一个结果是原来的代码，时间精确到小数点后 3 位，第二个结果是修改后的代码，时间至少精确到小数点后 6 位。

```
C:\Users\mibyb\Desktop>canny1 collage.pgm
*** PGM file recognized, reading data into image struct ***
*** image struct initialized ***
*** performing gaussian noise reduction ***
Gaussian noise reduction - time elapsed: 0.002000
Calculate gradient Sobel - time elapsed: 0.001000
*** performing non-maximum suppression ***
Non-maximum suppression - time elapsed: 0.000000
Estimate threshold - time elapsed: 0.000000
Hysteresis - time elapsed: 0.001000

C:\Users\mibyb\Desktop>
```

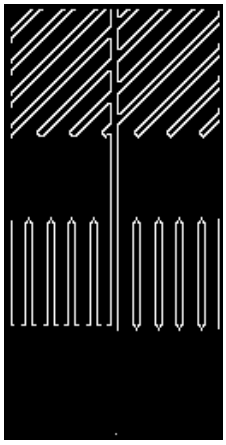
（图 1 之前用 clock 函数显示的结果）

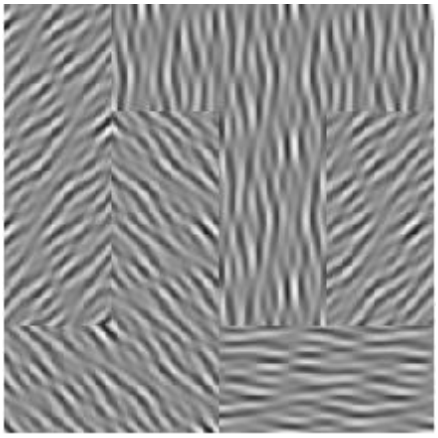
```
C:\Users\mibyb\Desktop>canny1 collage.pgm
*** PGM file recognized, reading data into image struct ***
*** image struct initialized ***
*** performing gaussian noise reduction ***
Gaussian noise reduction - time elapsed: 0.001638
Calculate gradient Sobel - time elapsed: 0.001588
*** performing non-maximum suppression ***
Non-maximum suppression - time elapsed: 0.000583
Estimate threshold - time elapsed: 0.000143
Hysteresis - time elapsed: 0.001596

C:\Users\mibyb\Desktop>
```

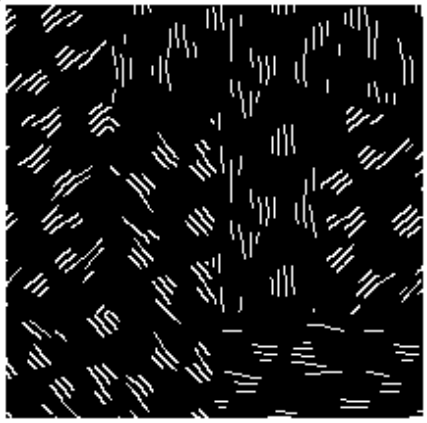
（图 2 修改后用嫡嗒函数显示的结果）

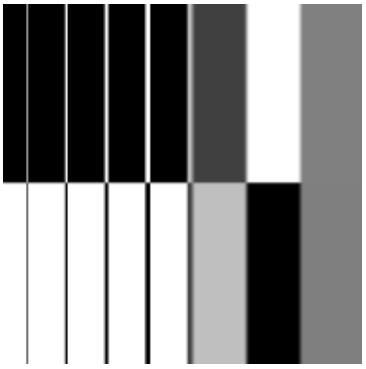
3.3 边缘检测结果

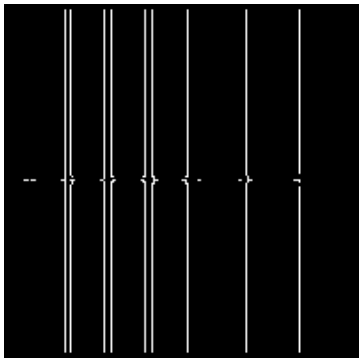
用修改后的串行代码，得到了几组输入、输出图像。见下表。











（表 1 Canny 算法的实现结果）

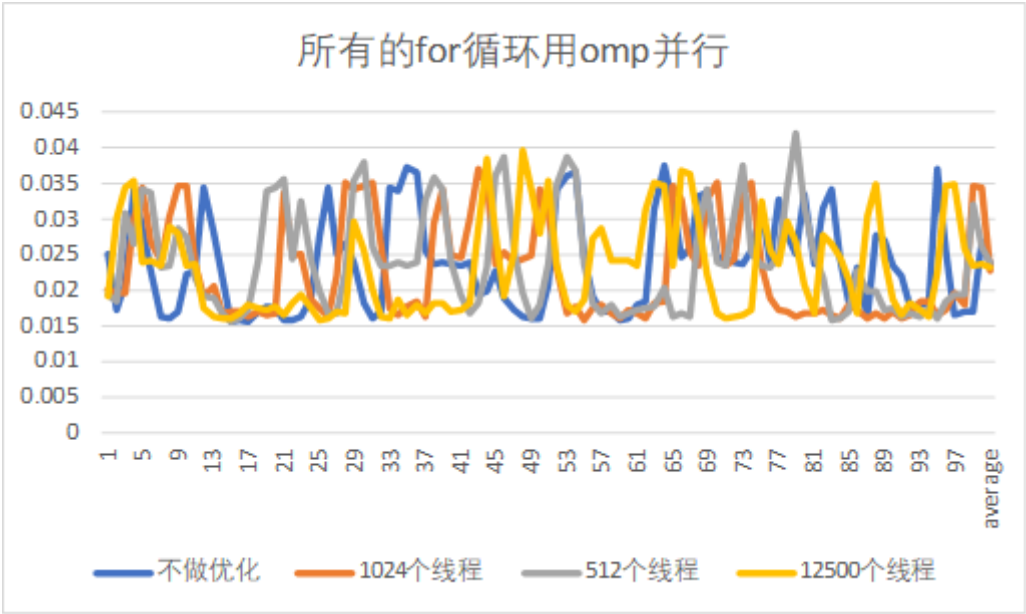
4. （尝试 1）用 OpenMP 进行 SIMD 优化

4.1 代码说明

在已有的串行代码基础上，用 OpenMP 对各个 for 循环进行并行。

4.2 优化结果

优化结果不明显，线程设置数量过多时甚至用时比串行结果更长，下图为运行 100 次用时的对比。6.1 小节反思了优化失败的原因。



（图 3 串行、并行用时的对比）

5. （尝试 2）用 IPP 库进行 SIMD 优化

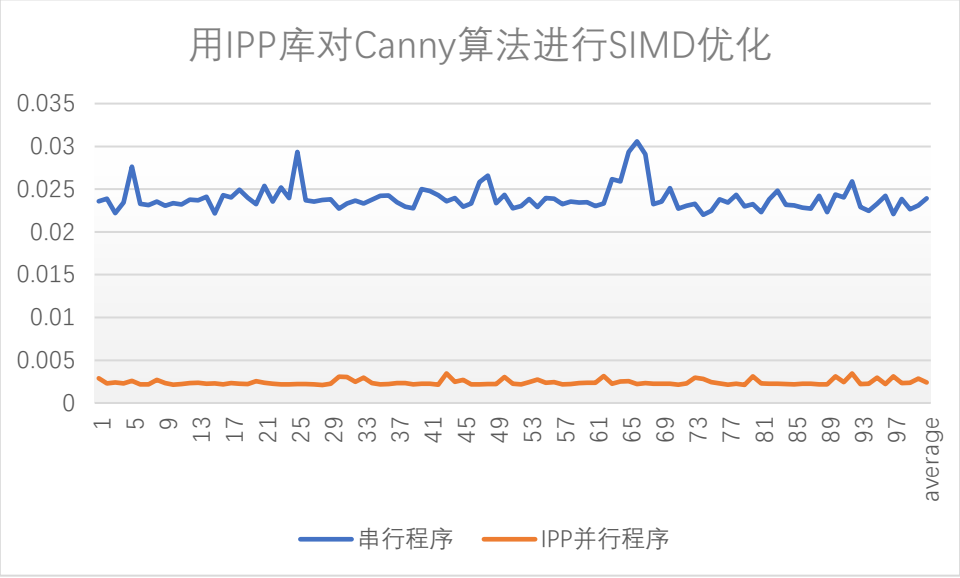
5.1 代码说明

OpenMP 后尝试使用 IPP 库进行优化。IPP 库中包含了 Canny 算法中各步骤的函数，且这些函数是使用 SIMD 优化的。也就是说使用这些封装好了的边缘检测函数，即可实现对 Canny 算法的 SIMD 的优化。

我在 Intel 官网的 IPP 手册上找到了 Canny 算法实现的示例源代码，并在这份代码中加写了图像文件读写、计时等部分。

5.2 优化结果

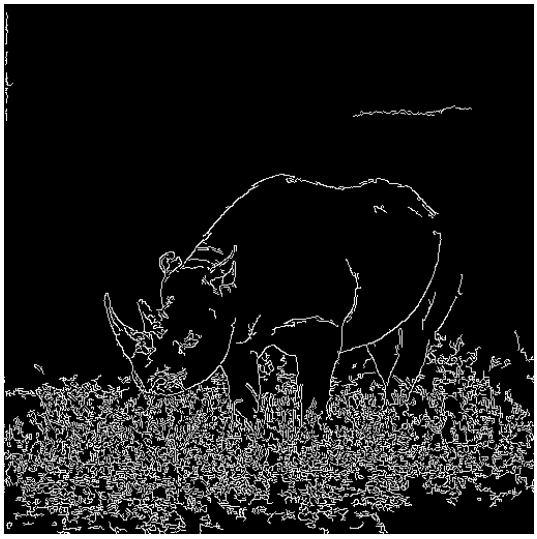
用 IPP 库中的边缘检测函数来进行优化的效果十分明显！以一副相同 512x512 的图像作为输入，以优化先后代码各自的 100 次运行用时为样本进行观察，平均用时的加速比达到了 $0.02392/0.002402 \approx 9.96$，见下图。



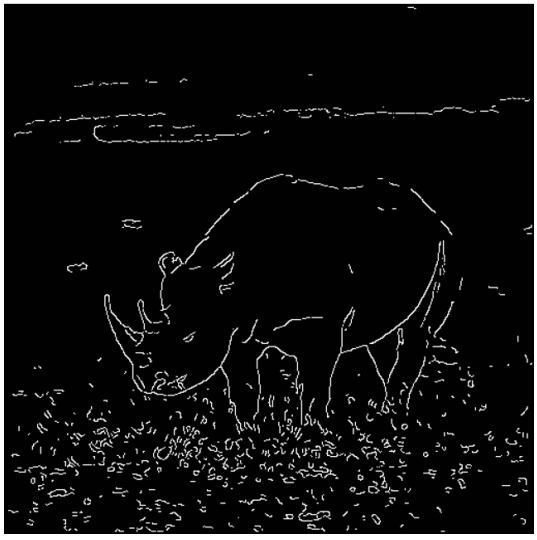
（图 4 用 IPP 库优化后的代码和原代码用时对比）

两份代码边缘检测输出图像略有不同，可以猜测 IPP 优化的代码在检测图片细小边缘时更为精确（草地部分、犀牛皮肤纹理等），但原来的代码对属性的变化更加敏感（背景中变化不太明显的天空部分，原来的代码检测出来了，但 IPP 优化的代码没有），见图 5，6，7。

（图 5 原图）



（图 6 IPP 代码得到的结果）



（图 7 串行代码得到的结果）

6. 反思与心得

6.1 为什么 OpenMP 没起到优化作用?

我在用 OpenMP 尝试优化的时候，课堂上正好在讲 OpenMP 编程中需要注意的问题，其中提到了如果 parallel for 中的计数器变量是公用的话，很容易出现问题，导致达不到期望的优化结果。查看了我修改的 OpenMP 代码，其中的计数器变量（即 i）就是不是线程私有

```
#pragma omp parallel for num_threads(16) schedule (dynamic,1)
for (i = 0; i < w * h; i++) {//reading PGM file
    img_data[i] = fgetc(fp);
}
```

的，所以我猜测可能是这里出了问题。

在那节课上也讲到 CPU 中可用的物理线程并不多，一般不超过 20 个，而在我之前修改出来的 OpenMP 代码中，将 num_threads(…)中的参数设成了 512，1024 等过多的线程数，因为机器并没有那么多处理单元来同时做这么多线程，这样简单粗暴地增加线程数并没有多大的优化意义。

6.2 IPP 库配置过程中需要注意的问题

要注意在项目的“属性→链接器→输入→添加附加项”中添加使用到的各个 IPP 库中的头文件。

参考文献

https://baike.baidu.com/item/%E8%BE%B9%E7%BC%98%E6%A3%80%E6%B5%8B

（边缘检测的介绍）

https://software.intel.com/en-us/ipp-dev-reference-canny-sample

（Intel 官网上提供的 Canny 算法实现代码）

http://www.cnblogs.com/darkknightzh/p/5473890.html>

（IPP 配置指导博文）

http://agner.org/optimize/

http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/imgproc/table_of_content_imgproc/table_of_content_imgproc.html

http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html

http://marathon.csee.usf.edu/edge/edge_detection.html

https://en.wikipedia.org/wiki/Canny_edge_detector

http://matlabserver.cs.rug.nl/