作业B-C++图像处理代码性能优化

2024.11

基本说明

设计目标

在现代计算机系统中，体系结构相关优化对于非专业方向的程序员来说已经变成了一个由编译器完成的黑盒。但是在日常程序的编写过程中，程序员仍然需要在程序设计过程中考虑一些相关的设计与优化，以便写出更加具有硬件友好性的程序。

在本次作业中，我们将对于一个简单的图像处理 C++ 程序进行优化，以便更好地利用现代计算机体系结构的特性。

运行环境

本次课程大作业支持常见的体系结构和操作系统作为运行环境，同学们可以根据自己的电脑配置选择合适的运行环境。

> **Tip**
> 事实上，如果在作业中并没有使用到特定的体系结构特性（SIMD）优化等，那么你可以使用任意的编译器进行编译，但是请注意在报告中说明你的编译器版本和编译选项。

助教将使用本地部署的 GitHub Actions 进行自动化测试，因此，请确保你的代码可以在 Github Actions 所提供的任意一种软硬件环境 参考1,参考2 上运行，即为：

- x86 架构 （SIMD 支持 AVX2/SSE 指令集）
  - Windows Server (g++/clang++/MSVC)
  - Ubuntu (适配 LINUX g++/clang++)

    - macOS （g++/clang++）
- arm64 架构（SIMD 支持 NEON 指令集）：
    - Ubuntu (适配 LINUX g++/clang++)
    - macOS（苹果 M 系列芯片, g++/clang++）

助教将在测试前将编译器更新，因此请放心使用新版本C++特性。同时，浮点数误差导致的一些微小差异在评测时会被忽略。

请在提交时说明你的代码的运行环境。如果有同学对于运行环境有疑问或使用其他体系结构的运行环境，可以在企业微信或邮件联系助教杨翼飞同学进行咨询。

作业要求

- 对于给定的图像处理程序，你需要按照文件中的注释对其进行优化，以便更好地利用现代计算机体系结构的特性。
- 你需要在你的报告中详细说明你的优化思路、实现过程、编译指令、优化效果，并提供一定的截图或数据来证明你的优化效果。
    - 我们要求报告统一使用 A4 格式，最小字体不小于小四字体（12号/12pt），行距不小于一倍行距，总页数不得超过 6 页。
- 提交要求： 你需要提交一个包含源代码、报告的压缩包，压缩包命名请按照两个大作业统一要求格式进行，压缩包内部的打包格式如下：
    - `[学号]-[姓名].pdf`：报告文件，包含你的优化思路、实现过程、优化效果等内容。
        - 一个示例的文件名为 `223xxxxx-张三.pdf`。
    - `[(avx2|sse|neon)]-[(g++|clang++|msvc)]-[学号]-[姓名].cpp`：优化后的源代码文件，文件名中包含你的优化使用的指令集和编译器版本。
        - 一个示例的文件名为 `avx2-g++-223xxxxx-张三.cpp`。
    - 一个示例压缩文件为：

```
B-223xxxxx-张三.zip
├── 223xxxxx-张三.pdf
└── avx2-g++-223xxxxx-张三.cpp
```

评分标准

- **[30%] 优化效果。** 包括优化前后的性能对比和对结果的分析，其中绝大多数的分数并不要求优化后的程序性能达到很高的水平，但是需要有一定的提升，同时需要在报告中对优化效果进行合理的解释。
  - 如果能做到相对于优化前的程序运行时间有一倍以上的提升（运行时间小于等于原运行时间的 50%），即可满足对于优化效果的要求，如能正确解释优化效果，可获得优化效果分中80%以上的分数，此后根据加速效果进行加分直至满分。
    - （参考报告中的加速比和代码实现，助教测试相差不大的情况下不会扣分数）。
  - 如果能做到相对于优化前的程序运行时间有四十倍以上的提升（运行时间小于等于原运行时间的 2.5%），且能合理解释优化效果，可在满分基础上获得少量可抵用其他部分分数的额外加分。
    - （本项测试以助教评测为准）
- **[30%] 优化思路。** 包括对程序的分析、优化思路的合理性、优化方法的选择等。
- **[15%] 代码质量。** 包括代码的可读性、可维护性、注释的完整性等。
- **[20%] 报告质量。** 包括报告的结构、文字表达、数据展示等。
- **[5%] 提交质量。** 包括是否按要求规范文件命名等。

代码简述

数字图像处理相关算法程序的编写是一种常见的计算机程序设计任务。在本次作业中，我们假设一个 16384×16384 的灰度图像（你也可以通过修改 `main` 函数中的 `size` 来修改维度，但请记得在提交时改回来），依次对其进行高斯模糊和幂次变换操作。

数据的生成与读取

> ⚠ **Warning**

> 你不可以修改随机数生成部分的代码逻辑，这会导致自动化测试出现潜在的问题。
>
> 但是你可以修改存储数据的数据结构和初始化方式，只需要小心注意数组的随机生成顺序即可。

原代码使用二维 `vector` 存储图像数据，其中 `figure` 存储原始图像数据，`result` 存储处理后的图像数据。对于每一个像素点，其值为一个取值在 $[0, 256)$ 内的非负整数值，所以我们使用 `unsigned char` 数据类型来存储。

```
// !!! ---------------------------------------- !!!
std::random_device rd;
std::mt19937_64 gen(seed == 0 ? rd() : seed);
std::uniform_int_distribution<unsigned char> distribution(0, 255);
// !!! ---------------------------------------- !!!

// 数组的初始化在这里，可以改动，**注意** gen 的顺序是从上到下从左到右即可。
for (size_t i = 0; i < size; ++i) {
    figure.push_back(vector<unsigned char>());
    for (size_t j = 0; j < size; ++j) {
    figure[i].push_back(static_cast<unsigned char>(distribution(gen)));
    }
}
```

其中被感叹号注释包围的部分是随机数生成的部分，请不要修改它，但是你可以修改存储 `figure` 和 `result` 的数据结构和初始化方式，注意随机数是按照从上到下从左到右的顺序生成的即可。

高斯模糊

高斯模糊是一种常见的滤波处理算法，其基本思想是对图像中的每一个像素点，以其为中心，取一个固定大小的窗口，然后计算窗口中所有像素点的加权平均值，作为该像素点的新值。

高斯模糊实际上是一个二维卷积操作，对于一个大小为 $n \times n$ 的窗口，其卷积核为一个 $n \times n$ 的矩阵，其中每一个元素的值为：

$$\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

在实现中，我们定义了一个 3×3 的卷积核，假设 $\sigma=1$，即卷积核为：

$$\begin{bmatrix}0.0751136 & 0.123841 & 0.0751136\\ 0.123841 & 0.20418 & 0.123841\\ 0.0751136 & 0.123841 & 0.0751136\end{bmatrix}$$

为了便于计算，我们将卷积核进行了近似和归一化，即：

$$\begin{bmatrix}1/16 & 2/16 & 1/16\\ 2/16 & 4/16 & 2/16\\ 1/16 & 2/16 & 1/16\end{bmatrix}$$

我们对图像的每一个像素点均进行卷积操作。对于边缘像素点我们使用局部的卷积核进行计算。

幂次变换

幂次变换是一种常见的图像处理算法，其对图像的每一个像素点进行如下变换：

$$f(x)=\left(\frac{f(x)}{255}\right)^{\gamma}\times255$$

其中 $f(x)$ 为原图像的像素值，γ 为幂次变换的参数。

幂次变换可以对灰度图像的对比度进行调整，当 $\gamma>1$ 时，可以增加图像的对比度，当 $0<\gamma<1$ 时，可以减小图像的对比度。在本大作业的实现中，我们取 $\gamma=0.5$。

运行与测试

对于优化前的程序，你可以使用任意的编译器进行编译，为了便于优化，请设置编译优化等级为 `-O0`（如果使用 Visual Studio，请设置为 Debug 模式），但是请注意在报告中说明你的编译器版本和编译选项。

这里给出了一个使用 `g++` 编译器的编译命令：

```
g++ -O0 -std=c++ -m64 -o before_optimize before_optimize.cpp
```

运行时如果不加入参数，程序将会使用随机数生成一个 16384×16384 的灰度图像进行处理，你也可以指定一个随机数种子用于生成固定的图像：

```
./before_optimize 2333
```

程序运行将会有两行输出，分别为计算结果矩阵的校验和和两个计算任务的运行时间。请不要修改这两行输出的内容，也不要增加任何额外的输出。

优化提示

> **Tip**
> 以下提示并不是必须的，你可以根据自己的理解和思考，采用任何与计算机体系结构相关的优化方法对所给代码进行优化。对于下列优化方法，我们不要求实现所有的优化方法（实际上，在助教的测试中，理解并实现其中的部分内容就可以实现很不错的加速效果），但是无论你采用什么优化方案，你需要对你的优化方法进行合理的解释。

> **Important**
> 在优化实现中，我们不反对使用 OpenMP 等并行计算优化方式，但请注意，我们的评测机只有不多于4个逻辑核心，请评估并行计算的代价和收益之间的关系。

局部性保证

- 你可以考虑存储 `figure` 和 `result` 形式是否有利于局部性，时间和空间局部性在计算机体系结构中也是非常重要的一个知识点。
  - `vector` 数据结构在内存中可能是不连续的，你可以考虑使用连续的内存空间。

查找表的构建

- 某些对于每一个像素都进行复杂运算的操作结果可能只有有限的若干种，你可以考虑在编译时计算出所有可能的结果，并构建一个查找表，在运行时查表即可。

- 查找表在本大作业的实现中似乎和体系结构并不是一个很相关的话题，但在 FPGA 等体系结构相关的研究中，LUT 是一个非常基本的组成部分。

SIMD 指令加速

- 你可以考虑使用 SIMD 指令加速一些复杂的运算，例如高斯模糊中的卷积操作。
- 不同体系结构环境下的 SIMD 指令集所对应的函数可能有显著的差异，这也是我们为同学们提供不同支持环境的原因之一。
  - 当然，你也可以使用 `GCC` 或 `Clang` 的内置函数来进行 SIMD 加速，参考 cppreference 中的介绍文档，但是请注意这些函数可能在不同的编译器版本中有所不同。
  - 如何查询自己设备 CPU 支持的指令集？
    - 请自行上网查询资料。下面提供一些参考：
    - 对于 x86/64 架构，你可以考虑使用 `AVX2` `SSE` 等指令集。
    - 对于 arm64 架构，你可以考虑使用 `NEON` 指令集。
- 请注意，SIMD 指令的使用需要考虑数据的对齐问题，你可能需要使用一些特殊的处理来保证数据的对齐。
- 在编译时，你可能需要使用一些特殊的编译选项来启用 SIMD 指令集，例如对于 `g++` 编译器，你可以使用 `-mavx2` 来启用 `AVX2` 指令集，在 Windows 下如果使用 Visual Studio 编辑并使用内置 `MSVC` 编译器编译，你可以使用 `/arch:AVX2` 来启用 `AVX2` 指令集。
  - 以使用 `avx2` 的 `g++` 编译器为例，编译命令可以为：

```
g++ -O0 -std=c++20 -m64 -mavx2 -o after_optimize
after_optimize.cpp
```

这里我们对于 `AVX2` 指令集的使用给出一个简单的示例：

我们假设有一个数组处理函数：

```
void processArrays() {
    for (size_t i = 0; i < size; ++i) {
        result[i] = left[i] * left[i];
        if (right[i] > 0) {
        result[i] += right[i];
        }
        if (result[i] > 1000.0f) {
        result[i] = 1000.0f;
        }
    }
}
```

在加入头文件 `immintrin.h` 后，我们可以使用 `AVX2` 指令集进行优化：

```
    void processArrays() {
    // AVX_SIZE = 8, 这是因为 AVX2 指令集中一个寄存器可以存储 8 个 float 类型的数据
    // 而在本实验中，我们使用 unsigned char 类型，可以计算一下能够存储的数据量
    const size_t simdSize = size - (size % AVX_SIZE);
    for (size_t i = 0; i < simdSize; i += AVX_SIZE) {
        // load the data to the registers
        __m256 l = _mm256_load_ps(&left[i]);
        __m256 r = _mm256_load_ps(&right[i]);

        __m256 res = _mm256_mul_ps(l, l); // result[i] = left[i] * left[i]

        // right[i] > 0 ? result[i] += right[i] : result[i] = result[i]
        __m256 mask = _mm256_cmp_ps(r, _mm256_setzero_ps(), _CMP_GT_OQ);
        __m256 condAdd = _mm256_and_ps(mask, r);
        res = _mm256_add_ps(res, condAdd);

        // maxvalue = 1000.0f
        __m256 threshold = _mm256_set1_ps(1000.0f);
        mask = _mm256_cmp_ps(res, threshold, _CMP_GT_OQ);
        res = _mm256_blendv_ps(res, threshold, mask);

        // store the result
```

```
            _mm256_store_ps(&result[i], res);
        }

        // 相对需要注意的地方就是最终没有被 8 整除的部分的处理
        // 但是在此次大作业中，我们假设 size 为 16384
        // 可以被你计算到的值整除，所以不做要求
        for (size_t i = simdSize; i < size; ++i) {
            result[i] = left[i] * left[i];
            if (right[i] > 0) [[likely]] {
            result[i] += right[i];
            }
            if (result[i] > 1000.0f) [[likely]] {
            result[i] = 1000.0f;
            }
        }
    }
```

参考资料

- C++ SIMD 加速的标准库实验内容可以参考 cppreference 中的介绍文档。
- x86/64 相关的 `AVX2` / `AVX512` / `SSE` 指令集的使用可以参考 Intel Intrinsics Guide。
- arm 相关的 `NEON` 指令集的使用可以参考 ARM NEON Intrinsics Reference。
- 在必要时刻，你可以参考搜索引擎和大模型给你的帮助，但请注意辨别信息的真实性和可靠性，也不要直接粘贴网络上的和其他同学的代码，助教将在提交后进行代码查重。
- 如对该作业有任何其他疑问，请在企业微信或邮箱中联系助教杨翼飞同学 (yangyf83@mail2.sysu.edu.cn)。

当作业完成之后

当你做出了很杰出的工作，将运行耗时压缩到了一个很低的数字，请尝试将原代码的优化等级调整到 `-02` 或 `-03`（Windows Visual Studio 对应为 Release 模式），你会发现编译器用几秒钟时间就把这个计算任务优化到了手工工作需要几天时间的水平。但是当你对优化后的代码也使用高优化等级时，你会发现你的手工优化也对编译器优化有一定的帮助，这也说明了在现代计算机体系结构中，程序员和编译器是相辅相成的关系，我们必须在代码编写过程中写出更加硬件友好的代码，编译器才能够更好地执行优化任务。

以下是助教提供的一个优化方案在不同优化等级下的表现，可作为一个优化前后的应用程序在不同编译器优化等级下表现的参考。请注意，在一些情况下， `g++ -03` 对性能的优化效果可能还不如 `-02` ，其原因已经超出了本课程的知识范围，感兴趣的同学可以自行查找静态分析和编译优化相关知识进行学习。

```
(base)
yangyf at netlab5 in [~/code/TA/arch_project]
10:36:41 › g++ -O0 -march=native -mavx2 -std=c++20 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 583222290
Benchmark time: 21759 ms
(base)
yangyf at netlab5 in [~/code/TA/arch_project]
10:37:49 › g++ -O0 -march=native -mavx2 -std=c++20 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Checksum: 583222290
Benchmark time: 847 ms
(base)
yangyf at netlab5 in [~/code/TA/arch_project]
10:38:21 › g++ -O2 -march=native -mavx2 -std=c++20 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 583222290
Benchmark time: 4805 ms
(base)
yangyf at netlab5 in [~/code/TA/arch_project]
10:38:40 › g++ -O2 -march=native -mavx2 -std=c++20 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Checksum: 583222290
Benchmark time: 258 ms
(base)
yangyf at netlab5 in [~/code/TA/arch_project]
10:38:55 › g++ -O3 -march=native -mavx2 -std=c++20 -fopenmp before_optimize.cpp -o before_optimize && ./before_optimize 100
Checksum: 583222290
Benchmark time: 4782 ms
(base)
yangyf at netlab5 in [~/code/TA/arch_project]
10:39:17 › g++ -O3 -march=native -mavx2 -std=c++20 -fopenmp after_optimize.cpp -o after_optimize && ./after_optimize 100
Checksum: 583222290
Benchmark time: 299 ms
(base)
```