山东大学

SHANDONG UNIVERSITY

计算机系统原理第二次实验报告

小组成员及分工

姓名	班级	学号	分工
鲍泓睿	密码1班	202200460076	SIMD指令集优化初步实现以及最终优化、性能分析与改进以及相应部分的报告撰写
高钲超	密码1班	202200460136	SIMD指令集优化的SSE指令集尝试、卷积运算的GPU优化以及相应的报告撰写
陈万里	网安2班	202200460153	SIMD指令集最终优化、SIMD与多线程优化、加速分析对比以及相应部分的报告撰写
郑傲宇	网安2班	202222460130	基准测试、实验结果与优化分析以及相应部分的报告撰写

目录

1 实验要求

使用SIMD指令集(可以结合其他优化方式)，对卷积算法进行优化，分析实验结果是否符合预期，探讨优化结果符合预期的原因或导致优化效果未达预期的因素。

- 注意事项：

1. SIMD指令集的头文件根据编译器的实现不同而略有差异，有的代码的写法是依赖编译器版本的，但是文档中提到的方法都会被实现。
2. “其他优化方式”包括但不限于利用程序的时间、空间局部性进行程序算法的优化，如果你用到了这些方法，请在报告中说明这些方法的原理和效果。
3. 关于矩阵的输入规模，请自行调整，方便测试即可。一般来说，卷积核的长度是奇数的，为了降低难度，使用偶数长度的卷积核即可。另外卷积核规模不宜过大。
4. 矩阵中单个数据的位宽不作限制，可以灵活调整，多做尝试。
5. 在报告中展示加速思路、方法、数据规模和对应的加速效果。程序实际执行时间还取决于编译器种类、优化选项、操作系统、CPU型号等因素，本实验需要控制这些无关因素不变，对比原始卷积运算和优化后的加速效果。

- 考察重点：

1. SIMD编程
2. 实验设计、数据统计方法与分析结论

2 实验环境

若无特殊说明，基准测试均在以下环境内运行

2.1 硬件环境

CPU	**Intel i7-12700H 6P8E P Core @4.70 GHz E Core@3.50 GHz （在 Linux 内核下，CPU 0-11 为 P 核心，12-19 为 E 核心）**
RAM	64G DDR5 4800MHz Dual Channel

2.2 软件环境

操作系统内核	**Linux kernel 6.8.4**
编译器	GCC 13.2.1 target x86_64-pc-linux-gnu
编译选项（无 SIMD）	`gcc -o convolution ./convolution.c -O3`
编译选项（AVX2）	`gcc -o convolution ./convolution.c -O3 -mavx2`
测试命令	`taskset -c 0-11 /usr/bin/perf stat -d -d -d ./convolution`

3 实验原理及优化思路

以下将介绍本实验的相关背景知识，以及根据研究得出来的相应优化思路。

3.1 SIMD 指令集

通过查阅相关知识，我们学习了关于 SIMD 指令集的相关知识。以下我们将简要介绍与本次实验相关的背景知识，以便后续开展实验操作。

SIMD（Single Instruction, Multiple Data）指令集是一种计算机处理器支持的并行计算指令集。它允许一条指令同时操作多个数据元素，从而实现高效的并行计算。SIMD 本质上是采用一个控制器来控制多个处理器，同时对一组数据中的每一条分别执行相同的操作，从而实现空间上的并行性的技术。

SIMD 架构的计算机之所以能够并行化执行四个浮点数（甚至更多）操作的原因是支持 SIMD 指令的 CPU 在设计时增加了一些专用的向量寄存器。SIMD 向量寄存器的长度往往大于通用寄存器，比如 SEE 的 XMM寄存器的长度为 128 位，AVX 和 AVX2 的 YMM 寄存器为 256 位。因此，这些专用的向量寄存器可以同时放入多个数据。但需要注意，这里放入的多个数据需要保证数据类型是一致的。

目前，大多数 CPU 已经支持了 MMX、SSE/SSE2/SSE3/SSE4/SSE5、AVX、AVX2等指令集。具体查看电脑支持何种指令集可以用 CPU-Z 等软件或者是在 Linux 中，键入 lscpu 来查看 CPU 的基础信息。

其中，不同的指令集有不同的数据类型、向量寄存器及其指令函数（intrinsic function），以下只做简要介绍：

- 数据类型

1. SSE 有三种类型定义 _m128, _m128d 和 _m128i，分别用以表示单精度浮点型、双精度浮点型和整型。
2. AVX/AVX2 有三种类型定义 _m256, _m256d 和 _m256i，分别用以表示单精度浮点型、双精度浮点型和整型。

- 向量寄存器

1. SSE 和 AVX 各自有16个寄存器，SSE 的16个寄存器为 XMM0 - XMM15，XMM 是 128 位寄存器，而 YMM 是 256 位寄存器。XMM 寄存器也可以用于使用类似 x86-SSE 的单精度值或者双精度值执行标量浮点运算。
2. 支持 AVX 的x86-64处理器包含 16 个 256 位大小的寄存器，名为 YMM0 ~ YMM15 。每个 YMM 寄存器的低阶 128 位的别名是相对应的XMM寄存器。大多数 AVX 指令可以使用任何一个 XMM 或者 YMM 寄存器作为 SIMD 操作数。

- Intrinsic Function

Intrinsic function 类似于 high level 的汇编，开发者可以无痛地将 instinsic function 同 C/C++ 的高级语言特性（如分支、循环、函数和类）无缝衔接。因为篇幅原因，在此只对 AVX2 指令集中常用的 instinsic function 做出介绍，以便后续实验操作。

instinsic function	**操作**
_mm256_setzero_si256	生成一个 256 位的向量，其中所有位都被设置为 0
_mm256_loadu_si256	从内存地址加载 256 位（或 32 字节）的数据到 256 位的 SIMD 寄存器
_mm256_set1_epi32	将一个 32 位整数复制到 256 位 SIMD 寄存器的所有元素中

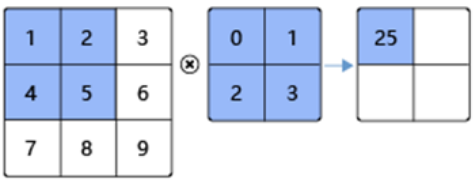
instinsic function	操作
_mm256_mullo_epi32	用于执行两个 256 位 SIMD 寄存器中的 32 位整数的逐元素乘法
_mm256_storeu_si256	用于将一个 256 位的 SIMD 寄存器的内容存储到内存中
_mm256_add_epi32	用于执行两个 256 位 SIMD 寄存器中的对应 32 位整数的加法
_mm256_hadd_epi32	用于执行两个 256 位 SIMD 寄存器中的 32 位整数的水平加法（相邻两个数的加法）
_mm256_extract_epi32	用于从一个 256 位 SIMD 寄存器中提取一个 32 位整数

利用 SIMD 指令集进行程序优化主要是将代码重写为向量操作的形式，以便利用 SIMD 指令并行处理数据。其中要将数据以向量的形式存储，一般要确保数据在内存中的地址对齐，以避免因未对齐访问而引起的性能损失，例如在矩阵运算时，按照先行索引后列索引的方式访问内存地址，提取一行的数据（因为在内存中是连续存放的）比先按列后按行索引访问要更快。之后利用指令对整个向量进行相应数据操作。利用该操作可以同时对多个数据元素执行相同的操作，通过并行性来提高性能。

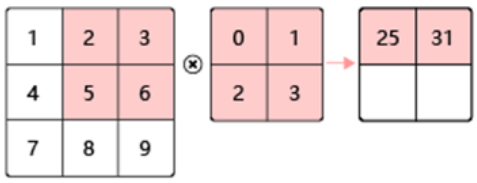
3.2 卷积运算

关于卷积运算，实验指导书中已经介绍的较为详细，以下只对其原理做简要介绍，主要结合代码分析原代码的实现操作。

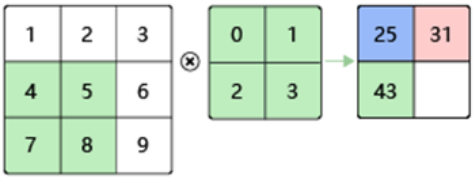
本实验考虑的是两个矩阵之间进行的运算，其中一个是“输入矩阵”，规模较大，另一个是“核”，规模较小。卷积执行过程中，从输入矩阵的左上角开始。核覆盖输入矩阵的一部分，这部分与核的大小相匹配。然后，将核的每个元素与其下方输入矩阵相对应位置的元素相乘，并将这些乘积求和，结果作为卷积操作的结果的一部分存储在一个新的“输出矩阵”中。然后滑动“核”，覆盖新的部分元素，重复这个过程并将结果存储在输出矩阵的新的位置。其运行过程如下图所示：



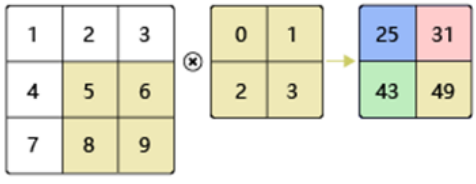
(a) 0x1+1x2+2x4+3x5=25



(b) 0x2+1x3+2x5+3x6=31



(c) 0x4+1x5+2x7+3x8=43



(d) 0x5+1x6+2x8+3x9=49

下面结合实验相关原代码进行分析。代码也是按照图中所示顺序，从左上往右下移动卷积核进行卷积运算。首先对图像（规模较大的输入矩阵）中的每个像素进行迭代，之后对卷积核中的每个元素进行迭代，并进行卷积运算，完成一次就存到结果矩阵的相应位置，之后移动卷积核。以下是原代码添加注释后的详细解释：

```
//原代码
// 定义卷积函数，它接受一个图像、一个卷积核和一个结果数组作为参数
void convolve66(const int* image, size_t image_rows, size_t image_cols,
```

```
    const int* kernel, size_t kernel_rows, size_t kernel_cols,
    int* result) {
    // 对图像中的每个像素进行迭代
    for (size_t i = 0; i <= image_rows - kernel_rows; ++i) { // 先按照行从上往下进行迭代
        for (size_t j = 0; j <= image_cols - kernel_cols; ++j) { // 再按照列从左往右进行迭代
            // 初始化求和变量
            int sum = 0;
            // 对卷积核中的每个元素进行迭代，并进行相应的乘加操作
            for (size_t ki = 0; ki < kernel_rows; ++ki) { // 先按照行从上往下进行迭代
                for (size_t kj = 0; kj < kernel_cols; ++kj) { // 再按照列从左往右进行迭代
                    // 将图像的当前像素与卷积核的当前元素相乘，然后加到求和变量上
                    sum += image[(i + ki) * image_cols + (j + kj)] * kernel[ki * kernel_cols + kj];
                }
            }
            // 进行完一个卷积核的迭代，将求和结果存储在结果数组中
            result[i * (image_cols - kernel_cols + 1) + j] = sum;
        }
    }
}
```

3.3 优化思路

根据上述分析，在进行卷积运算的时候，卷积核与输入矩阵是按照先行后列，从左上往右下的顺序依次进行的，此时无论是卷积核还是输入矩阵中的数据在内存中都是连续排列的。因此我们可以从每一行的开始取出一片连续的内存地址加载到 SIMD 的向量寄存器中，即把矩阵中的数据加载到 SIMD 指令集的向量中，对一行的所有元素进行整体操作。之后再在下一行进行相同的操作。由于该实验中，卷积核的规模是 8*8 大小的矩阵，而在 AVX2 指令集中，正好可以对 256-bit 的数据进行相应的向量操作，也就是 8 个 32-bit 的整数，一个向量对应卷积核的一行，每个卷积核操作中由上往下通过 SIMD 指令，对一行的数据进行卷积运算，之后由上往下进行累加，最后再对一行向量的数据累加得到最后结果。之后移动卷积核，进行下一个卷积运算。

也就是说，优化的关键是拆除最内层 for 循环，转化成将矩阵一整行数据的存储到 256-bit 的向量中进行整体运算。

4 实验过程

以下会介绍我们的 SIMD 指令集优化、多线程与 SIMD 指令集结合优化、GPU 优化等多种优化过程。其中包括一些“其他优化方式”（包括但不限于利用程序的时间、空间局部性进行程序算法的优化），会有所介绍。

4.1 利用 SIMD 指令集优化

根据上述优化思路，我们利用 AVX2 指令集实现了 SIMD 指令集优化。

4.1.1 简单的正确性测试

在进行 SIMD 编程的过程中，会遇到各种小问题，我们需要 debug 进行解决。在这里我们编写了一个 debug 打印向量的代码，用来进行 debug 和分析，方便处理正确性问题，发现问题和解决问题。使用的时候直接将需要 debug 的向量作为参数传入函数即可。

```
void print_m256i(__m256i vec) {
    int values[8];
    _mm256_storeu_si256((__m256i*)values, vec);
    printf("Values: ")
    for (int i = 0; i < 8; ++i) {
        printf("%d ", values[i]);
    }
    printf("\n");
}
```

可以利用该代码进行后续操作的正确性测试。

4.1.2 初步实现

卷积核的移动过程不变，即总体 for 循环的迭代顺序不变。在进行卷积核元素的迭代之前，首先利用 _mm256_setzero_si256() 指令，初始化求和变量为 256-bit 长的向量，之后取消内层 for 循环遍历卷积核的每一行中的元素，改为取卷积核以及输入矩阵每一行的第一个元素的地址，再利用 _mm256_loadu_si256() 指令，从两个地址分别取连续 256-bit 长度的地址，加载到两个 SIMD 256-bit 寄存器中，即将每行的所有数据加载到 8 个 32-bit 整数的向量中，然后利用 _mm256_mullo_epi32() 指令，将这两个向量整体进行对应元素的相乘操作，完成后再由上往下每层迭代以上操作，最后利用 _mm256_hadd_epi32() 指令将 sum 向量中的每个元素相加求和，利用 _mm256_storeu_si256() 指令存储到结果数据矩阵相应位置中。具体实现代码如下：

```
void convolve(const int* image, size_t image_rows, size_t image_cols,
    const int* kernel, size_t kernel_rows, size_t kernel_cols,
    int* result) {
    // 对图像中的每个像素进行迭代
    for (size_t i = 0; i <= image_rows - kernel_rows; ++i) {
        for (size_t j = 0; j <= image_cols - kernel_cols; ++j) {
            // 初始化求和变量
            __m256i sum = _mm256_setzero_si256();
            // 对卷积核中的每个元素进行迭代
            for (size_t ki = 0; ki < kernel_rows; ++ki) {
                // 将图像的当前像素与卷积核的当前元素相乘，然后加到求和变量上
                const int* image_ptr = &image[(i + ki) * image_cols + j];
                const int* kernel_ptr = &kernel[ki * kernel_cols];
```

```
                // 加载数据到 SIMD 寄存
                __m256i image_vec = _mm256_loadu_si256((__m256i*)image_ptr);
                __m256i kernel_vec =
_mm256_loadu_si256((__m256i*)kernel_ptr);
                __m256i prod = _mm256_mullo_epi32(image_vec, kernel_vec);
                sum = _mm256_add_epi32(sum, prod);
            }
            // 将求和结果存储在结果数组中
            sum = _mm256_hadd_epi32(sum, sum);
            sum = _mm256_hadd_epi32(sum, sum);
            sum = _mm256_hadd_epi32(sum, sum);
            _mm256_storeu_si256((__m256i*)(&result[i * (image_cols -
kernel_cols + 1) + j]), sum);
        }
    }
}
```

4.1.3 初步实现分析

以上代码实现了初步的 SIMD 指令集优化，该优化在初步运行分析中，正确性没有问题，运算速度大约可以实现 2 倍的加速（严格分析见后续测试）。但我们在后续运行分析中，发现该代码仍有可以优化的空间。例如在最后进行向量的元素求和中，我们利用 _mm256_hadd_epi32() 指令将向量中的相邻元素两两求和，重复三次该操作，一共要进行 3*4 次加和操作，且每次操作都需要等待上条指令执行完毕，无法进行流水线优化，而且在循环判定条件中有多次运算操作增加指令数，同时，还可以通过循环展开进一步提高并行性进行优化。

4.1.4 最终优化

根据初步实现后的分析，我们决定进一步优化。首先我们将最后的向量的元素求和操作从之前的整体向量累加改为先将向量的元素赋值到数组中，再对数组中的元素进行依次累加，且通过循环展开，每次处理两个元素，以提高运行速度，这样运算次数降低到 8 次且指令具有并行性；同时，我们在取地址的时，进行运算得到地址后直接将该地址加载到寄存器中，而不是先取值再传址，造成不必要的开销；最后，我们将`image_rows - kernel_rows`提前运算好，在循环判断中直接利用该值，以避免循环过程中每次判断时不必要的计算带来的指令数增加而造成开销。以下是修改之后的代码：

```
void convolve(const int* image, size_t image_rows, size_t image_cols,
    const int* kernel, size_t kernel_rows, size_t kernel_cols,
    int* result) {
    // 对图像中的每个像素进行迭代
    int sum_array[8];
    size_t size= image_rows - kernel_rows ;
    for (size_t i = 0; i <= size; ++i) {
        for (size_t j = 0; j <= size; ++j) {
            // 初始化求和变量
            __m256i sum = _mm256_setzero_si256();
            // 对卷积核中的每个元素进行迭代
            for (size_t ki = 0; ki < kernel_rows; ++ki) {
                // 将图像的当前像素与卷积核的当前元素相乘，然后加到求和变量上
                // 加载数据到 SIMD 寄存
                __m256i image_vec = _mm256_loadu_si256((__m256i*)(image + (i
+ ki) * image_cols + j));
                __m256i kernel_vec = _mm256_loadu_si256((__m256i*)(kernel +
ki * kernel_cols));
                __m256i prod = _mm256_mullo_epi32(image_vec, kernel_vec);
```

```
                sum = _mm256_add_epi32(sum, prod);
            }
            // 将求和结果存储在结果数组中
            int final_sum = 0;
            _mm256_storeu_si256((__m256i*)sum_array, sum);
            for (int k = 0; k < 8; k+=2) {
                sum_array[k] += sum_array[k + 1];
                final_sum += sum_array[k];
            }
            result[i * (size + 1) + j] = final_sum;
        }
    }
}
```

4.1.5 最终优化分析

使用以下 `main()` 函数进行初步性能测试：

```
#define IMAGE_ROWS 2048
#define IMAGE_COLS 2048
// 将图像大小设置为 2048x2048
#define KERNEL_ROWS 8
#define KERNEL_COLS 8
#define RESULT_ROWS (IMAGE_ROWS - KERNEL_ROWS + 1)
#define RESULT_COLS (IMAGE_COLS - KERNEL_COLS + 1)
int main() {
    int kernel[KERNEL_ROWS][KERNEL_COLS] = {
        {1, 2, 1, 2, 1, 2, 1, 2},
        {2, 1, 2, 1, 2, 1, 2, 1},
        {1, 2, 1, 2, 1, 2, 1, 2},
        {1, 2, 1, 2, 1, 2, 1, 2},
        {1, 2, 1, 2, 1, 2, 1, 2},
        {1, 2, 1, 2, 1, 2, 1, 2},
        {1, 2, 1, 2, 1, 2, 1, 2},
        {1, 2, 1, 2, 1, 2, 1, 2}
    };
    srand(time(NULL));
    for (int i = 0; i < IMAGE_ROWS; ++i) {
        for (int j = 0; j < IMAGE_COLS; ++j) {
            image[i][j] = rand() % 256; // 使用0-255之间的模式填充图像
        }
    }
    convolve((int *)image, IMAGE_ROWS, IMAGE_COLS, (int *)kernel,
KERNEL_ROWS, KERNEL_COLS, (int *)result);
    long long R=0;
    for (int i = 0; i < RESULT_ROWS; ++i) {
        for (int j = 0; j < RESULT_COLS; ++j) {
            R+=result[i][j];
        }
    }
    printf("%lld\n", R); // 防止整个函数被编译器优化掉
    return 0;
}
```

编译，运行`taskset -c 0-11 /usr/bin/perf stat -d -d -d ./convolution`：

```
 Performance counter stats for './convolution':

             77.04 msec task-clock:u                     #    0.992 CPUs utilized
                 0      context-switches:u               #    0.000 /sec
                 0      cpu-migrations:u                 #    0.000 /sec
             7,715      page-faults:u                    #  100.142 K/sec
     <not counted>      cpu_atom/cycles/u                                             (0.00%)
       303,679,551      cpu_core/cycles/u                #    3.942 GHz                    (30.06%)
     <not counted>      cpu_atom/instructions/u                                       (0.00%)
       502,296,623      cpu_core/instructions/u                                       (37.83%)
     <not counted>      cpu_atom/branches/u                                           (0.00%)
        98,781,664      cpu_core/branches/u              #    1.282 G/sec                  (45.62%)
     <not counted>      cpu_atom/branch-misses/u                                      (0.00%)
           178,808      cpu_core/branch-misses/u                                      (53.41%)
             TopdownL1 (cpu_core)                 #     62.0 %  tma_backend_bound
                                                  #      0.8 %  tma_bad_speculation
                                                  #      3.1 %  tma_frontend_bound
                                                  #     34.1 %  tma_retiring          (61.20%)
     <not counted>      L1-dcache-loads:u                                             (0.00%)
       124,288,173      L1-dcache-loads:u                #    1.613 G/sec                  (68.99%)
   <not supported>      L1-dcache-load-misses:u
         1,745,500      L1-dcache-load-misses:u                                       (69.21%)
     <not counted>      LLC-loads:u                                                   (0.00%)
             5,271      LLC-loads:u                      #   68.419 K/sec                  (73.11%)
     <not counted>      LLC-load-misses:u                                             (0.00%)
             5,703      LLC-load-misses:u                                             (77.00%)
     <not counted>      L1-icache-loads:u                                             (0.00%)
```

```
   <not supported>      L1-icache-loads:u

     <not counted>      L1-icache-load-misses:u
                    (0.00%)
             4,292      L1-icache-load-misses:u
                    (31.01%)
     <not counted>      dTLB-loads:u
                    (0.00%)
       148,852,367      dTLB-loads:u                  #    1.932 G/sec
                    (30.79%)
     <not counted>      dTLB-load-misses:u
                    (0.00%)
            18,414      dTLB-load-misses:u
                    (26.89%)
   <not supported>      iTLB-loads:u

   <not supported>      iTLB-loads:u

     <not counted>      iTLB-load-misses:u
                    (0.00%)
               291      iTLB-load-misses:u
                    (23.00%)
   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetch-misses:u

   <not supported>      L1-dcache-prefetch-misses:u


       0.077655416 seconds time elapsed

       0.073958000 seconds user
       0.003362000 seconds sys
```

对比无 SIMD 优化的性能:

```
 Performance counter stats for './convolution':

           117.01 msec task-clock:u                     #    0.995 CPUs
utilized
                0      context-switches:u             #    0.000 /sec

                0      cpu-migrations:u               #    0.000 /sec

              557      page-faults:u                  #    4.760 K/sec

     <not counted>      cpu_atom/cycles/u
                    (0.00%)
       518,618,018      cpu_core/cycles/u              #    4.432 GHz
                    (30.80%)
     <not counted>      cpu_atom/instructions/u
                    (0.00%)
```

```
     1,972,722,785      cpu_core/instructions/u
                       (38.49%)
     <not counted>      cpu_atom/branches/u
                       (0.00%)
       252,702,884      cpu_core/branches/u              #    2.160 G/sec
                       (46.16%)
     <not counted>      cpu_atom/branch-misses/u
                      (0.00%)
           173,241      cpu_core/branch-misses/u
                      (53.85%)
             TopdownL1 (cpu_core)                 #     35.4 %
tma_backend_bound
                                                  #      0.8 %
tma_bad_speculation
                                                  #      3.9 %
tma_frontend_bound
                                                  #     59.9 %  tma_retiring
            (61.54%)
     <not counted>      L1-dcache-loads:u
                       (0.00%)
       330,713,506      L1-dcache-loads:u                #    2.826 G/sec
                       (69.23%)
   <not supported>      L1-dcache-load-misses:u

         1,841,798      L1-dcache-load-misses:u
                       (70.26%)
     <not counted>      LLC-loads:u
                       (0.00%)
             2,361      LLC-loads:u                      #   20.177 K/sec
                       (72.83%)
     <not counted>      LLC-load-misses:u
                       (0.00%)
             2,036      LLC-load-misses:u
                       (75.39%)
     <not counted>      L1-icache-loads:u
                       (0.00%)
   <not supported>      L1-icache-loads:u

     <not counted>      L1-icache-load-misses:u
                       (0.00%)
               991      L1-icache-load-misses:u
                       (30.77%)
     <not counted>      dTLB-loads:u
                      (0.00%)
       384,217,307      dTLB-loads:u                     #    3.284 G/sec
                       (29.74%)
     <not counted>      dTLB-load-misses:u
                      (0.00%)
                99      dTLB-load-misses:u
                      (27.17%)
   <not supported>      iTLB-loads:u

   <not supported>      iTLB-loads:u

     <not counted>      iTLB-load-misses:u
                      (0.00%)
```

```
                93      iTLB-load-misses:u
                   (24.61%)
   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetches:u

   <not supported>      L1-dcache-prefetch-misses:u

   <not supported>      L1-dcache-prefetch-misses:u


       0.117559937 seconds time elapsed

       0.113778000 seconds user
       0.003314000 seconds sys
```

可见 CPU Instuction 数量在优化后仅为优化前的 25.46%，但由于 SIMD 指令 Latency 较高与 cache miss 等因素，优化后的 cycles 为优化前的 58.55% 。在 -O3 编译器优化等级下，编译器会自动进行循环展开等，由于关键计算已经被向量化，所以在不对原始代码进行自动向量化的情况下，编译器自动向量化对本实验影响较小。

4.2 其它优化

除了利用 AVX2 指令集优化外，我们还尝试做了其他方面的优化，具体情况如下。

4.2.1 利用多线程与 SIMD 指令集优化

SIMD的优化方式，在相同思路下还有如下代码，对于寄存器具有更强的节约性，使得更加适用于多线程。基本思路相同的，就是将卷积核的数据一行八个作为数据传入256的向量寄存器，从而实现以八个为一次的计算从而达到加速的效果。

```
void convolve_better(const int* image, size_t image_rows, size_t image_cols,
                     const int* kernel, size_t kernel_rows, size_t
kernel_cols,
                     int* result) {
    for (size_t i = 0; i <= image_rows - kernel_rows; ++i) {
        for (size_t j = 0; j <= image_cols - kernel_cols; ++j) {
            __m256i sum = _mm256_setzero_si256();
            for (size_t k = 0; k < kernel_rows; ++k) {
              //读取image和kernel的值
              __m256i image_val = _mm256_loadu_si256((__m256i*)(image +
(i+k) * image_cols + j));
              __m256i kernel_val = _mm256_loadu_si256((__m256i*)(kernel + k
* kernel_cols));
              sum = _mm256_add_epi32(sum, _mm256_mullo_epi32(image_val,
kernel_val));
            }
            int sum_array[8];
            _mm256_storeu_si256((__m256i*)sum_array, sum);
            int final_sum = 0;
            for (int k = 0; k < 8; k++) {
                final_sum += sum_array[k];
            }
            result[i * (image_cols - kernel_cols + 1) + j] = final_sum;
```

```
  }
}
}
```

将以上代码进行多线程编写。最初的思路是想将所有的k内循环做多线程，后来发现这样太过频繁开启线程，不利于优化。我们改变思路，将结果的卷积平分为线程数的份数，然后对这些值进行多线程计算。其中最后一份，以result_row-1为值。这样能够保证线程不被频繁的开启，而且还能够进行优化。

简单来说，我们把所有的工作平分给了各个线程。

```
 size_t start_i = t * rows_per_thread;
 size_t end_i = (t == num_threads - 1) ? (image_rows - kernel_rows) :
(start_i + rows_per_thread - 1);
```

我们使用这个结构体来传递数据。

```
typedef struct {
    const int* image;
    size_t image_rows;
    size_t image_cols;
    const int* kernel;
    size_t kernel_rows;
    size_t kernel_cols;
    int* result;
    size_t start_i;
    size_t end_i;
} ThreadArgs;
```

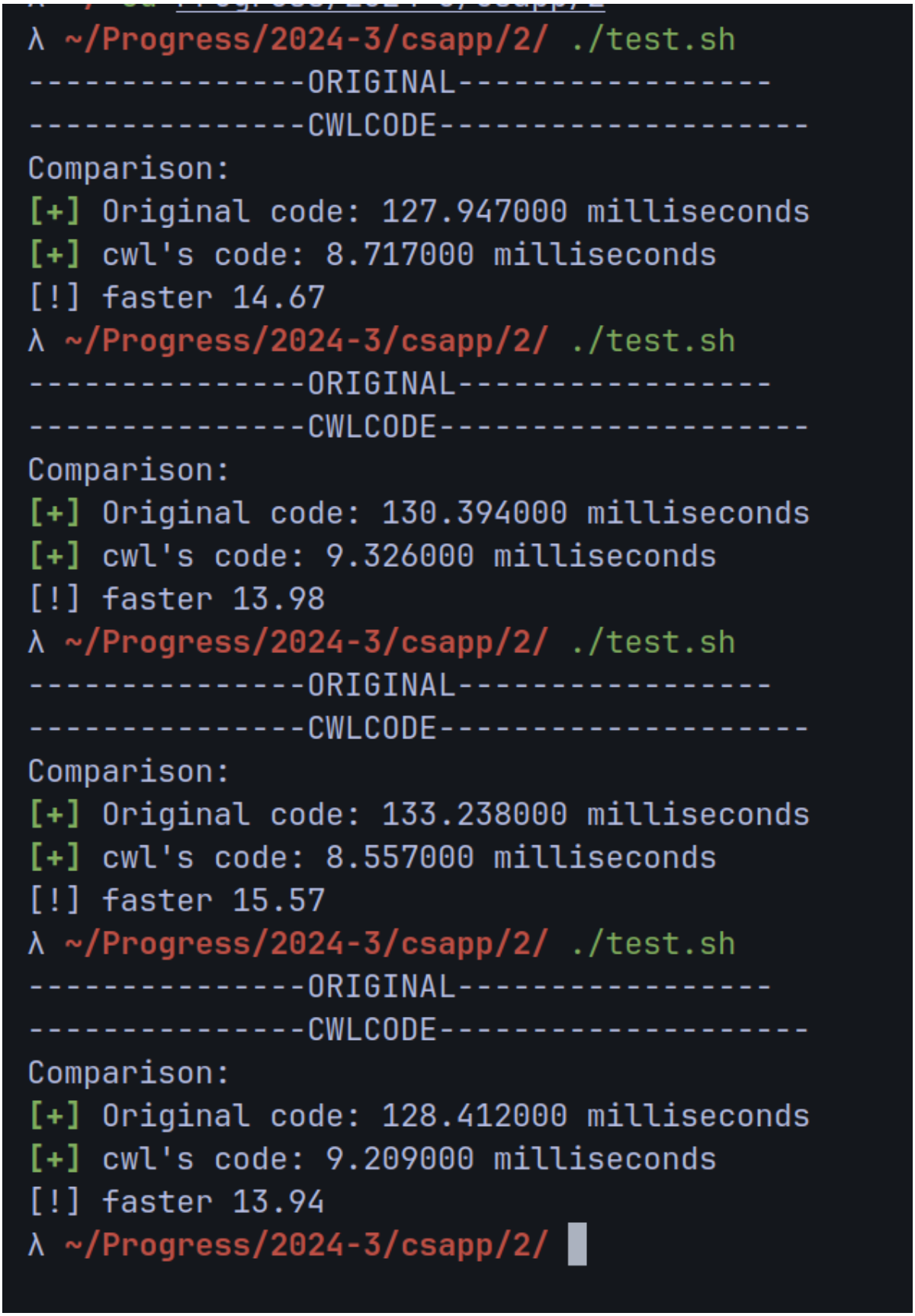
我们编写测试脚本进行测试：

```
echo "---------------ORIGINAL-----------------"
original_output=$(gcc -g -O3 -mavx2 bhr.c -o bhr && ./bhr)
rm original.log
echo "$original_output" >>original.log
original_time=$(echo "$original_output" | grep -oE '[0-9]+\.[0-9]+' | head
-1)


echo "---------------CWLCODE--------------------"
newcode_output=$(gcc -g -O3 -mavx2 ./cwl_thread.c -o cwl && ./cwl)
rm cwl.log
echo "$newcode_output" >>cwl.log
newcode_cwltime=$(echo "$newcode_output" | grep -oE '[0-9]+\.[0-9]+' | head
-1)

echo "Comparison:"
echo -e "\033[1;32m[+] \033[0mOriginal code: $original_time milliseconds"
# echo -e "\033[1;32m[+] \033[0mbhr's code: $newcode_time milliseconds"
echo -e "\033[1;32m[+] \033[0mcwl's code: $newcode_cwltime milliseconds"
# echo -e "\033[1;32m[+] \033[0mzay's code: $newcode_zaytime milliseconds"
# 计算加速比(original_time和newcode_cwltime的比值)
echo "[!] faster $(echo "scale=2; $original_time / $newcode_cwltime" | bc)"
```

不使用O3，初步测试这样的代码比原始代码加速14～15倍。



使用O3，我们初步测试加速为7～8倍。

```
λ ~/Progress/2024-3/csapp/2/ ./test.sh
---------------ORIGINAL-----------------
---------------CWLCODE------------------
Comparison:
[+] Original code: 11.891000 milliseconds
[+] cwl's code: 1.588000 milliseconds
[!] faster 7.48
λ ~/Progress/2024-3/csapp/2/ ./test.sh
---------------ORIGINAL-----------------
---------------CWLCODE------------------
Comparison:
[+] Original code: 12.668000 milliseconds
[+] cwl's code: 1.775000 milliseconds
[!] faster 7.13
λ ~/Progress/2024-3/csapp/2/ ./test.sh
---------------ORIGINAL-----------------
---------------CWLCODE------------------
Comparison:
[+] Original code: 12.895000 milliseconds
[+] cwl's code: 1.667000 milliseconds
[!] faster 7.73
```

最后我们的完整代码如下，最终实现了相应的加速。

```c
#include <stdio.h>
#include <immintrin.h>
#include <stdlib.h>
#include <sys/time.h>
#include <pthread.h>

#define IMAGE_ROWS 1024
#define IMAGE_COLS 1024
#define KERNEL_ROWS 8
#define KERNEL_COLS 8
#define RESULT_ROWS (IMAGE_ROWS - KERNEL_ROWS + 1)
#define RESULT_COLS (IMAGE_COLS - KERNEL_COLS + 1)

typedef struct {
    const int* image;
    size_t image_rows;
    size_t image_cols;
    const int* kernel;
    size_t kernel_rows;
    size_t kernel_cols;
    int* result;
    size_t start_i;
```

```
    size_t end_i;
} ThreadArgs;

void* convolve_thread(void* arg) {
    ThreadArgs* args = (ThreadArgs*)arg;
    const int* image = args->image;
    size_t image_rows = args->image_rows;
    size_t image_cols = args->image_cols;
    const int* kernel = args->kernel;
    size_t kernel_rows = args->kernel_rows;
    size_t kernel_cols = args->kernel_cols;
    int* result = args->result;
    size_t start_i = args->start_i;
    size_t end_i = args->end_i;

    for (size_t i = start_i; i <= end_i; ++i) {
        for (size_t j = 0; j <= image_cols - kernel_cols; ++j) {
            __m256i sum = _mm256_setzero_si256();
            for (size_t k = 0; k < kernel_rows; ++k) {
                __m256i image_val = _mm256_loadu_si256((__m256i*)(image + (i + k) * image_cols + j));
                __m256i kernel_val = _mm256_loadu_si256((__m256i*)(kernel + k * kernel_cols));
                sum = _mm256_add_epi32(sum, _mm256_mullo_epi32(image_val, kernel_val));
            }
            int sum_array[8];
            _mm256_storeu_si256((__m256i*)sum_array, sum);
            int final_sum = 0;
            for (int k = 0; k < 8; k++) {
                final_sum += sum_array[k];
            }
            result[i * (image_cols - kernel_cols + 1) + j] = final_sum;
        }
    }

    pthread_exit(NULL);
}

void convolve_better(const int* image, size_t image_rows, size_t image_cols,
                     const int* kernel, size_t kernel_rows, size_t kernel_cols,
                     int* result) {
    const int num_threads = 8; // 设置线程数量
    pthread_t threads[num_threads];
    ThreadArgs thread_args[num_threads];

    size_t rows_per_thread = (image_rows - kernel_rows + 1) / num_threads;

    for (int t = 0; t < num_threads; ++t) {
        size_t start_i = t * rows_per_thread;
        size_t end_i = (t == num_threads - 1) ? (image_rows - kernel_rows) : (start_i + rows_per_thread - 1);
        printf("start %d  end %d\n", start_i, end_i);

```

```
        thread_args[t].image = image;
        thread_args[t].image_rows = image_rows;
        thread_args[t].image_cols = image_cols;
        thread_args[t].kernel = kernel;
        thread_args[t].kernel_rows = kernel_rows;
        thread_args[t].kernel_cols = kernel_cols;
        thread_args[t].result = result;
        thread_args[t].start_i = start_i;
        thread_args[t].end_i = end_i;

        pthread_create(&threads[t], NULL, convolve_thread,
(void*)&thread_args[t]);
    }

    for (int t = 0; t < num_threads; ++t) {
        pthread_join(threads[t], NULL);
    }
}

int main() {
    int image[IMAGE_ROWS][IMAGE_COLS];
    int kernel[KERNEL_ROWS][KERNEL_COLS] = {
        {1, 0, -1, 0, 1, 0, -1, 0},
        {1, 0, -1, 0, 1, 0, -1, 0},
        {1, 0, -1, 0, 1, 0, -1, 0},
        {1, 0, -1, 0, 1, 0, -1, 0},
        {1, 0, -1, 0, 1, 0, -1, 0},
        {1, 0, -1, 0, 1, 0, -1, 0},
        {1, 0, -1, 0, 1, 0, -1, 0},
        {1, 0, -1, 0, 1, 0, -1, 0}
    };
    int result[RESULT_ROWS][RESULT_COLS];
    // Transpose the kernel matrix
    // Use a simple pattern to fill the image matrix
    for (int i = 0; i < IMAGE_ROWS; ++i) {
        for (int j = 0; j < IMAGE_COLS; ++j) {
            image[i][j] = (i * IMAGE_COLS + j) % 256; // Use a 0-255
pattern to fill the image
        }
    }
    struct timeval start, end;
    double interval;
    gettimeofday(&start, NULL);
    convolve_better((int *)image, IMAGE_ROWS, IMAGE_COLS, (int *)kernel,
KERNEL_ROWS, KERNEL_COLS, (int *)result);
    // convolve_bhr((int *)image, IMAGE_ROWS, IMAGE_COLS, (int *)kernel,
KERNEL_ROWS, KERNEL_COLS, (int *)result);
    gettimeofday(&end, NULL);
    interval = (end.tv_sec - start.tv_sec) * 1000.0 + (end.tv_usec -
start.tv_usec) / 1000.0;

    // Print the execution time
    printf("Convolution operation time: %f milliseconds\n", interval);
    printf("\n[!] Result matrix:\n");
    // Print the result matrix
    for (int i = 0; i < RESULT_ROWS; ++i) {
```

```
        for (int j = 0; j < RESULT_COLS; ++j) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

4.2.2 利用 GPU 进行优化

4.2.2.1 GPU 尝试思路

由于本次实验是进行卷积的运算，我们不难想到卷积和图形处理有关，而GPU比起CPU更擅长对图形数据的处理，我们可以利用GPU来进行卷积的运算来达到优化的目的

4.2.2.2 代码实现

我们修改convolve函数，使可以在GPU下运行convolve_gpu的新函数

```
#include <stdio.h>
#include <sys/time.h>

#define IMAGE_ROWS 14
#define IMAGE_COLS 14
#define KERNEL_ROWS 8
#define KERNEL_COLS 8
#define RESULT_ROWS (IMAGE_ROWS - KERNEL_ROWS + 1)
#define RESULT_COLS (IMAGE_COLS - KERNEL_COLS + 1)

// 原始的 CPU 版本的 convolve 函数
void convolve_cpu(const int *image, size_t image_rows, size_t image_cols,
                  const int *kernel, size_t kernel_rows, size_t
kernel_cols,
                  int *result) {
    for (size_t i = 0; i <= image_rows - kernel_rows; ++i) {
        for (size_t j = 0; j <= image_cols - kernel_cols; ++j) {
            int sum = 0;
            for (size_t ki = 0; ki < kernel_rows; ++ki) {
                for (size_t kj = 0; kj < kernel_cols; ++kj) {
                    sum += image[(i + ki) * image_cols + (j + kj)] *
kernel[ki * kernel_cols + kj];
                }
            }
            result[i * (image_cols - kernel_cols + 1) + j] = sum;
        }
    }
}

// CUDA 核函数
__global__ void convolve_gpu(const int *image, size_t image_rows, size_t
image_cols,
                             const int *kernel, size_t kernel_rows, size_t
kernel_cols,
                             int *result) {
```

```
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < image_rows - kernel_rows + 1 && j < image_cols - kernel_cols + 1) {
        int sum = 0;
        for (size_t ki = 0; ki < kernel_rows; ++ki) {
            for (size_t kj = 0; kj < kernel_cols; ++kj) {
                sum += image[(i + ki) * image_cols + (j + kj)] * kernel[ki * kernel_cols + kj];
            }
        }
        result[i * (image_cols - kernel_cols + 1) + j] = sum;
    }
}

double getCurrentTime() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

int main() {
    int image[IMAGE_ROWS * IMAGE_COLS];
    int kernel[KERNEL_ROWS * KERNEL_COLS] = {
        1, 0, -1, 0, 1, 0, -1, 0,
        1, 0, -1, 0, 1, 0, -1, 0,
        1, 0, -1, 0, 1, 0, -1, 0,
        1, 0, -1, 0, 1, 0, -1, 0,
        1, 0, -1, 0, 1, 0, -1, 0,
        1, 0, -1, 0, 1, 0, -1, 0,
        1, 0, -1, 0, 1, 0, -1, 0,
        1, 0, -1, 0, 1, 0, -1, 0
    };
    int result_cpu[RESULT_ROWS * RESULT_COLS];
    int result_gpu[RESULT_ROWS * RESULT_COLS];
    int *d_image, *d_kernel, *d_result;

    // 使用简单的模式填充图像矩阵
    for (int i = 0; i < IMAGE_ROWS; ++i) {
        for (int j = 0; j < IMAGE_COLS; ++j) {
            image[i * IMAGE_COLS + j] = (i * IMAGE_COLS + j) % 256; // 使用0-255之间的模式填充图像
        }
    }

    // 分配设备内存
    cudaMalloc((void **)&d_image, IMAGE_ROWS * IMAGE_COLS * sizeof(int));
    cudaMalloc((void **)&d_kernel, KERNEL_ROWS * KERNEL_COLS * sizeof(int));
    cudaMalloc((void **)&d_result, RESULT_ROWS * RESULT_COLS * sizeof(int));

    // 将数据从主机内存复制到设备内存
    cudaMemcpy(d_image, image, IMAGE_ROWS * IMAGE_COLS * sizeof(int), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_kernel, kernel, KERNEL_ROWS * KERNEL_COLS * sizeof(int),
cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((RESULT_COLS + threadsPerBlock.x - 1) /
threadsPerBlock.x,
                   (RESULT_ROWS + threadsPerBlock.y - 1) /
threadsPerBlock.y);

    double start, end;

    // 在 CPU 上运行原始的 convolve 函数并计时
    start = getCurrentTime();
    convolve_cpu(image, IMAGE_ROWS, IMAGE_COLS, kernel, KERNEL_ROWS,
KERNEL_COLS, result_cpu);
    end = getCurrentTime();
    printf("CPU Convolution operation time: %f milliseconds\n", (end -
start) * 1000.0);

    // 调用 CUDA 核函数并计时
    start = getCurrentTime();
    convolve_gpu<<<numBlocks, threadsPerBlock>>>(d_image, IMAGE_ROWS,
IMAGE_COLS,
                                                 d_kernel, KERNEL_ROWS,
KERNEL_COLS,
                                                 d_result);
    cudaDeviceSynchronize();
    end = getCurrentTime();
    printf("GPU Convolution operation time: %f milliseconds\n", (end -
start) * 1000.0);

    // 将结果从设备内存复制到主机内存
    cudaMemcpy(result_gpu, d_result, RESULT_ROWS * RESULT_COLS *
sizeof(int), cudaMemcpyDeviceToHost);

    // 输出结果比较
    printf("\nCPU Convolution Result:\n");
    for (int i = 0; i < RESULT_ROWS; ++i) {
        for (int j = 0; j < RESULT_COLS; ++j) {
            printf("%d ", result_cpu[i * RESULT_COLS + j]);
        }
        printf("\n");
    }

    printf("\nGPU Convolution Result:\n");
    for (int i = 0; i < RESULT_ROWS; ++i) {
        for (int j = 0; j < RESULT_COLS; ++j) {
            printf("%d ", result_gpu[i * RESULT_COLS + j]);
        }
        printf("\n");
    }

    // 释放设备内存
    cudaFree(d_image);
    cudaFree(d_kernel);
    cudaFree(d_result);
```

```
129     return 0;
130 }
```

其核心思路就是分配设备内存，将数据从主机内存复制到设备内存，然后在GPU上进行运行，运行结束后可以释放设备内存。

4.2.2.3 运行结果

我们运行程序，发现结果输出如下

```
chao@GaoZhengChao:~/cuda$ ./convolution_gpu
CPU Convolution operation time: 0.015020 milliseconds
GPU Convolution operation time: 1.977921 milliseconds

CPU Convolution Result:
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32

GPU Convolution Result:
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
-32 -32 -32 -32 -32 -32 -32
sh: 1: paues: not found
```

4.2.2.4 结果分析

我们运行程序不难发现，GPU的运行时间远远大于CPU的运行时间，这不仅另外陷入了深深的思考，我们类比上一次的实验我猜测可能是以下原因：

1. 数据传输时间：在GPU计算之前和计算之后，需要进行数据的传输，包括将输入数据从主机内存复制到设备内存，以及将计算结果从设备内存复制回主机内存。这些数据传输的时间可能在GPU计算时间中占据了很大的比例，尤其是对于像素数目较少的小图像，数据传输所占时间可能更加显著。
2. 并行度不足：尽管GPU具有大量的线程并行计算的能力，但是在这个例子中，你的线程块（blocks）大小为16x16，这可能不足以充分利用GPU的计算资源。特别是，如果你的GPU具有大量的CUDA核心，但是每个线程块中的线程数量较少，可能会导致GPU计算效率较低。
3. 算法实现不优化：尽管使用了GPU并行计算，但是你的算法实现可能并没有充分地利用GPU的优势。例如，尝试使用共享内存来减少全局内存访问，或者尝试使用更高效的算法来执行卷积运算。
4. 内存访问模式：GPU对内存访问的延迟很敏感，如果访问模式不佳，会导致性能下降。在卷积操作中，内存访问通常是不连续的，可能会导致访问效率不高。

综上所述，要解决GPU运行时间远大于CPU运行时间的问题，我认为可以尝试以下方法：

1. 优化数据传输，减少主机和设备之间的数据传输量。
2. 调整线程块大小以更好地利用GPU的并行计算能力。

3. 优化算法实现，以更好地利用GPU的计算资源。

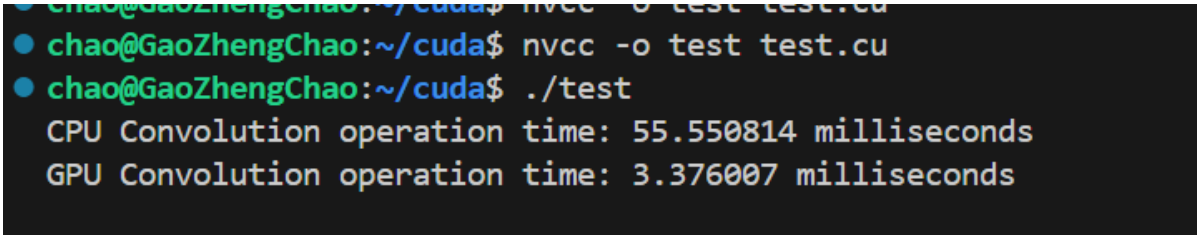
4. 优化内存访问模式，以提高内存访问效率。

4.2.2.5 新的尝试

基于上次实验，我猜测是数据量太小的缘故，我发现原本的矩阵大小是8x8大小的，数据规模十分小，于是我将数据改大为500x500的矩阵。

```
#define IMAGE_ROWS 500
#define IMAGE_COLS 500
```

运行结果如下，我们果然发现在大数据面前，GPU比起CPU对数据的处理更加快捷有效



5 基准测试

为评估优化效果，在多线程实验前，先进行单线程的基准测试。

除 5.5 部分外，其余部分均使用 2048x2048 的 Image。

5.1 环境说明

通过查询资料得知，实验环境中的 CPU 采用了 Intel 的“性能混合架构”，其 14 个核心中，有 6 个属于高性能（主频高，核心规模大，完整的超标量设计，支持超线程）的 P 核心，8个属于性能较低（主频低，核心规模小，超标量设计少，不支持超线程），能耗较低的 E 核心。

在实验中，我们发现：

- 程序并非自始至终运行在同一个核心上，操作系统内核会对其进行核心间的调度
- 同类核心间的调度对程序运行时间影响不大
- 不同核心间的调度对程序的运行时间影响较大，会产生较大的性能开销

5.2 单线程基准性能

- 为保证单线程效率最优，使用 `-O3` 编译器优化，最大程度降低因代码写法和函数调用造成的性能开销
- 使用 `taskset -c 0-11` 实现核心绑定，绑定 NTT 可执行文件到该 CPU 的 P 核心，从而最大程度减少因程序在 P 核心与 E 核心之间调度而造成的性能开销
- 使用 `perf` 工具实现简单的 Profiling，在指令层面测试性能，参数为 `stat -d -d -d ./ntt`
- 最终运行程序的完整命令为：`taskset -c 0-11 /usr/bin/perf stat -d -d -d ./ntt`
- 为保证 CPU 不出现较为严重的过热降频，两次测试之间间隔 5s，共测试 10 次

5.2.1 测试数据

- T：perf 工具采集的 CPU 时间，单位为 ms
- F： perf 工具采集的 CPU 频率，单位为 GHz
- I： perf 工具采集的 CPU 指令数
- C：perf 工具采集的 CPU 周期数
- IPC：Instruction Per Cycle，每个周期执行的指令数，体现指令 Latency

次数	T	F	I	C	IPC
1	117.23	4.488GHz	2,085,870,139	526,163,652	3.96
2	116.63	4.396GHz	1,908,236,033	512,708,690	3.72
3	116.09	4.394GHz	1,881,994,240	510,153,366	3.69
4	116.07	4.377GHz	1,908,471,289	508,020,127	3.76
5	117.56	4.422GHz	2,046,204,177	519,866,634	3.94
6	116.99	4.417GHz	1,993,222,271	516,791,875	3.86

次数	T	F	I	C	IPC
7	122.38	4.181GHz	2,072,068,168	511,642,947	4.05
8	120.21	4.150GHz	2,012,480,969	498,830,764	4.03
9	117.37	4.401GHz	1,974,808,995	516,534,570	3.82
10	119.96	4.166GHz	1,939,442,804	499,800,950	3.88
avg	118.05	4.34	1982279908.50	512051357.50	3.87

5.3 最终优化性能

- T：perf 工具采集的 CPU 时间，单位为 ms
- F： perf 工具采集的 CPU 频率，单位为 GHz
- I： perf 工具采集的 CPU 指令数
- C：perf 工具采集的 CPU 周期数
- IPC：Instruction Per Cycle，每个周期执行的指令数，体现指令 Latency

次数	T	F	I	C	IPC
1	76.02	3.813GHz	528,306,004	289,913,944	1.82
2	71.12	4.422GHz	385,079,653	314,491,586	1.22
3	73.67	3.987GHz	456,454,890	293,716,099	1.55
4	71.18	4.301GHz	465,015,054	306,158,747	1.52
5	72.63	4.135GHz	408,014,340	300,330,166	1.36
6	72.34	4.191GHz	392,112,900	303,138,686	1.29
7	72.29	4.144GHz	414,704,785	299,553,286	1.38
8	70.56	4.323GHz	375,516,391	305,068,929	1.23
9	74.87	3.878GHz	506,310,443	290,297,536	1.74
10	74.98	3.918GHz	466,228,341	293,797,295	1.59
avg	72.97	4.11	439774280.10	299646627.40	1.47

5.4 初步结论

观察以上基准测试数据，发现 SIMD 可以大幅减少 CPU 的指令数和周期数，但是也会大幅劣化 IPC，猜测有以下原因：

- 虽然本次实验中使用的 CPU 是超标量设计，但是 SIMD 指令仍需要更多的 CPU 资源，同时由于用于测试的 CPU 未关闭超线程，这些资源在其他地方也有需求，那么使用SIMD可能会导致资源竞争，从而降低IPC。
- SIMD指令通常需要大量的数据。本次实验使用的内存为 DDR5 4800MHz，内存带宽约为 60G/s，内存带宽有可能不足以满足需求，CPU 花费了更多的时间等待数据，从而降低 IPC。

同时观察最终优化的 perf 数据：

```
Performance counter stats for './convolution':

             74.87 msec task-clock:u                     #    0.992 CPUs
utilized
                 0      context-switches:u               #    0.000 /sec
                 0      cpu-migrations:u                 #    0.000 /sec
             8,222      page-faults:u                    #  109.824 K/sec
     <not counted>      cpu_atom/cycles/u
                       (0.00%)
       290,297,536      cpu_core/cycles/u                #    3.878 GHz
                       (27.96%)
     <not counted>      cpu_atom/instructions/u
                       (0.00%)
       506,310,443      cpu_core/instructions/u
                       (35.88%)
     <not counted>      cpu_atom/branches/u
                       (0.00%)
        98,797,978      cpu_core/branches/u              #    1.320 G/sec
                       (43.90%)
     <not counted>      cpu_atom/branch-misses/u
                      (0.00%)
           179,760      cpu_core/branch-misses/u
                      (51.91%)
             TopdownL1 (cpu_core)                 #     61.2 %
tma_backend_bound
                                                  #      0.8 %
tma_bad_speculation
                                                  #      3.1 %
tma_frontend_bound
                                                  #     34.9 %  tma_retiring
            (59.93%)
       123,770,444      L1-dcache-loads:u                #    1.653 G/sec
                       (67.94%)
         1,625,661      L1-dcache-load-misses:u
                       (69.86%)
               932      LLC-loads:u                      #   12.449 K/sec
                       (73.87%)
               625      LLC-load-misses:u
                       (77.88%)
             3,665      L1-icache-load-misses:u
                       (32.06%)
       150,763,503      dTLB-loads:u                     #    2.014 G/sec
                       (30.14%)
                57      dTLB-load-misses:u
                      (26.13%)

               122      iTLB-load-misses:u
                      (22.12%)
```

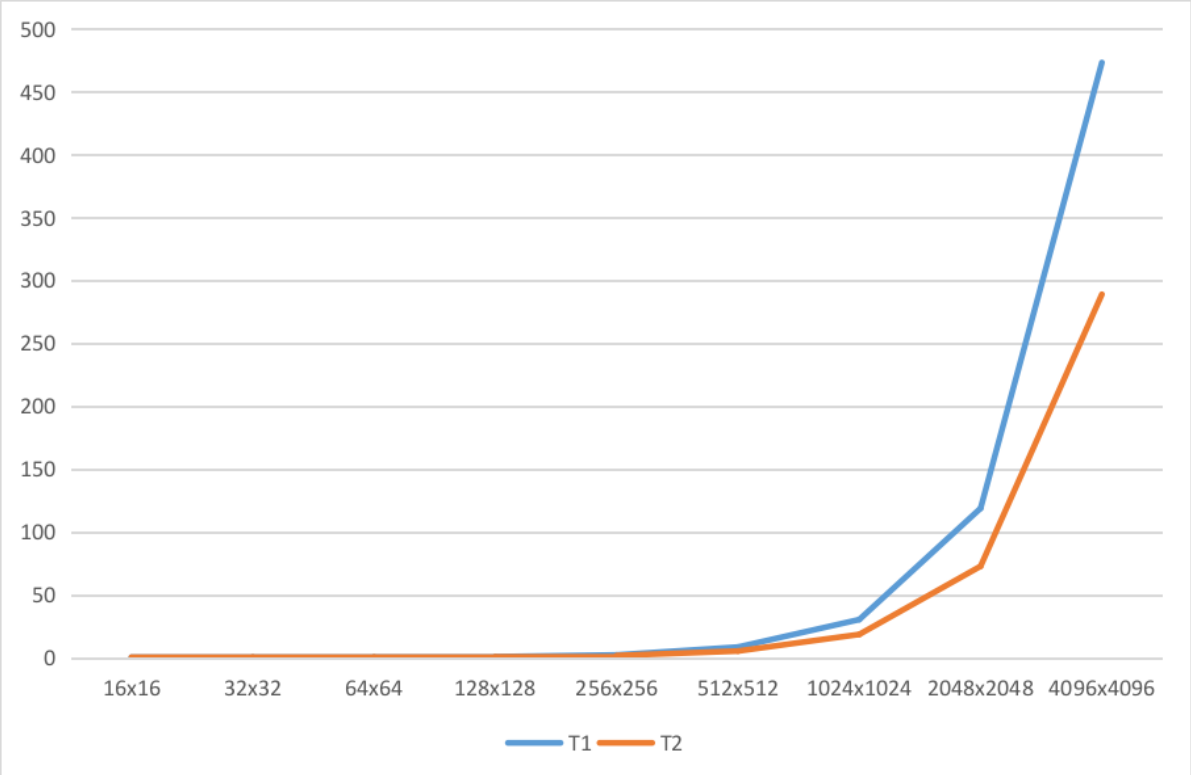
可以发现 cache miss 率较低，同时分支预测失败率较低，同时根据性能计数器数据， perf 工具推算出后端瓶颈为 61.2%，我们推测优化后程序主要为 Memory_Bound，即内存读写是瓶颈。最终优化后的程序已经能够较为充分地利用 CPU 的执行单元。

5.5 单线程与优化后性能对比，不同 Image 大小下的性能

从 16x16 开始，每次将 Image 的两边长加倍，分别使用无优化和最终优化两种方式运行，计算运行时间：

- Size：Image 的大小
- T1：无优化的运行时间，单位为 ms ，下同
- T2：最终优化的运行时间

Size	T1	T2	T1/T2
16x16	0.30	0.32	0.94
32x32	0.34	0.33	1.03
64x64	0.42	0.37	1.14
128x128	0.77	0.62	1.24
256x256	2.18	1.66	1.31
512x512	8.43	5.25	1.61
1024x1024	30.28	18.47	1.64
2048x2048	118.76	72.68	1.63
4096x4096	473.19	288.89	1.64
10	74.98	#REF!	



可见随 Image Size 增大，T2 与 T1 差值增大，但 T1/T2 逐渐保持稳定，最终稳定在 1.6 左右。

5.6 Profiling

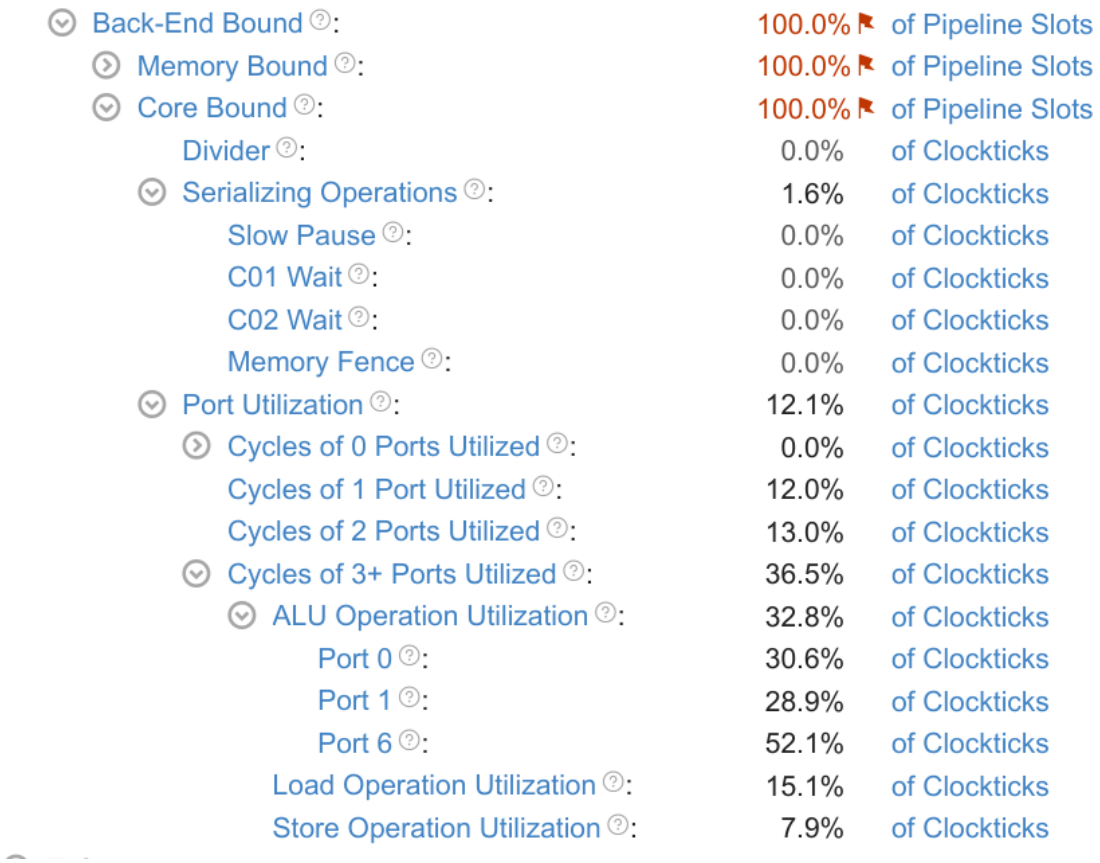
本次实验主要使用 SIMD 指令集，因此使用 Intel VTune 进行 Microarchitecture 分析：

> **Elapsed Time: 4.899s**
>
> | | | |
> |---|---|---|
> | Clockticks: | 20,401,200,000 | |
> | Instructions Retired: | 45,762,300,000 | |
> | CPI Rate: | 0.446 | |
> | P-Core: | | |
> | Retiring: | 100.0% | of Pipeline Slots |
> | Front-End Bound: | 25.5% | of Pipeline Slots |
> | Bad Speculation: | 0.0% | of Pipeline Slots |
> | Back-End Bound: | 100.0% | of Pipeline Slots |
> | E-Core: | | |
> | Average CPU Frequency: | 4.3 GHz | |
> | Total Thread Count: | 3 | |
> | Paused Time: | 0s | |
>
> **N/A is applied to metrics with undefined value. There is no data to calculate the metric.*

观察到主要为 Back-End Bound：

Back-End Bound:	100.0%	of Pipeline Slots
Memory Bound:	100.0%	of Pipeline Slots
L1 Bound:	25.8%	of Clockticks
DTLB Overhead:	0.2%	of Clockticks
Loads Blocked by Store Forwarding:	0.1%	of Clockticks
Lock Latency:	22.4%	of Clockticks
Split Loads:	6.4%	of Clockticks
4K Aliasing:	0.0%	of Clockticks
FB Full:	1.5%	of Clockticks
L2 Bound:	0.3%	of Clockticks
L3 Bound:	0.2%	of Clockticks
DRAM Bound:	0.2%	of Clockticks
Store Bound:	2.2%	of Clockticks
Core Bound:	100.0%	of Pipeline Slots

在 Memory Bound 中，主要为 L1 Bound 中的 Lock Latency，说明优化后仍存在资源竞争和数据依赖性，导致性能瓶颈，可以在访存上做进一步优化。



在 Core Bound 中 Cycles of 3+ Ports Utilized 较高，即需要 3 个以上 uop 的指令使用较多，因此该优化能够有效利用 ALU 等核心资源，达到了使用 SIMD 优化的目的，符合使用 SIMD 优化的原理。

6 实验结果与优化分析

根据 基准测试中的数据，计算得出加速比：

$$\frac{118.05}{72.97} \approx 1.62$$

相比于理想的加速结果（加速比为 4~8），该加速比并不高，通过 Profiling 得知，造成该现象的原因主要是存在资源竞争和数据依赖性，导致性能瓶颈，要想继续提高加速比，应该在访存顺序上继续优化，继续减少 cache miss，提高 cache 利用率。

SIMD 指令以较高的 Latency 为代价实现了更高吞吐量，使得 CPU 在常数个 cycle 内可以处理更多数据。但对于本题的场景，两行之间的访存会造成一定的 cache miss，从而导致性能提升不及预期。在 SIMD 相关的实现中，需要重视访存顺序，在配合多线程的 SIMD 实现中，需要考虑 CPU 本身超线程带来的核内微架构资源竞争问题。

7 参考资料

1. https://zhuanlan.zhihu.com/p/591900754
2. https://www.cnblogs.com/MrSaver/p/10356293.html

8 附录

1. simd.c SIMD 指令集的AVX2优化最终实现代码
2. cwl_thread.c 指令集与多线程优化实现代码
3. my_convolution.c SIMD 指令集优化的SSE优化
4. convolution_gpu.cu 卷积运算的GPU优化