《汇编与接口技术》课程

实验报告

实验名称： **Arm64 研究性实验之 perspectiveTransform 优化**

姓　　名： **衡勇睿（22281067）**

万绍文（22281285）

李辰璋（22301092）

实验分工： 全部流程（代码和报告编写）

均三人共同完成

一、实验概况

实验目的：我们要进行的实验是利用 arm64 汇编优化 OpenCV 库中的 perspectiveTransform 函数，使得加速比达到 2 倍以上。

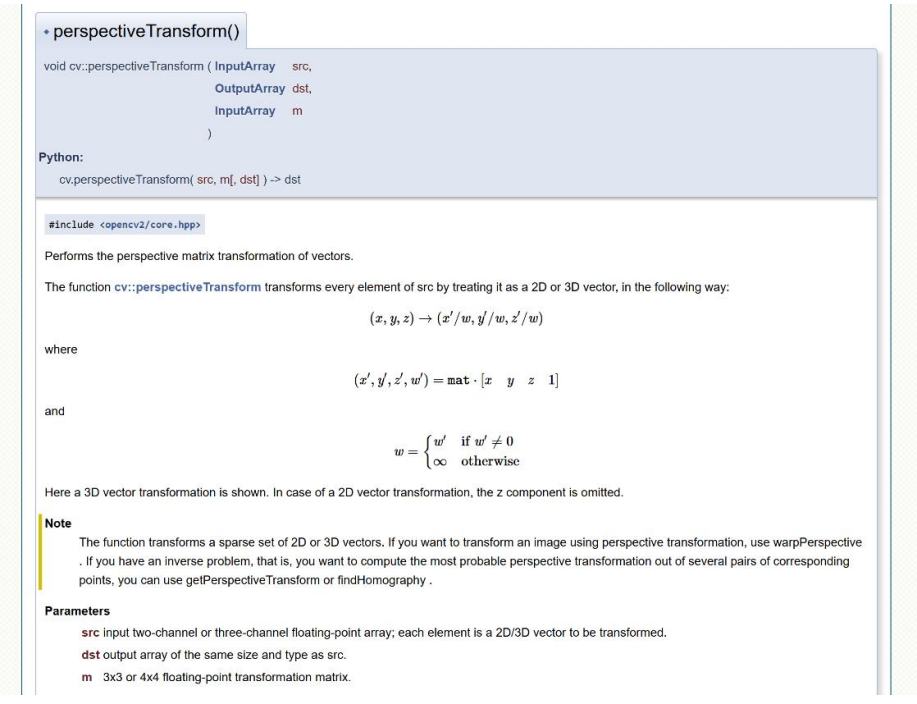
实验原理：perspectiveTransform 函数是对一组三维或二维的坐标进行仿射变换操作，通俗来说，就是进行了一个矩阵乘法操作。我们小组主要打算利用 arm64 的内存连续读写以及 SIMD 指令，来对矩阵乘法内部 for 循环语句的效率进行优化。

二、实验环境

本研究型实验将在华为泰山服务器上进行，华为泰山服务器的处理器为鲲鹏 920，其采用了 Arm64 指令集设计。

三、实验过程

3.1 函数功能说明



• perspectiveTransform()

void cv::perspectiveTransform (InputArray src,
OutputArray dst,
InputArray m
)

Python:
cv.perspectiveTransform(src, m[, dst]) -> dst

#include <opencv2/core.hpp>

Performs the perspective matrix transformation of vectors.

The function cv::perspectiveTransform transforms every element of src by treating it as a 2D or 3D vector, in the following way:

$$(x, y, z) \to (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot [x \quad y \quad z \quad 1]$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

Here a 3D vector transformation is shown. In case of a 2D vector transformation, the z component is omitted.

Note
The function transforms a sparse set of 2D or 3D vectors. If you want to transform an image using perspective transformation, use warpPerspective . If you have an inverse problem, that is, you want to compute the most probable perspective transformation out of several pairs of corresponding points, you can use getPerspectiveTransform or findHomography .

Parameters
src input two-channel or three-channel floating-point array; each element is a 2D/3D vector to be transformed.
dst output array of the same size and type as src.
m 3x3 or 4x4 floating-point transformation matrix.

图 3.1.1 OpenCV 中的 perspectiveTransform 函数

数学原理：perspectiveTransform 函数的功能是将一组二维或三维坐标与一个权重矩阵相乘得到一组新的坐标。具体公式如下：

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

其中：

$$(x', y', z', w') = mat \cdot [\text{x} \quad \text{y} \quad \text{z} \quad 1]$$

算法过程：本实验中，我们仅考虑较复杂的情况，即对三维坐标进行运算。所以算法过程为输入一个 n*3 的源矩阵和一个 4*4 的权重矩阵，进行维度变化及矩阵乘法操作后，得到一个 n*3 的结果矩阵。

参数形式：perspectiveTransform_arm(float32_t (*src)[3], float32_t (*dest)[3], float32_t (*mat)[4], uint32_t num)

测试用例：我们在 C 语言中采用随机生成矩阵的方式，因此我们不在这里给出具体的测试样例。此外，我们与传统方式实现矩阵乘法的计算结果来对比，来验证 arm 汇编优化算法的正确性并计算加速比（C 语言主程序见附录）。

3.2 优化流程

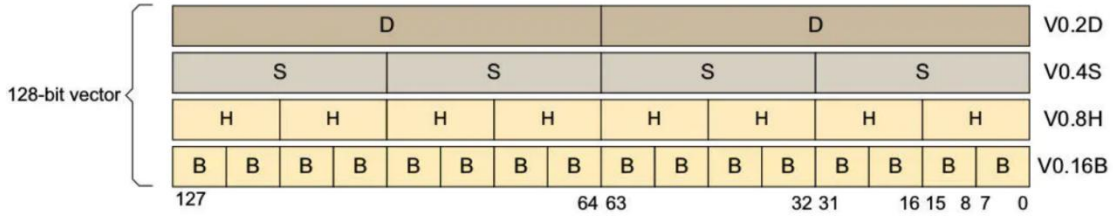
3.2.1 最初思路

我们考虑 4×4 矩阵乘法的运算过程：

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \cdot \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} x & y & z & w \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \\ \mathbf{a}_4 \end{bmatrix}$$

$$= x \cdot \mathbf{a}_1 + y \cdot \mathbf{a}_2 + z \cdot \mathbf{a}_3 + w \cdot \mathbf{a}_4$$

由此分析，我们只需要实现向量的数量乘法与向量加法运算即可完成任务。我们知道 Aarch64 中的 NEON 单元专用于支持增强型单指令批量运算(SIMD)，NEON 单元为我们提供了 32 个 128bit 的向量寄存器，这些寄存器可以支持符合 IEEE754 标准的浮点数运算与 SIMD 操作。此处一个 128bit 的向量寄存器刚好可以用于存储 4 个 32bit 的 float 类型用于向量运算，这就是设计的核心思路。



SIMD 中提供了现成的向量按位加与向量按位乘运算的指令。为了实现向量的数量乘法，我们考虑将一个单独的操作数复制四份再进行按位乘法：

$$x \cdot \mathbf{a} = \begin{bmatrix} x \cdot a_1 & x \cdot a_2 & x \cdot a_3 & x \cdot a_4 \end{bmatrix}$$

$$= \begin{bmatrix} x & x & x & x \end{bmatrix} \odot \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix}$$

于是我们的汇编代码设计方案就可以确定：

按行遍历所有的待转换点坐标，读取其各坐标分量

↓

将点的坐标分量值复制 4 份并存入一个向量寄存器中

↓

从变换矩阵中取出一列存入一个向量寄存器中

↓

两个寄存器执行按位乘法，并将结果累加至另外一个向量寄存器中

↓

对每个坐标分量重复上述过程

↓

将最后得到的累加结果从寄存器写入内存中的对应位置即可

以下为最初的汇编代码设计：

```
.section .text
.global perspectiveTransform_arm_realization
perspectiveTransform_arm_realization:

step:
    mov x4, x1
    mov x6, #4
    mov x7, #16
    ld1 {v0.s}[0], [x4], x6
    ld1 {v0.s}[1], [x4], x6
```

```
    ld1 {v0.s}[2], [x4], x6
    ld1 {v0.s}[3], [x4], x6

    mov x4, x0
    mov x5, x2
    ld1 {v1.s}[0], [x5], x7
    ld1 {v1.s}[1], [x5], x7
    ld1 {v1.s}[2], [x5], x7
    ld1 {v1.s}[3], [x5], x7

    ld1 {v2.s}[0], [x4]
    ld1 {v2.s}[1], [x4]
    ld1 {v2.s}[2], [x4]
    ld1 {v2.s}[3], [x4], x6

    fmul v1.4s, v1.4s, v2.4s
    fadd v0.4s, v0.4s, v1.4s

    mov x5, x2
    add x5, x5, #4
    ld1 {v1.s}[0], [x5], x7
    ld1 {v1.s}[1], [x5], x7
    ld1 {v1.s}[2], [x5], x7
    ld1 {v1.s}[3], [x5], x7

    ld1 {v2.s}[0], [x4]
    ld1 {v2.s}[1], [x4]
    ld1 {v2.s}[2], [x4]
    ld1 {v2.s}[3], [x4], x6

    fmul v1.4s, v1.4s, v2.4s
    fadd v0.4s, v0.4s, v1.4s

    mov x5, x2
    add x5, x5, #8
    ld1 {v1.s}[0], [x5], x7
    ld1 {v1.s}[1], [x5], x7
    ld1 {v1.s}[2], [x5], x7
    ld1 {v1.s}[3], [x5], x7

    ld1 {v2.s}[0], [x4]
    ld1 {v2.s}[1], [x4]
    ld1 {v2.s}[2], [x4]
    ld1 {v2.s}[3], [x4], x6
```

```
    fmul v1.4s, v1.4s, v2.4s
    fadd v0.4s, v0.4s, v1.4s

    mov x5, x2
    add x5, x5, #12
    ld1 {v1.s}[0], [x5], x7
    ld1 {v1.s}[1], [x5], x7
    ld1 {v1.s}[2], [x5], x7
    ld1 {v1.s}[3], [x5], x7

    ld1 {v2.s}[0], [x4]
    ld1 {v2.s}[1], [x4]
    ld1 {v2.s}[2], [x4]
    ld1 {v2.s}[3], [x4], x6

    fmul v1.4s, v1.4s, v2.4s
    fadd v0.4s, v0.4s, v1.4s

    st1 {v0.s}[0], [x1], x6
    st1 {v0.s}[1], [x1], x6
    st1 {v0.s}[2], [x1], x6
    st1 {v0.s}[3], [x1], x6

    mov x0, x4

    sub x3, x3, #1
    cmp x3, #0
    bne step

    ret
```

可以看到，在上述代码设计中我们的所有内存读写操作都是单字操作，即一次操作只从内存中读取一个字存入向量寄存器的对应位置；并且只将向量寄存器中的一个 32 位浮点数写入内存。这样的访存操作无疑是低效的，在访存上的优化将极大改善我们代码的性能，这也为我们后续的优化方案指明了方向。

3.2.2 最终思路

这里先给出具体代码，后面进行具体分析。

arm.S 代码：

```
.section .text
.global perspectiveTransform_arm
perspectiveTransform_arm:

    ld4 {v2.4s, v3.4s, v4.4s, v5.4s}, [x2]    // Load matrix M

step:
    mov w5, #0
    dup v1.4s, w5    // clear

    ld1 {v6.4s},[x0]    // load src[i][0...2]
    add x0, x0, #12

    dup v0.4s, v6.s[0]
    fmul v0.4s, v0.4s, v2.4s
    fadd v1.4s, v1.4s, v0.4s


    dup v0.4s, v6.s[1]
    fmul v0.4s, v0.4s, v3.4s
    fadd v1.4s, v1.4s, v0.4s


    dup v0.4s, v6.s[2]
    fmul v0.4s, v0.4s, v4.4s
    fadd v1.4s, v1.4s, v0.4s

    fadd v1.4s, v1.4s, v5.4s

    mov w6,v1.s[3]
    cmp w6,#0
    bne jp1
    dup v1.4s, w5
    b jp2
jp1:
    dup v7.4s,v1.s[3]
    fdiv v1.4s, v1.4s, v7.4s

jp2:
    cmp x3, #1
    bne jp3
    st1 {v1.s}[0], [x1],#4   // store dest[i][0...2]
    st1 {v1.s}[1], [x1],#4
```

```
    st1 {v1.s}[2], [x1],#4
    ret


jp3:
    st1 {v1.4s},[x1]
    add x1, x1, #12

    sub x3, x3, #1
    cmp x3, #0
    bne step

    ret
```

在与初版代码的对比上，优化主要体现在以下地方：

(1)在循环外把变换矩阵 M 提前 load 进 v2.4s，v3.4s，v4.4s，v5.4s 中，避免在循环内对 M 进行读操作。这样相比原版有两点优化：第一、终版相比原版代码显著减少了访存次数；第二、终版代码使用 1 条 ld4 指令（每次读 16 个字节）读入了整个 M 矩阵，而原版为了实现同样的操作，需使用 4 条 ld1 指令（每次读 4 字节）。ld4 相比 ld1 更快，原因将在(2)中详细说明。

```
mov x5, x2
ld1 {v2.s}[0], [x4], x6
ld1 {v2.s}[1], [x4], x6
ld1 {v2.s}[2], [x4], x6
ld1 {v2.s}[3], [x4], x6
```

图 3.2.1 原版读 M 矩阵的方式（循环内）

```
ld4 {v2.4s, v3.4s, v4.4s, v5.4s}, [x2]    // Load matrix M
```

图 3.2.2 终版读 M 矩阵的方式（循环外）

(2)终版尽量一次读写 16 个字节（即 4 个 32 位浮点数），原版代码每次读/写内存都是以一个字节为单位进行的，其访存效率较低。可以采用一次读/写 16 个字节的方法，充分利用内存突发传输方式的优势，同时提高 cache 命中率。

```
mov x4, x1
mov x6, #4
ld1 {v4.s}[0], [x4], x6
ld1 {v5.s}[0], [x4], x6
ld1 {v6.s}[0], [x4], x6
ld1 {v7.s}[0], [x4], x6
```

```
st1 {v4.s}[0], [x1], x6
st1 {v5.s}[0], [x1], x6
st1 {v6.s}[0], [x1], x6
st1 {v7.s}[0], [x1], x6
```

图 3.2.3 原版一次读/写 4 个字节

```
ld1 {v6.4s},[x0]    // load src[i][0...2]
add x0, x0, #12
```

```
st1 {v1.4s},[x1]
add x1, x1, #12
```

图 3.2.4 终版一次读/写 16 个字节

ld1 {v6.4s},[x0]，st1 {v6.4s},[x0]可以一次读/写 16 个字节的内存数据，其读/写内存的连续性非常高，可以有效降低访存延时。

但这里有一个问题，我们在每个循环内只对 3 个 32 位浮点数进行处理，使用上述指令会多读一个 32 位浮点数，所以每次地址 x0 只加 12；这相当于 v6.s[3]是多余的，但是即使多余了，仍旧比原版一个一个数读快很多。

(3)终版代码利用 SIMD 指令一次性处理多元素数据，尽量使用并行操作；例如 dup 指令将一个标量复制到四个向量寄存器中，之后再利用 SIMD 中的 fadd 和 fmul 指令进行并行计算，提高了数据处理效率；而原版没有使用 dup 指令，只能机械地从内存中一个一个读数，而这 4 个数在内存中又是不连续的，所以只能“跳读”，这无疑是费时的。实验表明，用 dup 指令替换 ld1 和 ins 可以带来很大的性能提升。

```
mov x4, x0
mov x5, x2
ld1 {v1.s}[0], [x5], x6
ld1 {v1.s}[1], [x5], x6
ld1 {v1.s}[2], [x5], x6
ld1 {v1.s}[3], [x5], x6

ld1 {v2.s}[0], [x4]
ld1 {v2.s}[1], [x4]
ld1 {v2.s}[2], [x4]
ld1 {v2.s}[3], [x4], x6

fmul v1.4s, v1.4s, v2.4s
fadd v0.4s, v0.4s, v1.4s
```

图 3.2.5 原版未使用 dup 指令

```
dup v0.4s, v6.s[0]
fmul v0.4s, v0.4s, v2.4s
fadd v1.4s, v1.4s, v0.4s
```

图 3.2.6 终版使用 dup 指令

综上，我们重写的汇编代码相比 C 语言源码主要有以下方面的优化：（1）在循环外提前将 M 矩阵读入向量寄存器，之后需要访问 M 矩阵时直接从向量寄存器中获取数据即可，显著减少访存次数；（2）批量读写内存，一次读写多个字节；（3）利用 SIMD 中提供的向量按位加与向量按位乘运算的指令，进行并行的加法、乘法运算。

下面两图分别是对 C 语言原始代码和汇编代码运算过程的可视化。

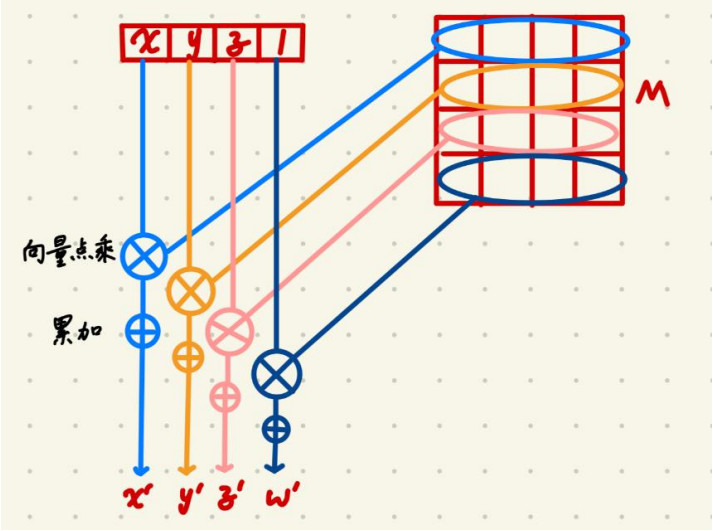


图 3.2.7 C 语言代码流程

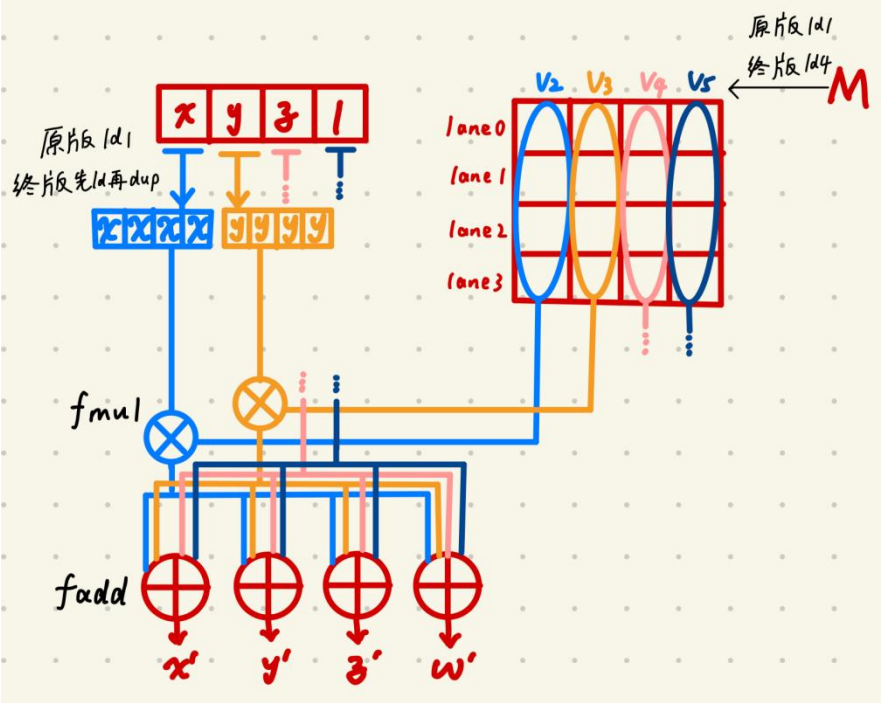


图 3.2.8 arm 汇编代码流程

3.3 运行结果

```
The Acceleration ratio: 8.622
The MSE: 0.000000
```

图 3.3.1 终版代码运行结果（截取部分）

从实验结果可知，均方误差为 0，说明我们的优化结果是正确的。同时，加速比提升至 8.6 左右，说明性能较初版代码得到巨大提升，优化非常成功（完整的输出示例见附录 2）。

四、疑难困惑及解决方式

在此次实验过程中，我们主要遇到了以下几种困难：

1. 运行时越界报错：

```
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

经调试，我们发现报错原因来源于以下指令：

```
st1 {v1.4s},[x1]
add x1, x1, #12
```

图 4.1 没有考虑越界情况的代码

前面我们提到，st1 {v6.4s},[x0]可以一次写 16 个字节的内存数据，而我们的实际功能只对 3 个 32 位浮点数进行处理，使用上述指令会多写一个 32 位浮点数，即将 v6.s[3]写入内存是多余的操作。这一个多写的操作在最后一个循环会造成数组越界，是很危险的行为（在除了最后一个循环的其他循环不会有问题，因为多写的部分还在 dest 内存的内部，接下来的循环会把多写的地方覆盖掉；但是在最后一个循环多写的 32 位不会被覆盖）。所以这里需要一个特判，即如果是最后一个循环，改用分 3 次写 4 字节的方式，如下图所示。

```
jp2:
    cmp x3, #1
    bne jp3
    st1 {v1.s}[0], [x1],#4   // store dest[i][0...2]
    st1 {v1.s}[1], [x1],#4
    st1 {v1.s}[2], [x1],#4
    ret


jp3:
    st1 {v1.4s},[x1]
    add x1, x1, #12

    sub x3, x3, #1
    cmp x3, #0
    bne step
```

图 4.2 考虑越界情况的代码（加入特判）

2. 向量寄存器使用问题

我们在本次实验中反复使用的 v 寄存器，本质上使用是 128 位 NEON SIMD 寄存器，这意味着如果操作 32 位浮点数，可同时操作 4 个 s 单元；如果操作 16 位整数（short），可同时操作 8 个 h 单元；而如果操作 8 位整数，则可同时操作 16 个 b 单元。

这就意味着，如果不能很好理解 SIMD 中各种寄存器的宽度，进行操作时很

容易出现位数不匹配的问题而报错。比方说，本次实验中我需要将数值广播到 v 寄存器 4 个 32 位单元中，而我预先将这个数值存入 x 寄存器，再使用 dup 指令，就出现了报错（如下图所示）

```
arm.S: Assembler messages:
arm.S:9: Error: operand mismatch -- `dup v1.4s,x5'
arm.S:9: Info:    did you mean this?
arm.S:9: Info:          dup v1.4s, w5
```

图 4.3 报错信息

而将这个数值送入 w 寄存器（x 寄存器的低 32 位）或某个 v 寄存器的 s 通道（如 v0.s[0]）,就不会出现这个问题。

五、实验心得体会

开展研究性学习的目的之一是培养学生自学一门新语言的能力，但是对于计算机专业的普通学生而言，这项任务是富有挑战性的。更何况网络上的教程质量参差不齐，往往令人难以理解。

在本次研究性学习实践中，网络上几乎难以找到关于 Aarch64 中 SIMD 指令的系统化教程，大多数都是使用高级语言的方式实现(例如使用 C 语言的 intrinsic 方式)。我们在实验中的早期实践往往都是胡乱尝试、盲人摸象，即使是偶然蒙对了结果，也是知其然而不知其所以然，这样的学习是极其低效的。

在我们看来，对于有一定编程基础的人而言，学习一门新语言最快的方法就是通过实例学习。例如在 Aarch64 HelloWorld 的代码示例中，我们可以了解到如何通过 arm 汇编分离实现函数，如何在 C 语言中调用汇编函数以及传递函数参数；在 memorycopy()函数优化的示例中，我们可以学习如何实现内存与寄存器之间的数据传输，以及如何通过循环展开的方式优化代码执行效率；在矩阵乘法优化的示例中，我们可以了解到如何使用 Aarch64 NEON 单元所提供的向量寄存器进行批量浮点数运算，如何指定数据类型，如何使用函数的前缀修饰符和后缀修饰符，以及如何操作向量寄存器的指定通道；在后续

的例子中我们也了解了 C 语言内嵌汇编代码的基本实现方式等等。

总之，这次实验让我们受益匪浅，不仅提升了我们的代码能力，更重要的是教会了我们自学一门陌生编程语言的能力。我们会利用好这段宝贵经历，在未来的学习和工作中取得更大的进步。

附录 1：

C 语言代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <arm_neon.h>
#include <string.h>
#include <time.h>
#include <math.h>

#define eps 1e-6
#define NUM_POINTS 5000

void perspectiveTransform_raw(float32_t (*src)[3], float32_t (*dest)[3],
float32_t (*mat)[4], uint32_t num) {
    for (int i=0;i<num;i++){
        for (int j=0;j<3;j++){
            float32_t des = 0.0;
            for (int k=0;k<3;k++){
                des += src[i][k]*mat[j][k];
            }
            des += mat[j][3];
            dest[i][j] = des;
        }
        float32_t w = 0;
        for (int k=0;k<3;k++){
            w += src[i][k]*mat[3][k];
        }
        w += mat[3][3];

        if (w<-eps||w>eps) {
            dest[i][0] /= w;
            dest[i][1] /= w;
            dest[i][2] /= w;
        } else {
```

```
            dest[i][0]=0.0;
            dest[i][1]=0.0;
            dest[i][2]=0.0;
        }
    }

    return;
}

extern void perspectiveTransform_arm(float32_t (*src)[3], float32_t
(*dest)[3], float32_t (*mat)[4], uint32_t num);
float32_t getMSE(float32_t (*a)[3], float32_t (*b)[3]);


int main() {
    srand((unsigned)time(NULL));

    //Stopwatch
    struct timespec start_time, end_time;
    double time_before, time_after;

    //This is a test case:
    float32_t original_points[NUM_POINTS][3];
    for (int i=0;i<NUM_POINTS;i++) {
        int x=rand()%1000000, y=rand()%1000000, z=rand()%1000000;
        original_points[i][0]=(float32_t)x/1000.0;
        original_points[i][1]=(float32_t)y/1000.0;
        original_points[i][2]=(float32_t)z/1000.0;
    }
    float32_t transform_matrix[4][4];
    for (int i=0;i<4;i++) {
        int x=rand()%1000000, y=rand()%1000000, z=rand()%1000000,
w=rand()%1000000;
        transform_matrix[i][0]=(float32_t)x/1000000.0;
        transform_matrix[i][1]=(float32_t)y/1000000.0;
        transform_matrix[i][2]=(float32_t)z/1000000.0;
        transform_matrix[i][3]=(float32_t)w/1000000.0;
    }
    float32_t transformed_points[NUM_POINTS][3];
    float32_t transformed_points2[NUM_POINTS][3];
    printf("Transform matrix:\n");
    for (int i=0;i<4;i++)
        printf("%.2f, %.2f, %.2f, %.2f\n", transform_matrix[i][0],
transform_matrix[i][1], transform_matrix[i][2], transform_matrix[i][3]);
```

```
    printf("-------------------------------------------------------\n
");

    //perspectiveTransform_raw
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    perspectiveTransform_raw(original_points, transformed_points,
transform_matrix, NUM_POINTS);
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    printf("Points before transform:\n");
    for (int i=0;i<10;i++)
        printf("(%.2f, %.2f, %.2f)\n", original_points[i][0],
original_points[i][1], original_points[i][2]);
    printf("...\n");
    printf("Points after transform:\n");
    for (int i=0;i<10;i++)
        printf("(%.2f, %.2f, %.2f)\n", transformed_points[i][0],
transformed_points[i][1], transformed_points[i][2]);
    printf("...\n");
    time_before=end_time.tv_nsec-start_time.tv_nsec;
    printf("Time consumed by raw perspective transformation: %3lf ns\n",
time_before);
    printf("-------------------------------------------------------\n
");

    //perspectiveTransform_arm
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    perspectiveTransform_arm(original_points, transformed_points2,
transform_matrix, NUM_POINTS);
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    printf( "Points before transform:\n");
    for (int i=0;i<10;i++)
        printf("(%.2f, %.2f, %.2f)\n", original_points[i][0],
original_points[i][1], original_points[i][2]);
    printf( "...\n");
    printf( "Points after transform:\n");
    for (int i=0;i<10;i++)
        printf("(%.2f, %.2f, %.2f)\n", transformed_points2[i][0],
transformed_points2[i][1], transformed_points2[i][2]);
    printf("...\n");
    time_after=end_time.tv_nsec-start_time.tv_nsec;
    printf("Time consumed by arm perspective transformation: %3lf ns\n",
time_after);
```

```
    printf("------------------------------------------------------------\n
");
    printf("The Acceleration ratio: %.3lf\n", time_before/time_after);
    printf("The MSE: %.6f\n",
getMSE(transformed_points,transformed_points2));
    return 0;
}

float32_t getMSE(float32_t (*a)[3], float32_t (*b)[3]){
    float32_t mse = 0;
    for(int i = 0;i<NUM_POINTS;i++){
        for(int j = 0;j<3;j++){
            mse += (a[i][j] - b[i][j])*(a[i][j] - b[i][j]);
        }
    }
    mse /= 3*NUM_POINTS;
    return mse;
}
```

附录 2：

输出结果

```
Transform matrix:
0.46, 0.70, 0.55, 0.52
0.07, 0.41, 0.10, 0.49
0.01, 0.30, 0.41, 0.87
0.93, 0.85, 0.10, 0.57
------------------------------------------------------------
Points before transform:
(179.94, 940.63, 180.14)
(1.00, 95.26, 723.81)
(1.00, 15.71, 966.59)
(1.00, 194.84, 775.55)
(1.00, 115.61, 319.66)
(1.00, 333.05, 400.93)
(1.00, 847.07, 991.59)
(1.00, 538.53, 189.04)
(1.00, 454.82, 519.96)
(1.00, 240.10, 343.33)
...
Points after transform:
(0.85, 0.43, 0.36)
```

```
(3.02, 0.71, 2.11)
(4.93, 0.91, 3.63)
(2.31, 0.64, 1.54)
(1.96, 0.60, 1.26)
(1.40, 0.55, 0.81)
(1.39, 0.55, 0.80)
(1.01, 0.51, 0.50)
(1.38, 0.55, 0.79)
(1.49, 0.56, 0.89)
...
Time consumed by raw perspective transformation: 535679.000000 ns
------------------------------------------------------------
Points before transform:
(179.94, 940.63, 180.14)
(1.00, 95.26, 723.81)
(1.00, 15.71, 966.59)
(1.00, 194.84, 775.55)
(1.00, 115.61, 319.66)
(1.00, 333.05, 400.93)
(1.00, 847.07, 991.59)
(1.00, 538.53, 189.04)
(1.00, 454.82, 519.96)
(1.00, 240.10, 343.33)
...
Points after transform:
(0.85, 0.43, 0.36)
(3.02, 0.71, 2.11)
(4.93, 0.91, 3.63)
(2.31, 0.64, 1.54)
(1.96, 0.60, 1.26)
(1.40, 0.55, 0.81)
(1.39, 0.55, 0.80)
(1.01, 0.51, 0.50)
(1.38, 0.55, 0.79)
(1.49, 0.56, 0.89)
...
Time consumed by arm perspective transformation: 62130.000000 ns
------------------------------------------------------------
The Acceleration ratio: 8.622
The MSE: 0.000000
```