第 2 章 Chapter 2

ARM NEON SIMD 指令优化

目前在移动平台上，主流的处理器是 ARM。ARM 既是一家公司的名称，也是该公司设计的系列处理器的统称。实际上 ARM 公司并不生产处理器，而是通过转让 ARM 处理器相关的技术来获利。目前绝大多数厂商生产的手机、平板电脑和移动开发板使用的都是 ARM 或基于 ARM 定制的处理器。一些厂商，如苹果、高通、三星、联发科、英伟达等会购买 ARM 的处理器技术，然后做一些配置增强，以满足市场的需要。

ARM 处理器分成 A、R、M 三个系列，其中 A 系列是为了性能设计的，主要用于高性能手机、平板电脑、开发板和服务器，本章主要介绍基于此系列中的 A15 多核向量处理器编程。市场上的 ARM A 系列处理器有基于 v7 的 Cortex A9 和 Cortex A15，笔者使用的是 NVIDIA 生产的 Tegra TK1 自带的 ARM A15 处理器。

为了节能，ARM 处理器的核心频率会依据负载情况自动调整，如果负载增大，频率会自动上升，直到最高频率；如果温度过高，频率又会自动下降。在做性能测试时需要特别注意这点。

2.1 NEON 指令集综述

NEON 是 ARM Cortex A 系列处理器支持的数据并行技术，和 SSE/AVX 类似：一条指令以指令级 SIMD 的方式同时对多个数据进行操作，同时，操作的数据个数由向量寄存器的长度和数据类型共同决定。 NEON SIMD 寄存器的长度为 128 位，如果操作 32 位浮点数，可同时操作 4 个；如果操作 16 位整数（short），可同时操作 8 个；而如果操作 8 位整数，则可同

时操作 16 个。可见，选择合适的数据类型也是优化性能的重要方法之一，本章中就有相关的实例。

ARMv7 NEON 指令集架构具有 16 个 128 位的向量寄存器，命名为 q0 ～ q15。这 16 个寄存器又可以拆分成 32 个 64 位寄存器，命名为 d0 ～ d31。其中 q_n 和 d_{2n}，d_{2n+1} 是一样的，故使用汇编编写代码时要注意避免产生寄存器覆盖。

使用 NEON 指令读写数据时，不需要保证数据对齐到 16 字节。GCC 支持的 NEON 指令集的 C 语言接口（内置函数，intrinsic）声明在 arm_neon.h 头文件中。 NEON 指令集支持的映射到向量寄存器的向量数据类型命名格式为 type size x num。

其中：

① type 表示元素的数据类型，目前只支持 float、int 和 uint。

② size 表示每个元素的数据长度位数，float 只支持 32 位浮点数，int 和 unit 支持 8 位、16 位、32 位和 64 位整数。

③ num 表示元素数目，即向量寄存器的位数。由于 NEON 只支持 64 位和 128 位向量寄存器，故 size 和 num 的乘积只能是 64 或 128。

如 uint16x8_t 表示每个元素数据类型为 uint，大小为 16 位，每个向量保存 8 个数，故使用的向量寄存器长度为 128 位；如 float32x4_t 表示每个元素的数据类型为 32 位浮点，向量寄存器可操作 4 个数据，故使用 128 位向量寄存器。

NEON 内置函数命名方式有两种，分别对应源操作数是否涉及标量，具体解释如下。

1）源操作数涉及标量时，数据类型表示为 v op dt_n/lane_type。

其中：

① n 表示源操作数是标量而返回向量，lane 表示运算涉及向量的一个元素。

② op 表示操作，如 dup、add、sub、mla 等。

③ dt 是目标向量和源向量长度表示符。

- 如果目标向量和源向量长度都为 64 位，dt 为空。
- 如果源向量和目标向量长度一致都为 128 位，dt 为 q。
- 如果目标向量长度比源数向量长度大，且源向量长度都为 64 位、目标向量长度为 128 位，dt 为 l（英文字母，不是数字 1）。
- 如果多个源向量长度不一致且都不大于目标向量长度（一个源向量长度为 64 位，另一个为 128 位，目标向量长度为 128 位），dt 为 w。
- 如果目标向量长度比源向量长度小，即目标向量长度 dt 为 n。

④ type 表示源数据类型缩写，如 u8 表示 uint8；u16 表示 uint16；u32 表示 uint32；s8 表示 int8；s16 表示 int16；s32 表示 int32；f32 表示 float32。

2）源操作数全是向量时，数据类型表示为 v op dt_type，其中 op、dt 和 type 的含义和源

操作数为标量时一致。

下面给出几个实例以增加读者理解。

1）内置函数 vmla_f32 表示使用 64 位向量寄存器操作 32 位浮点数据，即源操作数使用的向量寄存器和目标操作数使用的向量寄存器表示都是 float32x2_t。

2）内置函数 vmlaq_f32 表示使用 128 位向量寄存器操作 32 位浮点数据，即源操作数使用的向量寄存器和目标操作数使用的向量表示都是 float32x4_t。

3）内置函数 vmlal_u32 表示使用的目标寄存器是 128 位向量，源寄存器是 64 位向量，操作 32 位无符号整数。

4）内置函数 vaddw_s32 表示使用的目标寄存器是 128 位向量，源寄存器一个是 64 位向量，一个是 128 位向量。

5）内置函数 vmovn_u64 表示目标寄存器是 64 位向量，源寄存器是 128 位向量，即同时操作两个数。

2.2　ARM A15 处理器性能

ARM A15 处理器具有 4 个核心，核心的最高频率为 2.3 GHz，因此其单核理论 32 位浮点峰值计算能力为 2.3×4×2 = 18.4 GFLOPS，4 核即为 73.6 GFLOPS，这大约是 8 年前单路服务器级 X86 处理器的性能。如果未来的 ARM 处理器支持使用 16 位浮点数运算的话，则其理论峰值计算能力将翻倍。

A15 处理器每个核心都具有独立的 L1 指令缓存和数据缓存，两者大小都是 32 KB，采用 2 路组相联映射策略，缓存线长度为 64 字节，替换策略采用最近最少使用策略。笔者测试发现，其 L1 缓存读带宽为每核心每时钟周期 16 字节，延迟为 8 个时钟周期，故要发挥其性能需要的并行度为 128 字节，即总共有 128 字节的读访问正在流水线上进行。如果使用 128 位 SIMD 读指令，则需要 8 条；如果使用 256 位的 SIMD 读指令，则需要 4 条。

A15 处理器 4 个核心共享统一的 L2 缓存。L2 缓存的大小可配置为 512 KB、1MB、2MB，甚至 4MB 大小；采用 16 路组相联映射策略，缓存线长度为 64 字节，替换策略采用随机策略。

NVIDIA tegra tk1 开发板配置的内存容量为 2GB 的 DDR3L，内存频率为 933 MHz，数据宽度为 64 位，故其理论峰值带宽为 0.933×8 = 7.46 GB/s。

2.3　NEON 支持的操作

本节根据操作类型和数据类型，分别列出 GCC 支持的 ARM NEON 内置函数，并加以简

单解释以供参考。

由于 NEON 具有许多不同的内置函数，本节不可能把它们全都列出来，故只列出常见的部分，更详细的列表可参考 ARM 官方手册。

2.3.1 基本算术运算

NEON 支持丰富的算术运算指令，基本上包括了所有的常见算术运算，如表 2-1 所示。

表 2-1 ARM NEON 常见算术运算

操　作	数据类型	描　述
add	u8 u16 u32 u64 s8 s16 s32 s64 f32	求和
addq	u8 u16 u32 u64 s8 s16 s32 s64 f32	
addl	u8 u16 u32 s8 s16 s32	
addw	u8 u16 u32 s8 s16 s32	
padd	u8 u16 u32 s8 s16 s32	每个向量中相邻元素相加，结果合并成一个向量
paddq	u8 u16 u32 s8 s16 s32	
qadd	u8 u16 u32 u64 s8 s16 s32 s64	求和，如果溢出，则取临界值
qaddq	u8 u16 u32 u64 s8 s16 s32 s64	
sub	u8 u16 u32 u64 s8 s16 s32 s64 f32	减
subq	u8 u16 u32 u64 s8 s16 s32 s64 f32	
subl	u8 u16 u32 s8 s16 s32	
subw	u8 u16 u32 s8 s16 s32	
qsub	u8 u16 u32 u64 s8 s16 s32 s64	减，如果溢出，则取临界值
qsubq	u8 u16 u32 u64 s8 s16 s32 s64 f32	
mul	u8 u16 u32 s8 s16 s32 f32	乘
mulq	u8 u16 u32 s8 s16 s32 f32	
mull	u8 u16 u32 s8 s16 s32	
mul_n	u16 u32 s16 s32 f32	标量乘向量
mulq_n	u16 u32 s16 s32 f32	
mull_n	u16 u32 s16 s32 f32	
mla	u8 u16 u32 s8 s16 s32 f32	乘加
mlaq	u8 u16 u32 s8 s16 s32 f32	
mlal	u8 u16 u32 s8 s16 s32	
mla_n	u16 u32 s16 s32 f32	标量与向量乘后加向量
mlaq_n	u16 u32 s16 s32 f32	
mlal_n	u16 u32 s16 s32 f32	

（续）

操　作	数据类型	描　述
mls	u8 u16 u32 s8 s16 s32 f32	乘减
mlsq	u8 u16 u32 s8 s16 s32 f32	
mlsl	u8 u16 u32 s8 s16 s32	
mls_n	u16 u32 s16 s32 f32	乘减，乘操作数之一为标量
mlsq_n	u16 u32 s16 s32 f32	
mlsl_n	u16 u32 s16 s32	
fma	f32	融合乘加
fmaq	f32	
fms	f32	融合乘减
fmsq	f32	
neg	s8 s16 s32 f32	负数
negq	s8 s16 s32 f32	
qneg	s8 s16 s32	求负数，如果溢出，则取临界值
qnegq	s8 s16 s32	
abd	u8 u16 u32 s8 s16 s32 f32	差的绝对值
abdq	u8 u16 u32 s8 s16 s32 f32	
abdl	u8 u16 u32 s8 s16 s32	
aba	u8 u16 u32 s8 s16 s32	差的绝对值再加
abaq	u8 u16 u32 s8 s16 s32	
abal	u8 u16 u32 s8 s16 s32	
abs	s8 s16 s32 f32	求绝对值
absq	s8 s16 s32 f32	
qabs	s8 s16 s32	求绝对值，如果溢出，则取临界值
qabsq	s8 s16 s32	
max	u8 u16 u32 s8 s16 s32 f32	最大值
maxq	u8 u16 u32 s8 s16 s32 f32	
min	u8 u16 u32 s8 s16 s32 f32	最小值
minq	u8 u16 u32 s8 s16 s32 f32	
recpe	f32 u32	估计倒数
recpeq	f32 u32	
rsqrte	f32 u32	开方倒数
rsqrteq	f32 u32	

从表中可以看出，NEON 已经支持了绝大多数常见算术运算指令，故可以预知，大多数计算只要满足数据并行的特征，就有可能使用 NEON 加速。

由于某些笔者不知道的原因，在 ARM 官方网站和手册上，除了乘加指令外，并没有优化程序性能所需的各个指令的吞吐量和延迟数据，甚为遗憾。不过笔者的测试表明：MLA 指令中，如果有一个乘数是标量的话，其吞吐量为每周期每核心 1 条，延迟大约为 4 个时钟周期，这意味着要发挥 mla 指令的吞吐量，代码中至少要有 4 个相互不依赖的 mla 指令；如果乘数都是向量的话，那么吞吐量为每个周期每核心 1 条，延迟为 8 个周期。

一些算法可能可以使用不同的指令系列实现，不同的指令系列的吞吐量和延迟都不相同，不知道指令的吞吐量和延迟会极大地阻碍代码的优化工作。

2.3.2 基本比较运算

如表 2-2 所示，ARM 提供了丰富的比较运算内置函数，这些函数完全能够满足编写程序时的比较功能需要。

表 2-2 比较运算内置函数

操 作	数 据 类 型	描 述
ceq	u8 u16 u32 s8 s16 s32 f32	等于
ceqq	u8 u16 u32 s8 s16 s32 f32	
cge	u8 u16 u32 s8 s16 s32 f32	大于等于
cgeq	u8 u16 u32 s8 s16 s32 f32	
cle	u8 u16 u32 s8 s16 s32 f32	小于等于
cleq	u8 u16 u32 s8 s16 s32 f32	
cgt	u8 u16 u32 s8 s16 s32 f32	大于
cgtq	u8 u16 u32 s8 s16 s32 f32	
clt	u8 u16 u32 s8 s16 s32 f32	小于
cltq	u8 u16 u32 s8 s16 s32 f32	
cage	f32	绝对值大于等于
cageq	f32	
cale	f32	绝对值小于等于
caleq	f32	
cagt	f32	绝对值大于
cagtq	f32	
calt	f32	绝对值小于
caltq	f32	

（续）

操　作	数据类型	描　述
tst	u8 u16 u32 s8 s16 s32	位测试
tstq	u8 u16 u32 s8 s16 s32	

NEON 比较指令的返回值为无符号整型向量，这使得向量化一些简单的类 C 中的三目运算成为了可能。在实践中，笔者不建议在 ARM 处理器上向量化分支代码。

2.3.3　基本数据类型转换及舍入运算

很多程序具有将浮点数舍入到整数的代码片段，ARM 内置函数支持的舍入模式如表 2-3 所示。需要注意的是：其返回值类型依旧是 32 位浮点数。而在 32 位整数和浮点数之间转换的内置函数如表 2-4 所示。

表 2-3　舍入运算

操　作	数据类型	描　述
rndn	f32	舍入到整数，向偶舍入
rndqn	f32	
rnda	f32	舍入到整数，远离零舍入
rndqa	f32	
rndp	f32	舍入到整数，向正无穷舍入
rdqp	f32	
rndm	f32	舍入到整数，向负无穷舍入
rndqm	f32	
rnd	f32	舍入到整数，向零舍入
rndq	f32	

表 2-4　类型转换运算

操　作	数据类型	描　述
cvt_f32	u32 s32	转换成 32 位浮点数
cvtq_f32	u32 s32	
cvt_u32	f32	转换成 32 位无符号整数
cvtq_u32	f32	
cvt_s32	f32	转换成 32 位有符号整数
cvtq_s32	f32	

通常使用特定的模式主要是因为两个原因：精度或速度。这两者通常来说是相对的，即精度低的速度快，精度高的速度慢。

在同样大小的数据类型中，执行数据类型转换可能意味着精度的损失，比如从单精度浮点数转换为无符号整型，因为有些数据用单精度能够表示，而无符号整型不能表示，反之也成立。

2.3.4 基本位运算

NEON 提供了左移、右移操作的许多变种，以及一些位统计操作，可惜不支持逻辑位运算，如表 2-5 所示。

表 2-5 ARM NEON 位运算

操　作	数　据	描　述
shl	u8 u16 u32 u64 s8 s16 s32 s64	左移，移动位数由向量指定
shlq	u8 u16 u32 u64 s8 s16 s32 s64	
qshl	u8 u16 u32 u64 s8 s16 s32 s64	左移，如果溢出，则返回临界值
qshlq	u8 u16 u32 u64 s8 s16 s32 s64	
shl_n	u8 u16 u32 u64 s8 s16 s32 s64	左移，移动位数由标量指定
shlq_n	u8 u16 u32 u64 s8 s16 s32 s64	
shll_n	u8 u16 u32 s8 s16 s32	
qshl_n	u8 u16 u32 u64 s8 s16 s32 s64	和 qshl 类似，参数为标量
qshlq_n	u8 u16 u32 u64 s8 s16 s32 s64	
shr_n	u8 u16 u32 u64 s8 s16 s32 s64	右移，移动位数由标量指定
shrq_n	u8 u16 u32 u64 s8 s16 s32 s64	
mvn	s8 s16 s32 u8 u16 u32	位求反
mvnq	s8 s16 s32 u8 u16 u32	
cls	s8 s16 s32	统计符号位数量
clsq	s8 s16 s32	
clz	s8 s16 s32 u8 u16 u32	前缀 0 的数目
clzq	s8 s16 s32 u8 u16 u32	
cnt	s8 u8	二进制表示中 1 的数目

移位运算、求无符号数的二进制表示中，1 的个数广泛用于一些加密解密算法中。

2.3.5 基本逻辑运算

NEON 支持的逻辑运算如表 2-6 所示。其中没有意外也没有惊喜，虽然没有非操作，但是可以通过其他操作得到。

表 2-6　ARM NEON 逻辑运算

操　作	数　据	描　述
and	u8 u16 u32 u64 s8 s16 s32 s64	与
andq	u8 u16 u32 u64 s8 s16 s32 s64	
orr	u8 u16 u32 u64 s8 s16 s32 s64	或
orrq	u8 u16 u32 u64 s8 s16 s32 s64	
eor	u8 u16 u32 u64 s8 s16 s32 s64	异或
eorq	u8 u16 u32 u64 s8 s16 s32 s64	
bic	u8 u16 u32 u64 s8 s16 s32 s64	与非
bicq	u8 u16 u32 u64 s8 s16 s32 s64	
orn	u8 u16 u32 u64 s8 s16 s32 s64	或非
ornq	u8 u16 u32 u64 s8 s16 s32 s64	

NOEN 支持的逻辑运算，无论是从种类上还是从支持的数据类型上已经非常完备。

2.3.6　基本设置加载存储操作

NEON 支持的基本数据加载存储操作如表 2-7 所示。从表中可以看出，NEON 从指令层次支持由结构体组成的数组（AOS）和由数组组成的结构体（SOA）之间的转换。

表 2-7　ARM NEON 加载存储

操　作	数　据	描　述
vmovn	s16 s32 s64 u16 u32 u64	复制向量寄存器数据，并更改数据元素大小
vmovl	s16 s32 s8 u16 u32 u8	
ld1	u8 u16 u32 u64 s8 s16 s32 s64 f32	从内存中加载数据到一个向量中
ld1q	u8 u16 u32 u64 s8 s16 s32 s64 f32	
st1	u8 u16 u32 u64 s8 s16 s32 s64 f32	将一个向量中的数据存储到内存中
st1q	u8 u16 u32 u64 s8 s16 s32 s64 f32	
ld2	u8 u16 u32 u64 s8 s16 s32 s64 f32	从内存中加载数据（以 AOS 存储）到两个向量中（SOA）
ld2q	u8 u16 u32 u64 s8 s16 s32 s64 f32	
st2	u8 u16 u32 u64 s8 s16 s32 s64 f32	把两个向量（SOA）中数据存储到内存中（AOS）
st2q	u8 u16 u32 u64 s8 s16 s32 s64 f32	
ld3	u8 u16 u32 u64 s8 s16 s32 s64 f32	从内存中加载数据（以 AOS 存储）到 3 个向量中（SOA）
ld3q	u8 u16 u32 u64 s8 s16 s32 s64 f32	
st3	u8 u16 u32 u64 s8 s16 s32 s64 f32	把 3 个向量（SOA）中数据存储到内存中（AOS）
st3q	u8 u16 u32 u64 s8 s16 s32 s64 f32	

（续）

操　作	数　据	描　述
ld4	u8 u16 u32 u64 s8 s16 s32 s64 f32	从内存中加载数据（以 AOS 存储）到 4 个向量中（SOA）
ld4q	u8 u16 u32 u64 s8 s16 s32 s64 f32	
st4	u8 u16 u32 u64 s8 s16 s32 s64 f32	把 4 个向量（SOA）中数据存储到内存中（AOS）
st4q	u8 u16 u32 u64 s8 s16 s32 s64 f32	

和 SSE/AVX 不同，NEON 没有显式地加载不对齐到向量长度的访存指令，这主要是因为：NEON 的 load/store 指令原生支持不对齐到向量长度的访问。

ldx/stx（x 代表 1、2、3、4）能够从结构体类型的内存数据中加载得到独立的向量，比如 ld3 指令能够一次从内存中加载 192 位数据，保存到 3 个向量中，其中内存相对地址 0 的数据加载到向量 0 中，内存相对地址 1 的数据加载到向量 1 中，内存相对地址 2 的数据加载到向量 2 中，以此类推。ldx/stx 指令的存在极大地方便了向量化一些数据类型。

2.3.7　特殊操作

NEON 提供了一些特殊的操作，如复制一个标量成向量、将两个 64 位向量合并成一个 128 位向量或相反。其他的一些有类似矩阵转置的运算和打包、拆包等，如表 2-8 所示。

表 2-8　特殊操作

操　作	数 据 类 型	描　述
dup_n	u8 u16 u32 u64 s8 s16 s32 s64 f32	将标量值复制到向量的每个元素
dupq_n	u8 u16 u32 u64 s8 s16 s32 s64 f32	
combine	u8 u16 u32 s8 s16 s32 f32	将两个 64 位向量合并成 128 位
get_high	u8 u16 u32 u64 s8 s16 s32 s64 f32	返回 128 位向量的高 64 位
get_low	u8 u16 u32 u64 s8 s16 s32 s64 f32	返回 128 位向量的低 64 位
trn	u8 u16 u32 s8 s16 s32 f32	将两个向量看成 2 行的矩阵，然后做转置
trnq	u8 u16 u32 s8 s16 s32 f32	
zip	u8 u16 u32 s8 s16 s32 f32	打包
zipq	u8 u16 u32 s8 s16 s32 f32	
unzip	u8 u16 u32 s8 s16 s32 f32	拆包
unzipq	u8 u16 u32 s8 s16 s32 f32	
ext	u8 u16 u32 u64 s8 s16 s32 s64 f32	依据偏移量从两个向量中生成一个向量
extq	u8 u16 u32 u64 s8 s16 s32 s64 f32	

dup 操作将一个标量扩展成向量，类似于 AVX 中的 broadcast_ss 指令。而 trn 则可实现类似 2 行多列矩阵转置的操作。

ext 内置函数从两个向量中依据偏移量生成一个新向量，如偏移量为 2，则从第一个向量中取出后两个元素和第二个向量的前两个元素组成一个新向量。对于许多算法，ext 内置函数能够大量减少读写存储器的次数，对于存储器限制的程序来说，能够派上大用场。

2.4　应用实例

本节以几个简单的实例来展示如何使用 NEON 指令来优化 ARM 应用。

2.4.1　彩色图像转灰度图像

本节以将 RGB 彩色图像转换成灰度图像为例，展示如何使用 NEON 指令集来加速计算。假设图像为 256 色，即每个像素通道为一个字节。原始的将 RGB 像素转换成灰度像素算法如下所示：

```
unsigned char gray = (unsigned char)(r*0.3f + g*0.59f + b*0.11f);
```

由于 r、g、b 为一个字节，因此编译器会插入指令将其先转成浮点型，计算完成后再转成整型。

1. 变浮点运算为整数运算

依据 0 计算时，编译器需要插入指令将像素值转换成浮点值，如果能够提前将像素权重系数转化成整数，那么就可以去掉浮点转换指令。每个通道像素为 256 色，那么可以先将权重乘以 256，然后将结果除以 256 即可。由于 C 语言在做算术运算时，会将字节数据转化成整型数据，因此权重参数类型采用 int。此算法如代码清单 2-1 所示。

代码清单2-1　变浮点为定点核心代码

```
int r_ratio = 77;//0.3*256
int g_ratio = 151;//0.59*256
int b_ratio = 28;//0.11*256
int temp = r*r_ratio + g*g_ratio + b*b_ratio;
unsigned char gray = temp / 256;
```

加上输入和输出，代码清单 2-1 对应的 CPU 代码如代码清单 2-2 所示。

代码清单2-2　整体代码

```
for(int i = 0; i < n; i++){
      int r = *src++; // load red
      int g = *src++; // load green
      int b = *src++; // load blue

      int r_ratio = 77;
      int g_ratio = 151;
      int b_ratio = 28;
```

```
        int temp = r*r_ratio;
        temp += g*g_ratio;
        temp += (b*b_ratio);

        dest[i] = (temp>>8);
    }
```

测试发现程序整体的效果并不是十分理想。使用 objdump 反汇编后，发现其核心运算汇编代码如代码清单 2-3 所示。

代码清单2-3　反汇编后代码

```
vld3.8  {d16,d18,d20}, [r7]!
vld3.8  {d17,d19,d21}, [r7]
vmovl.u8        q15, d19
vmovl.u8        q1, d16
vmovl.u8        q2, d18
vmovl.u8        q3, d17
vmovl.u8        q14, d20
vmovl.u8        q8, d21
vmull.s16       q9, d2, d22
vmull.s16       q0, d4, d24
vmull.s16       q10, d6, d22
vmull.s16       q5, d30, d24
vmull.s16       q4, d31, d25
vmull.s16       q1, d3, d23
vadd.i32        q0, q0, q9
vmull.s16       q2, d5, d25
vmull.s16       q3, d7, d23
vadd.i32        q10, q5, q10
vmull.s16       q15, d28, d26
vmull.s16       q9, d29, d27
vadd.i32        q2, q2, q1
vmull.s16       q14, d16, d26
vadd.i32        q3, q4, q3
vmull.s16       q8, d17, d27
vadd.i32        q15, q15, q0
vadd.i32        q14, q14, q10
vadd.i32        q9, q9, q2
vadd.i32        q8, q8, q3
vshr.s32        q15, q15, #8
vshr.s32        q9, q9, #8
vshr.s32        q8, q8, #8
vshr.s32        q14, q14, #8
vmovn.i32       d20, q15
vmovn.i32       d21, q9
vmovn.i32       d18, q14
vmovn.i32       d19, q8
vmovn.i16       d16, q10
vmovn.i16       d17, q9
vst1.8  {d16-d17}, [lr]!
```

从汇编代码中可以看出以下几点：

- 编译器自动向量化了代码，可采用 vld3 来加载 RGB 像素数据到 3 个向量寄存器中，以方便后面的计算。
- 编译器自动循环展开了 4 次。分析可以得知：编译器自动执行循环展开主要是因为算法中只使用了少量的寄存器，因此还有大量的寄存器可以用来通过循环展开，来发掘并行性，提高流水线效率。
- 编译器生成了许多 vmovl 和 vmovn 指令，编译器生成这些指令是因为在代码中混合使用了 32 位和 16 位整数，这些指令用于在计算前后转换数据。
- 乘法采用了 16 位整数，而加法采用了 32 位整数。

2. 使用更小的数据类型

由于编译器向量化了运算（向量化运算会直接在 char 上运算），那么 C 语言在运算时将 char 转换成 int 的前提已经不存在了，而正是由于这个前提导致了汇编代码中有许多 vmovn、vmovl 指令，以转换数据大小。为了去掉 vmovn、vmovl 指令，只需改变 r、g、b 变量及其权重的数据类型为 uint8_t，而乘法结果采用 uint16_t 即可。优化后代码如代码清单 2-4 所示。

代码清单2-4　使用uint8_t表示像素

```
for(int i = 0; i < n; i++){
            uint8_t r = *src++; // load red
            uint8_t g = *src++; // load green
            uint8_t b = *src++; // load blue

            uint8_t r_ratio = 77;
            uint8_t g_ratio = 151;
            uint8_t b_ratio = 28;
            uint16_t temp = r*r_ratio;
            temp += g*g_ratio;
            temp += (b*b_ratio);
            dest[i] = (y>>8);
}
```

为了确认修改数据类型大小产生了期望的结果，笔者反汇编了程序。对应的核心汇编指令如代码清单 2-5 所示。

代码清单2-5　汇编代码

```
    vld3.8  {d16,d18,d20}, [r7]!
    vld3.8  {d17,d19,d21}, [r7]
    vmull.u8        q1, d18, d26
    vmull.u8        q15, d16, d24
    vmull.u8        q3, d17, d25
    vmull.u8        q2, d19, d27
```

```
vmull.u8        q8, d21, d23
vmull.u8        q14, d20, d22
vadd.i16        q15, q1, q15
vadd.i16        q9, q2, q3
vadd.i16        q10, q15, q14
vadd.i16        q9, q9, q8
vshr.u16        q10, q10, #8
vshr.u16        q9, q9, #8
vmovn.i16       d16, q10
vmovn.i16       d17, q9
vst1.8  {d16-d17}, [lr]!
```

从反汇编代码来看，编译器生成的 vmovl、vmovn 指令大幅度减少，但是还遗留两个 vmovn 指令，这是因为移位生成的结果是 16 位整型而非 8 位整型。

3. 使用 NEON 内置函数生成所需指令

从中可以看出 mov 指令已经去掉，而编译器自动循环展开次数变成了 2 次。但是奇怪的是编译器并没有生成 mla 指令。为此使用 NEON 内置函数改写循环，如代码清单 2-6 所示。

代码清单2-6　使用内置函数生成mla指令

```
n /= 8;
uint8x8_t r_ratio = vdup_n_u8(77);
uint8x8_t g_ratio = vdup_n_u8(151);
uint8x8_t b_ratio = vdup_n_u8(28);

for(int i = 0; i < n; i++){
     uint8x8x3_t rgb = vld3_u8(src); // load red
     uint8x8_t r = rgb.val[0];
     uint8x8_t g = rgb.val[1];
     uint8x8_t b = rgb.val[2];

     uint16x8_t y = vmull_u8(r, r_ratio);
     y = vmlal_u8(y, g, g_ratio);
     y = vmlal_u8(y, b, b_ratio);
     uint8x8_t ret = vshrn_n_u16(y, 8);

     vst1_u8(dest, ret);

     src += 3*8;
     dest += 8;
}
```

为了去掉代码清单 2-5 中的 vmovn 指令，右移采用的内置函数是 vshrn，它能够在移位的同时减少数据类型大小。

和前面一样，为了保证结果每像素大小是 8 字节，类型转换是需要的。只是 vshrn_n_u16 会在做右移的同时将 2 字节无符号型转成 1 字节无符号型。笔者反汇编后发现对应的核心汇编指令如代码清单 2-7 所示：

代码清单2-7　汇编代码

```
vmov.i8 d19, #77        ; 0x4d
vmov.i8 d22, #151       ; 0x97
vmov.i8 d23, #28

vld3.8  {d16-d18}, [r1]!
adds    r3, #1
cmp     r3, r2
vmull.u8        q10, d16, d19
vmlal.u8        q10, d17, d22
vmlal.u8        q10, d18, d23
vshrn.i16       d20, q10, #8
vst1.8  {d20}, [r0]!
```

生成的循环内指令只有 8 条，而且已经完全没有多余指令了。

4. 小结及展望

此节所使用的一系列简单的算法优化、C 代码转换、数据类型使用、NEON 内置函数使用等手段，将核心代码从近 40 条降到了 8 条。

从代码清单 2-7 来看，已经没有多余的指令了，但是可以看到的是：核心计算只使用了很少的几个向量寄存器，还有大量的向量寄存器空闲。读者可以思考如何更好地利用这些空闲的寄存器进一步提升性能。

笔者实际测试表明，最终优化后的代码在 ARM A15 处理器上性能是未做任何优化前的几十倍。

2.4.2　矩阵转置

在某些矩阵运算中，为了更好地实现某些算法，需要将矩阵转置，如果矩阵转置性能不好的话，有可能成为瓶颈。因此矩阵转置的性能非常重要。

1. 寄存器分块算法

为了简化问题，首先考虑如何实现一个 4×4 的单精度浮点矩阵的转置。假设每一行已经加载到向量寄存器中，那么问题转化为如何转置 4 个长度为 4 的向量。ARM 提供了 vtrnq 指令来转置两个向量，但是目前不支持处理 64 位数据类型，因此需要特别处理。

vtrnq 指令的详细功能是：有 4 个长度为 128 位的向量，标记为 a、b、c、d，使用下标 0、1、2、3 标记其元素。vtrnq(a, b) 返回的结果是两个 128 位的向量 a' 和 b'，其中：a' 的值为 $a_0b_0a_2b_2$，b' 的值为 $a_1b_1a_3b_3$。类似 vtrnq(c, d) 返回的 c' 的值为 $c_0d_0c_2d_2$，d' 的值为 $c_1d_1c_3d_3$。可以看出，如果将 a'、b'、c'、d' 中每两个元素看成一个整体，再执行 vtrnq(a', c') 和 vtrnq(b', d') 操作，即可返回转置好的 4 个向量。

由于 NOEN 的 vtrn 指令目前还不支持 64 位元素，无法简单使用多个 vtrn 指令完成 4×4

的矩阵转置。但是可以通过向量分割和合并以利用 vtrn 获得的部分结果。基于 ARM NEON 内置函数的实现如代码清单 2-8 所示。

代码清单2-8 4×4矩阵寄存器转置

```
inline void transpose32x4x4(float32x4_t& q0, float32x4_t& q1, float32x4_t& q2,
    float32x4_t& q3){
   float32x4x2_t q01 = vtrnq_f32(q0, q1);
   float32x4x2_t q23 = vtrnq_f32(q2, q3);

   q0 = q01.val[0];
   float32x2_t d00 = vget_low_f32(q0);
   float32x2_t d01 = vget_high_f32(q0);
   q1 = q01.val[1];
   float32x2_t d10 = vget_low_f32(q1);
   float32x2_t d11 = vget_high_f32(q1);
   q2 = q23.val[0];
   float32x2_t d20 = vget_low_f32(q2);
   float32x2_t d21 = vget_high_f32(q2);
   q3 = q23.val[1];
   float32x2_t d30 = vget_low_f32(q3);
   float32x2_t d31 = vget_high_f32(q3);

   q0 = vcombine_f32(d00, d20);
   q1 = vcombine_f32(d10, d30);
   q2 = vcombine_f32(d01, d21);
   q3 = vcombine_f32(d11, d31);
}
```

基于 4×4 的小矩阵，可以实现 8×8、16×16 的矩阵分块。由于 NEON 128 位向量寄存器数量的限制，测试发现使用 8×8 的矩阵分块性能最好。

2. 一级缓存分块算法

由于矩阵转置是一个访存密集型的应用，因此发挥内存带宽非常重要。故考虑寄存器分块后，还需要考虑 L1 缓存分块。考虑到 A15 上一级数据缓存的大小为 32 KB，L1 分块的初始大小设置为 64×64，而实际测试也表明这个值比较好。

使用 L1 分块优化后的代码如代码清单 2-9 所示。其中参数 M、N 是寄存器分块参数，参数 numRowsComputed、numColsComputed 是 L1 分块优化参数。

代码清单2-9 一级缓存分块

```
template<int M, int N, int numRowsComputed, int numColsComputed>
void l1_blocking_MxN(int numRows, int numCols, const float* src, float* dst){
   for(int row = 0; row < numRowsComputed; row += 4*M){
      for(int col = 0; col < numColsComputed; col += 4*N){
         float32x4_t q[4*M][N];
         for(int i = 0; i < 4*M; i++){
            for(int j = 0; j < N; j++){
```

```
                q[i][j] = vld1q_f32(src + (row+i)*numCols + col+4*j);
            }
        }
        for(int i = 0; i < M; i++){
            for(int j = 0; j < N; j++){
               transpose32x4x4(q[4*i][j], q[4*i+1][j], q[4*i+2][j], q[4*i+3][j]);
               }
         }

          for(int i = 0; i < N; i++)
              for(int j = 0; j < M; j++){
                  vst1q_f32(dst+(col+4*i)*numRows + row+4*j, q[4*j][i]);
                  vst1q_f32(dst+(col+4*i+1)*numRows + row+4*j, q[4*j+1][i]);
                  vst1q_f32(dst+(col+4*i+2)*numRows + row+4*j, q[4*j+2][i]);
                  vst1q_f32(dst+(col+4*i+3)*numRows + row+4*j, q[4*j+3][i]);
               }
          }
    }
}
```

为了给读者一个使用 L1 分块算法的全貌，给出调用 L1 缓存分块的代码，如代码清单 2-10 所示。为了简单，并没有考虑矩阵的行列大小和 L1 缓存分块大小不匹配的情况。

代码清单2-10　调用L1缓存分块

```
template<int blockingRows, int blockingCols>
void optVersion_MxN(int numRows, int numCols, const float* src, float* dst){
     for(int row = 0; row < numRows; row += blockingRows){
            for(int col = 0; col < numCols; col += blockingCols){
                  l1_blocking_MxN<2, 2, blockingRows, blockingCols>(numRows,
                      numCols, src+row*numCols+col, dst+col*numRows+row);
            }
    }
}
```

本节通过矩阵转置展示了如何使用寄存器分块和 L1 缓存分块技术，这两种技术的使用大大提升了矩阵转置的性能。测试表明：优化后比优化前性能提升 10 倍左右。

2.4.3　矩阵乘

矩阵乘是很多算法的核心，而且目前没有基于 ARM NEON 的高性能矩阵乘库，因此编写一个高效的、运行在 ARM 上的矩阵乘法很有意义。

矩阵乘法依据输入矩阵是否转置可分成 4 种类别，即 $A \times B$、$A \times B^T$、$A^T \times B$、$A^T \times B^T$。本节基于 $A \times B^T$ 实现，将 B 转置后，$A \times B$ 转化成 A 的每一行和 B 的每一行做向量内积运算。

1. 向量内积版矩阵乘法

代码代码清单 2-11 是使用 NEON 内置函数的矩阵乘代码（计算前将矩阵 b 做了转置操

作)。为了便于示例，作者并没有考虑 M、K 和 N 不是 4 的倍数的情况。

代码清单2-11 向量内积版本

```
for(int i = 0; i < M; i++){
     for(int j = 0; j < N; j++){
             float32x4_t temp = vdupq_n_f32(0.0f);
             for(int k = 0; k < K; k += 4){
                    float32x4_t v_a = vld1q_f32(a+i*K+k);
                    float32x4_t v_b = vld1q_f32(b_T+j*K+k);
                    temp = vmlaq_f32(temp, v_a, v_b);
            }
            c[i*N + j] = (temp[0]+temp[1])+(temp[2]+temp[3]);
    }
}
```

反汇编后查看发现，其最内层循环代码如代码清单 2-12 所示。

代码清单2-12 向量内积版本最内层汇编代码

```
    vld1.32 {d18-d19}, [r4]!
    adds    r3, #4
    vld1.32 {d20-d21}, [r5]!
    cmp     r1, r3
    vmla.f32        q8, q9, q10
```

循环内有两条 vld1 指令加载 A、B 数据到向量寄存器中，然后使用 vmla 指令做一次乘加运算。由于循环内只有一条 mla 指令，并行度不够，故无法达到峰值性能。从指令数量来看，计算指令只占总指令数据的 20%。从计算访存比例来看，循环内加载指令和计算指令的比例为 2∶1，故性能应当限制于访存环节。

2. 循环展开

从代码清单 2-12 可以看出，代码清单 2-11 的实现性能限制在计算的比例太小（即 vmla 指令数量和比例太小)。如果增加循环内 vmla 指令的数量，那么就增加了计算指令的比例。考虑到代码清单 2-11 中两个外层循环访问的数据是相互重用的，故可对两个外层循环采用循环展开优化以提高计算比例，减少访存数量。执行循环展开优化后的代码如代码清单 2-13 所示。

代码清单2-13 循环展开版本

```
#define IB 4
#define JB 2
for(int i = 0; i < M; i += IB){
   for(int j = 0; j < N; j += JB){
      float32x4_t temp[IB][JB];
        for(int ii = 0; ii < IB; ii++){
            for(int jj = 0; jj < JB; jj++){
                temp[ii][jj] = vdupq_n_f32(0.0f);
          }
     }
```

```
        for(int k = 0; k < K; k += 4){
             float32x4_t v_a[IB];
             for(int ii = 0; ii < IB; ii++){
                 v_a[ii] = vld1q_f32(a+(i+ii)*K+k);
             }
             float32x4_t v_b[JB];
             for(int jj = 0; jj < JB; jj++){
                 v_b[jj] = vld1q_f32(b_T+(j+jj)*K+k);
             }
             for(int ii = 0; ii < IB; ii++){
                 for(int jj = 0; jj < JB; jj++){
                     temp[ii][jj] = vmlaq_f32(temp[ii][jj], v_a[ii], v_b[jj]);
                  }
             }
        }
        for(int ii = 0; ii < IB; ii++){
            for(int jj = 0; jj < JB; jj++){
                float32x4_t temp_c = temp[ii][jj];
                c[(i+ii)*N + j+jj] = (temp_c[0]+temp_c[1])+(temp_c[2]+temp_c[3]);
             }
        }
       }
    }
    #undef IB
    #undef JB
```

为了确定优化是否可以产生期望的效果，笔者反汇编后发现，对应的核心汇编代码如代码清单 2-14 所示。

代码清单2-14　循环展开版本汇编代码

```
    vld1.32 {d16-d17}, [r2]
    vld1.32 {d8-d9}, [r0]
    adds    r0, r3, r6
    vld1.32 {d18-d19}, [r5]
    adds    r1, #4
    vld1.32 {d26-d27}, [r3]
    cmp     fp, r1
    vld1.32 {d24-d25}, [r4]
    add.w   r2, r2, #16
    vld1.32 {d22-d23}, [r0]
    add.w   r3, r3, #16
    vmla.f32        q3, q4, q8
    vmla.f32        q2, q4, q9
    vmla.f32        q14, q13, q9
    vmla.f32        q10, q13, q8
    vmla.f32        q5, q12, q9
    vmla.f32        q0, q12, q8
    vmla.f32        q15, q11, q9
    vmla.f32        q1, q11, q8
```

生成的指令中，有 8 条 vmla，6 条 vld1，5 条其他指令，计算比例已经超过 40%，计算访存比为 4∶3。从代码中可以看出，使用了 14 个向量寄存器，故再执行循环展开会导致寄存器溢出。

测试表明，代码清单 2-13 代码的性能在 2048×2048×2048 的矩阵上已经可以达到 A15 理论峰值的近 50%，实际编译器生成的指令系列代码清单 2-14 并不是最优的，还有其他可优化的地方，比如可以更好地安排指令顺序，以利用双发射能力；固定分块大小，以去掉加法和比较指令等。

2.5 本章小结

本章首先介绍了 ARM 处理器和如何计算 ARM 处理器的性能，然后详细介绍了 ARM NEON 指令集的常见指令，最后以如何使用 ARM NEON 指令优化彩色图像转灰度图像、矩阵转置和矩阵乘法为例，详细介绍了如何使用 ARM NEON 指令优化程序性能。

本章介绍的使用 NEON 指令的优化技巧有：变浮点运算为整数运算；采用更小的数据类型；寄存器分块；一级缓存分块；循环展开。