Neon

## 一、Introdaction

## 1.1、SIMD

1. SIMD 指令

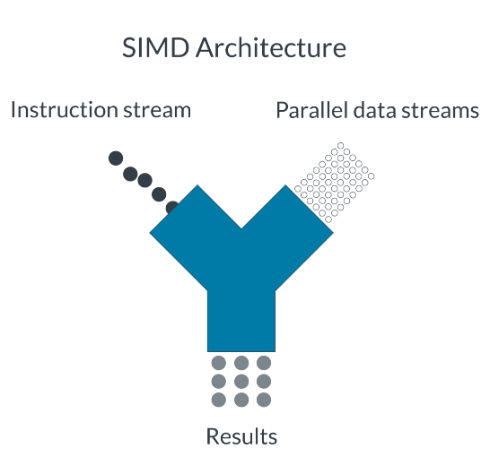
Arm Neon technology is an advanced SIMD (single instruction multiple data) architecture extension for the Arm Cortex-A series and Cortex-R52 processors.

Neon technology was introduced to the Armv7-A and Armv7-R profiles. It is also now an extension to the Armv8-A and Armv8-R profiles.

→：用于视觉和深度学习。

Neon technology is intended to improve the multimedia user experience by accelerating audio and video encoding/decoding, user interface, 2D/3D graphics or gaming. Neon can also accelerate signal processing algorithms and functions to speed up applications such as audio and video processing, voice and facial recognition, **computer vision and deep learning.**

## 二、Overview

****

## 三、Use Neon

Neon enabled libraries, compiler's auto-vectorization feature, Neon intrinsics, and finally, Neon assembly code

编译器的自动矢量化功能

## 四、Libraries

## 五、AutoVectorization

## 六、Compiler Intrinsics

# neon\_memcpy.cpp

**对于 ARM 中使用的 Neon 仅仅有这个程序？**

函数原型

void \*memcpy(void\*dest, const void \*src, size\_t n);

功能

由src指向地址为起始地址的连续n个字节的数据复制到以destin指向地址为起始地址的空间内。

头文件

#include<string.h>

返回值

　　函数返回一个指向dest的指针。

说明

　　1.source和destin所指内存区域不能重叠，函数返回指向destin的指针。

　　2.与strcpy相比，memcpy并不是遇到'\0'就结束，而是一定会拷贝完n个字节。

**namespace neon**

**{**

**void neon\_memcpy(void\* dst, void\* src, long size);**

**void neon\_memcpy\_1080to720(void\* dst, void\* src);**

**void neon\_memcpy\_rotate\_720to1080(void\* dst, void\* src);**

**void neon\_memcpy\_720to1080(void\* dst, void\* src);**

**}**

# Ne10 库函数

## 一、参考资料部分

### 1.1、[原创] Ne10 加速ARM平台图像处理应用

https://community.arm.com/cn/b/blog/posts/ne10-arm

### 1.2、Compute Library

文档来源

https://arm-software.github.io/ComputeLibrary/latest/

### 1.3、Ne10库入门

地址：https://community.arm.com/cn/b/blog/posts/ne10

## 二、NEON优化库(Optimized libraries)

**1、编译 Ne10 库文件**

[**https://www.jianshu.com/p/83da8280cc2d**](https://www.jianshu.com/p/83da8280cc2d)

**#!/bin/bash**

**export ARCH=arm64**

**export CROSS\_COMPILE=aarch64-linux-gnu-**

**export PATH=$PATH:/home/djiango/bsp15.0/gcc-linaro-4.9-2015.05-x86\_64\_aarch64-linux-gnu/bin**

**cd /home/djiango/Ne10/projectNe10**

**mkdir build && cd build**

**cmake -DCMAKE\_TOOLCHAIN\_FILE=/home/djiango/Ne10/projectNe10/GNUlinux\_config.cmake ..**

**make**

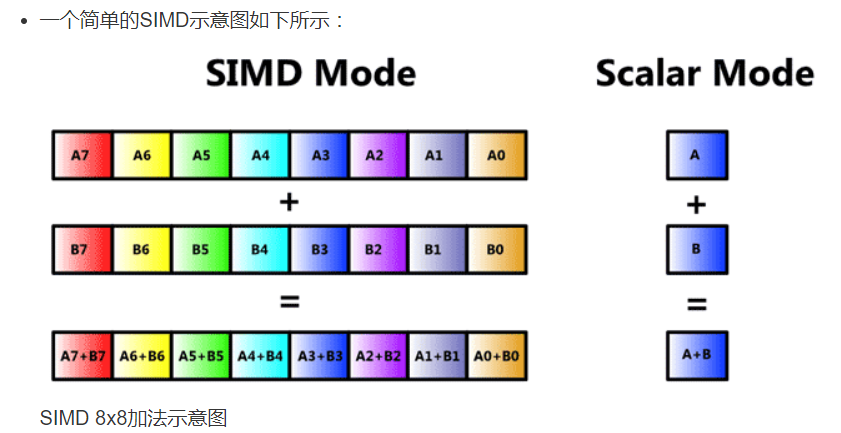
# 参考资料

## 一、ARM NEON 编程系列3——使用ARM NEON Intrinsics加速Video Codec

[**https://www.veryarm.com/118963.html/amp**](https://www.veryarm.com/118963.html/amp)

### 1.1、SIMD简介

Single Instruction Multiple Data (SIMD)，单指令多数据。从字面理解，就是在CPU执行中，一条操作指令可以同时操作多个寄存器，从而在物理上倍数的加速运行。我理解范畴内的X86平台上最早的SIMD指令应该是奔腾MMX上自带的MMX指令，其寄存器宽度是64位，可以同时操作8个字节。MultiMedia eXtensions (MMX)是多媒体扩展的意思，其最初的设计目的就是为了加速图像／视频等高并行数据的处理速度。

****

**在这里，一条SIMD加法指令可以同时得到8个加法结果。就计算步骤本身而言，比单独使用8条加法指令能够获得8倍的加速比。从该示例也可以看出，随着寄存器长度的变长，单指令能够处理的数据量也越来越大，从而获得更高的加速性能。在Intel最新的AVX2指令集中，寄存器最大长度已经达到512位。**

### 1.2、ARM NEON Intrinsics简介

NEON指令是从Armv7架构开始引入的SIMD指令，其共有16个128位寄存器。发展到最新的Arm64架构，其寄存器数量增加到32个，但是其长度仍然为最大128位，因此操作上并没有发生显著的变化。对于这样的寄存器，因为可以同时存储并处理多组数据，称之为向量寄存器。Intrinsics是使用C语言的方式对NEON寄存器进行操作，因为相比于传统的使用纯汇编语言，具有可读性强，开发速度快等优势。如果需要在代码中调用NEON Intrinsics函数，需要加入头文件"arm\_neon.h"。

### 1.3、数据类型

NEON Intrinsics内置的整数数据类型主要包括以下几种:

(u)int8x8\_t;

(u)int8x16\_t;

(u)int16x4\_t;

(u)int16x8\_t;

(u)int32x2\_t;

(u)int32x4\_t;

(u)int64x1\_t;

其中，第一个数字代表的是数据类型宽度为8/16/32/64位，第二个数字代表的是一个寄存器中该类型数据的数量。如int16x8\_t代表16位有符号数，寄存器中共有8个数据。

### 1.4、常用指令

NEON Intrinsics支持的所有指令可参看ARM NEON Intrinsics，其包含了常用的arm汇编指令类型，如数学运算，逻辑运算等。另外，其引入了有针对性的加载／存储／转置／交叉存取等指令。部分常见的指令在会下面的示例环节中予以说明。需要注意的是，指令中的助记符与arm汇编是相同的。

#### 示例1:

|  |
| --- |
| int16x8\_t vqaddq\_s16 (int16x8\_t, int16x8\_t)  int16x4\_t vqadd\_s16 (int16x4\_t, int16x4\_t) |

* 第一个字母'v'指明是vector向量指令，也就是NEON指令；
* 第二个字母'q'指明是饱和指令，即后续的加法结果会自动饱和；
* 第三个字段'add'指明是加法指令；
* 第四个字段'q'指明操作寄存器宽度，为'q'时操作QWORD, 为128位；未指明时操作寄存器为DWORD，为64位；
* 第五个字段's16'指明操作的基本单元为有符号16位整数，其最大表示范围为-32768 ~ 32767；
* 形参和返回值类型约定与C语言一致。

其它可能用到的助记符包括:

l 长指令，数据扩展

w 宽指令，数据对齐

n 窄指令, 数据压缩

#### 示例2

|  |
| --- |
| uint8x8\_t vld1\_u8 (const uint8\_t \*) |

第二个字段'ld'表示加载指令

第三个字段'1'(注意是1，不是l)表示顺次加载。如果需要处理图像的RGB分量，可能会用到vld3。关于vld/vst指令更详细的说明，请自己参阅arm官方文档。

### 1.5、函数改写示例

#### 1. 简单示例

原始代码

|  |
| --- |
| // uint8\_t \*\_dst, uint8\_t \*\_src, int16\_t \*src2  // int height, int width  for (y = 0; y < height; y++) {  for (x = 0; x < width; x++) {  dst[x] = av\_clip\_pixel(((src[x] << 6) + src2[x] + offset) >> shift);  }  src += srcstride;  dst += dststride;  src2 += MAX\_PB\_SIZE;  } |

原始代码的理解：原始代码是在求平均值。

改写代码

|  |
| --- |
| int16x8\_t result\_16x8;  int16x8\_t offset\_16x8 = vmovq\_n\_s16(offset);  int16x8\_t minusshift\_16x8 = vmovq\_n\_s16(-1 \* shift);  int16x8\_t min\_16x8 = vmovq\_n\_s16(0);  int16x8\_t max\_16x8 = vmovq\_n\_s16(255);  for (y = 0; y < height; y++) {  for (x = 0; x < width; x+=8) {  result\_16x8 = vshlq\_n\_s16(vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x]))), 6);  result\_16x8 = vshlq\_s16(vqaddq\_s16(vqaddq\_s16(result\_16x8, vld1q\_s16(&src2[x])), offset\_16x8), minusshift\_16x8);  vst1\_u8(&dst[x], vqmovn\_u16(vreinterpretq\_u16\_s16(vmaxq\_s16(vminq\_s16(result\_16x8, max\_16x8), min\_16x8))));  }  src += srcstride;  dst += dststride;  src2 += MAX\_PB\_SIZE;  } |

`说明：

1. 这里只针对宽度为8的倍数进行了改写，实际代码中需要对传入参数进行判断
2. vld1\_u8读取8字节数据，vmovl\_u8对读取的uint8x8进行宽度扩展

vld1\_u8(&src[x])：读取 8 字节数据。

vmovl\_u8(vld1\_u8(&src[x]))：读取的 uint 8x8 进行宽度拓展。

1. vreinterpretq\_s16\_u16对数据类型进行强制转换

vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x])))

关于类型转化的问题

4.5.5 Casting NEON variables between different types

NEON intrinsics are strongly typed, so they must be explicitly cast between vectors of different

types. To cast vectors use vreinterpret for D registers or vreinterpretq for Q registers. These

intriniscs do not generate any code, but just enable you to cast the NEON types:

uint8x8\_t byteval;

uint32x2\_t wordval;

byteval = vreinterpret\_u8\_u32(wordval);

uint8x16\_t byteval2;

uint32x4\_t wordval2;

byteval2 = vreinterpretq\_u8\_u32(wordval2);

Note

The output type, u8, is listed after vreinterpret, before the input type, u32.

1. vshlq\_n\_s16对数据进行左移处理(P.S. NEON提供了右移指令，但是只能使用整数常量。需要根据变量进行右移时，只能使用左移负数位的方法。)

vshlq\_n\_s16(vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x]))), 6);

数据左移处理？对应的是原始代码的 <<6 ，左移6位的操作。

1. vqmovn\_u16对处理结果进行宽度压缩

处理结果宽度压缩：将数值压缩在 0-255 之间即可。

|  |
| --- |
| vqmovn\_u16(vreinterpretq\_u16\_s16(vmaxq\_s16(vminq\_s16(result\_16x8, max\_16x8), min\_16x8))) |

在开始有定义 min\_16x8 = vmovq\_n\_s16(0);

Max\_16x8=vmovq\_n\_s16(255);

定义了压缩在最大最小的范围内。

1. vst1\_u8将处理后的int16x8\_t数据写回内存

|  |
| --- |
| vst1\_u8(&dst[x], vqmovn\_u16(vreinterpretq\_u16\_s16(vmaxq\_s16(vminq\_s16(result\_16x8, max\_16x8), min\_16x8)))); |

存储数据到 dst[x] 中。

改写代码的思路：到底进行了哪些操作？怎么从寄存器的角度来进行解释？

#### 2.进阶示例

原始代码

|  |
| --- |
| /\*  #define QPEL\_FILTER(src, stride)  (filter[0] \* src[x - 3 \* stride] +  filter[1] \* src[x - 2 \* stride] +  filter[2] \* src[x - stride] +  filter[3] \* src[x ] +  filter[4] \* src[x + stride] +  filter[5] \* src[x + 2 \* stride] +  filter[6] \* src[x + 3 \* stride] +  filter[7] \* src[x + 4 \* stride])  DECLARE\_ALIGNED(16, const int8\_t, ff\_hevc\_qpel\_filters[3][16]) = {  { -1, 4,-10, 58, 17, -5, 1, 0, -1, 4,-10, 58, 17, -5, 1, 0},  { -1, 4,-11, 40, 40,-11, 4, -1, -1, 4,-11, 40, 40,-11, 4, -1},  { 0, 1, -5, 17, 58,-10, 4, -1, 0, 1, -5, 17, 58,-10, 4, -1}  };  \*/  filter = ff\_hevc\_qpel\_filters[mx - 1];  for (y = 0; y < height + QPEL\_EXTRA; y++) {  for (x = 0; x < width; x++)  tmp[x] = QPEL\_FILTER(src, 1);  src += srcstride;  tmp += MAX\_PB\_SIZE;  } |

改写代码

|  |
| --- |
| /\*  DECLARE\_ALIGNED(16, const int8\_t, ff\_hevc\_qpel\_filtersT[3][64]) = {  { -1, -1, -1, -1, -1, -1, -1, -1, 4, 4, 4, 4, 4, 4, 4, 4,//(0)  -10,-10,-10,-10,-10,-10,-10,-10, 58, 58, 58, 58, 58, 58, 58, 58,  17, 17, 17, 17, 17, 17, 17, 17, -5, -5, -5, -5, -5, -5, -5, -5,  1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0},  { -1, -1, -1, -1, -1, -1, -1, -1, 4, 4, 4, 4, 4, 4, 4, 4,//(1)  -11,-11,-11,-11,-11,-11,-11,-11, 40, 40, 40, 40, 40, 40, 40, 40,  40, 40, 40, 40, 40, 40, 40, 40,-11,-11,-11,-11,-11,-11,-11,-11,  4, 4, 4, 4, 4, 4, 4, 4, -1, -1, -1, -1, -1, -1, -1, -1},  { 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,//(2)  -5, -5, -5, -5, -5, -5, -5, -5, 17, 17, 17, 17, 17, 17, 17, 17,  58, 58, 58, 58, 58, 58, 58, 58,-10,-10,-10,-10,-10,-10,-10,-10,  4, 4, 4, 4, 4, 4, 4, 4, -1, -1, -1, -1, -1, -1, -1, -1}  };  \*/  int16x8\_t filteT\_16x8\_0, filteT\_16x8\_1, filteT\_16x8\_2, filteT\_16x8\_3, filteT\_16x8\_4, filteT\_16x8\_5, filteT\_16x8\_6, filteT\_16x8\_7;  int16x8\_t result\_16x8;  filter = ff\_hevc\_qpel\_filtersT[mx - 1];  filteT\_16x8\_0 = vmovl\_s8(vld1\_s8(&filter[0]));  filteT\_16x8\_1 = vmovl\_s8(vld1\_s8(&filter[8]));  filteT\_16x8\_2 = vmovl\_s8(vld1\_s8(&filter[16]));  filteT\_16x8\_3 = vmovl\_s8(vld1\_s8(&filter[24]));  filteT\_16x8\_4 = vmovl\_s8(vld1\_s8(&filter[32]));  filteT\_16x8\_5 = vmovl\_s8(vld1\_s8(&filter[40]));  filteT\_16x8\_6 = vmovl\_s8(vld1\_s8(&filter[48]));  filteT\_16x8\_7 = vmovl\_s8(vld1\_s8(&filter[56]));  for (y = 0; y < height + QPEL\_EXTRA; y++) {  for ( x = 0; x < width; x += 8 ) {  // init the output reg  result\_16x8 = vmovq\_n\_s16(0);  // (0)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x-3]))), filteT\_16x8\_0);  // (1)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x-2]))), filteT\_16x8\_1);  // (2)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x-1]))), filteT\_16x8\_2);  // (3)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x]))), filteT\_16x8\_3);  // (4)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x+1]))), filteT\_16x8\_4);  // (5)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x+2]))), filteT\_16x8\_5);  // (6)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x+3]))), filteT\_16x8\_6);  // (7)  result\_16x8 = vmlaq\_s16(result\_16x8, vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x+4]))), filteT\_16x8\_7);  // store the output data  vst1q\_s16(&tmp[x], result\_16x8);  }  src += srcstride;  tmp += MAX\_PB\_SIZE;  } |

说明:

关键是不知道原来的函数是什么，仅仅是输入。需要处理的数据是什么？

在C实现中，每个结果需要读取包括自身在内的8个输入，乘以相应的系数并累加。最简单直观的实现方法是

|  |
| --- |
| output\_16x8 = vmulq\_s16( vreinterpretq\_s16\_u16(vmovl\_u8(vld1\_u8(&src[x-3]))), vmovl\_s8(vld1\_s8(ff\_hevc\_qpel\_filters[mx - 1]))); |

这样实现，会使得8个乘积分布在同一个向量寄存器中，需要通过取寄存器的不同元素实现累加，加法部分无法并行。

在C实现中，其数学表示为两个1x8和8x1的矩阵之间的乘法。分析数据间的关系，将矩阵乘法转换为矩阵转置乘法，可以得出前文改写代码的实现。在该实现中，由于滤波器系统固定，因此预先定义了其转置矩阵并扩展。在进行'乘加'操作的过程中，一个循环将8个结果全部计算完毕，使得乘法/加法均实现了并行化。

P.S. 这里，单独设置了8个向量寄存器变量并展开使得代码较长，使用循环+数组的方式也可以得到同样的结果，且代码较短。但是在底层高频函数中，尽量展开循环可以最大化的提升效率。

这一部分暂时停止，不用特别的在意。相对来说太过于复杂.

结语

本文只介绍了使用ARM NEON Intrinsics的原理和基本应用。实际中需要对待优化的函数原理及能使用的资源了解清楚才能使用最有效的方法并行化程序。

## 二、ARM NEON编程初探——一个简单的BGR888转YUV444实例详解

### 2.1、BGR888ToYUV444

NEON是一种SIMD（Single Instruction Multiple Data）指令，也就是说，NEON可以把若干源（source）操作数（operand）打包放到一个源寄存器中，对他们执行相同的操作，产生若干目的（dest）操作数，这种方式也叫向量化（vectorization）。

这样的话，能打包多少数据同时做运算就取决于寄存器位宽，在ARMv7的NEON unit中，register file总大小是1024-bit，可以划分为16个128-bit的Q-register（Quadword register）或者32个64-bit的D-register（Dualword register），也就是说，最长的寄存器位宽是128-bit（详见Guide第一章）。

以上面的R888ToYUV444函数为例，假设我们采用32-bit单精度浮点数float来做浮点运算，那么我们可以 把最多128/32=4个浮点数打包放到Q-register中做SIMD运算，一次拿4个BGR算出4个YUV，从而提高吞吐量，减少loop次数。

### 2.2、从浮点运算到整型运算

|  |
| --- |
| // 1. Multiply transform matrix (Y′: unsigned, U/V: signed)  uint16\_t y\_tmp = 76 \* r + 150 \* g + 29 \* b;  int16\_t u\_tmp = -43 \* r - 84 \* g + 127 \* b;  int16\_t v\_tmp = 127 \* r - 106 \* g - 21 \* b;  // 2. Scale down (">>8") to 8-bit values with rounding ("+128") (Y′: unsigned, U/V: signed)  y\_tmp = (y\_tmp + 128) >> 8;  u\_tmp = (u\_tmp + 128) >> 8;  v\_tmp = (v\_tmp + 128) >> 8;  // 3. Add an offset to the values to eliminate any negative values (all results are 8-bit unsigned)  yuv[i \* 3] = (uint8\_t) y\_tmp;  yuv[i \* 3 + 1] = (uint8\_t) (u\_tmp + 128);  yuv[i \* 3 + 2] = (uint8\_t) (v\_tmp + 128); |

从这段代码我们不难发现，32-bit的float运算被16-bit的加减、乘法和移位运算所代替。这样的话，我们可以把最多128/16=8个整型数放到Q-register中做SIMD运算，一次拿8个BGR算出8个YUV，把向量化程度再提一倍。使用整型运算还有一个好处：一般而言，整型运算指令所需要的时钟周期少于浮点运算的时钟周期。所以，我以这段代码为基准（baseline），用NEON来加速它。（细心的看官也许已经看到我说法中的纰漏：虽然单个整型指令的周期小于单个相同运算的浮点指令的周期，但整型版本的BGR888ToYUV444比起浮点版本的多了移位和加法的overhead，指令数目是不同的，总的时钟周期不一定就短。Good point! 看官不妨看完本文后也用NEON改写浮点版本练练手，两相比较就不难得出最终的结论啦XD）

对于相关数据来说，里面的公式来源于理论。不用特别的证明这个理论，知道如何将浮点转化为整型运算即可。

### 2.3、NEON Intrinsics

|  |
| --- |
| void BGR888ToYUV444(unsigned char \* \_\_restrict\_\_ yuv, unsigned char \* \_\_restrict\_\_ bgr, int pixel\_num)  {  const uint8x8\_t u8\_zero = vdup\_n\_u8(0);  const uint16x8\_t u16\_rounding = vdupq\_n\_u16(128);  const int16x8\_t s16\_rounding = vdupq\_n\_s16(128);  const int8x16\_t s8\_rounding = vdupq\_n\_s8(128);  int count = pixel\_num / 16;  int i;  for (i = 0; i < count; ++i) {  // Load bgr  uint8x16x3\_t pixel\_bgr = vld3q\_u8(bgr);  uint8x8\_t high\_r = vget\_high\_u8(pixel\_bgr.val[0]);  uint8x8\_t low\_r = vget\_low\_u8(pixel\_bgr.val[0]);  uint8x8\_t high\_g = vget\_high\_u8(pixel\_bgr.val[1]);  uint8x8\_t low\_g = vget\_low\_u8(pixel\_bgr.val[1]);  uint8x8\_t high\_b = vget\_high\_u8(pixel\_bgr.val[2]);  uint8x8\_t low\_b = vget\_low\_u8(pixel\_bgr.val[2]);  int16x8\_t signed\_high\_r = vreinterpretq\_s16\_u16(vaddl\_u8(high\_r, u8\_zero));  int16x8\_t signed\_low\_r = vreinterpretq\_s16\_u16(vaddl\_u8(low\_r, u8\_zero));  int16x8\_t signed\_high\_g = vreinterpretq\_s16\_u16(vaddl\_u8(high\_g, u8\_zero));  int16x8\_t signed\_low\_g = vreinterpretq\_s16\_u16(vaddl\_u8(low\_g, u8\_zero));  int16x8\_t signed\_high\_b = vreinterpretq\_s16\_u16(vaddl\_u8(high\_b, u8\_zero));  int16x8\_t signed\_low\_b = vreinterpretq\_s16\_u16(vaddl\_u8(low\_b, u8\_zero));  // NOTE:  // declaration may not appear after executable statement in block  uint16x8\_t high\_y;  uint16x8\_t low\_y;  uint8x8\_t scalar = vdup\_n\_u8(76);  int16x8\_t high\_u;  int16x8\_t low\_u;  int16x8\_t signed\_scalar = vdupq\_n\_s16(-43);  int16x8\_t high\_v;  int16x8\_t low\_v;  uint8x16x3\_t pixel\_yuv;  int8x16\_t u;  int8x16\_t v;  // 1. Multiply transform matrix (Y′: unsigned, U/V: signed)  high\_y = vmull\_u8(high\_r, scalar);  low\_y = vmull\_u8(low\_r, scalar);  high\_u = vmulq\_s16(signed\_high\_r, signed\_scalar);  low\_u = vmulq\_s16(signed\_low\_r, signed\_scalar);  signed\_scalar = vdupq\_n\_s16(127);  high\_v = vmulq\_s16(signed\_high\_r, signed\_scalar);  low\_v = vmulq\_s16(signed\_low\_r, signed\_scalar);  scalar = vdup\_n\_u8(150);  high\_y = vmlal\_u8(high\_y, high\_g, scalar);  low\_y = vmlal\_u8(low\_y, low\_g, scalar);  signed\_scalar = vdupq\_n\_s16(-84);  high\_u = vmlaq\_s16(high\_u, signed\_high\_g, signed\_scalar);  low\_u = vmlaq\_s16(low\_u, signed\_low\_g, signed\_scalar);  signed\_scalar = vdupq\_n\_s16(-106);  high\_v = vmlaq\_s16(high\_v, signed\_high\_g, signed\_scalar);  low\_v = vmlaq\_s16(low\_v, signed\_low\_g, signed\_scalar);  scalar = vdup\_n\_u8(29);  high\_y = vmlal\_u8(high\_y, high\_b, scalar);  low\_y = vmlal\_u8(low\_y, low\_b, scalar);  signed\_scalar = vdupq\_n\_s16(127);  high\_u = vmlaq\_s16(high\_u, signed\_high\_b, signed\_scalar);  low\_u = vmlaq\_s16(low\_u, signed\_low\_b, signed\_scalar);  signed\_scalar = vdupq\_n\_s16(-21);  high\_v = vmlaq\_s16(high\_v, signed\_high\_b, signed\_scalar);  low\_v = vmlaq\_s16(low\_v, signed\_low\_b, signed\_scalar);  // 2. Scale down (">>8") to 8-bit values with rounding ("+128") (Y′: unsigned, U/V: signed)  // 3. Add an offset to the values to eliminate any negative values (all results are 8-bit unsigned)  high\_y = vaddq\_u16(high\_y, u16\_rounding);  low\_y = vaddq\_u16(low\_y, u16\_rounding);  high\_u = vaddq\_s16(high\_u, s16\_rounding);  low\_u = vaddq\_s16(low\_u, s16\_rounding);  high\_v = vaddq\_s16(high\_v, s16\_rounding);  low\_v = vaddq\_s16(low\_v, s16\_rounding);  pixel\_yuv.val[0] = vcombine\_u8(vqshrn\_n\_u16(low\_y, 8), vqshrn\_n\_u16(high\_y, 8));  u = vcombine\_s8(vqshrn\_n\_s16(low\_u, 8), vqshrn\_n\_s16(high\_u, 8));  v = vcombine\_s8(vqshrn\_n\_s16(low\_v, 8), vqshrn\_n\_s16(high\_v, 8));  u = vaddq\_s8(u, s8\_rounding);  pixel\_yuv.val[1] = vreinterpretq\_u8\_s8(u);  v = vaddq\_s8(v, s8\_rounding);  pixel\_yuv.val[2] = vreinterpretq\_u8\_s8(v);  // Store  vst3q\_u8(yuv, pixel\_yuv);  bgr += 3 \* 16;  yuv += 3 \* 16;  }  // Handle leftovers  for (i = count \* 16; i < pixel\_num; ++i) {  uint8\_t r = bgr[i \* 3];  uint8\_t g = bgr[i \* 3 + 1];  uint8\_t b = bgr[i \* 3 + 2];  uint16\_t y\_tmp = 76 \* r + 150 \* g + 29 \* b;  int16\_t u\_tmp = -43 \* r - 84 \* g + 127 \* b;  int16\_t v\_tmp = 127 \* r - 106 \* g - 21 \* b;  y\_tmp = (y\_tmp + 128) >> 8;  u\_tmp = (u\_tmp + 128) >> 8;  v\_tmp = (v\_tmp + 128) >> 8;  yuv[i \* 3] = (uint8\_t) y\_tmp;  yuv[i \* 3 + 1] = (uint8\_t) (u\_tmp + 128);  yuv[i \* 3 + 2] = (uint8\_t) (v\_tmp + 128);  }  } |

注释相关：

**1、vdup\_n\_u8/s8/u16/s16 含义**

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

复制寄存器到向量中。

**2、uint8x16x3\_t pixel\_bgr = vld3q\_u8(bgr);**

我用vld3q\_u8指令从内存一次加载16个像素（也就是uint8\_t类型的B、G、R三个通道的数值），将各个通道的16个数值放到一个Q-register中（也就是用了三个Q-register，每个分别存放16个B、16个G和16个R），vst3q\_u8的写入操作也是类似的，这充分利用128-bit的带宽，使内存存取（memory access）的次数尽可能少。

但是，后边运算的向量化程度其实仍然不变，只能同时进行8个16-bit整型运算，也就是说，对于运算的部分，理想加速比（speedup）是8而非16。（当然，一次从内存加载多少数据是一个design choice，没有绝对的答案，看官也可以尝试别的方式XD）

对于像素个数不能被16整除的情况，可能会有剩下的1到15个像素没被上述NEON代码处理到，这里我把它们称作leftovers，简单粗暴地跑了之前的for循环。其实leftover的情况如果占比高的话，还有更高明的处理方式，请各位看官参见Guide的6.2 Handling non-multiple array lengths一节。

我把Y通道计算的一部分放到一起，稍作解释，其他的很多运算都是类似的：

|  |
| --- |
| // 复制8个值为128的uint16\_t类型整数到某个Q-register，该Q-register对应的C变量是u16\_rounding  const uint16x8\_t u16\_rounding = vdupq\_n\_u16(128);  // 复制8个值为128的uint8\_t类型整数到某个D-register，该D-register对应的C变量是scalar  uint8x8\_t scalar = vdup\_n\_u8(76);  // pixel\_bgr.val[0]为一个Q-register，16个uint8\_t类型的数，对应R通道  // pixel\_bgr.val[1]为一个Q-register，16个uint8\_t类型的数，对应G通道  // pixel\_bgr.val[2]为一个Q-register，16个uint8\_t类型的数，对应B通道  uint8x16x3\_t pixel\_bgr = vld3q\_u8(bgr);  // 一个Q-register对应有两个D-register  // 这是拿到对应R通道的Q-register高位的D-register，有8个R值  // 参见Guide中Register overlap一节（这部分内容很重要！）  // 其他如low\_r、high\_g的情况相似，这里从略  uint8x8\_t high\_r = vget\_high\_u8(pixel\_bgr.val[0]);  // 对8个R值执行乘76操作  // vmull是变长指令（常以单词末尾额外的L作为标记），源操作数是两个uint8x8\_t的向量，目的操作数是uint16x8\_t的向量  // 到这一步：Y = R \* 76  // low\_y的情况类似，从略  high\_y = vmull\_u8(high\_r, scalar);  // 把scalar的8个128改为8个150  scalar = vdup\_n\_u8(150);  // 执行乘加运算  // 到这一步：Y = R \* 76 + G \* 150  high\_y = vmlal\_u8(high\_y, high\_g, scalar);  // 把scalar的8个150改为8个29  scalar = vdup\_n\_u8(29);  // 执行乘加运算  // 到这一步：Y = R \* 76 + G \* 150 + B \* 29  high\_y = vmlal\_u8(high\_y, high\_b, scalar);  // 到这一步：Y = (R \* 76 + G \* 150 + B \* 29) + 128  high\_y = vaddq\_u16(high\_y, u16\_rounding);  // vqshrn\_n\_u16是变窄指令（常以单词末尾额外的N作为标记，N为Narrow），将uint16x8\_t的向量压缩为uint8x8\_t  // 到这一步：Y = ((R \* 76 + G \* 150 + B \* 29) + 128) >> 8  // vcombine\_u8用于将两个D-register的值组装到一个Q-register中  pixel\_yuv.val[0] = vcombine\_u8(vqshrn\_n\_u16(low\_y, 8), vqshrn\_n\_u16(high\_y, 8)); |

# 三、arm NEON简介

资料地址：

<https://wenku.baidu.com/view/f912b09ea48da0116c175f0e7cd184254b351bd0.html>

NEON的寄存器：**有16个128位四字寄存器Q0-Q15，32个64位双字寄存器D0-D31，两个寄存器是重叠的**，在使用的时候需要特别注意，不小心就会被覆盖掉。

NEON的数据类型：无符号整数、有符号整数、未指定类型的整数、浮点数、{0,1}上的多项式。数据类型针对的是操作数，而不是目标数。NEON中的数据类型说明符由一个指示数据类型的字母构成，该字母通常后跟一个指示宽度的数字。

NEON指令可处理：(1)、由以下内容构成的双字向量：八个8位元素、四个16位元素、两个32位元素、一个64位元素；(2)、由以下内容构成的四字向量：十六个8位元素、八个16位元素、四个32位元素、两个64位元素。

**NEON中的正常指令、宽指令、窄指令、饱和指令、长指令**：  
(1)、**正常指令**：生成大小相同且类型通常与操作数向量相同的结果向量；  
(2)、**长指令**：对双字向量操作数执行运算，生成四字向量的结果。所生成的元素一般是操作数元素宽度的两倍，并属于同一类型；（注意是和窄指令相反）  
(3)、**宽指令**：一个双字向量操作数和一个四字向量操作数执行运算，生成四字向量结果。所生成的元素和第一个操作数的元素是第二个操作数元素宽度的两倍；  
(4)、**窄指令**：四字向量操作数执行运算，并生成双字向量结果，所生成的元素一般是操作数元素宽度的一半；（和长指令相反）  
(5)、**饱和指令**：当超过数据类型指定的范围则自动限制在该范围内。

NEON标量：有些NEON指令可处理与向量组合使用的标量。NEON标量可以为8位、16位、32位或64位。除乘法指令之外，访问标量的指令也可访问寄存器组中的任何元素。指令语法通过在双字向量中使用索引来引用标量，从而使Dm[x]表示Dm中的第x个元素。乘法指令仅允许使用16位或32位标量，并且只能访问寄存器组中的前32个标量。这在乘法指令中意味着：

(1)、16位标量限定为寄存器D0-D7，其中x位于范围0-3内；

(2)、32位标量限定为寄存器D0-D15，其中x为0或1。

{0,1}上的多项式算法：使用布尔算法规则处理系数0和1：(1)、0+0=1+1=0；(2)、0+1=1+0=1；(3)、0\*0=0\*1=1\*0=0；(4)、1\*1=1. 也就是说，将两个{0,1}上的多项式相加与按位异或的运算相同，而将两个{0,1}上的多项式相乘则与整乘的运算相同，但部分积执行的是异或运算，而不是相加运算。

**NEON注意事项：**  
(1)、load数据的时候，第一次load会把数据放在cache里面，只要不超过cache的大小，下一次load同样数据的时候，则会比第一次load要快很多，会直接从cache中load数据；  
(2)、在做NEON乘法指令的时候会有大约2个clock的阻塞时间，如果你要立即使用乘法的结果，则就会阻塞在这里。乘法的结果不能立即使用，可以将一些其它的操作插入到乘法后面而不会有时间的消耗；  
(3)、使用饱和指令的时候，如乘法饱和的时候，在做乘法后会再去做一次饱和，所用时间要比直接做乘法要慢；  
(4)、在对16位数据进行load或者store操作的时候，需要注意的是字节移位。

