**dct的aarch64优化经验**

王永昶 张芳 赵宏智

## 前期准备工作

1. 完成《GDB 工具的基本用法》几个小例子，熟悉一下GDB用法，顺便温习一下汇编语言基本语法。
2. 阅读《GNU ARM 汇编语法》，重点阅读8.GNU 内嵌(inline)ARM 汇编代码及其格式，里面小例子可酌情实现以助理解。（改编汇编基础）
3. 建议《研究型汇编实验指导书》完成小例子，其中例子3.5是重点，后面dct汇编代码在这个例子的基础上进行演化

## dct的c代码解析与优化思路

1. C代码中的DCT函数可分为三部分：

第一部分对8\*8的像素点进行处理（6个函数），得到中间结果矩阵rows2。

第二部分是将rows2进行转置。

第三部分与第一部分处理手段十分相似（就改了一个参数），最后得到最终结果data矩阵

1. 第一部分，每一行像素点都用一组（6个）函数进行处理，且每个函数所完成的功能十分简单。这有几点好处：便于代码阅读，便于调试，节省代码量。当然坏处也很明显：过多的调用函数，传递参数会与内存频繁打交道，严重降低性能。
2. 优化思路1：将这些函数合并为一个过程，形如：

for (i = 0; i < 8; i++)

{

for(j=0; j<8; j++)

{

sum[j] = 0;

for (k = 0; k < 8; k++)

sum[j] = sum[j] + cola[i\*8+k] \* pixels[j][k];

irows2[j][i] = sum[j];

rows2[j][i] = (short)(sum[j] >> sh14[i]);

}

}

优化思路2：尽量使用Arm64的SIMD指令来并行处理多个像素。

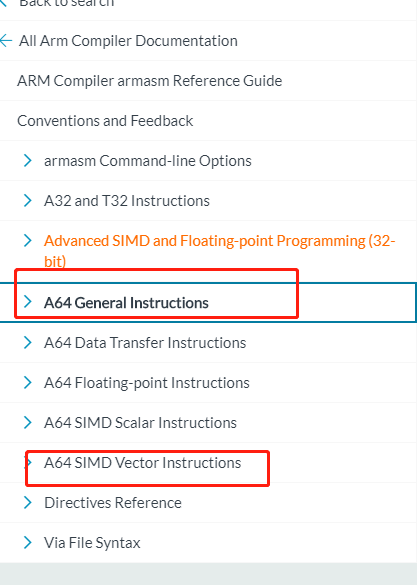
## 三、主要用到的汇编指令

1. 先从ld，st指令开始：

参照《研究型汇编实验指导书》例子3.5，其使用LD4与ST4把一组数据从内存取到向量寄存器中或从向量寄存器中存入内存。在这里我们的目标矩阵是8\*8的矩阵，所以我们使用LD1与ST1指令代替LD4与ST4指令。

用《8000页指令集手册》查指令可能比较费力，这里推荐网站<https://developer.arm.com/documentation/dui0802/b/A64-General-Instructions?lang=en>

索引如下图所示：



其中有两块是我们主要使用的，一个是aarch64一般指令集，一个是simd向量指令集（两个红色框的）。

LD1，LD2，LD4与ST 1，ST 2，ST4指令均是simd向量指令集。

至于我们为什么用LD1，ST1不用LD2，LD4，ST2，ST4。在这里给出一个测试矩阵：

short aa[8][8]={

{0,1,2,3,4,5,6,7},

{8,9,10,11,12,13,14,15},

{16,17,18,19,20,21,22,23},

{24,25,26,27,28,29,30,31},

{32,33,34,35,36,37,38,39},

{40,41,42,43,44,45,46,47},

{48,49,50,51,52,53,54,55},

{56,57,58,59,60,61,62,63},

感兴趣的可以使用两次“ld1 {v0.8h, v1.8h, v2.8h, v3.8h}, [地址]” 取出，再用两次“st4 {v16.8h, v17.8h, v18.8h, v19.8h}, [地址]”存入（或LD4先取出，ST1再存入）显示结果，比较前后矩阵变化。

1. 再把最内层循环的乘法翻译成汇编：

参照《研究型汇编实验指导书》例子3.5，其使用mul指令进行一组数据的乘法运算。

但是

例子3.5的数太小了，两个较小的short类型数相乘的积short类型还是可以正确放下的。感兴趣的同学可以把例子3.5两个矩阵的第一行数据分别放大1000倍，这样用mul得出的结果就不对了（short范围-32768~+32767）

1. 乘法溢出问题

经试验在YUV转换到jpeg时两个short矩阵相乘时，其结果一定会超出short范围。在C代码中也是先用int类型数组暂存，再算数右移14或16位才又成为short矩阵的（就是下面packword\_lower函数）

void packword\_lower(int NumSrc, short\* PDest, int\* PSrc)

{

int i;

int m = 0;

for (i = 0; i < NumSrc; i++)

{

PDest[i] = (short) PSrc [i];

}

return;

}

所以不能用mul指令，那有没有这样一种乘法指令可以完成我们的任务呢？simd指令集丰富，完善。我们找到了指令“smull”与“smull2”。这组指令可以让一组short类型数据相乘得到int类型数据，或让一组int类型数据相乘得到long类型数据等。具体用法参考网站或手册。

1. 累加数组累加和

源代码中，每次元素乘完都会进行累加。那有没有这样一种指令可以完成我们的任务呢？我们找到了addv指令

1. 加法溢出问题

还是要考虑一下加法的溢出情况的（C代码中sum是否溢出）。经试验在YUV转换到jpeg时sum累加时，其结果不会超出int类型范围。所以我们不用考虑addv指令的高级形式，可以直接使用addv指令。

1. 循环右移

C语言的运算符“>>”代表算数右移，与之对应的还有逻辑右移和循环右移。移位操作对于寄存器来说是最简单的工作所以肯定有与之对应的指令。我们查到算数右移，逻辑右移和循环右移分别为ASR，LSR，ROR。依题意我们选择ASR。（数据在计算机中都是以补码方式存储）

1. 通用寄存器与向量寄存器的数据交换

但是算数右移只能在通用寄存器执行，所以我们需要把向量寄存器的数据逐一取出到通用寄存器进行移位。

## 四、汇编改写顶层设计

1. 首先再回顾源C代码：第一部分数据处理，第二部分矩阵转置，第三部分同第一部分。其中第二部分是“多余的操作”，看一下我们整理成三层循环的第一部分：

for (i = 0; i < 8; i++)

{

for(j=0; j<8; j++)

{

sum[j] = 0;

for (k = 0; k < 8; k++)

sum[j] = sum[j] + cola[i\*8+k] \* pixels[j][k];

irows2[j][i] = sum[j];

rows2[j][i] = (short)(sum[j] >> sh14[i]);

}

}

如果我们把最后两行代码中的i，j互换：irows2[i][j] = sum[j];rows2[i][j] = (short)(sum[j] >> sh14[i]);就可以顺便完成转置。

Cola与pixels都是8\*8的short矩阵，我们可以分别用8个向量寄存器存储，0-7号寄存器存cola，8-15存pixels。然后用16-23存结果矩阵rows2.由于在移位操作中我们需要把矩阵中每一个元素逐一取出到通用寄存器移位后再逐一放到结果矩阵中，这个过程正好可以顺便转置，所以可以把C代码的一，二部分合并。且一部分与三部分只有细微差别所以我们只需完成第一部分的汇编任务就完成了一大半。

最后我们使用内嵌汇编的方法，把COLA，pixels，14,16作为四个输入参数，data作为输出参数。把dct里面所有计算过程汇总成一个比较完整同一的汇编代码。

## 五、函数输出结果的差异性评价

由于本函数是对图像进行处理的。需要采用PSNR指标和优化前后两幅图像的每个像素的颜色分量值的对比值指标来评估函数输出结果的差异性。在本例子中，代码改进前的输出图像和代码改进后的输出图像的PNSR值和每个像素的颜色分量值的对比值均无差异。

## 六、优化的加速比

说明：dct函数经ARM64的SIMD指令优化后，按加速比来衡量函数性能的提升程度。

C语言编码对一个8\*8的像素点处理耗时28589397 ns，ARM64汇编优化后耗时3833403 ns ，加速比7.457968。