ARM NEON优化（二）——NEON编程, 优化心得及内联汇编使用心得

NEON编程基础

使用NEON主要有四种方法：

1. NEON优化库(Optimized libraries)
2. 向量化编译器(Vectorizing compilers)
3. NEON intrinsics
4. NEON assembly

根据优化程度需求不同，第4种最为底层，若熟练掌握效果最佳，一般也会配合第3种一起使用。本文将会重点介绍第3、4种方法。先简要介绍前两种。

- Libraries：直接在程序中调用优化库
  - OpenMax DL：支持加速视频编解码、信号处理、色彩空间转换等；
  - Ne10：一个ARM的开源项目，提供数学运算、图像处理、FFT函数等。

- Vectorizing compilers：GCC编译器的向量优化选项
  - 在GCC选项中加入向量化表示能有助于C代码生成NEON代码，如`-ftree-vectorize`。

NEON intrinsics
提供了一个连接NEON操作的C函数接口，编译器会自动生成相关的NEON指令，支持ARMv7-A或ARMv8-A平台。

所有的intrinsics函数都在GNU官方说明文档。

一个简单的例子：

```
//add for int array. assumed that count is multiple of 4

#include<arm_neon.h>

// C version
void add_int_c(int* dst, int* src1, int* src2, int count)
{
  int i;
  for (i = 0; i < count; i++)
    dst[i] = src1[i] + src2[i];
  }
}

// NEON version
void add_float_neon1(int* dst, int* src1, int* src2, int count)
{
  int i;
  for (i = 0; i < count; i += 4)
  {
    int32x4_t in1, in2, out;
    in1 = vld1q_s32(src1);
    src1 += 4;
    in2 = vld1q_s32(src2);
    src2 += 4;
    out = vaddq_s32(in1, in2);
    vst1q_s32(dst, out);
    dst += 4;
  }
}
```

代码中的`vld1q_s32`会被编译器转换成`vld1.32 {d0, d1}, [r0]`指令，同理`vaddq_s32`和`vst1q_s32`被转换成`vadd.i32 q0, q0, q0`，`vst1.32 {d0, d1}, [r0]`。若不清楚指令意义，请参见ARM® Compiler armasm User Guide - Chapter 12 NEON and VFP Instructions。

NEON assembly
NEON可以有两种写法：

1. Assembly文件
   - 纯汇编文件，后缀为”.S”或”.s”。注意对寄存器数据的保存。具体对通用寄存器的详解不是本文的重点，有兴趣的读者请自行补充该部分知识。

2. inline assembly内联汇编
   - 优点：在C代码中嵌入汇编，调用简单，无需手动存储寄存器；
   - 缺点：有较为复杂的格式需要事先学习，不好移植到其他语言环境。

比如上述intrinsics代码产生的汇编代码为：

```
// ARMv7-A/AArch32

void add_float_neon2(int* dst, int* src1, int* src2, int count)
{
  asm volatile (
    "1:                                         \n"
    "vld1.32         {q0}, [%[src1]]!           \n"
    "vld1.32         {q1}, [%[src2]]!           \n"
    "vadd.f32        q0, q0, q1                  \n"
    "subs            %[count], %[count], #4     \n"
    "vst1.32         {q0}, [%[dst]]!            \n"
    "bgt             1b                         \n"
    : [dst] "+r" (dst)
    : [src1] "r" (src1), [src2] "r" (src2), [count] "r" (count)
    : "memory", "q0", "q1"
  );
}
```

NEON优化心得

笔者在前段时间连续使用NEON做ARM平台的优化，由于中文资料少得可怜，且英文资料零散琐碎，期间也遇到不少坑，先摘出部分经验至此，希望能够帮助到大家。ヘ(￣ω￣)ヘ

建议的NEON调优步骤

1. **理清所需的寄存器、指令。** 建议根据要实现的任务，画出数据变换流程，和每步所需的具体指令，尽可能找到最优的实现流程。这一步非常关键，如果思路出错或是不够优化，则会影响使用NEON的效果，并且对程序修改带来麻烦，一定要找到最优的实现算法哦~

2. **先实现intrinsics（可选）。** 初学者先实现intrinsics是有好处的，字面理解性更强，且有助于理解NEON指令。建议随时打印关键步骤的数据，以检查程序的正误。

3. **写成汇编进一步优化。** 将intrinsics生成的汇编代码进行优化调整。一般来说，有以下几点值得注意**【干货】**：

   - 只要intrinsics运算指令足够精简，运算类的汇编指令就不用大修；
   - **大部分的问题会出在存取、移动指令的滥用、混乱使用上**；
   - 优化时要尽量减少指令间的相关性，包括结构相关、数据相关控制相关，保证流水线执行效率更高；
   - 大概估算所有程序指令取指、执行、写回的总理论时间，以此估算本程序可以优化的空间；
   - 熟练对每条指令准备发射、写回时间有一定的认识，有助于对指令的优化排序；
   - 一定要多测试不同指令的处理时间！！原因是你所想跟实际有出入，且不同的编译器优化的效果可能也有些不同；
   - 一定要有一定的计算机体系结构基础，对存储结构、流水线有一定的体会！！

【注意】在此笔者温馨提示各位看官(⊙o⊙)不仅是NEON，所有的性能优化是个经验活儿，需要自己动手才能领悟更多的诀窍，总结一下NEON优化就是：

- 第一优化算法实现流程；
- 第二优化程序存取；
- 第三优化程序执行；
- 第四哪儿能优化，就优化哪儿~~

对NEON优化使用的好坏直接导致优化效果，优化效果好的会节省70%以上的时间。

内联汇编使用心得

当读者熟练后就可以直接上手内联汇编了。时间有限，本文中不具体介绍inline assembly的使用方法，我后续可能会将这部分单独写成一篇博客。感兴趣者请参见ARM GCC Inline Assembler Cookbook

一些使用心得：

- **inline assembly下面的三个冒号一定要注意**
  - output/input registers的写法一定要写对，clobber list也一定要写完全，否则会造成令你头疼的问题 (T-T) …
  - 这个问题在给出的cookbook中也有介绍，但是并不全面，有些问题只有自己碰到了再去解决。 笔者就曾经被虐了很久，从生成的汇编发现编译器将寄存器乱用，导致指针操作完全混乱，毫无头绪…
  - 一般情况下建议的写法举例：

```
asm volatile (
  ... /* assembly code */

  : "+r"(arg0)  // %0
    "+r"(arg1)  // %1  // Output Registers
  : "r"(arg2)   // %2  // Input Registers
  : "cc", "memory", r0, r1
);
```

- **传入内联汇编程序段的C参数是有限的**
  - 笔者亲测对于Cortex-A7平台output/input registers基本在9以内才可保证，否则会报出`can't find a register in class 'GENERAL_REGS' while reloading 'asm'`错误。

内容太多，笔者就先写到这儿，容看官慢慢消化，后续可能还会写一篇NEON优化进阶版，讲解一些优化小trick~敬请期待！