ffmpeg RISC-V 优化介绍

孙越池 sunyuechi@iscas.ac.cn

开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler

目录

- ffmpeg介绍及其RISC-V优化历史
- SIMD, Vector, RISC-V V扩展
- 简单的rvv实例
- ffmpeg中自动向量化和手动实现

ffmpeg简介

ffmpeg是一个开源的，重要的多媒体方面的基础库/工具，在很多领域都需要它

比如在 编解码 方面有： 播放器(vlc..)，浏览器(chromium, firefox..), 录制/直播(obs), 各种剪辑软件等等总是用到它

ffmpeg的性能很关键，例如

视频的流程程度，能否处理高分辨率, 直播/视频会议等延迟
资源有限的情况使用效率（嵌入式）
转码速度（剪辑）

ffmpeg简介

软件方面如何优化ffmpeg性能?

通用来说可以设计更好的编解码器，改进算法

在这个基础上，x86, arm, aarch64 都直接通过汇编进行了大量的优化工作

ffmpeg的RISC-V历史

ffmpeg的RISC-V支持只需要简单的配置，在2021 10月添加后，就可以在RISC-V构建了

之后主要由现在的RISC-V maintainer: Rémi Denis-Courmont 在2022 9月开始提交RISC-V相关的patch, 包括测试，cpu检测，一些dsp(Digital Signal Processing)的优化等

ffmpeg在2023年11月10号发布的6.0.1 release中，正式包含了RISC-V的优化

最近20240405 7.0已经发布了 合入了更多RISC-V优化

SIMD

x86

64 bits: MMX (1997)
128 bits: SSE (1999), SSE2 (2000). . . AVX (2008)
256 bits: AVX2 (2011)
512 bits: AVX-512 (2013 2017)

ARM

32 bits: ARMv6 SIMD (2002)
128 bits: ARMv7 AdvSIMD, a.k.a. NEON (2005)
128 bits: ARMv8 A64 AdvSIMD, also a.k.a. NEON (2012)

ref:
https://archive.fosdem.org/2023/schedule/speaker/remi_denis_courmont/
https://www.youtube.com/watch?v=Z4DS3jiZhfo

SIMD-Vector

RISC-V

128（除嵌入式）-65536 bits： RVV1.0

Vector

x86:

avx-10 (2023)

ARM:

sve(2016)

sve2(2019 2022)

RISC-V:

rvv1.0(2023)

Vector编程常见模式

csrr, t0, vlenb // 按字节数读取向量长度

读向量长度

unroll

处理边界情况

Vector编程推荐的模式

向量长度可变时，比较推荐的向量化方式是基于活跃元素。

对应mask(RISC-V)/predicate(arm), 或者x86上相似的概念

mask是布尔值组成的，在向量寄存器中但不是存数据 。它用于指定向量寄存器中哪些值是活跃的，寄存器中对应活跃的值才被修改，或者是活跃的值被load/store覆盖

Vector编程推荐的模式

向量方式循环的流程：

首先count有多少元素要处理

如果要处理的元素 >= cpu可以处理的元素，所有值都活跃，尽可能处理，使用完整的向量寄存器
如果要处理的元素 < cpu可处理长度，忽略向量寄存器中多余的元素(mask tail)

这种方法不需要处理edge，因为mask和count结合，不用在任何一次循环关心要处理的元素数量，这里和定长unroll的逻辑不同。

对齐

SIMD的一个常见问题是load/store希望按照向量长度对齐，但在Vector方式如sve/rvv只要按照元素对齐

rvv的使用和特色

vsetvli t0, a4, e16, m1, ta, ma

a4: 需要处理的元素数量

t0: 这次处理的元素数量 它会结合硬件的向量长度和需要处理的元素数量被设置

e16: 元素大小 有8,16,32,64

m1: unroll / grouping 1/8 1/4 … 4, 8 表示向量寄存器分组，用于方便的unroll, 后面详细说

ta: agnostic 尾部处理策略

ma: agnostic mask处理策略

multipler

mf8 mf4 mf2 m1 m2 m4 m8

1/8 1/4 1/2 1 2 4 8

RVV的unroll: 配置multipler

默认m1时 有32个向量寄存器，v0,v1,v2,v3…v31

m2时 有16个向量寄存器 v0,v2,v4,…v30 是所有的偶数，每个向量寄存器长度加倍

m4时 有8个向量寄存器 v0,v4,v8… 每个向量寄存器长度4倍 …

这样在unroll时，只需要修改vset的配置，而且不需要让代码变长和麻烦的修改寄存器名

另外也可以模拟cpu支持更长的向量长度，只是向量寄存器变少

简单的rvv实例

```c
static inline void abs_pow34_v(float *out, const float *in, const int size)
{
    for (int i = 0; i < size; i++) {
        float a = fabsf(in[i]);
        out[i] = sqrtf(a * sqrtf(a));
    }
}
```

```
func ff_abs_pow34_rvv, zve32f
1:
        vsetvli     t0, a2, e32, m8, ta, ma
        sub         a2, a2, t0              // size -= t0
        vle32.v     v0, (a1)                // load in
        sh2add      a1, t0, a1              // in += t0
        vfabs.v     v0, v0                  // fabsf(in[i]);
        vfsqrt.v    v8, v0                  // sqrtf(a)
        vfmul.vv    v8, v8, v0              // a * sqrtf(a)
        vfsqrt.v    v8, v8                  // sqrtf(a * sqrtf(a));
        vse32.v     v8, (a0)                // out[i] =
        sh2add      a0, t0, a0              // out += t0
        bnez        a2, 1b                  // if size != 0, loop

        ret
endfunc
```

ffmpeg中自动向量化和手动实现

```
static void vp8_v_loop_filter_simple_c(uint8_t *dst, ptrdiff_t stride, int flim) {
  int i;
  for (i = 0; i < 16; i++)
    if (vp8_simple_limit(dst + i, stride, flim))
      vp8_filter_common(dst + i, stride, 1);
}
```

```
static av_always_inline int vp8_simple_limit(uint8_t *p, ptrdiff_t stride,
                                             int flim)
{
    LOAD_PIXELS
    return 2 * FFABS(p0 - q0) + (FFABS(p1 - q1) >> 1) <= flim;
}
```

ffmpeg中自动向量化和手动实现

```
static av_always_inline void filter_common(uint8_t *p, ptrdiff_t stride,
                                           int is4tap, int is_vp7)
{
    int __attribute__((unused)) p1 = p[-2 * stride];
    int __attribute__((unused)) p0 = p[-1 * stride];
    int __attribute__((unused)) q0 = p[0 * stride];
    int __attribute__((unused)) q1 = p[1 * stride];
    int a, f1, f2;

    a = 3 * (q0 - p0);

    if (is4tap)
        a += clip_int8(p1 - q1);

    a = clip_int8(a);

    // We deviate from the spec here with c(a+3) >> 3
    // since that's what libvpx does.
    f1 = FFMIN(a + 4, 127) >> 3;

    if (is_vp7)
        f2 = f1 - ((a & 7) == 4);
    else
        f2 = FFMIN(a + 3, 127) >> 3;

    // Despite what the spec says, we do need to clamp here to
    // be bitexact with libvpx.
    p[-1 * stride] = av_clip_uint8(p0 + f2);
    p[ 0 * stride] = av_clip_uint8(q0 - f1);

    // only used for _inner on blocks without high edge variance
    if (!is4tap) {
        a              = (f1 + 1) >> 1;
        p[-2 * stride] = av_clip_uint8(p1 + a);
        p[ 1 * stride] = av_clip_uint8(q1 - a);
    }
}
```

ffmpeg中自动向量化和手动实现

vp8_v_loop_filter_simple_c函数在用最新的RISC-V gcc -O3 生成汇编代码包含标量实现和向量实现，首先会判断使用标量实现还是向量实现，然后跳转到了使用标量实现

如果把选择标量还是向量的过程删掉，只留下向量实现，那么测试结果是，它生成的向量版本比标量部分的实现性能差50%

但是手写向量实现版本，能得到是标量版本4倍的性能，所以在这个例子中手写的向量版本是生成的向量版本8倍的性能

这个例子中的问题

1. 没能用好multipler， 生成的 multipler <= 1, 而手写可以用到multipler = 2
2. 生成的向量指令大量使用了splat模式，总是进行向量和向量之间操作，这是不必要的，很多时候v扩展可以在向量和标量之间操作
3. 计算中涉及两个16位数操作，在生成的向量版本中，会加宽到32位, 再裁剪，多次用到vset, 加宽裁剪指令，其实从解码器context来看，是不可能到32位的 这些指令手写就省下了
4. 尽可能多的使用了mask, 但mask是比较慢的，大部分ffmpeg中的向量代码都用不到mask, 这里虽然必须用到，但手写会选择尽可能少使用

ffmpeg中自动向量化和手动实现

5. 其中vp8_filter_common涉及将整数裁剪到0-255或-128-127, 生成的代码对应了c语言的方式，用了多条指令，其实
裁剪0-255只需要 vmax和vnclipu两条指令，裁剪-128-127从c语言看比裁剪无符号复杂，但是其实向量指令更简单，只需要一条vnclip指令

```
static av_always_inline av_const uint8_t av_clip_uint8_c(int a)
{
    if (a&(~0xFF)) return (~a)>>31;
    else           return a;
}

#define clip_int8(n) (av_clip_uint8((n) + 0x80) - 0x80)
```

进行中的重点

maintainer remi的review

孙越池在优化多个编解码器，比较重要的可能是vp8,vp9, vp8最近刚刚基本完成，vp9大概完成了一半（都在review）

rise 在202312发布了ffmpeg RISC-V h264优化项目，在做，也许会突然公开出来

THANKS

开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler